A META-LEVEL ARCHITECTURE FOR PROTOTYPING OBJECT SYSTEMS

by
Jeff McAffer

A Dissertation

Submitted to

The Graduate School of
The University of Tokyo
in Partial Fulfillment of the Requirements
for the Degree of Doctor of Science
in Information Science

September 1995

©1995, Jeff McAffer

ABSTRACT

As applications become larger and more complex, it is frequently the case that system components require varying models of computation. The use of different computational models is not well supported by standard object-oriented mechanisms and systems. Typical mechanisms implicitly encapsulate meta-level (i.e., computational) semantics along with the base-level (i.e., domain) behaviour. Objects defined using one model cannot easily be executed under another and so cannot be reused. A major problem is the inclusion of base-level language constructs in the meta-level architecture design. Meta-levels typically only facilitate concepts which are similar to those in the original base-level language and so cannot describe widely differing models of execution. We present a meta-level architecture founded on the novel principle of fine-grained, operational decomposition of the meta-level into objects. Unlike others, our approach bases the design of the architecture on the operations which occur during object execution (e.g., send, lookup) rather than the structural nature of an object's representation (e.g., class, method). This clearly separates the elements of the meta-level from those of the base-level language and so opens the meta-level to more radical change. The power of this approach is shown via several markedly different object models and their combination and non-intrusive application to user code. We detail how computational domains are completely altered with almost no modification of the original application code or its semantics. This capability is applied to real-world problems in object reuse, object behaviour investigation and in novel application design. Detailed examples relating to distributed computing and object communication are presented. It is shown that this approach to meta-level design is more open and flexible, and better supports the application of common software engineering practices (e.g., encapsulation and reuse) to the components of the meta-level — Properties desired by anyone designing complex systems.

Contents

1 Introduction **1**
- 1.1 Existing architectures . . . 3
- 1.2 Our approach . . . 6
- 1.3 Meta-level applications . . . 8
- 1.4 Contributions . . . 10
  - 1.4.1 Conceptual contributions . . . 10
  - 1.4.2 Practical contributions . . . 11
- 1.5 Overview . . . 12

2 Background **14**
- 2.1 3-Lisp . . . 14
- 2.2 3-KRS . . . 15
- 2.3 CLOS . . . 15
- 2.4 ClassTalk . . . 16
- 2.5 ABCL/R2 . . . 16
- 2.6 AL-1/D . . . 16
- 2.7 RbCl . . . 17
- 2.8 Apertos . . . 18
- 2.9 OpenC++ . . . 18
- 2.10 Actalk . . . 18
- 2.11 Summary . . . 19

3 CodA **20**
- 3.1 The CodA object system . . . 20
- 3.2 Operational decomposition into objects . . . 20
- 3.3 Meta-level decomposition . . . 22
- 3.4 Behaviours . . . 23
  - 3.4.1 Send . . . 24
  - 3.4.2 Accept . . . 24
  - 3.4.3 Queue . . . 25
  - 3.4.4 Receive . . . 25
  - 3.4.5 Protocol . . . 26
  - 3.4.6 Execution . . . 26

3.4.7 State . . . 27
3.5 Meta-level framework . . . 27
3.5.1 Object models . . . 29
3.5.2 Object model combination . . . 31
3.6 The ConcurrentObject model . . . 33
3.7 Abstraction, compression and expansion . . . 34
3.8 Summary . . . 35

4 Ported Objects **36**
4.1 Meta-level design . . . 37
4.1.1 Send . . . 39
4.1.2 Accept . . . 39
4.1.3 Queue . . . 40
4.1.4 Receive . . . 40
4.1.5 Execution . . . 40
4.1.6 Example . . . 41
4.2 Applying PortedObjects . . . 41
4.3 Compound ported objects . . . 43
4.4 Summary . . . 44

5 *Tj* **46**
5.1 Distributed system infrastructure . . . 47
5.1.1 Object spaces . . . 47
5.1.2 Remote references . . . 48
5.2 The DistributedObject model . . . 49
5.2.1 Distributed object execution . . . 49
5.2.2 Marshaling . . . 50
5.2.3 Marshaling policies and optimization . . . 52
5.3 The MigrantObject model . . . 54
5.3.1 Computational migration . . . 55
5.3.2 State migration . . . 56
5.4 The ReplicatedObject model . . . 56
5.4.1 Replication example . . . 57
5.5 Implementation . . . 60
5.6 Summary . . . 61

6 CodA implementation and use **63**
6.1 Implementation strategy . . . 65
6.2 Per-object meta-levels . . . 66
6.3 Behaviours and meta-components . . . 70
6.3.1 Changing behaviours . . . 71
6.3.2 Adding behaviours . . . 72
6.4 Messaging . . . 72

6.4.1 Message sending reification . . . 72
6.4.2 Message accumulators . . . 74
6.4.3 Debugging with messages . . . 75
6.5 Programming with CodA . . . 75
6.5.1 Code changes . . . 75
6.5.2 Optimization . . . 76
6.6 Summary . . . 77

7 Applications **79**
7.1 N-Body . . . 79
7.1.1 Problem description . . . 79
7.1.2 Adding concurrency and distribution . . . 80
7.1.3 Changes to original code . . . 81
7.1.4 Experiments . . . 83
7.1.5 Summary . . . 86
7.2 Expert system . . . 86
7.2.1 Problem description . . . 86
7.2.2 Adding concurrency and distribution . . . 87
7.2.3 Changes to original code . . . 88
7.2.4 Experiments . . . 89
7.2.5 Summary . . . 90
7.3 Vibes . . . 91
7.3.1 Problem description . . . 91
7.3.2 Analysis framework . . . 92
7.3.3 Monitoring . . . 93
7.3.4 Summary . . . 94

8 Evaluation **96**
8.1 Capability . . . 96
8.1.1 CodA . . . 98
8.1.2 CLOS MOP . . . 99
8.1.3 ABCL/R2 . . . 99
8.1.4 AL-1/D . . . 100
8.1.5 RbCl . . . 101
8.1.6 Apertos . . . 101
8.2 Performance . . . 102
8.2.1 Execution performance . . . 103
8.2.2 Performance perspective . . . 104
8.3 Summary . . . 105

9 Conclusions **106**
9.1 Perspectives and future work . . . 107

A Default Meta-component code **113**

B Example code **115**
B.1 N-Body . . . 115
B.1.1 Particle . . . 115
B.1.2 QuadTree . . . 116
B.1.3 Solver . . . 118
B.1.4 Distributed QuadTree . . . 119
B.1.5 Distributed Solver . . . 121
B.1.6 Invocations . . . 121
B.2 Expert system . . . 123
B.2.1 Annotations . . . 123
B.2.2 Additions . . . 126

List of Figures

1.1 Receiver dependent message sending . . . 3
1.2 Retro-reification . . . 4
1.3 Receiver dependent sending implemented with methods . . . 5
1.4 Receiver dependent sending implemented with objects . . . 8
1.5 Project overview . . . 10

3.1 Sample meta-level configuration and interaction . . . 23
3.2 A standard class configuration . . . 30
3.3 Class object models . . . 30
3.4 Object model collision . . . 32
3.5 Meta-level behaviour compression and expansion . . . 34

4.1 The PortedObject meta-level . . . 39
4.2 Correlation annotations . . . 42
4.3 Compound object example . . . 43
4.4 Compound object parameter handling . . . 44

5.1 RemoteReferences and the meta-level . . . 49
5.2 Distributed object layout . . . 58
5.3 The making of a replica . . . 59

6.1 Meta-level structure . . . 65
6.2 The CodA meta-level class hierarchy . . . 66
6.3 The object-identity problem . . . 67
6.4 Dynamic reification of the meta . . . 69
6.5 Implementation of the meta-level . . . 70

7.1 Messaging behaviour without replication . . . 84
7.2 Messaging behaviour with replication . . . 85
7.3 Overview of **ENVY**/*Expert* . . . 87
7.4 Example modified rule . . . 90
7.5 Overview of Vibes . . . 92
7.6 Monitoring Send activity . . . 95

B.1 N-Body application class hierarchy . . . 115

List of Tables

6.1 Message sending forms 73

7.1 Summary of N-Body code changes and additions 82
7.2 Summary of expert system changes and additions 89

8.1 The relationship between meta-level architectures. 97
8.2 Levels of support. 97
8.3 Mapping from AL-1/D to CodA. 100
8.4 CodA/*Tj* performance 103

Acknowledgements

I would like to thank my supervisor, Professor Akinori Yonezawa for providing such a rich environment of ideas, visitors and machines in which to work. Key in that environment was Satoshi Matsuoka whose questions were always as incisive as his memory is vast.

I would not have been able to do this work if it were not for Dave Thomas and Brian Barry. Their continuing moral, technical and financial support has been more than that for which any person could hope.

Perhaps most importantly, I thank all the people who made the Yonezawa lab what it was during my time there; Jean-Pierre Briot, my cohort and fellow Smalltalk evangelist. Jacques Garrigue and Kentaro Torisawa, constant dinner companions, philosophical combatants and providers of much needed sounding board technology. Kenjiro Taura, for lots of discussion and cheerfully answering my endless stream of AP1000 and other questions. Hidehiko Masuhara for his thoughtful comments and discussion and Naohito Sato for guiding me through the wilderness of TEX and Mathematica and generally making life in the lab pleasant. And finally, all the other lab members (particularly the systems managers) over the years.

Very special thanks go to all the people at Object Technology International who built much of the technology I used in my work. You are an exceptional group of people and I am proud to be associated with you.

Finally, I thank my friends and family around the world for being who they are and doing what they do.

This work was supported in part by Object Technology International, the Japanese Ministry of Education (Monbusho) and the Canadian Government's Japan Science and Technology Fund.

Chapter 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 are opaque to the user/client who does not need to see inside the box to solve their problems. In practice, this is not always possible or desirable.

Consider someone wishing to migrate their existing applications from a uniprocessor environment to a distributed environment? Should they have to rewrite a large part of their system to accommodate the change in computational models? The typical view of objects as encapsulators of domain and computational behaviour says, "Yes, programmers must open the whole black box to change part of it.".

The more general issue is that library designers cannot know or guess all possible uses of the classes they build. 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. In many cases we want to reuse and object's semantics but not necessarily its computational behaviour.

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

Looking at the same problem from a different point of view we see that ideally, (sub)systems are designed in whatever computing paradigm best suits their requirements. Unfortunately, what is good for one part of a large or complex system may not be appropriate for another. Furthermore, what is good at one point in time is not necessarily good at the next. Designers cannot accurately predict the actual or best behaviour of their application elements when combined. For the most part, they cannot even see a way to combine elements from radically different domains.

While 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 designing their objects' *computational* behaviour. Unfortunately, the

computational behaviour of an object is generally hidden in the language environment or is defined inside the object's black box.

These situations lead to a requirement for an explicit representation of an object's computational model. Given this, interaction between objects using disparate styles is facilitated by interfacing the computational models. Moreover, given a generalized framework for modeling these behaviours, much of the work of their combination and interaction is eliminated. New models are created to suit evolving demands.

The idea of separating implementation from semantics is not completely new. The *parameterization* available in systems like the Mach Virtual Memory manager [45] and *annotations* (or *pragmas*) in languages like High Performance Fortran (HPF) are but two examples. Parameterization allows users to adjust the implementation by changing tolerances or strategies within the framework of a particular implementation. Annotations are used to change the implementation strategy itself. Regardless of the approach, the key is a clear and explicit exposure of the computational concepts which is independent of domain semantics.

This is the goal of researchers in meta-level architectures and reflective computing. One of the fundamental tenets 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 matter, but changes at the meta-level affect the execution of the base-level. The base-level defines the domain-specific behaviour of the object while its meta-level defines the computational behaviour of the base-level. 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 emerging field of *open implementations* [32]. Open implementations trade-off the clarity and simplicity of the black-box model and the freedom and flexibility of more open but less protected 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 their situation.

The general goals of an architecture for open implementation are:

Separation The implementation of an object must be explicitly exposed and clearly distinguished from the object's domain-specific behaviour description.

Expressiveness The expression of a wide range of computational behaviours must be supported.

Extensibility Unanticipated computing patterns must be facilitated and integrated in a seamless way.

Programmability The architecture as a whole must follow and support standard software engineering concepts such as frameworks and reuse. It must also limit its intrusion on both the host environment and the base-level code.

Much of the work in meta-level architectures and reflection has gone into the base- and meta-level separation. As a result, most architectures provide a clean and discernible interface between

the two levels while the remaining three goals are not completely supported. This is unfortunate since meta-levels are potentially complex pieces of software. They could benefit greatly from the inclusion of better engineering support.

1.1 Existing architectures

Building open meta-level architectures is particularly challenging because of the diversity of behaviours (e.g., kinds of message sending) we wish to describe while maintaining a uniform base-level view of object behaviour (e.g., the existence of message sending in the system). Without the support of sound software engineering techniques, the meta-level becomes confused and unmanageable. Base-level programmers may benefit from the architecture but meta-level programmers will find it wanting.

Most of the current architectures are expressive and extensible but in restricted senses. The main problem is that they reify only the physical concepts which exist in the base-level language (e.g., classes, methods, slots, etc.). We call this a *top-down* approach. The top-down approach has a number of positive attributes. The restricted domain allows the meta-level programmer interface and use-patterns to be simple, clear and well-defined. Implementations can take advantage of this to make earlier, more efficient strategy choices resulting in performance improvements. The overhead to programmers is reduced as much of the configuration and administration work is done by the system.

Despite these benefits, top-down models also suffer due to their restriction. Their expressiveness is limited to concepts which are in the original language system. Many do not provide facilities for significant extension. Concepts or behaviours which are not in the original design cannot (easily) be added. Consider the case shown in Figure 1.1 where an object wishes to vary its message passing mechanism based on the identity or type of the message receiver. (See Section 4.3 for a real-world instance of this problem.) In the figure, the code for object Source sends the foo message to the objects Dest1 and Dest2. foo messages sent to Dest1 should be handled normally while foos sent to Dest2 should be modified in some way to be foo++ messages.

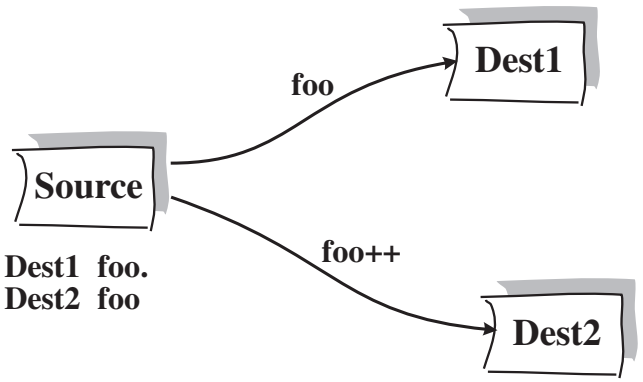


Figure 1.1: Receiver dependent message sending

The non-meta-level solution to this problem is to embed the desired behaviour in the base-level

program. This is approach restricts the reuse of the object in different situations. It also requires considerable effort on the part of the programmer (to add the required control structures) and clutters the domain code with behaviour code.

The meta-level approach is to implement the different sending mechanisms at the meta-level and then implicitly or explicitly determine which to use from the base-level. Depending on the architecture, this may or may not be easy. Many languages have only one notion of message sending and their meta-levels do not provide any facilities for its modification — Message sending is a given constant for that language. Languages and meta-level architectures such as the CLOS MOP, Smalltalk and C++ are examples of this. Since message sending is not explicitly reified, the best we can do is to *retro-reify* them.

An example of retro-reification is shown in Figure 1.2. Here we have the same message sending situation as in the earlier example but now Source's and Dest2's meta-levels are shown. Note that Dest1 is not relevant here and so is not shown. Retro-reification is implemented by observing that message reception can always be reified simply by intercepting arriving messages. Taking advantage of this, Dest2's meta-level is modified to trap foo messages and pass them back to the sender's meta-level for possible modification by its message sending operation.

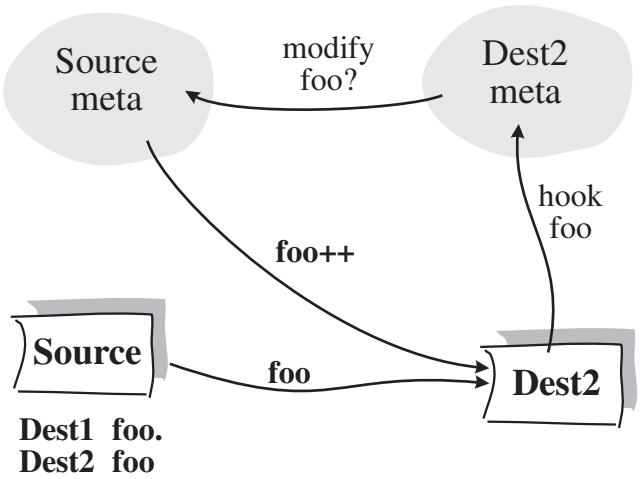


Figure 1.2: Retro-reification

When the foo message arrives at Dest2 it is trapped by the meta-level and passed back to Source's meta-level. Source's meta-level checks the message and its receiver and determines that a modification is required. The foo message is modified into the foo++ message which is finally passed to Dest2.

While this approach works, it has a number of drawbacks in addition to begin somewhat indirect and confusing. The first is that we must know all possible receivers of foo messages and modify their meta-levels to retro-reify (i.e., trap) the sending of those messages. This incurs overhead on all sends of foo regardless of whether or not modification is required. It also requires the ability to describe instance-specific behaviour (i.e., modify just Dest2's behaviour). This is quite difficult in systems like the CLOS MOP because the CLOS language implementation is based on method

invocations rather than message sends. The CLOS language model does not include facilities for object-specific behaviour specification.

Retro-reification is an issue only where message sending is not explicitly reified by the system's meta-level. This is typical of non-concurrent systems which have no particular need for different kinds of message sending. Concurrent systems on the other hand, typically provide reifications of sending, in the form of a method on some meta-level object. This approach suffers from engineering and scalability problems.

Figure 1.3 shows a typical meta-level with the send operation reified as methods. Note that the send operation's lasting state (if any) is maintained in instance slots of the metaobject itself. The figure shows this approach applied to the message sending problem discussed above. The different behaviours are implemented in different methods, send1 and send2, on the metaobject.

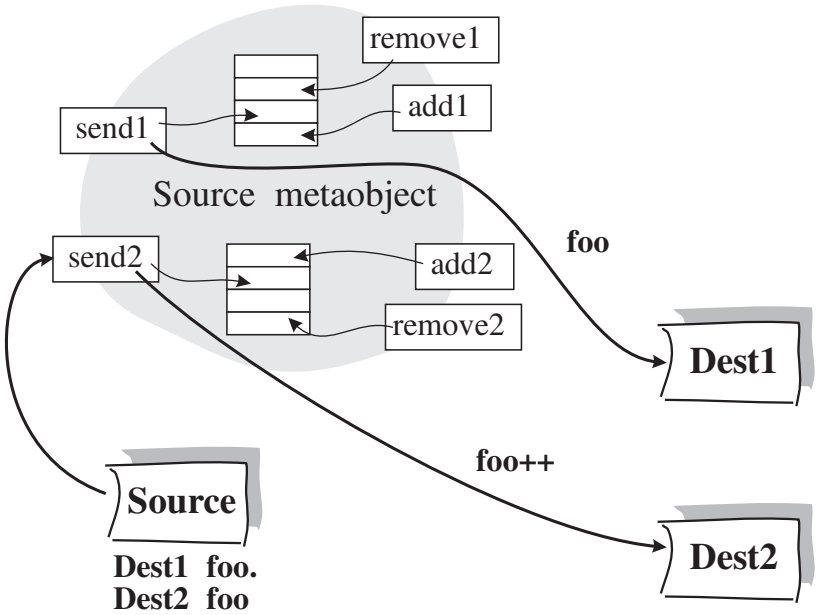


Figure 1.3: Receiver dependent sending implemented with methods

While this approach works for small numbers of behaviours, it does not scale well. For example, we see in Figure 1.3 that send1 and send2 each have independent state vectors required to describe their behaviour. Since the send operations are methods on some metaobject, these state vectors are held in instance slots of the metaobject. As such, the methods for modifying and maintaining the state vectors must also be added to the metaobject (e.g., add1, remove2).

In addition to simple naming problems related to having many methods and slots with similar purposes, there is the issue of duplicate code and behaviour management. In many cases the send related methods are very similar or identical. The primary difference is in the values in the state vectors they manipulate. As such, these methods (e.g., add1 and add2) are largely duplicate code.

Further, the maintenance and infrastructure methods are logically part of their related send operation but are not physically represented as such. They are simply methods on some metaobject.

Manipulating operations as whole, consistent modules is not supported. All of these problems are exacerbated when we consider that typical metaobjects describe several different operations (e.g., sending, receiving and queuing). Each of these has its own state vector and infrastructure methods. Combining these on one (or a small number of) metaobject(s) makes the meta-level a cluttered and confusing place.

Fundamentally, *methods* are not suitable structures for describing meta-level operations. Methods simply do not provide the infrastructure and abstraction necessary for describing more than very simple behaviours. They do not directly support reuse, combination or composition. They are not suitable units of encapsulation for engineering the meta-level.

1.2 Our approach

Objects on the other hand do provide the necessary abstractions and infrastructure for describing complex interactions. While objects are generally used in meta-level architectures, we claim that their coarse-grained top-down application restricts the overall scope and usefulness of the architecture.

In contrast, we take a fine-grained *bottom-up* approach. Rather than starting with and then opening particular base-level language elements (e.g., classes), we start by describing the basic elements of generic object behaviour (e.g,. message sending, state). We then provide infrastructure for composing these behaviours into specific object models. This is very much related to basic concepts in object-oriented software engineering — decompose a specific problem into generic components and then compose the pieces to solve the problem. From this comes both a solution to the problem and a set of components which can be used in other solutions.

Using the bottom-up approach we have developed CodA[27], a meta-level architecture capable of 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. Our approach is summarized in the following statement:

> *The CodA meta-level architecture is based on an operational decomposition of meta-level behaviour into objects and the provision of a framework for managing the resultant components.*

This statement captures the architecture's three major principles; *operational decomposition*, *decomposition into objects* and *provision of a framework*. We claim that following these principles in designing the meta-level is novel and that it leads us to a system which is more capable and more expressive than existing systems. Furthermore, these principles directly support the general meta-level design goals as set out above.

An operational decomposition is a factoring of the meta-level by operation (e.g., message sending or method lookup) rather than physical entity or programming language element (e.g., evaluator or class). Each meta-level operation which occurs during the execution of an object is treated as a separate element of the meta-level. The focus is on the dynamic rather than static structure of the system. That is, on what occurs not how it is organized.

Typical decompositions seek to represent the physical structure of a system (e.g., how its objects are defined and organized). As we saw with the message sending example above, this leads to meta-levels in which we cannot easily express unanticipated dynamic behaviours. In addition, by focusing on particular language elements, these architectures further limit their expressiveness. For example, architectures which use only classes as the unit of object description do not admit the description of prototype-based object behaviour. We can design an architecture which specifically facilitates both classes and prototypes but then there will be some third notion which does not fit either mold and so cannot be integrated.

By focusing on execution operations, we avoid these issues and get at the very essence of an object — its executional semantics. All objects, regardless of their structure, configuration and base-level semantics, reduce to the same common set of basic conceptual operations (i.e., primitives). The definitions of these operations may differ but if the operational concepts are simple/basic enough, we can capture all elements of common languages.

It can be said that other systems reify the operational side of object behaviour because they define meta-level methods which describe the operations. As we saw above, this approach is severely flawed with respect to the engineering of the meta-level. We apply object encapsulation to the meta-level in a bottom-up fashion. Each of these operational descriptions is individually encapsulated in a distinct object at the meta-level. The result is a reification of object behaviour in which each operation has clearly defined responsibilities and interfaces, and can exist in relative independence of other operations.

This closely matches the needs of sound object-oriented software engineering practice: Factor out common attributes, 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 behaviour of objects. That is, the meta-level is just an application whose domain happens to be the behaviour[1] of objects.

Applying this approach to the message sending problem above, we take the send method, its associated state vector and maintenance methods and group them together in one Send object as shown in Figure 1.4. The definition of this operation is then instantiated multiple times and used in different contexts and for different objects. The figure shows one instance of the same kind of Send used to handle each of the sending behaviours for Source.

Unfortunately, the simple creation of fine-grained operational meta-level objects is not enough. Larger-grained structural (i.e., traditional) decompositions provide users with abstractions which closely match the concepts they manipulate while programming (e.g., classes, inheritance). While beneficial for initial understanding and ease of use of the meta-level, this technique is not fully extensible. New operations which span the responsibility borders of these implicit abstractions or perhaps do not fall within the domain any existing abstraction, have no clear host object at the meta-level. Such operations tend to be 'tacked onto' or 'spread over' somewhat inappropriate meta-level objects.

The operational approach removes these abstractions to gain generality but in doing so, leaves the programmer in a sea of meta-level objects with no organizational infrastructure. To take the place of the implicit abstractions found in most systems, we add a generic framework for organizing

[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.

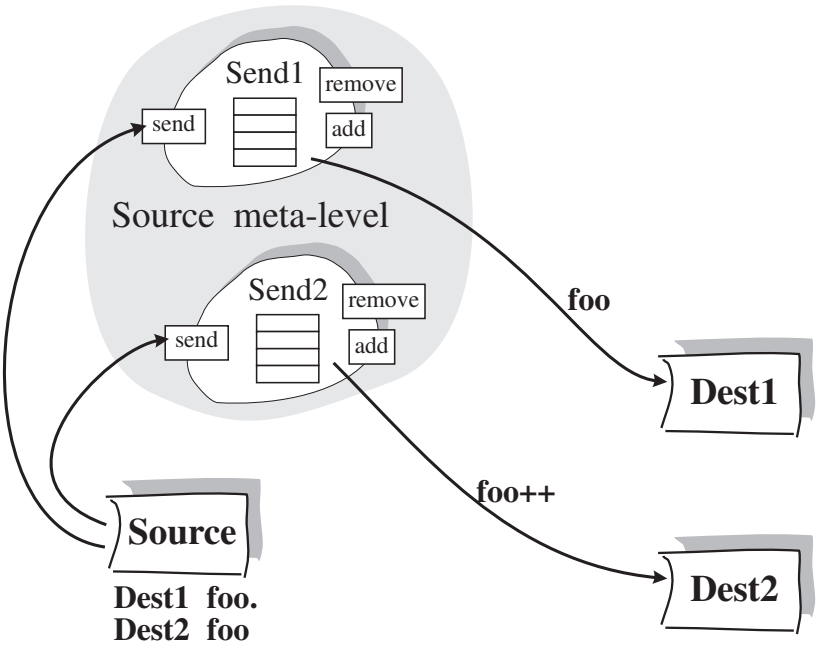


Figure 1.4: Receiver dependent sending implemented with objects

the meta-level. The most important feature of the framework is that it is free of semantics. It is uniform. All operations are created equal and are treated the same way. There is no confusion or ambiguity as to where operations are described or how they are managed. They are all hosted by objects of their own and structurally managed by a distinct framework. Arbitrary extension is easy because there are no semantics attached to the structure. Such frameworks are fast becoming a standard object-oriented software engineering tool. We apply it here to managing object behaviour at the meta-level.

Throughout this document we show that by following the three principles outlined above (i.e., (1) operational decomposition (2) into objects and (3) the provision of a framework), we can meet our goals of creating an expressive, extensible and programmable environment in which to describe an object's computational behaviour as distinct from its domain behaviour. Furthermore, we show that because this approach supports a wide range of object behaviours and can be closely integrated with existing language systems, we can provide an environment for exploration and experimentation in many behaviour domains.

1.3 Meta-level applications

Key to demonstrating the usefulness and expressiveness of our architecture are the descriptions of several diverse object models, in particular, those related to communications (PortedObjects) and distribution (DistributedObject, ReplicatedObject and MigrantObject). These models represent non-trivial changes to standard object behaviour and are motivated by real-world application

requirements.

The models related to distributed computing create an environment which equals dedicated distributed languages in terms of sophistication but far outstrips them with respect to generality. The distribution components and models are largely orthogonal to normal object computation and so can be completely integrated with the underlying implementation environment in an seamless way. Very few existing systems employ explicit meta-level architectures as the basis of their system design. We claim that this approach, and primarily, the ability to change arbitrary objects into distributed (e.g., replicated, migrated) objects without affecting their semantics, is a major step forward in design and building of distributed systems. In addition to using these facilities for the creation of a completely transparent distributed Smalltalk system, *Tj*, we demonstrate the benefits of our approach both in the areas of behaviour investigation and execution domain changes.

Tj incorporates the full power of our meta-level architecture and adds distributed computing specific mechanisms. The use of this general architecture is warranted because describing distributed object behaviour is more than just implementing remote references. *Tj*'s DistributedObject model contains comprehensive notions of object spaces, machines, topologies, remote references and object marshaling. Further models describe replicated and migratory objects. These mechanisms are implemented in an open and extensible way so as to accommodate user changes and additions. The mechanisms themselves are reified as meta-level components and provide a place to define both their implementation and their use (i.e., policies).

To be able to effectively investigate object behaviour we must be able to view and modify application object implementations. The provision of an explicit meta-level decomposed operationally clearly and decisively separates the base-level semantics from the meta-level operations and policies. As a result, application behaviour can be changed with very little impact on base-level code and semantics. We present an N-Body problem solver application to demonstrate how this is done and show how effective a tool this technique can be.

In addition to behaviour investigation, there is also a large demand for computational domain changes or adaptation. For example, many people have large bodies of code (e.g., class libraries) which are targeted at a particular execution environment or architecture (e.g., sequential, uniprocessor). The spread of distributed and multiprocessor machines call this restriction into question. Application builders would like to be able to effectively reuse their class libraries in these new environments without major changes to the original code. The transparency and integration of CodA and *Tj* enables this. We show that applications written for single processor (i.e., non-distributed) environments can be modified to run in distributed environments with almost no disruption of the application itself.

In a different vein, work with PortedObjects (see Chapter 4) and the Vibes application (see Section 7.3) demonstrates the building of an application which is based on the capabilities of an advanced meta-level architecture. The Vibes problem domain, data analysis, is best served by a dataflow architecture but the desired generality of analysis techniques motivates us to allow the use of generic objects in the analysis graph. The tension between these two demands is broken by enabling the modification, at the meta-level, of arbitrary objects to have dataflow, or *ported*, behaviour. Using this facility, we design and build an analysis system in which analysts compose everyday (and special-purpose) objects to create specific analysis systems. This gives Vibes a major advantage over dedicated analysis systems in which analysts must explicitly program analysis

objects rather than simply reusing existing libraries.

Such power is a direct result of our architecture's explicit reification of object computational semantics separate from base-level semantics which is in turn facilitated by our unique approach to meta-level factoring — operational decomposition. The operational approach is shown to produce a clearer separation of, and a wider scope for, behaviour description.

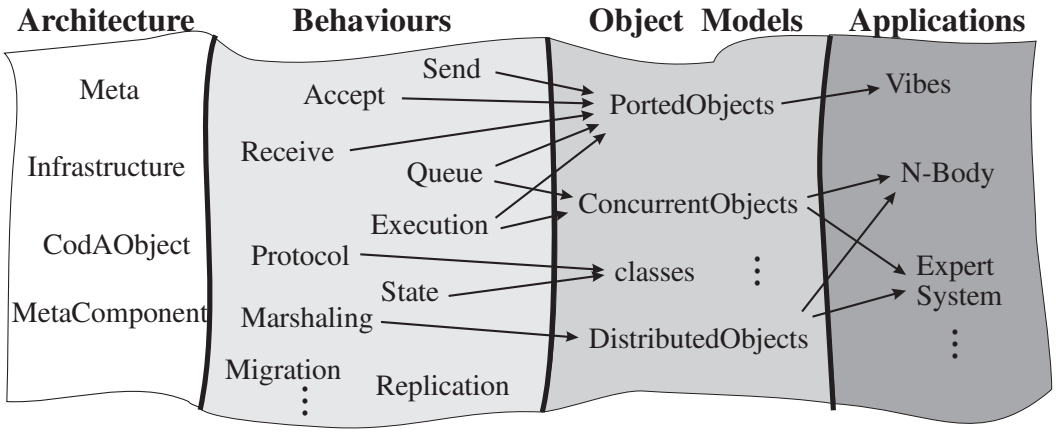


Figure 1.5: Project overview

An overview of CodA, *Tj* and the developed object models and applications is shown in Figure 1.5. On the far left are the fundamental building blocks of our system — Abstract reifications of the main meta-level concepts and the infrastructure for performing meta-level operations. Using this framework we developed a number of meta-components, some optional, some required, which realize our operational decomposition of the meta-level. These components are combined into object models, or coherent definitions of object behaviour. Finally, using the developed framework, components and models, we demonstrate the capabilities and expressiveness of the system by implementing or modifying the behaviour of various real applications.

1.4 Contributions

The main contributions of our work come in two broad categories; conceptual and practical. The conceptual contributions add ideas to the base of meta-level architecture theory while the practical contributions add experience to the current practice in employing meta-level architectures. The contributions in both areas are outlined below.

1.4.1 Conceptual contributions

Fine-grained operational decomposition into objects

CodA employs a novel strategy for structuring the meta-level; operational decomposition. Most other systems use a structural approach which results in larger-grained meta-levels which are closely tied to the base-levels they represent. In CodA, meta-level *operations*

are abstracted into concrete objects. These components are fully independent fine-grained objects with clearly defined interfaces and properties. They support composition, reuse and other common software engineering concepts.

Framework for extension

The CodA architecture contains a generic framework for managing the components of the meta-level. This infrastructure is independent of the behaviours being described. It is fully extensible. Additional components can be added and existing components can be modified or have their implementations combined. Other systems have either no real framework or a specific architecture which admits only certain kinds of behaviour description.

Meta-level engineering

The framework for behaviour extension scales up to the combination of entire object models. Object models abstract higher-level behavioural concepts from the lower-level operations which are the foundation of the architecture. Object models are constructed by the specification of constraints on relevant behaviours. Models are combined by merging these constraints. This approach inherently supports; the easier identification of points of conflict between models, isolation of the effects of changing individual behaviours, increased reusability of components derived from the combination of object models and easier/explicit management of the behaviour space.

Complete integration with a standard programming language/environment

We have implemented CodA in, and integrated it with, a standard, industrial quality Smalltalk system. Though the CodA architecture itself is independent of any particular language features, the Smalltalk implementation matches closely the underlying object system and virtual machine. Users retain all of the power of the Smalltalk programming environment (e.g., classes, browsers, inspectors, debuggers) and gain the ability to modify object behaviour on a per-use, per-object, per-class or global basis. Virtually any Smalltalk object can be manipulated in this way. CodA modifications to meta-level behaviour descriptions are transparent to users of the modified objects to the point where most of the environment's some 1,000 classes and 6,000 methods continue to work unchanged.

The implementation of CodA in Smalltalk has given rise to a number of novel implementation techniques. In particular, we have solved the *object-identity* problem common in systems which attempt to extend Smalltalk runtime semantics. Novel mechanisms for prototype-based computing, *futures* and dynamic message reification where also developed.

Using the architecture and developed object models, CodA has been adapted to run in distributed environment such as clusters of workstations and MPP computers such as the Fujitsu AP1000[38]. Implicit in this is that we have developed the first industrial grade Smalltalk to run on MPP class machines.

1.4.2 Practical contributions

Concrete examples of meta-level use

Our work gives concrete justification for and examples of meta-level use in both object model

and application design and prototyping. We present several quite different models of object behaviour which demonstrate the power of using full scale meta-level architectures.

For example, *Tj* contains distributed object models which rival the capabilities of existing dedicated distributed systems such as Emerald [22]. It supports replication, migration, a sophisticated object marshaling framework and a wide variety of messaging types in a way which is transparent to the base-level and uniformly applied across the whole system. This is only possible with a general architecture as CodA.

We also apply these ideas to several non-trivial applications including the entire Smalltalk system in which CodA is implemented and a commercial expert system tool which is modified to be both concurrent and distributed (see Section 7.2). We show that using an explicit meta-level, such changes can be done with relatively minor modifications to the base-level objects of the original system.

Environment for exploration

Perhaps one of the most significant contributions of this work is the cumulative effect of all these contributions gathered together into one system. Overall, CodA provides an open, powerful and expressive software development environment for exploring and experimenting with object behaviours. The architecture is based on sound object-oriented software engineering practice and its implementation is completely integrated with a powerful programming environment (Smalltalk). Moreover, the environment as a whole is transparent to the user allowing full reuse of existing class libraries where applicable.

1.5 Overview

The remainder of the this thesis is organized into eight chapters and a series of appendices. Following immediately is a discussion of background material and related work. In this chapter we highlight positive and negative points of past work and the general directions in which our goals and efforts differ.

Chapter 3 presents the CodA architecture, in particular, our approach to meta-level factoring. Object models, our mechanism for abstracting object behaviour, is detailed along with an educational example, the ConcurrentObject model. Further sections discuss techniques for managing and merging abstract descriptions of meta-level behaviour.

A detailed example of how CodA is used is presented in Chapter 4. This chapter presents the PortedObject model for dataflow style computing. The PortedObject model demonstrates the creation of radically new paradigms through the modification of existing object behaviours.

Chapter 5 describes *Tj*, a distributed object system built using CodA. *Tj*'s object models are examples of how entirely new behaviours added to objects. *Tj* augments all objects from the underlying environment with distributed computing operations such as remote referencing/messaging, marshaling, migration and replication. Distribution is completely integrated and transparent. Our discussion deals mostly with distributed computing mechanisms such as replication and their design using the CodA architecture. Topics such as applications and performance issues are left to further chapters.

Chapter 6 details the implementation and use of CodA in the Smalltalk object-oriented environment. CodA is integrated with Smalltalk and allows CodA and Smalltalk objects to interact and inter-operate. This chapter gives a number of concrete programming examples and insights into the mechanisms used to support the architecture.

CodA and some of the object models we have designed are combined and applied to a set of three problems in Chapter 7. The first application is a relatively regular N-Body problem solver. The application is adapted to be concurrent and distributed, and the use of CodA as a behaviour/performance investigation platform is demonstrated. The next application is the addition of concurrency and distribution to a commercial expert system tool. Here we show the transparency, integration and non-intrusiveness of our architecture. Finally we present a behaviour analysis application which makes extensive use of the CodA meta-level both for data gathering and analysis.

Chapter 8 evaluates the capabilities CodA relative to a representative set of existing architectures. The analysis focuses on CodA's success (or failure) in meeting the goals and objectives set out above. We also discuss CodA's performance in absolute terms to show that it is usable in addressing real issues in application and behaviour design.

A final chapter summarizes our research and relates the goals and approach to the results. We also discuss some possible routes of future work. A series of appendices provide readers with a more detailed view of CodA and *Tj*, their implementation and use.

Chapter 2

Background

The area of meta-level architectures, reflective computing and open implementations has received considerable attention of late. While most of the systems related to our work have differed either in focus or approach, they have all achieved their goals by opening or exposing the implementation of fundamental system components. Regardless of what behaviours they describe and how they are described, by accessing and manipulating the exposed interfaces, users modify and control the environment in which their objects live and execute. The net result of that is the ability to adapt objects to different and changing computational environments.

2.1 3-Lisp

The area of reflective computing and meta-level architectures really began with Brian Smith and Jim des Rivieres's work on 3-Lisp [11, 40]. 3-Lisp is a sequential Lisp which incorporates a meta-circular interpreter and the ability to *level shift*. Level shifting is typically done via a *reflective procedure* call and allows, for example, base-level programs to shift to the meta-level and carry out some reflective computation. The computation at the meta-level is carried out by an interpreter which is logically distinct from that of the base-level. Level shifting is a uniform operation applicable to any level (e.g., meta-level to meta-meta-level) giving rise to a logically *infinite tower* of levels or interpreters.

3-Lisp explicitly reifies code, the execution environment and the current continuation giving meta-level programmers access to all the vital elements of Lisp execution. New language or system constructs are added by modifying or reimplementing the interpreter for a given level (i.e., the level's meta-level).

This model captures the fundamentals of reflective computing and meta-level architectures. It is however, closely tied to the Lisp model of computing. It does not address issues specific to object-oriented computing and the framework it provides for meta-level modification and extension is relatively unsophisticated.

2.2 3-KRS

Carrying on from the 3-Lisp work, Patti Maes developed 3-KRS[24]. The 3-KRS language is object-based and its meta-level was one of the first to be object-based. The 3-KRS meta-level, rather than being a meta-circular interpreter, is represented by a series of *metaobjects*. Level shifting is uniformly integrated into the language model using standard message sending operations. That is, a level shift is just a message send to, or invocation of, a metaobject. Other than open the possibility of using standard object-oriented software development techniques, 3-KRS does not provide any additional infrastructure for managing the meta-level. Nor does it prescribe a particular architecture for the meta-level.

2.3 CLOS

The CLOS Metaobject Protocol (MOP) [23] came about as an attempt to unify several disparate object models implemented in Lisp. The basic approach was to identify and reify those parts of the various systems which were essential to defining their behaviour. By generalizing, abstracting and finally combining these behaviours at the meta-level, a unifying framework was produced. The architecture has six kinds of objects at the meta-level; classes, slots, generic functions, methods, specializers and method combinations.

This model does an excellent job of reifying, in objects, the structural nature of CLOS language. That is, the metaobjects (e.g., classes, slots, etc.) are derived directly from structures that programmers write or define when using the base-level language. It does not however, reify its operational concepts. Operations like message sending and method lookup are represented by methods on the appropriate metaobject. This approach is not inherently bad but it is different from the approach we have taken with CodA (see Section 3).

The CLOS MOP's provides a strong basis for the modification of existing language constructs but does not support a framework for the addition of new concepts which are not part of the existing model. Of course, since it is an open architecture, new concepts can be added but there is no generalized infrastructure for doing this. Similarly, the MOP, while implemented with objects, is not *oriented towards objects*. That is, it is difficult to modify the behaviour of individual objects.

Consider a user wanting to count all invocations of method foo for some object O. A first cut at this is to implement a new kind of method metaobject which counts invocations and insert this into the generic method chain for the selector foo for objects of O's class. Unfortunately, this affects all instances of O's class and its subclasses (who do not redefine foo) — the count will not be just for O. To get around this, the counting method metaobject must also maintain a list of objects for which it is counting and check the receiver against the list for every invocation. While this solution works, it introduces overhead in unrelated objects and does not scale to handle many different behaviours for many different objects.

2.4 ClassTalk

ClassTalk[8, 10] is somewhat like CLOS in that it is a reification of an existing object model. ClassTalk opens structural aspects of the Smalltalk language environment. In particular, the meta-class structure. This allows users to explicitly program metaclasses and thus modify the meta-level (i.e., the class) of an object.. It also opens a number of other system operations such as method lookup and execution.

This is not a fully general system however. Like CLOS, ClassTalk is a useful system within its language environment but it does not open widely the world of objects. Behaviours having to do with individual object execution are not addressed nor are operational mechanisms such as message sending.

2.5 ABCL/R2

The ABCL/R2 [25] was one of the first systems to introduce meta-level computing for concurrent objects. In doing this it reifies the concept of computing power and allows it to be manipulated by users. ABCL/R2 uses a *group-wide* approach to organizing and scheduling concurrent objects. Objects within a group share computing resources and can be manipulated as a whole by controlling the group itself. Groups are meta-level concepts.

The ABCL/R2 meta-level takes the interpreter approach of traditional systems like 3-Lisp. That is, the meta-level is basically a meta-circular interpreter. Shifting to the meta-level causes (logically) another interpreter to be created and invoked to interpret the meta-level to which you just shifted. Inherent in this approach is the idea of complete meta-level reification. If we wish to change just one aspect of an object's behaviour, we must, at least logically, reimplement its entire interpreter. This is in contrast to the approach taken in RbCl and Apertos (and in fact our work) where only those portions of the meta-level actually changed are reified.

Since every meta-level is a complete (logical) reimplementation of the interpreter, there is no need for a framework to organize the meta-level. The organization can be left up to the interpreter's implementor. This discourages object behaviour description reuse by making it difficult to integrate unrelated or third-party behaviours.

2.6 AL-1/D

AL-1/D[34, 33] is an application of AL-1's Multi-Model Reflection Framework (MMRF) [21, 20] to distributed systems. Within this architecture, an object's meta-level is partitioned into a fixed set of six *models* or views of base-level object behaviour. Below we summarize the parts of base-level object behaviour described within each of these models.

Operation Describes the operational semantics related to the programming language. Concepts such as message sending, state storage and stack manipulation are detailed here.

Resource Represents shared system resources such as CPU and memory.

Statistics Maintains various statistics for various aspects of object behaviour (e.g., CPU time, memory usage, blocking, etc.).

Distributed Environment (DE) Models the distributed system in terms of topology, networks, CPU and communications channel loadings, etc. Also maintains a global name server for object-to-location resolution.

Migration Defines how an object moves from one host to another and what objects are moved with it.

System Specifies the primitive system operations on which the entire system is built.

Overall this partitioning appears to reflect implementation issues rather than design issues. This is useful but ultimately leads to problems and confusion. Behaviours one might expect to see together, are split over meta-models (e.g., object monitoring and migration behaviour).

The partitioning is also relatively coarse-grained. One model, Operation, describes almost all of an object's execution behaviour. This is closely tied to the base-level language and does not serve as a sound basis for behaviour reuse or combination.

The *logical weight* of the models is not *balanced*. While the Operation model defines a great deal of behaviour, the Statistics model is just a storage area for monitored values. These are grossly different in complexity, scope and significance. While it may be convenient to have system statistics grouped in one place, this would not seem to be sufficient motivation for an entire model.

The details of moving an object from one place to another are split over several models (DE and Migration) rather than being one logical unit. The Migration model is billed as an OS abstraction but it includes specification of things like object attachment and slot inclusion which would seem to be higher level issues. The DE model, in addition to specifying argument passing strategies (e.g., call-by-move), deals with completely unrelated issues such as object name/location mapping and processor loading.

2.7 RbCl

RbCl [19, 18] is a reflective concurrent language system in which the meta-level is factored into objects. Metaobjects with different implementations (in different languages) can be plugged in and used as long as they understand the calling conventions. This is demonstrated with the free substitution of RbCl and C++ objects. Unfortunately, very little is said about what objects exist at the meta-level and how they interact.

The authors claim that RbCl is *kernel-less*. The essence of this claim is that every function point in the description of object behaviour (i.e., the meta-level) can be reimplemented by the user. As such, there is no fixed kernel. This is a nice idea as it gives the user complete power over the system but with no framework or infrastructure organizing the meta-level, this power is hard to manage.

RbCl is also set apart from other systems in that it reifies many of the low-level system operations like scheduling, interprocessor messaging device interfaces, etc. Though there could be more at the higher end, it has a reasonable span from system- to object-level behaviour descriptions.

2.8 Apertos

Apertos[43, 44] is an object-oriented operating system. As such, the objects it deals with are things like schedulers, virtual memory servers and device drivers. Even though this differs from our domain, the Apertos' meta-level architecture is of interest to us. The Apertos meta-level is reified in to metaobjects which are grouped into *metaspaces*. Metaspaces are structural concepts which have no real meaning at runtime. They can be arranged in a hierarchy similar to classes and are used to describe relatively complete sets of behaviour.

An Apertos object's behaviour is defined by the metaspace(s) in which it resides. To change its behaviour, we move if to a new metaspace. Any number of objects can reside in the same metaspace at the same time. Though Apertos does not reify many higher-level object behaviour concepts such as message handling (e.g., lookup) and state storage format (e.g., slot structure), in later chapters we will see that its fundamental architecture has much in common with our work.

2.9 OpenC++

OpenC++[9] is a static or compile-time meta-level architecture which allows users to modify the implementation of their objects. OpenC++ metaobjects represent C++ language elements and are produced when an OpenC++ program is parsed. Which metaobjects are created is controlled by meta-level directives or annotations made by the user. The language system has a small framework for annotating code to allow generalized annotations to be used. Once the code is parsed, the metaobjects, which essentially form a parse tree, are asked to produce code which implements the behaviour they describe. So, by using annotations to control metaobject creation, users can control the implementation of language constructs.

This system is interesting but is not fundamentally different from current object-oriented compiler technology seen in languages like Smalltalk. Having said that, we find we would like a similar system for optimizing out many of the runtime meta-level operations in our system.

2.10 Actalk

Actalk [7] is a meta-level architecture implemented in Smalltalk. It introduces actor-based concurrent objects to an otherwise passive object environment using meta-level manipulation. Its main goal is to provide a testbed for describing object behaviours in areas relating primarily to concurrent execution and message passing.

The system itself has become more and more component-based at the meta-level, but the basic architecture has remained somewhat monolithic and code-based. There are however, many object behaviours implemented in Actalk (e.g., intra-object synchronization constraints) which we look forward to implementing in CodA.

2.11 Summary

Each of these systems is a powerful tool within a particular domain. By in large, they were created as reifications of specific, pre-existing systems in an effort to open their implementation and facilitate user input. The way in which this has been done varies widely from system to system. The approaches used are typically not compatible with one another and the object descriptions from one architecture do not easily map onto a different architecture. In our work we strive to unify aspects of various existing systems and enable users to create and use object models from various domains within the same environment.

Chapter 3

CodA

3.1 The CodA object system

In creating the CodA architecture we developed a relatively generic model of objects. This model forms a basis from which most other object systems can be constructed. The CodA base-level object system contains the following three concepts:

State Each object has associated with it a set of *slots* into which it can store and from which it can retrieve other objects. These slots are for the exclusive use of their owner.

Execution Each object has associated with it a set of *methods* which it can execute. These methods can access state and sent messages.

Message passing Every object has the capacity to send a message to another object. All method executions are the consequence of a message send.

The CodA object system is a *pure* object system. That is, it has no elements which are not objects. Additionally, CodA is run-time oriented. Rather than providing integral support for language constructs like classes, inheritance or delegation required for the static description of objects, CodA borrows these constructs from whatever language is used in its implementation.

This is in contrast to the meta-level facilities found in systems like CLOS[23] and ClassTalk[8, 10]. The focus of these systems is 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. While CodA is perhaps 'lower-level' than other systems, this approach allows us to gain a certain measure of language independence while retaining the potential of the architecture.

3.2 Operational decomposition into objects

In Chapter 1 we motivated our choice of meta-level decomposition into objects in terms of the goals for our system. This section makes concrete our claims and design. The CodA meta-level

is factored according to the operations required during the execution of an an object. Each of the factored operations (*behaviours*), is reified by an object (*meta-component*). Meta-components provide specific definitions or implementations of the behaviours they describe. Typical behaviours are concepts such as; object execution (both mechanisms and resources), message passing, message to method mapping and object state maintenance.

While logically the behaviours are distinct, in an implementation a single component may describe several behaviours. An object's characteristics are changed by explicitly redefining or changing the 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.

This approach to defining the meta-level has several distinct advantages over that found in other systems. One common approach is to create public interface methods on a small number of meta-level objects [25, 19, 23]). The design of these objects is often based on structural concepts found in the programming language used for the system. For example, the CLOS MOP's metaobjects are things like classes, methods and slots. These concepts relate to how objects are described by users, not how they are run by computers. This relates back to our discussion of bottom-up versus top-down approaches in Chapter 1. While the top-down approach is convenient for reifying existing or predicted behaviour, it does not lay a sound framework for the description of new and arbitrary behaviour.

A structural architecture forces an object's operational behaviour to be reified as meta-level code rather than objects. Changes to a behaviour are made by modifying the related interface methods on perhaps several metaobjects. Unfortunately, code is inherently more difficult to deal with than objects and is not a good unit of software engineering. While code is often represented as *method objects*, these 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 description is not clear. Unanticipated behaviours have no clear home for their description and/or state.

Decomposition of operational behaviour into objects gives us a higher-level view of object execution. Objects abstract code, define points of interaction and ease integration. They remove us from the details of implementations and allow us to concentrate on design. The operational approach results inherently in fine-grained objects. The advantages of this are highlighted in various parts of the system (e.g., object model combination).

Objects also provide a wider interface for the behaviours (i.e., operations) they define. While the execution-related interface of a behaviour may only contain a handful of methods, there may be a large number of support and configuration interfaces. Similarly for state. If behaviours were to share host objects as in other systems, these methods and slots would collide with those of other behaviours and be confused with structural code, hiding the real semantics of the system.

We apply this strategy of operational decomposition into objects to the CodA object system discussed in Section 3.1 by looking at the events which occur during the execution of objects. Each of these events is mapped onto an operation. Each operation is considered to be a behaviour with a meta-component to give its description. Since the decomposition is based on these operations, it is termed an *operational decomposition*.

In certain cases (e.g., State) two very closely related operations (e.g., slot value setting and getting) were merged into one behaviour. Regardless, the result is a fine-grained decomposition of the basic CodA object system which consists of seven components default behaviours; Send, Accept, Queue, Receive, Protocol, Execution and State.

The next section gives an example of how a meta-level is decomposed according to our strategy. Following that, Section 3.4 gives a detailed description of each of the seven default behaviours and its prescribed execution interface. Further sections describe how the components which define these behaviours are managed. A final section sets out a simple but instructive example of modifications to several behaviours to effect a coherent object model.

3.3 Meta-level decomposition

Above we noted that the default CodA meta-level is decomposed into seven different components, one for each execution-time object operation. The number 'seven', or more precisely, the actual seven components were determined by analyzing the execution events which occur during normal object execution in a number of different object systems.

It was found that all object systems reduced to a fundamental set of *primitive* operations which could not be usefully decomposed. We converted each operation into an explicit object at the meta-level (e.g., a meta-component). In some objects systems, some operations are trivial or not even explicitly represented. Regardless, the CodA architecture gives these operations an explicit description. For particular object systems and operations (e.g., queuing for non-concurrent object systems) the operation is optimized away in the implementation but a representation of this behaviour exists in the meta-level.

We do not claim that our decomposition is the only one possible but do claim that it is a powerful and effective decomposition. Further, the architecture is such that if users wish to change the granularity of the decomposition, they are free to do so. A later section (Section 3.7) details how this is done. The architecture is also extensible in that completely new behaviours can be added without affecting those which already exist.

The default decomposition forms a very basic model of object execution. Figure 3.1 depicts the events, meta-components and interactions required to effect all aspects of this model. In the figure, some object A is sending a message M 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. Only those meta-components relevant to this particular interaction are labeled.

The figure shows that A sends M by interacting with it's Send, invoking its send: interface (1). Note that for brevity the for: portion of the component interfaces has been omitted from the diagram. The Send communicates with B's Accept (2) and transfers M. The Accept queue:'s the message with the Queue (3).

Since the message is queued with B, the calling thread (i.e., A's) can return along the call chain over which it came. If the message is synchronous then A's Send blocks the thread pending a reply from B. If it is asynchronous, the thread returns to A's base-level code and continues running.

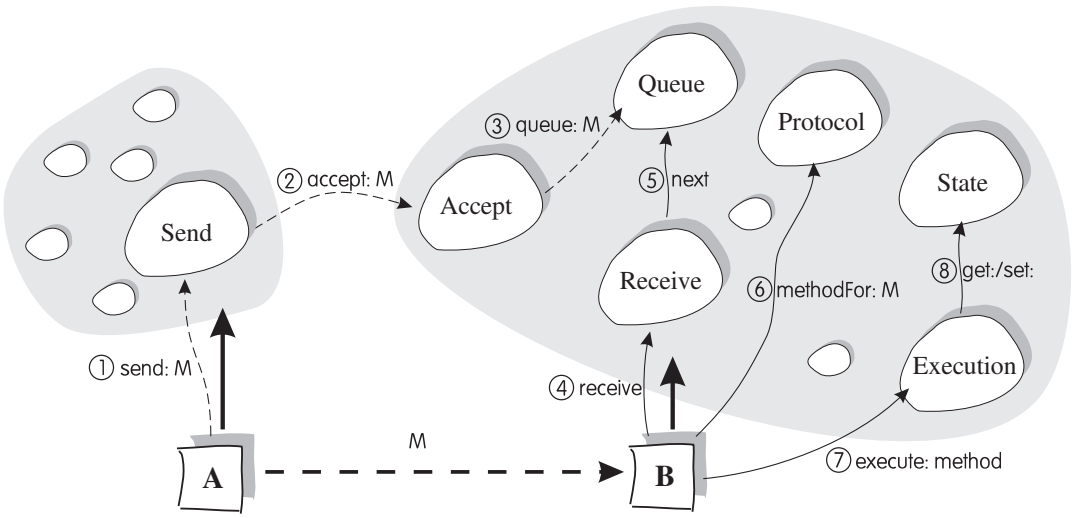


Figure 3.1: Sample meta-level configuration and interaction

At some point, perhaps in the past, *B* will execute a receive: operation which invokes the Receive (4) and attempts to fetch the next message from the Queue (5). Assuming that a blocking receive was used, *B*'s thread will block in the Queue until a message is available. When one is, it is removed from the Queue and returned via the Receive. The normal behaviour at this point is to find a methodFor: the message using the Protocol (6). Once a method is found it is execute:d by the Execution (7). During the execution of the method, the receiver's (*B*'s) state slots may be accessed via its State component (8).

3.4 Behaviours

Below, each of the default CodA behaviours is described and its operational (i.e., execution-time) interface specified. Interfaces required for configuration and infrastructure are not included as they will vary depending on the capabilities or properties of components and the actual implementation environment. All specifications are presented in terms of the Smalltalk implementation detailed in Chapter 6. The code for the default meta-components which implement these behaviours is given in Appendix A.

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 the meta-component implementing a behaviour maintains a one-to-one relationship to the base-level (e.g., in a state variable) then the base argument is ignored.

3.4.1 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 in Chapter 4 contains a Send behaviour which diverges substantially from the default model.

By default 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.

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

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.

3.4.2 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 choosing a message for processing.

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

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.

3.4.3 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). 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.

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 and responds to the following messages:

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.

3.4.4 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. It responds to the following messages:

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.

3.4.5 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. It responds to the following messages:

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

3.4.6 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.

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 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 and the logic for using it.

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 job to determine how to execute methods and then execute them. Concepts such as group-wide computation[25] are implemented by creating objects which share Execution components.

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). It responds to the following messages:

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 execution currently underway is interrupted with the processing of message.

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

3.4.7 State

States describe the physical storage and structure of objects. They implicitly define what slots an object has as well as explicitly define how the data in those slots is stored. The States *do not* actually hold the data, they simply know how it is accessed.

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

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.

3.5 Meta-level framework

In CodA, meta-levels are defined by compositions of the decomposed operations within a framework. We are guided here by a general approach to object-oriented software engineering (paraphrased here):

> *Decompose a problem into a number of smaller, relevant sub-problems. Supply a framework for sub-problem manipulation and then compose solutions to the sub-problems to solve the whole problem.*

This separates structure and semantics. The operational decomposition into objects discussed above gives us a set of guiding principles for factoring behaviour descriptions. The specific decomposition we propose in Section 3.2 fully defines the behaviour of individual objects in a generic object system. Since our decomposition strategy removes the implicit structure from the meta-level, we add it back in with an explicit framework for composing and configuring the meta-components.

A CodA object does not have a single entity which is "the meta-level" but rather defines the meta as a named conceptual collection of all the meta-components which describe a particular object. metas may take many forms but are always only *brokers* for the services provided by their meta-components. They are the foundation of the framework for managing the meta-level. They provide the mechanisms for manipulating the meta-components and forming coherent structures.

metas do not themselves define object behaviours. They may store internally the meta-components they broker, they may simply fetch them from somewhere when required or may even create new ones for each request. This is transparent to the user/programmer.

The framework we propose is completely extensible and free of semantics. It places no particular meaning on any particular behaviour or component. It is designed to support basic properties of one object's meta-level:

1. Behaviours are named.
2. There may be arbitrarily many behaviours and components.
3. Components may be represented differently.
4. One component may define several behaviours.
5. Components may be replaced by other compatible components.
6. Components (and thus behaviours) may be shared between meta-levels.

Meta-levels are formed by composing the components which define all of the behaviours of an object. There must be a definition for each. Behaviours which are not explicitly specified assume some default definition as supplied by the implementation environment.

The CodA meta-component framework is completely extensible allowing new behaviours and components to be added by users. Adding a new behaviour is a matter of defining its name within the framework and creating components to describe the desired operations. In general we develop at least two components for a behaviour; one which defines some default operation and one which defines the new operation we specifically want to add to objects. The default component is used when an object's behaviour is accessed but no component has been explicitly provided. It provides interface compatibility but typically defines all operations as no-ops. By doing this, all objects can be made to appear to have a definition for the new behaviour. Developers then create variations on this default behaviour and substitute them individual or groups of objects.

The physical representation of the meta is left to particular implementations of CodA (see Chapter 6 for details of the Smalltalk implementation). Similarly, the syntax and mechanisms for invoking and accessing the meta-level are implementation details. For the purposes of discussion however, we introduce here the Smalltalk syntax and use it throughout this document.

Most meta-level architectures are coupled with a particular language and so the language itself has been adapted/designed to facilitate meta-level access. For example, meta-level programmers need a way of indicating shifts to the meta-level or accesses to meta-level objects. ABCL/R2 uses the ↑ (up arrow) [25] to signal a shift to the meta-level.

In CodA we adopt the technique best suited to the base-level language. For the Smalltalk implementation we chose to limit changes to the language itself and so implemented level shifting via message passing which is already integrated in the language. In particular, sending the meta message to some object (e.g., anObject meta) returns an object which *represents* meta-level computation. Any messages sent to this object are executed in the context of the meta-level. As such, the result of anObject meta is the CodA meta.

3.5.1 Object models

The basic framework functions well as long as the number of behaviours it manages is small and the behaviours themselves are small and independent. As the system scales up and behaviours become inter-related, it becomes more and more difficult to manage — There are no higher level structuring mechanisms. To address this we introduce the idea of *object models*, the main mechanism for structural abstraction and organization.

Object models represent higher-level concepts and allow the grouping of interdependent behaviours. Models are used to describe things such as classes, concurrency and distribution. Their main role is to specify configurations of meta-components which form some logical or coherent behaviour and insure consistency across the operations of a particular meta-level. As such, an object model is really a set of *constraints* on the a group of behaviours relevant to its domain. Some models have descriptions of one or two behaviours while others specify many. For example, the ConcurrentObject model specifies constraints on two behaviours, Execution and Queue. All others are immaterial.

The constraints on a particular behaviour may take many forms from very specific to very broad. A model does not necessarily contain the components of which it is composed but may simply describe them. For example, a model may specify that a particular type (class) of object be used for some behaviour. Or that an object with particular *properties* be used.

Consider standard classes for example. Classes are really just specific meta-level configurations which are shared amongst *instances*. For an object to be an *instance* of some class, it must share its specifications of protocols (i.e., methods) and state (i.e., slots) with the other instances. Figure 3.2 illustrates this for the Point class and some instances.

CodA has no explicit class structures but the object model construct can be used to give the same effect. We define a class object model to be a model which constrains the Protocol and State behaviours to be defined by specific meta-component objects. This model is then forced on all base-level objects wishing to be considered instances of the 'class'. Using this approach, we define a different model for each class since the Protocol and State behaviours differ from class to class.

Figure 3.3 gives an example of a Point object model. It shows two objects, *A* and *B* which are both *instances* of Point. The Point model specifies particular Protocol and State components which define those behaviours for Points. *A* and *B*'s meta-levels share these meta-components but are free to redefine all others.

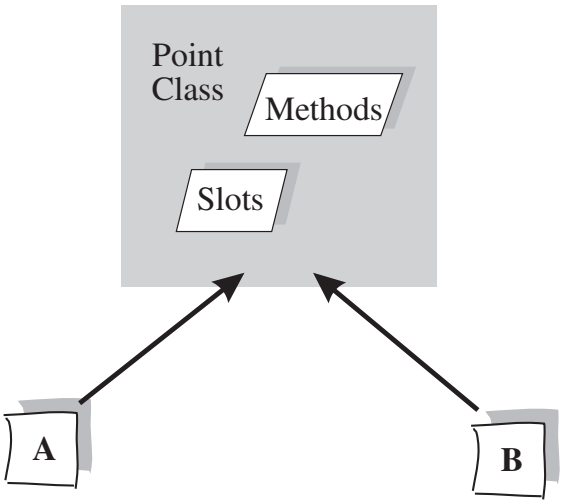


Figure 3.2: A standard class configuration

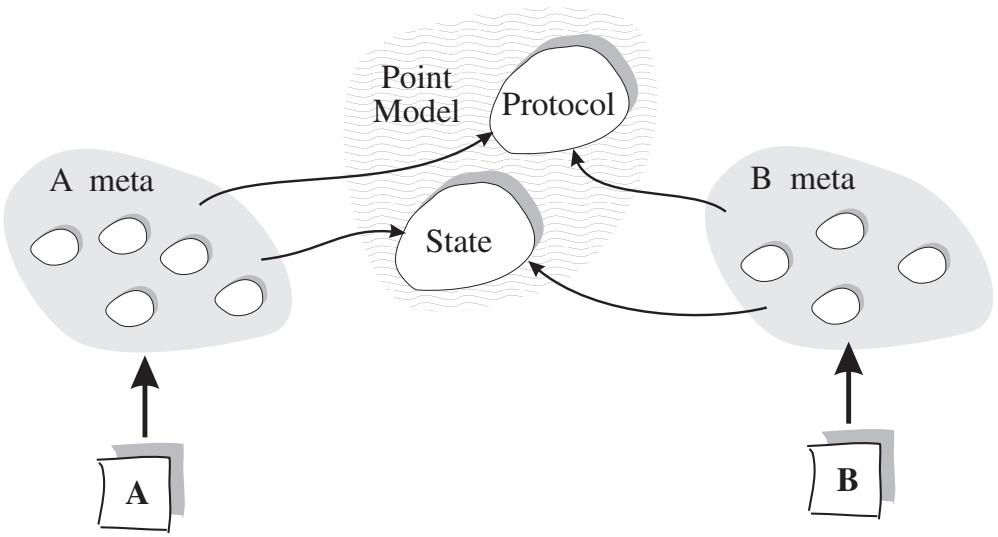


Figure 3.3: Class object models

To get the effects of inheritance, we create States and Protocols which take undefined elements (i.e., slots and methods) from the components of the superclass object model. The inheritance mechanism's implementation can include single and/or multiple inheritance, delegation or any other approach as appropriate. Notice also that State and Protocol inheritance are disjoint and can follow different rules.

Class models specify very particular constraints on the components which can be used to define behaviours. Other object models, such as the ConcurrentObject and ConcurrentObject models described later, provide only rough descriptions of the *properties* that suitable meta-components must have. A given model can combine many different kinds of constraints.

Overall, object models are somewhat like Apertos metaspaces [43, 44]. Metaspaces are generally larger and more concrete and are defined in a hierarchical fashion. Since object models can vary in size and sophistication from a simple constraint on one behaviour to a complex set of constraints on many behaviours, CodA does not dictate how object models are defined or managed. The best mechanism is likely one which is natural to CodA's implementation environment. We do however, supply some infrastructure for their combination.

3.5.2 Object model combination

CodA object models are smaller and more dynamic than Apertos metaspaces. Meta-levels are created by the dynamic *combination* of object models rather than static declaration. All objects start out behaving according to the default object model. To this we add other models such as ClassedObjects, ConcurrentObjects, etc.

When a model is added to a meta-level, the components it contains or describes are combined with those which already exist in the meta-level. Unfortunately, meta-components are general objects and can be arbitrarily complex. The automatic combination of such entities is an open problem which exists in all software systems. While CodA is no exception, its architecture contains certain features which inherently ease the resolution of conflicts.

Combining disjoint (i.e., non-overlapping) object models is straightforward. The new model or meta-level simply contains the union of the constraints from the original models. As long as all the constraints and components specified follow the standard CodA interfaces, the object will continue to run.

Combining overlapping object models may require programmer intervention since the nature of the collision may be arbitrarily complex. 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 the situation shown in Figure 3.4. In the figure there are two object models X and Y. Each defines constraints for a number of behaviours. As it happens, they both define constraints for the Send behaviour. If we wish to create a model XY (the combination of X and Y), we must resolve any conflicts between constraints XSend and YSend.

Assuming that the conflicts are non-trivial and cannot be satisfied by some existing meta-component, they must be resolved manually by the programmer. This results in the creation of a

Send component (e.g., XYSend) with the properties required by both the *X* and *Y* models. Having resolved the conflict between XSend and YSend (by implementing XYSend) once, it need never be resolved again. Other occurrences of the conflict, perhaps during the combination of some other models, are resolved simply by reusing the XYSend component. As we build a library of components, conflict resolution becomes more a problem of identifying the existing component which satisfies the constraints than of actually writing code.

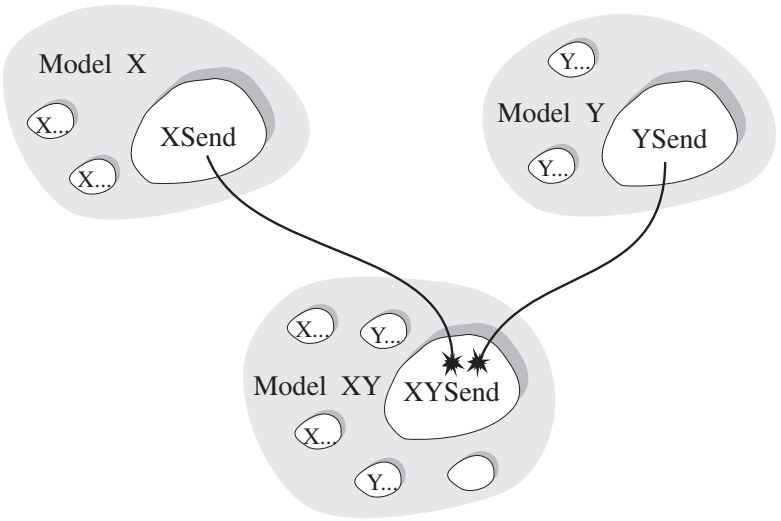


Figure 3.4: Object model collision

Unfortunately, automatically determining whether or not two components are semantically comparable is not easy. CodA addresses this problem using a *property*-based specification and combination mechanism. A property is a simple declarative token which points out one way in which a component is different from the default. Properties need not be statically declared but can be dynamically computed. The property set of a component can be dynamically modified and can contain any number of properties.

Comparing property lists provides an even more precise identification of component conflicts. They highlight not only the components which collide but the way in which they collide. Rather than hard-coding the use of particular components in an object model, programmers declaratively specify the properties that the components must have. For example, the ConcurrentObject model specifies that it requires an *active* Execution component. Any active Execution will do. Whether a change is required and which actual Execution is used is determined when the model is merged with others.

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 complementary approach is component generalization and parameterization. In creating the various object models discussed here, we applied the above techniques with success. As we

developed more and more overlapping models, 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. Object models then contain property constraints and parameter specifications.

3.6 The ConcurrentObject model

As an introductory example of how the meta-level is manipulated, we present the ConcurrentObject model. 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, a kind of Execution.

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, 7, 28]. 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 that 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 may contain no ConcurrentObjects but still require asynchronous sends.

3.7 Abstraction, compression and expansion

While we claim that the default meta-level decomposition discussed above covers every aspect of basic object execution with fine-grained meta-components, we also recognize that there may be situations where our decomposition is either too coarse or too fine. Addressing this requires no new mechanisms, just the realization that the meta-level is extensible.

If a particular behaviour is found to be too coarse in some situation, it can be *expanded* or broken into pieces, each of which becomes a new meta-level behaviour. These new behaviours are individually and independently addressable and manipulable. The original behaviour remains unchanged but its defining component is changed to delegate the various component interface methods to the components defining the new behaviours.

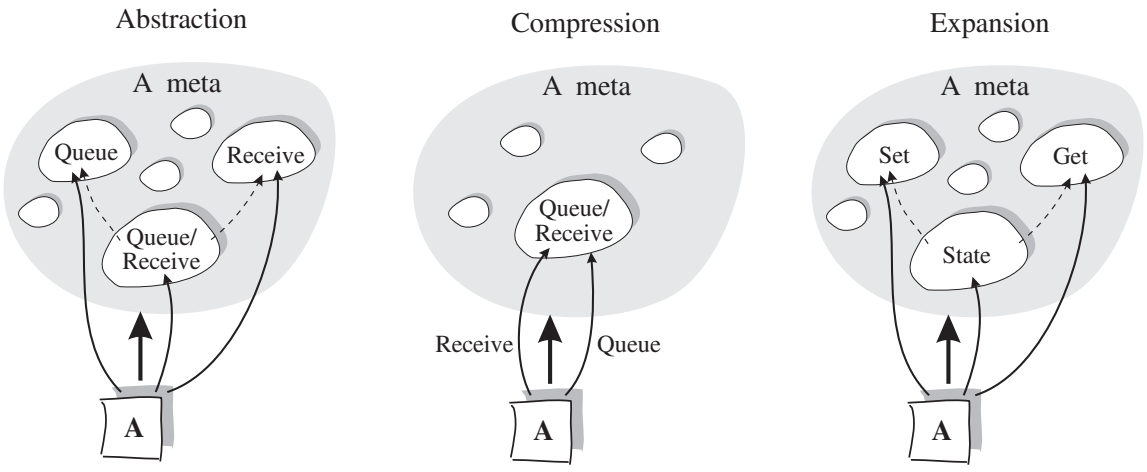


Figure 3.5: Meta-level behaviour compression and expansion

The right side of Figure 3.5 shows the expansion concept applied to the State behaviour. Expansion produced two new behaviours, Get and Set which are added to the meta-level. The implementation of the State component is modified to delegate the appropriate operations to the

new components. Note that the State component still exists and continues to provide valuable behaviour description (i.e., a description of the state slots and their representations).

Similarly, if part of the decomposition is found to be too fine, it can be *abstracted* by creating a new behaviour which is a logic combination of several existing behaviours. The left side of Figure 3.5 depicts the abstraction of the Queue and Receive behaviours into one. The abstraction results in a new behaviour, QueueReceive, being added to the meta-level. The component defining that behaviour combines and abstracts the original Queue and Receive components and adds some operational 'glue'.

In an effort to gain efficiency (in speed or space) we may also want to have the same physical object define several behaviours. This is shown in the *compression* part of Figure 3.5. The Queue and Receive behaviours are both defined by a single Queue/Receive object. Note that a new behaviour has not been added to the meta-level as with abstraction. Compression is an implementation technique which is transparent to the meta-level user.

3.8 Summary

Typical meta-level architectures are reifications of existing language systems. As such, their decompositions and frameworks contain embedded structure derived from the underlying models. This restricts the range of behaviours which can be described easily. We have designed CodA to be independent of base-level language concepts and so be capable of defining a wide range of object models. This was done by basing our decomposition on the operational aspects of a very basic object system containing just message sending, method execution and state accessing. Our basic *operational* decomposition contains seven default behaviours; Send, Accept, Queue, Receive, Protocol, Execution and State. The execution-time interfaces for each is outlined.

We show how the meta-components which describe these behaviours are arranged in a generic framework which provides naming and object attachment services. The framework itself defines no semantics and is completely extensible. Using this general framework, the structuring abstractions (e.g., classes) which were lost during the operational decomposition are reconstructed in a generic way using the concept of *object models*. Object models are groupings of meta-components which go together to describe some logically consistent unit. This is demonstrated by showing how a class is represented by an object model. CodA's inherent facilities for object model combination are also presented.

We have taken these design approaches in an effort to make the system as general as possible but still provide the user with support for abstraction and meta-level concept manipulation. Later chapters demonstrate the usefulness and capability of CodA by presenting non-trivial object models which are from varying domains but which are all described within the same architecture and can all be used together. These models are combined and an applied to significant applications with relative ease.

Chapter 4

Ported Objects

In the previous chapter we presented the CodA architecture and gave some simple examples of changing an object's behaviour (e.g., the ConcurrentObject model). In this chapter we present a farther reaching and more complex model, PortedObjects to show how deep changes in object behaviour are effected. Though this model defines radically different object semantics, it is implemented largely by changing the definitions of already existing behaviours rather than adding new ones.

PortedObjects are objects which communicate and behave in a dataflow-like way. They have *ports* or channels over which data flows. Users program PortedObject systems by building connections between the ports on objects. 'Programs' are run by feeding data into free parameter ports. The values put in a port are automatically broadcast to all the objects to which it is connected. When an 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 and is processed.

The need for the PortedObject model arose out of our work with Vibes [26], an object behavioural analysis toolkit. The application itself is described in detail in Section 7.3 but a brief overview is given here. Vibes is a data analysis tool much like AVS [41], IRIS/Explorer [39] and parallel system analysis tools such as Pablo [36]. The common denominator for these systems is their dataflow architecture. Computation, and thus analysis, is done by chaining together *nodes* which accept data, transform it in some way and output it to following nodes. Chains are allowed to split and merge to form a graph structure.

In most existing systems the computation done at each node is quite simple and largely mathematical. This is suitable for scientific and statistical analysis techniques but does not support more symbolic analysis techniques. Further, the mode of programming new node computations is rather rigid. Programmers must create procedures and data types which conform to the architecture's model. These analysis modules typically cannot be reused in other contexts (e.g., other analysis tools).

For Vibes we sought to build a system which is flexible, supports symbolic and numerical computation and facilitates standard software engineering practices such as reuse and factoring. The key to achieving these goals is a clear separation of the analysis operations and the dataflow architecture (i.e., base- from meta-level). The CodA architecture and the PortedObject model

enables this separation and allows programmers to use normal imperative objects in a dataflow computational environment without significant code modifications.

As a result, analysts simply take available objects which describe the desired analysis node operations (e.g., filters, collectors, expert systems, DSP processors), modify their meta-level to have ported behaviour and chain them together. This is possible with everyday, generic (i.e., not dataflow-specific) objects. If an object describing the desired computation does not exist, the programmer must create one. These new objects need not be written in a dataflow style since they can be adapted after-the-fact (i.e., at the meta-level). As such, they can be reused in many different applications and computational domains. Both modes of reuse are major advantages for people building complex analysis systems.

In addition, it is interesting to note that the PortedObject model is attractive to researchers working on formal expressions of communication and concurrency. For example, concurrency formalisms like the π-calculus [29] make use of channels and ports to specify object communication. We have not pursued this line of research but its obvious connection to the PortedObject model further justifies the model as a useful and important meta-level modification.

For our purposes, the PortedObject computational model must incorporate the following properties:

Pluggability and Extensibility Ordinary objects must be capable of being 'plugged' into arbitrary object graphs but be isolated at the base-level from the graph's details (i.e., objects are not directly accessible). If an object is not isolated from the details of its containing graph, the system would not be extensible. Changes in the graph would require changes in the object which may not be possible.

Nesting To handle large and complex subsystems, we must be able to partition a graph into coherent, non-intersecting subgraphs. Subgraph containment must be transparent to the containers and the containees.

Pluggability allows the construction of arbitrary computation graphs by connecting together sets of generic objects rather than imperative programming. Nesting supports the abstraction and grouping functional units both for design and reuse. Combined, these capabilities create the basis for a powerful and general analysis architecture.

4.1 Meta-level design

Since the PortedObject model is a dataflow paradigm in which an object's logical data (e.g., object state) takes on a new role; that of a communication mechanism. From the base-level, explicit message sends and receives between PortedObjects are not possible. Objects see the outside world only through their input ports and affect it only through their output ports. Ports are treated as state producers and consumers. Reads and writes from/to (logical) state slots are communications operations.

At the meta-level however, this is just a restriction on the nature of normal object message passing. Outgoing messages, rather than being triggered by some explicit base-level code, are

initiated by state slot modifications. If a slot is associated with some port(s) then the new value is transmitted to each object connected to that port. Incoming messages, rather than being mapped directly onto methods, update the availability of new data values (i.e., logical slots) and potentially trigger the execution of the object's code which depends on those values. There is a significant decoupling and abstraction of object intercommunication.

We talk about ports as being implicit or explicit. Explicit ports are ones which exist in the object's natural description. They typically occur as instance variable slots into which values arriving over ports can be stored. Often they set auxiliary parameters or configurations. Implicit ports on the other hand typically exist as method arguments. In the imperative paradigm, values are passed into these ports during a function call or message send. In the dataflow paradigm, parameter arrival and object execution are decoupled. The PortedObject model, implicit slots are added automatically. There are no other tangible differences between implicit and explicit ports.

The PortedObject model is implemented in changes to the definitions (i.e., meta-components) of a number of behaviours at the meta-level. The changes fall into two main categories: value transmission and coordination.

Value transmission takes the form of multicast messages between meta-level components. Objects with new values on an output port multicast the value to the meta-levels of all the objects connected to that port. Objects with multiple input ports have some mechanism for maintaining the separation of messages arriving over different ports, either by tagging or multiple queues. Definitions for these operations are found in the Send, Accept and Queue behaviours.

It is important to remember that message passing at the meta-level is independent of execution of the base-level. A message passing from one object's Send to another's Accept does not imply that the receiving base-level object will execute immediately or at all. The receiver's meta-level is free to delay, ignore or modify the incoming message. In the PortedObject case, messages at the meta-level are used as data carriers. Moving data from port to port.

Exactly how the multicast-related behaviours are defined depends on the model desired. For example, connection management is only needed if communication security is required. By maintaining lists of connections between specific ports on specific objects, we ensure that 'sent' values are only sent to connected objects and that 'received' values come only from connected objects. This also enables a certain degree of implementation hiding and abstraction. Ports can be typed allowing for safer interconnections. Connection objects can be used to map output ports to input ports rather than relying on implicit information about the source or destination of a value. These features decrease the effort required for maintenance and increase the pluggability of objects.

Coordination modifications are needed to effect a change in the way object execution is triggered. Normally, an object's base-level is triggered with the arrival of each new message. PortedObjects however, cannot receive messages, just data values in ports. As mentioned above, data values are transmitted by messages at the *meta-level*. These messages in turn affect the status for their receiver's base-level object's ports.

Some base-level objects should execute every time they a new data value is available on any of their ports. Others must only execute when values are available on some number or all of the ports. There are many possible of firing rules. Which is correct depends on the object, its status and the ports involved. We say that an object is *coordinated* when at least one of its firing rules has been satisfied. An object's Receive defines its coordination characteristics. In addition, its Execution is

slightly modified to take into account the requirement for coordination before execution.

Figure 4.1 shows the modified meta-level of a PortedObject. The changes to the default components for Send, Accept, Queue, Receive and Execution are detailed in the following sections.

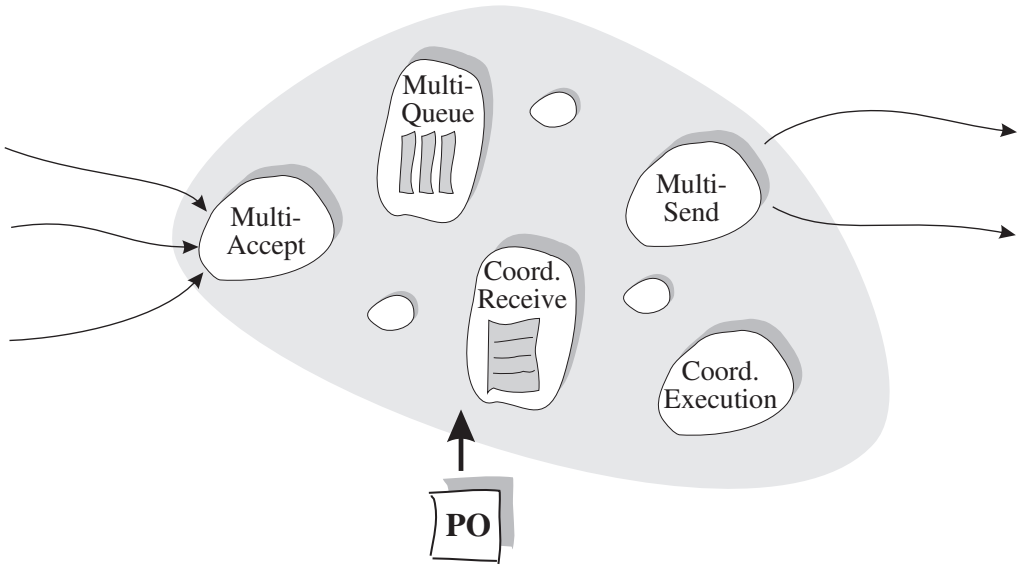


Figure 4.1: The PortedObject meta-level

4.1.1 Send

At the inter-object level, PortedObjects cannot explicitly send messages. They can only store values in their logical output (result) ports. A base-level 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 multicasting 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 component which provides infrastructure for multi-casting messages to a known set of receivers. For PortedObjects these receivers are represented by ports and connections. MultiSends have additional interfaces for managing who receives what messages.

4.1.2 Accept

PortedObjects use MultiAccept components to describe their Accept behaviour. Their definition differs from that of DefaultAccepts only in their support for maintaining and managing sets of associated objects (e.g., ports and connections). Like MultiSends, this consists mostly of add/remove and connect/disconnect methods in various forms. Operationally, as messages arrive (via accept:for:), MultiAccepts mark them with the port over which they came and queues them as per normal operation.

4.1.3 Queue

The PortedObject model uses MultiQueue components to define for their Queue behaviour. 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.

4.1.4 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 CoordinatedReceive components.

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
```

This example uses a very specific and simple constraint system. CoordinatedReceives in general can use any means of determining coordination. A generalized version takes a user-supplied coordination source. This object produces cSets for use in the dequeuing operation. After a message is fetched, the coordination source is told which message was retrieved so it can adjust its state. Using this technique, CoordinatedReceives are usable in many different situations and can manage arbitrarily complex constraint systems.

4.1.5 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 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.

The main change in a PortedObject's Execution is highlighted by the modified process:for: method shown below. After executing an infrastructure message (2+3), the Execution tests for

coordination (4). If the object is coordinated, it is evaluated (5). After evaluation, the coordination set is reset (6).

```
CoordinatedExecution»process: message for: base
1)  | method |
2)  method := base meta protocol methodFor: message for: base.
3)  self execute: method with: message for: base.
4)  base meta receive isCoordinated ifTrue: [
5)      self evaluate: base.
6)      base meta receive resetCoordinationSet]
```

The definition of evaluation varies from object to object. It is abstracted out and supplied by the user or the base-level object's definition. The evaluation of a PortedObject can perform arbitrary manipulations (e.g,. mapping) of the port values before invoking the actual operation represented by the object.

4.1.6 Example

Using the above definitions of object behaviour, the following steps occur during the interaction between two objects, A and B which are connected. Readers should compare this with the basic execution model described in Section 3.3.

A detects a change in the output port R so it takes the new value and wraps it in a meta-level message newValue: (the value is the argument). The message is then passed, at the meta-level, to the Accepts of all the objects connected to A's R port (including B). When the message arrives at B's Accept, its arrival port is checked and the message is tagged and queued. The MultiQueue maintains one (logical) queue for each input port of B. B is continually waiting for new messages and so at some point, fetches the one containing the value from A's R. Executing this message sets the value of the associated input port to the value contained in the message and updates the coordinated status of B. If B is now coordinated then its evaluation code can be run.

Overall the execution is very similar to the normal objects with the major exception being the disjoint nature of message receipt and object evaluation.

4.2 Applying PortedObjects

Clearly not all objects are suited to this sort of model. Typically objects with large and complex interfaces cannot be adapted or can only be adapted in a limited way. Often however, the kinds of objects you would like to use in a dataflow situation do lend themselves to the PortedObject model and can be used quite effectively.

The addition of porting to an object is done by first identifying its parameters and results as logical entities. Each parameter or result is made into a port on the surface of the object. The object being ported must also be analyzed to determine its execution requirements (i.e., coordination constraints). The analysis should determine the main functions of the object and their operating

conditions. Then an abstract firing specification is created which matches the coordinated state of the object onto an actual execution sequence.

For example, assume some object O is a signal processing element which can calculate correlations between two input data sets, A and B. If O is a normal object, it has some interface method (e.g., correlate:and:) which takes A and B as arguments and returns the result of their correlation. To make this into a PortedObject and hide O's interface, we first identify the two input ports, A and B and one output (result) port, say R. We then define the firing specification such that when there are new data values on both A and B, the operation correlate:and: is executed with the current port values as arguments. The return value is automatically mapped into the port R and passed on to any connected objects.

These changes can all be done in terms of annotations (see Section 6.5.1 which define the various porting properties. Figure 4.2 shows the annotations needed to add porting for correlation on the DSP object mentioned above. Most of these definitions can be generated from a little bit of user input. The evaluator function is an arbitrary piece of code supplied by the user. The ports are made available as methods on the object itself. Reading from a port is a typical *get* operation while writing to a port is a typical *set* operation. Naming collisions with pre-existing methods are avoided by automatic detection and renaming of the port accessing methods.

```
POdefaultParameters
   "The names of the default parameter ports"
   ^#(data1 data2)

POdefaultResults
   "The names of the default result ports"
   ^#(result)

POdefaultCoordinatedParameters
   "The set of input ports to coordinate before executing the object"
   ^#(data1 data2)

POdefaultEvaluator
   "The default mechanism for evaluating the object"
   ^[:anObject | | result |
      result := anObject correlate: anObject data1 and: anObject data2.
      anObject result: result]
```

Figure 4.2: Correlation annotations

With these declarations placed in the class of the object to be ported, we can port an instance of the class using the following expression.

```
anObject meta installModel: PortedObject for: anObject
```

The installation procedure automatically constructs the necessary meta-components, ports, virtual slots etc. to make anObject ported. Once this has been done, the object is free to be connected to others and run. Note that due to the coordination constraint requiring inputs on both input ports before the object will evaluate, both ports must be connected to some data source.

There are a variety of techniques for actually connecting ports together. Below we show one which uses the model definition itself to manipulate the meta-levels and make the connection. In the example, port result of the object called source is connected to the port called input of dest. Again, this automatically constructs all the necessary infrastructure and installs the connection.

```
PortedObject connect: #result of: source to: #input of: dest
```

Further discussion and examples of the PortedObject model are given in Section 7.3 where we present the Vibes data analysis tool developed for CodA.

4.3 Compound ported objects

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 4.3.

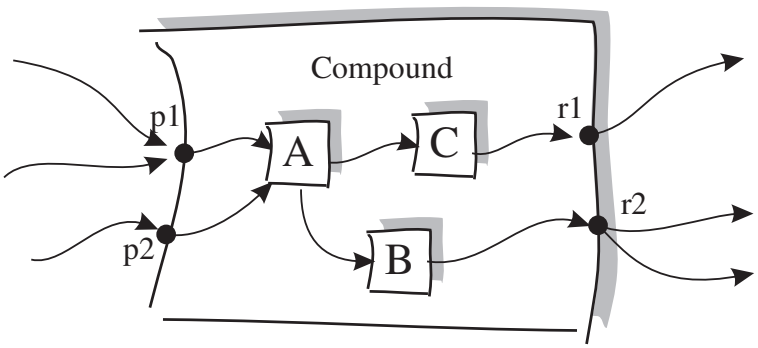


Figure 4.3: Compound object example

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. The design of Compound is an interesting problem.

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 and manage messages accordingly. 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 — The group's incoming and outgoing connections are connected to different objects.

We take a novel approach and extend Compound's meta-level to have two new behaviours, InternalSend and InternalAccept. These behaviours are actually defined 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 4.4 shows the configuration for the parameter side of Compound from Figure 4.3. Note that p1 and p2 in the two figures are the same.

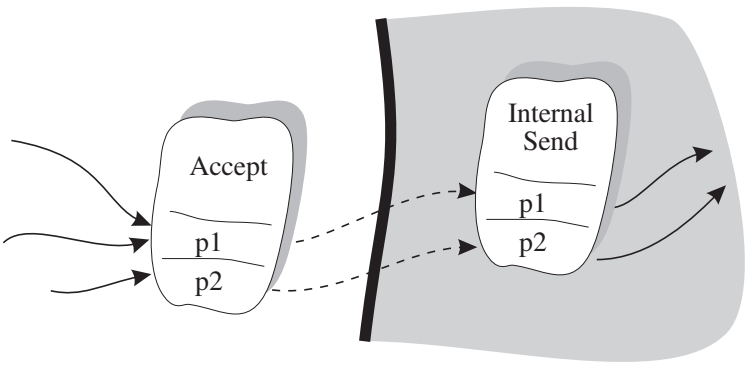


Figure 4.4: Compound object parameter handling

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.

The contained object's meta-levels have no idea that they are inside a compound PortedObject. The objects connected to Compound have no idea that it is a compound PortedObject. The encapsulation is completely transparent but for the addition of one message pass' overhead. The result is a very powerful abstraction mechanism which allows compound PortedObjects to be arbitrarily connected and nested.

4.4 Summary

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, Compound PortedObjects demonstrate 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 esthetically pleasing.

Furthermore, the meta-components developed for this model are quite generic. It is easy to see other situations, quite independent of the PortedObject model, in which we would like to have multicast behaviour or coordinated behaviour. In those cases, these components can be directly reused.

Chapter 5

Tj

Our previous example object models have largely involved only the modification of existing meta-level behaviours. In this chapter we explore a group of object models which define behaviours completely new to normal objects. In particular, we present *Tj*, a distributed object system. *Tj* contains models for generic distributed objects as well as replicated and migrated objects.

Our objective in implementing *Tj* was to create an environment for designing and experimenting with distributed applications and various forms of distributed object computing. The creation of distributed applications is often hindered by a lack of *a priori* knowledge of how objects will react to distribution. The use of strongly typed languages can help static analysis techniques determine call graphs and interaction patterns but it is difficult to account for the dynamic nature of distributed systems (e.g., the same application may run differently depending on the machine topology). This is also true of the distribution mechanisms themselves.

Notions such as distribution are often embedded in a language system or base-level code. This prevents users from easily experimenting with new forms and policies for distribution. They must use that language or distribution system. These environments are often restrictive and require changes to base-level semantics to incorporate meta-level (i.e., distribution) concepts.

In our approach, the separation of the base-level application code from the meta-level object behaviour (e.g., distribution) code plays a key role. This separation enables distributed application developers to prototype their applications and experiment with distribution models while minimizing the effects on the application's code. We allow normal objects to be reused in a distributed environment by transparently adding distribution behaviour to their meta-level. In addition, new distribution mechanisms are more easily integrated with existing object behaviours.

There has been relatively little work done using explicit meta-level architectures for implementing distributed object systems. Apertos [43, 44], AL/1-D [33] and GARF [15] being notable exceptions. These systems vary in their domain and approach.

GARF's domain is communications and transactions. The system provides various forms of replication and persistence for objects. It is based on a limited, two level meta-level architecture which is somewhat specific to their domain. Apertos and AL/1-D emphasize the operating system aspects of distributed systems. They focus on the reification of system-level notions such as, shared resources, memory paging and object naming, and do not provide object related operations such as replication. While these issues are important, CodA and *Tj* concentrate on the object aspects of

distributed object systems.

We start with a general meta-level architecture and add comprehensive distribution mechanisms. It is important to have a sound footing for describing object behaviour because the issues related to distribution are 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. The use of a general framework oriented towards objects enables behaviour reuse, combination and extension far beyond that available to special purpose (e.g., 'distributed') systems.

Tj provides both distributed computation mechanisms such as remote references, replication, migration, etc. and a general architecture for defining and describing policies for their use. Particular emphasis is placed on argument and return value passing techniques (i.e., *marshaling*). While most systems provide a means of specifying marshaling on a per-object or per-object group (e.g., class) basis, this is not enough. Objects are often used simultaneously in many different contexts. We must be able to specify marshaling on a per-use basis. *Tj* provides an open, extensible marshaling framework based on declarative *marshaling descriptors* which are specified by users or by the system via automatic analysis.

Based on these ideas, *Tj* adds three new object models;

- DistributedObject
- ReplicatedObject
- MigrantObject

The DistributedObject model reifies general distributed object behaviours such as marshaling, and augments objects with the infrastructure needed to exist in a distributed system (e.g., global addressing). ReplicatedObject and MigrantObject, as their names imply, provide replication and migration capabilities to objects. These operations are designed in terms of new and existing behaviours.

The following sections detail the underlying infrastructure provided by *Tj* and the new and modified meta-level behaviours required to add distribution, replication and migration to standard objects.

5.1 Distributed system infrastructure

A *Tj* distributed system is a collection of DistributedObjects living in *object spaces* and interconnected by *remote references*. These spaces are mapped onto *machines* in some *system topology* interconnected by *transport mechanisms*. For our discussion here, the important concepts are object spaces and remote references. The others are relatively low-level system details the nature of which will vary from implementation to implementation.

5.1.1 Object spaces

Object spaces are a grouping mechanism for objects. Typically there is one space for each processor in the system. Each of these spaces maintains a list of the objects and references it imports and

exports. Spaces are also used as part of the globally unique identifier assigned to all objects. An object's global id is based on the space in which it lives and some id within that space.

References to remote objects are always represented by the object's global identifier. On import, the global id is matched to any local representative (e.g., a RemoteReference or replica) which may exist in the receiving space. If none is found, one is created and installed automatically. By manipulating an object space's import and export tables and these global ids, we maintain system-wide object identity.

Object spaces also support a simple distributed garbage collection mechanism which releases export registration for local objects which are no longer referenced remotely. This allows the actual object to be reclaimed by the local garbage collector if it is not referenced locally.

5.1.2 Remote references

RemoteReferences are the local representation of some remote object. They are similar to Proxies[37]. Locally RemoteReferences are just like any other object. They can be stored in instance slots, assigned to variables, passed as arguments, etc. When sent a message, the simplest RemoteReference just forwards it to the space containing the real object — its *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 5.1. 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 target's Accept. 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!

A RemoteReference's meta-level is also interesting because the way its meta-components are managed. Since RemoteReferences are ubiquitous in the system and they are often short-lived, it is too costly to maintain an separate meta-level for each just to track the references to its components. Instead, whenever a RemoteReference's meta-level or one of its meta-components is needed locally (e.g., during the message sending shown in Figure 5.1), the system automatically creates a local representative (i.e., RemoteReference) of the correct kind.

The RemoteAccept in Figure 5.1 is an example of this. During the execution of A's Send code something like the following will be executed:

M receiver meta accept accept: M for: M receiver

When we fetch the receiver's (i.e., B''s) meta we create a RemoteReference whose handle is B but which knows it is a reference to its handle's meta-level. Similarly, when we then reference the remote meta's components (e.g., B' meta accept), we dynamically create analogous remote component references. This way we completely eliminate remote messaging for meta-level access. In a sense these are symbolic references which are resolved to actual objects only when they are

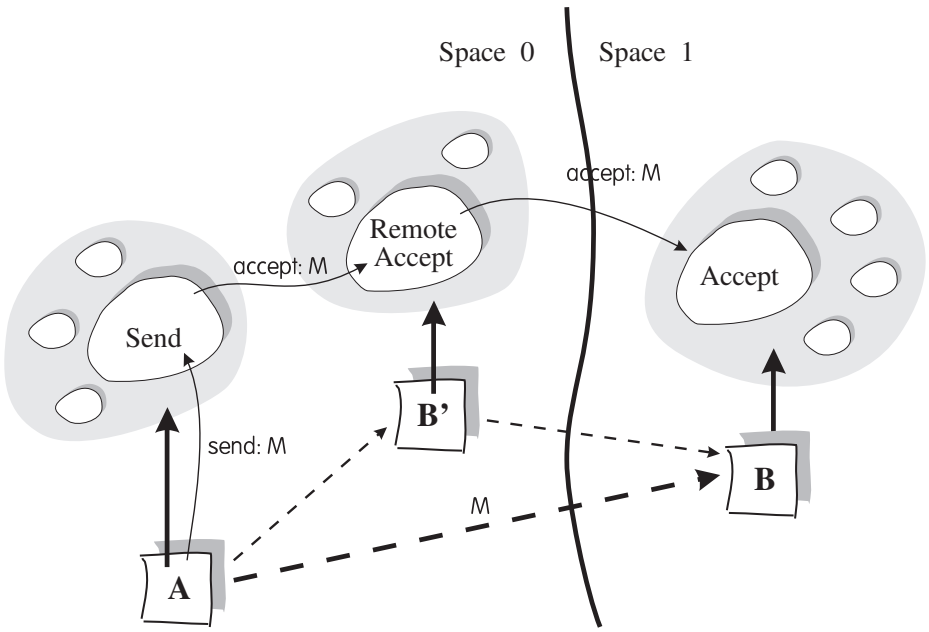


Figure 5.1: RemoteReferences and the meta-level

imported into the space which holds the target.

These sorts of mechanisms are very easily and consistently implemented in *Tj* because we have a uniform basis for their design (i.e., CodA). This is also a demonstration of the integration and flexibility of the environment. Here we use CodA to implement distributed objects and then apply the distribution mechanisms back onto CodA itself.

5.2 The DistributedObject model

The most important property of DistributedObjects is the reification of their state and execution behaviours into distinct objects. This is a direct consequence of the CodA object model's explicit definition of the State and Execution meta-components. In the distributed domain it means that an object can store its state in one space but execute in another. Extending this property, an object's state and execution can migrate independently. Such relationships and properties are exploited throughout *Tj*'s distribution mechanisms.

5.2.1 Distributed object execution

In general, the execution model of *Tj* objects is either *active* or *passive*. An active object has a thread of its own while passive objects do not. Normally, when a passive object receives a message, it *borrows* the thread of the sender method for the duration of the corresponding method execution. However, when a message is received from remote object, there is no local thread to borrow. The system provides a thread which does not belong to a particular object but simply executes a set of message sends from some root method. When that method exits, the thread dies or can be reused.

Active objects, on the other hand, have a thread of their own and execute independently of the other threads in the system. They also may have *active methods*. An active method is a method which is executed exclusively by the object's thread. In contrast, *passive methods* can be executed by any thread. These facilities are used largely in support of controlling concurrency and synchronization in a way similar to Emerald's monitors [22]. CodA also supports more sophisticated intra-object synchronization mechanisms but these are not discussed here.

Distribution is introduced into a system for a number of reasons; because it fits the problem domain, for fault tolerance or for increased performance. We can gain performance through concurrent or parallel execution of distributed objects. To describe parallelism within a distributed object, *Tj* includes notions of distributed *message arrival* and distributed *method execution*.

Distributed message arrival is the idea that when a message arrives at some object, it is simultaneously distributed to all versions (e.g., replicas) of the object. This is different from traditional message multicasting in that it is defined by the receiver rather than sender. Distributed method execution implies that when a method is invoked, it is invoked for all versions (e.g., replicas) of the object.

These notions are different in two ways. First, message arrival does not imply the execution of a particular (or any) method. This mapping is determined by the receiver's meta-level. Second, since messages may be queued before being processed, message arrival and the method execution which may follow are temporally decoupled events.

These differences are important in that they represent different points in an object's execution at which we can control concurrency. By automatically distributing messages on arrival to a group of objects (e.g., replicas), we leave control over their handling to the individual objects. Individuals may choose to ignore or delay some messages. On the other hand, distributing method execution centralizes message handling but injects concurrency into the execution of the corresponding method.

5.2.2 Marshaling

The *marshaling* problem is another important issues in building distributed object systems. To build a distributed system, it is technically sufficient to supply just a pass-by-reference mechanism. Unfortunately, the exclusive use of referencing leads to a dramatic increase in cross-space references and messages. This in turn leads to a decrease in performance of both user code and system code (e.g., distributed GC). Passing objects by value (i.e., by copying) reduces cross-space messages but at the expense of an increase in message size. There is also a loss of generality as copying is typically only applicable to *immutable* objects where it will not affect semantics.

Work with Emerald [22] has explored more sophisticated techniques such as pass-by-move and pass-by-visit and found them to be useful. We have developed a generalized marshaling mechanism based on the notion of *marshaling descriptors*. Used in conjunction with general Marshaling components, these descriptors give *hints* as to how an object should be marshaled. Examples are: reference, cached, shallow/deep copy, replica, etc.

In *Tj*, an object's marshaling policies are defined at the meta-level by the component filling the newly created Marshaling behaviour. These policy descriptions are an interface to the general marshaling mechanisms supplied by *Tj*.

Users trade-off flexibility and efficiency by manipulating the Marshaling component. The fully general mechanism allows the dynamic analysis of object graphs to determine the appropriate marshaling strategy. At the other extreme, we can specify that a primitive, no-frills strategy be used. These choices can be precompiled from abstract declarative specifications into efficient marshaling methods which are used directly.

Below we present the general marshaling framework and the supported marshaling techniques. Following that, we discuss the setting of marshaling policies and approaches to distributed system optimization by modifying marshaling strategies.

Every object has a default descriptor (supplied by its Marshaling) which is used if no other descriptor is supplied. In general, user-supplied descriptors can override a default however it is possible to prevent or constrain this. Descriptors are also *descriptor generators* in that they produce descriptors for the various parts of the object whose marshaling they describe.

At the heart of the generalized marshaling mechanism is a generic object graph walker or *marshaler*. The marshaler walks object graphs according to a series of marshaling descriptors. At each object in the graph it invokes the operations specified by corresponding descriptor. The marshaler maintains the minimum desired, current and maximum desired traversal depths as well other *global* information such as a marshaled object registry used for cutting cycles. These combine to give descriptors a global view of the marshaling process for use in determining how to proceed.

The available set of marshaling descriptors is completely extensible and allows users to add new marshaling techniques in support of new distributed object behaviours. For example, with the addition of the ReplicatedObject model came a **Replica** descriptor which specifies that an object be marshaled as a replica. Below is a list of the marshaling descriptors which are part of the basic system and the object models discussed in this chapter.

Constant Substitute some constant value held by the descriptor for the actual object being marshaled. Marshal the constant according to a descriptor also held internally.

Basic Marshal the object's instance variables according to its contents' default descriptor.

Depth Traverse the object graph from the current object to a minimum and maximum depth as specified by the descriptor. Using this mechanism we can specify an infinite range from shallow to deep copy.

Slot Specify, on a per-slot basis, descriptors to use in marshaling an object's slots.

Reference Marshal a global reference to the object.

Cached Reference Marshal a global reference such that the first time it is accessed, the reference is resolved locally according to a descriptor held by the reference. Note that this resolution descriptor can take any form.

Replica Replicate the object in the receiver's space. The object is replicated according to a further descriptor held internally.

Move Move the object to the receiver's space. The object is moved according to a further descriptor held internally.

Operation Specify a block of code to be used in marshaling the object. This is the escape mechanism which enables arbitrarily complex marshaling.

Use Marshal objects based on manual or automatic code analysis of the *receiving* object's use of the objects. This is suitable as a general default because if no analysis information is available for an object, its self-specified defaults are used.

Attach Specify a set of arbitrary objects to transparently marshal along with the target object. Also allows the specification of marshaling descriptors for those objects. This effects an Emerald-like object attachment facility.

Descriptors can be computed or declared. Computed descriptors are often the result of some analysis process while declared descriptors occur as annotations to messaging operations. A message annotated with a declarative marshaling descriptor is shown below. Note that computed and supplied descriptors are handled in the same way by the system. They are just derived and attached differently.

For example, the sequence:

someObject <- foo: arg1 {deep} bar: arg2 {replica} sends the foo:bar: message to someObject and marshals the first argument using deep copy. The second argument is replicated in the receiving space. In this case it is replicated according to the arg2's default replication descriptor (see below for a discussion of replication descriptors). If we wish to specify how the replication will take place we can specify a further, *nested*, descriptor such as, {replica: (-3 20)}. This specifies that the argument is passed as a replica which is a copy of the object graph starting at argument and going to a minimum of depth 3 and a maximum of depth 20.

This notion of nesting descriptors can be applied in many situations. For example, when using cached reference marshaling we can also specify a descriptor to be used to resolve the remote reference when it is located. We may even choose to resolve the reference with a replica (e.g., {cached: replica}). Variations on this nesting theme can be as complex as required and can involve almost any of the descriptors mentioned above.

5.2.3 Marshaling policies and optimization

Optimization of object marshaling has two facets; the mechanism and the policy. As discussed above, *Tj*'s generalize marshaling framework allows users to substitute arbitrary marshaling mechanisms in a declarative way. These user-defined mechanisms can be as simple or as complex as required. Users looking to reduce marshaling overhead should analyze their application requirements and perhaps replace some objects' fully general Marshaling component with one which is more specific and more efficient. Alternatively, they can continue to use the general mechanism but declare more specific descriptors to match their situation.

A rich, extensible set of marshaling mechanisms is only part of the answer. We must also be intelligent about how we determine which mechanism to use for a particular object in a particular situation. There are a number of ways of determining the best or most effective marshaling strategy.

By default each object follows a strategy determined by some internal property (e.g., class or type). *Tj*, like most systems, says that by default objects are passed as references unless they are

immutable, in which case they are passed as values. Clearly this is not sufficient. In the previous section we detailed how messages can be annotated with marshaling information for its arguments. These annotations override the defaults for an object. In many cases we have found that specifying such annotations does not have the expected effect. For example, caching or replicating an object may result in more remote messages than it eliminates. The problem is that once the object is local to some other processor it will execute its code on that processor. If that code executed accesses instance variables which were not made local then a remote message will be required. If this is done frequently it may outweigh the benefits of having copied the parent object local.

Emerald and other systems address this using the notion of *object attachment*. Under the attachment mechanism, users explicitly attach objects to one another. When an object is copied, all of its attached objects are copied. This solution works to a certain degree but does not take into account use-cases. By modifying the object itself, the attachment mechanism forces the object to have that behaviour for all uses. We have found that in certain situations we or the system know that the attached objects will not be required and so need not be copied/replicated/moved.

Use-case specific marshaling is tremendously important in the implementation of both system-level mechanisms and automatic optimization techniques. When implementing a distributed object operations (e.g., replication) the use of a particular marshaling strategy can be very important. These strategies are independent of the user's view of the object's marshaling strategy (i.e., the object's default). For example, while object attachment is useful for message passing marshaling, it may not be relevant to replication. The replication mechanism must be able to marshal objects independent of their implicit or explicit marshaling specifications.

Tj approaches this issue using marshaling descriptors. By modifying an object's Marshaling, we change its default marshaling descriptor and effect slot-based attachment. As shown above, these strategies (i.e., descriptors) can be specified as message send time. The form of an object under replication, copying, caching, etc. may be different from that of messaging and is specified with separate defaults and explicit arguments passed to those operations.

The declarative nature of these specifications lends itself to integration with automatic analysis and optimization techniques. These techniques depend largely on the analysis of particular base-level sends to determine how parameters and return values should be passed — It is, by definition, use-case specific. For example, if analysis reveals that a particular object is not used, then we simply generate a constant: nil descriptor which marshals it as nil. Since it is not used, its value does not matter.

Static analysis of the entire Smalltalk class library found that approximately 7% of all arguments are not used. While many of these represented error conditions (i.e., they would never occur), the majority were due to polymorphism and specialization. In some cases, methods at the top of an inheritance hierarchy provide for the specification of certain parameters but due to their general nature, do not actually use the parameter. They assume that the parameter will be used by an overriding method. More frequently however, subclass methods ignore arguments required by superclass methods in favour of some derived or stored value. In both cases, this behaviour is in an attempt to maintain protocol compatibility and facilitate polymorphism. As such, we expect the number of unused arguments to increase as class libraries grow.

Analysis may also determine that a particular message's return value is not used in any substantial way. We cannot simply say "don't return a value" because often the return of a value is

used as an indication of operation completion or synchronization. We can however, specify that the value to be returned should be some trivial object like nil. This will maintain the synchronization properties of the message while eliminating most of the overhead. The potential gains may be great. The code of our Smalltalk environment contains some 100,000 message send operations of which the return values of approximately 27% are not used (in static analysis).

Our work has been with untyped languages like Smalltalk and so our analysis capabilities are somewhat limited. While even this simple work can be useful, in general, an increase in analysis detail is accompanied by an increase in precision in the descriptor and thus optimization. Static analysis with strong-typing can determine a method's marshaling requirements quite accurately by looking at its parameter and return value types and references, and examining the call-graph. Systems like Orca [3, 42] and Munin [5] use such techniques. Unfortunately, even this analysis can result in sub-optimal marshaling at run-time.

For example, suppose we developed a metric for determining how many times a method argument is accessed. Using this metric we might suggest that arguments referenced more than X times be passed by value or by migration so they are local. On the other hand, at run-time we may find that one of those arguments is in fact a very large structure and that copying or moving of the object would be costly. Clearly there is a trade-off. Note that strong typing does not address this case as the copied size of an object is determined by its type *and* its attachments which may be dynamically determined.

We propose the use of run-time *negotiation* which would weight the accessing costs against the copying/movement costs and determine the appropriate action. By negotiation we do not mean some heavy-weight, multi-iteration conversation between the sender and receiver over the processor inter-connect channels. We look to the meta-level and see that an object, by giving out a reference to itself, is projecting a part of its Accept into the remote space (e.g., the RemoteAccept in figure 5.1). The sender's Send component can communicate and cooperate locally with the receiver's projected Accept to determine the best communication strategy.

By implementing *Tj* using a rich meta-level environment (i.e., CodA), we expose implementation information and allow the system to make informed choices. This exposure does not violate encapsulation because it is the object's choice to provide information. *Tj* provides the mechanisms and framework for using the available information to calculate declarative marshaling specifications which suit the needs of a particular object interaction. The mechanisms scale to handle a full range of marshaling descriptors from broad hints to precise forced specifications. These facilities are much more general and powerful than those seen in any other system. While it would be possible to add some of them to existing environments without explicit meta-level support, the availability of the CodA meta-level facilities has made their design and implementation very easy.

5.3 The MigrantObject model

As discussed in the overview of *Tj*'s distributed object model, an object's State and Execution are independent. As such, the issues related to the migration of their state and execution are somewhat orthogonal. In fact, since an object's Execution is just another object, the implementation of its migration is largely the same as that of base-level objects. To control or define the migration of an

object, we create a new behaviour in the meta-level, Migration.

5.3.1 Computational migration

We do not consider the possibility of a computation (e.g., an object's Execution) being replicated. Replication (see Section 5.4) generally implies some level of global consistency. *Tj* assumes a MIMD computing model and expects the various copies of an object's Execution to evolve independently making global consistency undesirable. An object's Execution can however, be copied or migrated to remote spaces in a way similar to the computational migration seen in [17]. Note that an object can have Executions in many spaces and still only maintain one version of its state. This is a direct consequence of the separation of the State and Execution behaviours of objects and gives rise to the following six possible relationships between versions of an object's State and its Execution:

1 : 0 Normal passive object. This is the default case. Objects have one state location and borrow threads from their senders.

1 : 1 Normal active object. Objects have one state location and a dedicated thread for their execution. These may not be co-located.

1 : N Parallel object with single state. Useful where the object makes many accesses to remote objects and few to itself. Simply copy the Execution to all the relevant spaces and the methods execute locally. Optional method distribution will result in all copies of the Execution executing the same method though not necessarily in lock-step. State accesses are serialized through the one copy of the state.

M : 0 Replicated passive object. Each replica is passive and borrows threads local to its state. This results in a degree of implicit parallelism. State accesses are controlled according to the replication consistency model.

M : 1 Replicated active object. Use cases for this may seem somewhat contrived but they are nonetheless feasible as potential behaviours. Consider the case where a particular object is large, as is the number of references it does to itself and the objects in any given space. The object's execution is such that it processes data in one space discretely and then moves to the next. In this case, it may be more efficient to replicate the state once in each space. Then we can avoid the iterative state migration and execution synchronization cost by migrating a single Execution among the replica.

M : N Replicated parallel object. This is a mixture of the above models.

Methods for an object executing remotely are such that the receiver (i.e., self) is a RemoteReference to the object which is the master copy of the object's state. All instance variable accesses are converted to remote message sends to the nearest space which maintains state for the object. Depending on how the Execution was created, the local version of the object may also

maintain local versions of the object's other meta-components such that some messages are handled locally.

Tj does not support full thread migration in the sense that arbitrary threads cannot be migrated at arbitrary points in time. There is no design limitation imposing this restriction, it has yet to be implemented. As it is, migration (and in fact copying) can only be done on message processing boundaries. Basically this is to avoid the need to copy and recreate arbitrary portions of the stack. Since we are on a message boundary, we can simply construct the appropriate stack base for the execution in the new space. There are some outstanding technical issues relating to the migration of active objects which will be addressed in future work as the need arises.

5.3.2 State migration

The mechanisms for State migration are quite similar to those for replication. Migration does not place any "after operation demands" on the originating space other than the need for a migrated object location mechanism [14]. Migration is essentially a copy operation followed by a global pointer update.

As with replication, objects are migrated according to the slot specifications given in a supplied migration descriptor. Migration descriptors are derived from replication descriptors. In the implementation they are simply the marshaling descriptor used to marshal the object when it is moved to its new location. As such, objects and their slots can be migrated in almost any form. For example, specifying *migration* for a particular slot effects an Emerald-like attachment. Using a *cached reference* will copy/migrate/replicate an object only if it is accessed in the remote space. Note also that an object can be passed as a migrant. That is, a mechanism similar to Emerald's pass-by-move is implemented by using the migrate marshaling descriptor with a message.

There are also issues related to the threads executing in an object when it is migrated. By in large, these are handled by the local object replacement mechanism. When an object's state is migrated, the local version of the object is replaced by a remote reference to the object in its new location. As such, any references or messages to the object will be forwarded to its new location.

In addition to specifying the form of the migration, the Migration also defines the policies for migration. For example, where and when the object is to be migrated. In many cases, the user/programmer may *hint* that an object should be migrated. Say in a parameter passing marshal descriptor. It is then up to the parameter object's Migration to provide additional information (e.g., cost, size, complexity) in support of the parameter passing negotiation techniques discussed above. This is also true of Replications.

5.4 The ReplicatedObject model

When an object's state is replicated, the *master* (the original) is copied to one or more remote spaces creating a number of *clients*. The master's meta-level is modified such that state changes are trapped by the Replication component. Replication is in fact independent of state form and can accommodate radically different representations in different spaces. Replication components themselves are quite simple.

The master's Replication maintains a list of spaces which contain replica. On state change, it coordinates with those spaces to update the clients according to some consistency model. The clients' meta-levels contain a counterpart Replication component which has agreed to an update protocol and knows the identity of the master object. Depending on the consistency model, state reads are also routed through the Replication.

The replication model for an object is specified via a descriptor. Replication descriptors are derived from the marshaling descriptors discussed in Section 5.2.2. In short, the replication descriptor is the marshaling descriptor used for the argument to the update: message which is invoked when a replicated object is modified. They also contain information related to multicast and master-only messages as discussed below.

The ReplicatedObject model also extends the set of available marshaling descriptors. Readers of Section 5.2.2 will note the presence of a *Replica* marshaling descriptor in the list of available marshaling descriptors. This in combination with the uniform descriptor mechanism admits the following possibilities:

- Parameter passing by replication. This is a novel mechanism in which the argument is copied with an indication that when it is received in a space, it should report back to the master for consistency management.

- Varying consistency models by slot. Allowing different consistency models recognizes the fact that objects may have different use patterns from application to application and that the demands they place on their state variables may not be homogeneous.

- Slot form specification. While the consistency model describes how and when replica are updated, specifying the form is directly analogous specifying the object marshaling descriptor used when updating a remote value. In message passing, a use-case may demand that a particular slot of an object be passed in a certain form (e.g., copied to a particular depth). In replication we may encounter the same use-case and would expect that slot of the object be replicated, in this case, as a copy to a particular depth. Here we allow the full range of marshaling mechanisms for use in specifying remote slot form. Replicated slots can even be updated by further replicating the value in the slot!

5.4.1 Replication example

To demonstrate replication we develop the partial replication scenario shown in Figure 5.2. The figure shows two objects, original (in space 0) and replica (in space 1). Though not shown, original is actually a 2D N-Body [4] 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. Readers are referred to Section 7.1 for a more detailed discussion of this application.

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 as all particles potentially

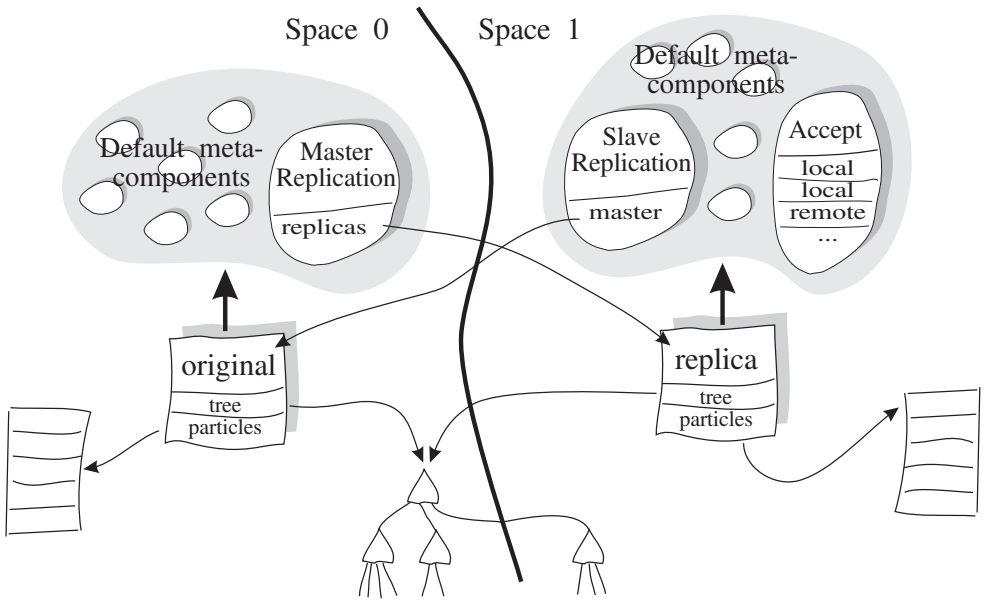


Figure 5.2: Distributed object layout

exert forces on all others. The tree is the central data structure for relating particles to one another and must be globally known and unique. The solver is a prime candidate for partial replication.

As shown in Figure 5.2, 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. All the replicas in the system share the same tree but have independent particle sets. The replication of original is done in a series of six steps. Figure 5.3 shows the required code while the discussion below explains each step.

1. Ensure that the original's Replication 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

```
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) replicaSpaces add: space

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

Figure 5.3: The making of a replica

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 *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.

```
MasterReplication»update: slot with: value for: base
  replicaSpaces 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.

5.5 Implementation

The current version of *Tj* is implemented in Smalltalk and runs on the Fujitsu AP1000 MPP machine [38] with up to 1024 nodes. It also runs on clusters of Sun workstations using MPI messaging. All of the mechanisms and models described here are implemented and running with several more in the design and implementation stage. On all platforms we use a third party messaging package to implement message passing between object spaces. The AP1000 implementation uses the Fujitsu CellOS messaging library and the workstation version uses an implementation of MPI.

While layering gives us portability and flexibility, it also has some affects on performance. In the case of CellOS and MPI, the messaging libraries copy incoming messages before the user can read them. Furthermore, since we allow concurrent receipt and unmarshaling of messages, we have to copy the message buffer once again before the objects it contains are reconstructed. The situation is similar on message sending. The net result is four more copying operations than may be strictly necessary. A more optimized implementation could eliminate some or all of these.

Another problem has to do the lack of interrupt driven messaging and timer interrupts. The messaging libraries do not provide interrupt driven messaging so we must poll the incoming message queues for new messages. Unfortunately, the AP1000's basic OS also does not provide timer interrupts so it is difficult to develop any sort of regular polling mechanism. The current implementation simply polls the message queue when there is nothing else to do (i.e., when idling). This works in practice but can lead to unnatural event (i.e., execution) ordering since incoming messages do not get into their receiver's message queue until *all* objects on the processor are idling. We can adjust priorities to reduce this problem somewhat but cannot eliminate it.

The addition of a timer interrupt as is available on the workstation version of *Tj* gives us regular polling but still limits responsiveness to, in the worst case, the polling period. On common workstations the fastest interrupts are 1ms. Future versions of the *Tj* messaging system will be more tightly coupled with the underlying system and improve latency.

The current system is unique in that it is the only implementation of an industrial grade, pure object-oriented language on an MPP class machine. The implementation of *Tj* in Smalltalk on this class of machine (i.e., MPP) is quite interesting and novel in and of itself. Our implementation gives users full access to all of Smalltalk's development tools including advanced configuration and version management utilities. They AP1000's front-end processor is integrated into the cell topology. Objects in cells and the front-end can directly reference each other and communicate transparently. Users are presented with a graphical front-end to the AP1000 cell array and can browse the executing code and inspect the resident objects. There is also basic support for remote debugging (e.g., stepping of remote processes).

The only other implementation similar to this is Hewlett-Packard's Distributed Smalltalk[16]. HPDST is implemented on ParcPlace Systems' VisualWorks system and runs on clusters of workstations. Fundamentally it is an implementation of the CORBA [31] distributed object system.

While it is an interesting system, it is based on a somewhat restrictive object model and does not include a general framework for meta-level computing or sophisticated marshaling etc. CORBA objects are relatively coarse-grained and heavy-weight.

5.6 Summary

We have detailed the various object models in *Tj*, our distributed object system. They go together to form a comprehensive distributed environment suitable for prototyping applications and experimenting with distribution mechanisms. In the distributed computing domain, *Tj*'s capabilities rival those of dedicated distributed systems like Emerald. *Tj* however, a retains and makes available to the programmer, the full power of the CodA architecture. Users take advantage of this to describe new behaviours which better suit their situation.

Tj provides a powerful and extensible framework for describing object marshaling on a per-object, per-object-group and per-use basis. Our model of marshaling descriptors presents the user/programmer with a single, clear and consistent model of inter-space object transport specification. The descriptors are used both for object marshaling and mobility operations (e.g., replication and migration). They are uniform and recursive.

Overall, *Tj* addresses problems related to introducing distribution to systems which previously had none and in the description of complex distributed object behaviour. The framework itself is robust and extensible. Future work in this area will be directed at the addition of distribution mechanisms (e.g., new replica coherency strategies) and automatic analysis of application code to direct the use of distributed mechanisms such as marshaling.

Tj is based on the idea of object behaviour change at the meta-level. It uses the CodA meta-level architecture to open the implementation of objects and provides a framework for the description of sophisticated distribution mechanisms (e.g., replication). The clear separation of base- and meta-levels facilitates the distribution of objects which were never intended to be distributed. With the CodA architecture users have a solid foundation for describing detailed policies for the use of distribution. They also have access to mechanisms for the control of the concurrency implicitly added with the distribution.

As we demonstrate in Chapter 7, the addition of distribution requires very little change at the base-level. As such, programmers can reuse entire class libraries and experiment with quite different distribution models without major concern for base-level behaviour.

This is in contrast to dedicated distributed systems like Emerald where users are forced to use the Emerald language and environment for which there is little existing software. It also changes the programmer's mindset. They are no longer programming a *distributed* system, but rather a system which *may* run in a distributed environment. We do not claim that programmers can completely ignore the rigors of distributed computing do claim that using a solid meta-level architecture give them an explicit place to describe how their objects should behave in that environment.

Our work is quite close to that of the Apertos team. The Apertos operating system 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.

Chapter 6

CodA implementation and use

We have implemented the CodA meta-level architecture in Smalltalk. This chapter gives the reader a look "under the hood" of CodA and will be of interest to users wishing to fully utilize or extend its capabilities. Typical programmers however, need not be familiar with all of this information to use CodA effectively. While we detail the Smalltalk implementation, readers are reminded that CodA is by-in-large independent of implementation language specifics. In general, CodA objects adopt as many of the properties of their host object system as possible. So, when we talk about CodA objects having some property, we are typically referring to CodA objects in the host environment (e.g., Smalltalk).

A key point of the implementation is that we transparently add CodA's capabilities to normal Smalltalk objects. This gives users full access to Smalltalk tools and class libraries, and gives Smalltalk objects full access to the flexibility and power of CodA. The high degree of integration achieved is possible largely because CodA deals with the runtime or operational behaviour of individual objects. It is orthogonal to language issues such as classes and inheritance. By separating these concerns we minimize the changes required to the host environment and improve CodA's integration.

While the normal Smalltalk object system is quite open, modification of the behaviours defined in the VM are often difficult if not impossible. CodA seeks to provide what might be called an *object engine*. That is, to provide the notion of *object* and a framework for defining the precise executional behaviour of these objects. Though CodA objects have *default* behaviours, most can be explicitly modified.

Another key point of the implementation is that reification is transparently introduced and only incurs overhead where it is used. Reification of one object or operation does affect others either in behaviour or performance. Fundamental in this is the use of the Smalltalk virtual machine (VM) wherever possible. For each Smalltalk object we enable the lazy and transparent addition of meta-level infrastructure and default meta-components (i.e., the conversion to a CodA object). The default components describe the standard Smalltalk object model. That is, a standard CodA object behaves just like a standard environment (e.g., Smalltalk) object.

In addition, the Smalltalk implementation has resulted in a number of unique and interesting implementation techniques and mechanisms. These include a novel solution to the object identity problem commonly encountered when trying to reify operations which are not explicit in the basic

Smalltalk system (e.g., message sending).

Since CodA and Smalltalk objects are almost completely interchangeable, the Smalltalk class library can be directly (re)used by CodA programmers. The capabilities of Smalltalk and CodA objects are however, substantially different. User's wishing to utilize CodA's flexible object structure must be aware of the differences and interactions between the object systems.

The most fundamental difference between Smalltalk and CodA is in how they treat classes. In Smalltalk an object's behaviour is defined by its class and the Smalltalk virtual machine. Classes define the object structure (i.e., the number and type of instance variables) and the methods to which instances respond. A class' location in the class hierarchy implicitly defines how methods are found (i.e., which classes are searched). The virtual machine (VM) defines all other object behaviours. These include how methods are executed, messages sent and instance variables arranged and stored in physical memory.

In CodA, an object need not have a class as such and if it does, the class generally provides only the default meta-components for the object. The CodA framework essentially opens both the class and VM portions of the Smalltalk object model[1]. Implicit in this departure from class-based objects is the ability to have object-specific behaviour. All Smalltalk objects of the same class behave in exactly the same way. They respond to the same messages, have same structure and handle messages in the same way. CodA allows the use of both class-based and prototype-based object definition techniques.

The next difference is the provision of arbitrary method resolution techniques. Smalltalk uses a single inheritance model where only classes are searched for messages applicable to a particular message. While the default behaviour of CodA objects is similar, users can explicitly define their own message/method resolution strategy. For example, multiple inheritance, state-based availability searching, double dispatching, etc.

Also embedded in the Smalltalk VM is the way in which messages are passed. Since Smalltalk is fundamentally a sequential system, the definition of message passing need not be sophisticated. Simple blocking synchronous sends are sufficient. Similarly, all objects are reactive. They are implicitly in an endless loop waiting for some other object to send them a message which they will process. The introduction of concurrency and distribution draws both of these behaviours into question. We require the ability to specify message types (e.g., synchronous, asynchronous, future, etc.) as well as how, why and when messages are received and processed.

As mentioned above, CodA takes lazy and partial approach to meta-level reification. While eventually all execution *bottoms-out* in the virtual machine, reifying a meta-component inserts another layer between the user code and the VM. In contrast to the interpreter approach, this *pushing away* of the underlying implementation platform (i.e., the VM) adds overhead to a restricted portion of the base-object's execution.

While the Smalltalk meta-level is implicitly defined in an object's class and the VM, a CodA object's meta-level is explicit. It is implemented as a concrete set of objects which are accessible and manipulable. As we saw in Section 3.5, the starting point for meta-level access is the object's meta. In general, users should not store or otherwise reference an object's meta directly. The actual object accessed using meta may be different from call to call. Rather, the meta message should be

[1] It could be said that since CodA is implemented in Smalltalk, then in fact, Smalltalk is not all that closed. This is of course true however CodA provides a framework or infrastructure for change where none existed in Smalltalk.

viewed as marking a level or viewpoint shift.

Once execution has shifted to the meta-level (i.e,. having executed anObject meta), programmers have access to the meta-components which define anObject's behaviour. There can be any number of meta-components related to a specific meta-level. By default, all metas have some definition for the standard behaviours as defined in Section 3.4. Some meta-levels are *fixed* in that they allow no changes or additions to this set. Others are *changeable* and allow existing components to be replaced. Still others are *extensible* such that entirely new behaviours and components can be added and existing ones changed.

6.1 Implementation strategy

As discussed in Chapter 3 and shown in Figure 6.1, there are two major elements in the CodA architecture, the meta and the components. Every object has a meta and a collection of components which describe its behaviour. An object's meta groups and organizes these components. In general, users do not need to explicitly program or manipulate the metas other than installing and removing component definitions. Components can be any object which responds to the interface protocol prescribed by the behaviour which it implements. So, as long as an object can do the job, it can be used as a meta-component.

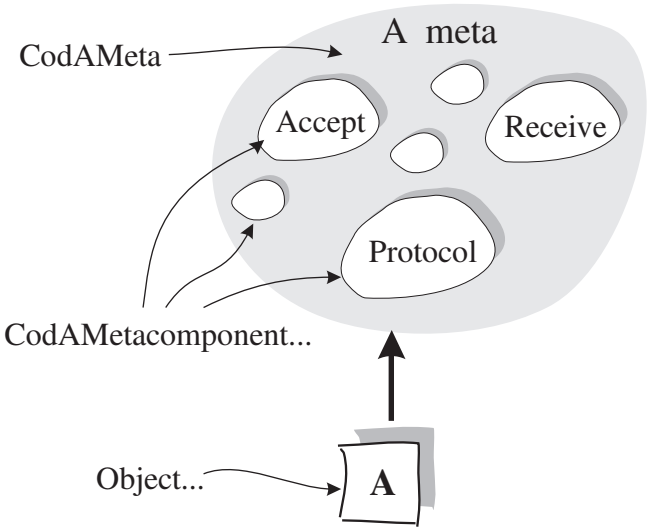


Figure 6.1: Meta-level structure

Figure 6.2 shows the developed class hierarchy for the major parts of the CodA meta-level. The *italic* classes are those which are part of the normal Smalltalk system. All others are unique to the implementation of CodA. These concepts and classes are detailed in the following two sections.

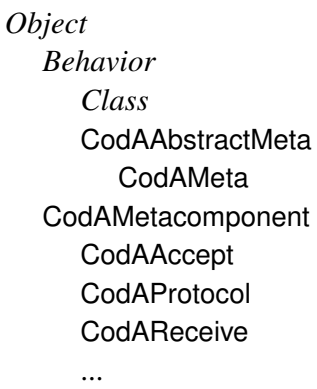


Figure 6.2: The CodA meta-level class hierarchy

6.2 Per-object meta-levels

To implement CodA's per-object meta-level definitions, we must be able to associate different behaviour descriptions (i.e., meta-levels) with different objects. In essence, we would like to associate a meta slot with every object. Unfortunately, Smalltalk does not directly support the runtime addition of slots to objects. There are four techniques which we can use to associate a meta with an individual object:

1. Have a global table which maps from object to meta.
2. Require CodA objects to be created as instances of classes which have a meta instance variable.
3. Use Encapsulators [35] or forwarders to 'wrap' normal Smalltalk objects with something indicating their meta.
4. Somehow embed the meta in the current object structure.

The first three options have severe limitations. The first requires the maintenance of a potentially huge table and extra overhead for every meta access. The second goes counter to our goal of reusing objects from the regular Smalltalk class library as CodA objects.

The third option, Encapsulators, has been a popular solution but suffers from the *object identity* problem. Stated briefly, the object identity problem occurs when one object has two addresses; that of the encapsulator and that of the actual object. As long as the encapsulator adds no semantic value to its target (i.e., it really just forwards *all* messages), this option is feasible. If the encapsulator does *anything* else, we have an identity problem: Users of one address will get different behaviour than users of the other. This is despite the two addresses representing the same logical object.

Consider the example in Figure 6.3. Here we see some object O with an associated Encapsulator E. In the example, E is not just a forwarder. It somehow transforms all incoming bar messages into foo messages and then pass them to O. Messages other than foo are passed on unchanged.

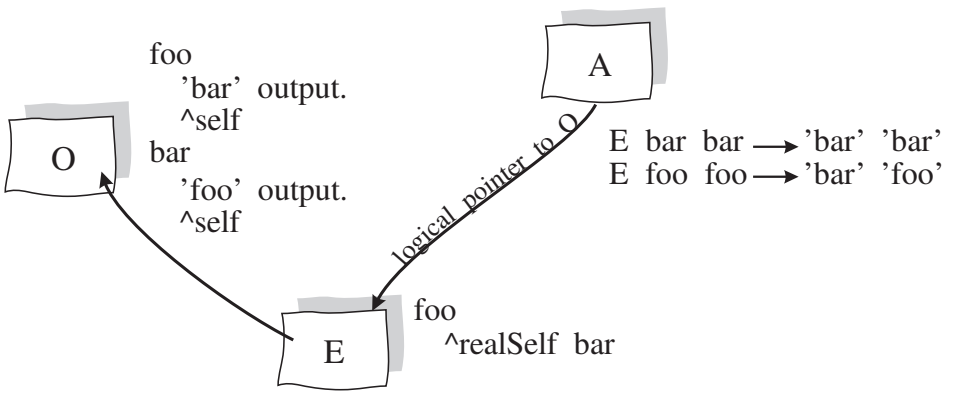


Figure 6.3: The object-identity problem

Since E encapsulates O, all objects which logically refer to O should actually refer to E. As long as this relation is maintained, everything is fine. All uses of O (i.e., E) will give consistent results. Unfortunately, the relationship is easily broken and E's true identity (i.e., O) can leak out.

The figure shows O's method bar which outputs some string and returns self. Executing E bar bar causes this string to be output twice. O also has a foo method which outputs a different string. Executing E foo foo causes bar's string to be output followed by foo's string. This is not what we expect since both foo and bar are defined as returning self. The behaviour is different because the first foo is intercepted by E and transformed into a bar message (as per the desired behaviour of E). O's bar method returns self which is O when the method is executed. The second foo message is sent to the result of the first (i.e., O). As a result, E is by-passed and the message is not transformed — The behaviour is inconsistent.

This specific example is indicative of the general problem. While it is theoretically possible to scan return values and substitute Es for Os, in practice, it is impossible for two reasons. First, return values can be arbitrarily complex and references to O may be nested arbitrarily deeply. Scanning the entire reachable world from the return value is impractical. Even if this scanning were done, it is impossible to tell which Os should actually be Es! There may exist some legitimate references to O.

The only remaining option is to add a new slot to hold the reified meta-level. It is impractical to blindly add this slot to all objects since only some will actually have metas and Smalltalk does not support the dynamic addition of slots to individual objects. So, rather than adding a new slot, we approach the problem differently and reuse an existing slot.

Every Smalltalk object has an implicit classField slot which holds its class. So, in fact this slot already contains the object's meta-level — its Smalltalk meta-level. Since the slot is embedded in the object structure and is never changed in the normal course of events, it is a perfect candidate for reuse. The slot is reused by replacing its original value with a object which mimics the original but adds new properties. In this case, the new object should point to the original class and be, or point to, the new CodA meta. Since the object in the classField is used directly by the VM, it must adhere to certain guidelines:

- Every object has a built-in class field called the classField. The VM uses the value in the classField to to lookup message selectors and find methods to execute and to determine the object's layout in memory (size, instance variable format etc.).

- The value in an object's classField must accurately describe the physical structure of the object as this information is used by the garbage collector when traversing objects. If the information is incorrect, a VM has little choice but to crash immediately in a most unpleasant way.

- Classes are not special or magic objects. They are instances of a subclass of Behavior. They have the peculiar property that they describe other objects. That is, they are at the meta-level.

- If an object's classField is changed, only that object is affected. Modifying the object contained in an object's classField however, affects all objects which also have that object in their classField (i.e., the other 'instances' of that 'class').

- It does not make sense to change the classField to some random class even if the structures are the same. For example, Associations and Points both have two named instance variables (i.e., the same structure) but the information they contain or represent is vastly different.

- Since the structure of an object cannot be changed once the object is created, changing the object's classField really only changes the methods to which it responds. Instance specific state can be added via indirect references through the classField.

All of these points are addressed if the object substituted (i.e., the meta) is an instance of Behavior or one of its subclasses[2]. When a custom meta is installed on a standard Smalltalk object, it generally should not override all of the methods defined by the object's class. By making the meta (an instance of a subclass of Behavior) be a subclass of the original class, the object automatically inherits all of its original methods from its original class.

Readers should take special care not to confuse the meta of an object with its class. *They are not necessarily the same!* It is simply an implementation detail that metas are stored in the spot normally reserved for classes and implemented as instances as subclasses of Behavior. CodA objects need not have any class (in the Smalltalk sense) at all. Having said that, by default, an object's class is its meta.

By implementing in Behavior, all of the methods normally associated with metas, classes are made to act as metas. Every object automatically has a default meta, its class, and is integrated into the CodA environment. Having done this, system-wide changes in behaviour are made by modifying the meta-related parts of Behavior. Class- and subclass-wide changes are done by modifying individual classes, and object-specific changes by creating a new meta and substituting it for the default as outlined above. Figure 6.4 shows a before-and-after view of dynamic meta-level reification. The procedure for effecting this transformation is as follows:

[2]Interestingly, it was to support techniques such as this that the Behavior class exists [12] but few people have put it to such use.

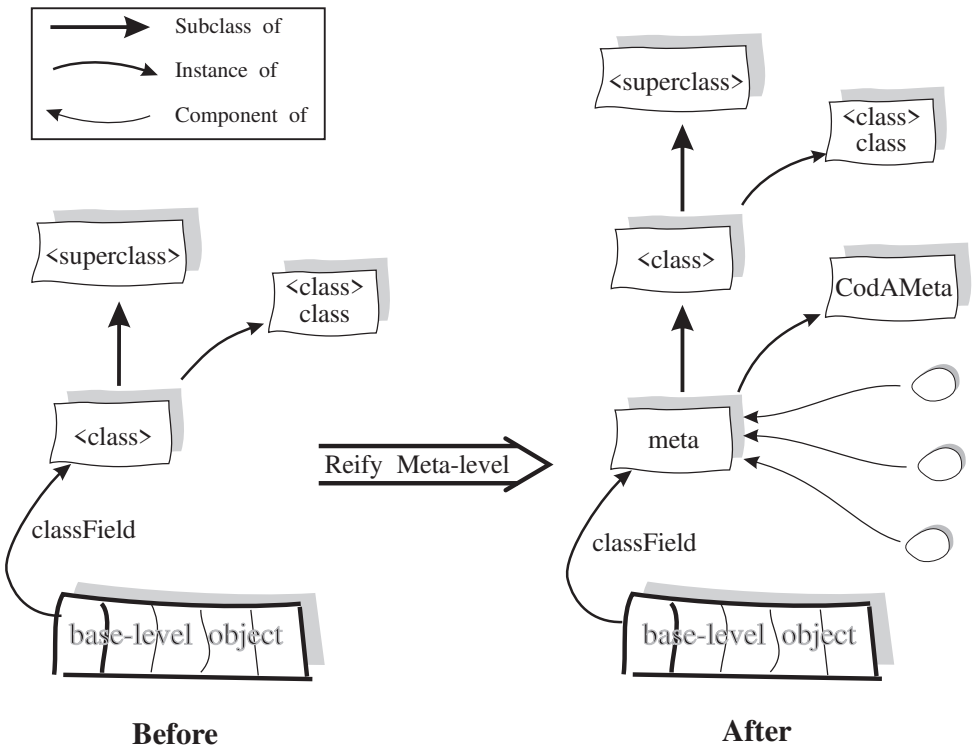


Figure 6.4: Dynamic reification of the meta

1. Find/create an instance of a subclass of Behavior which has the desired properties. This object will be the new meta.

2. Set the new meta's housekeeping fields (e.g., object structure description) to be the same as the object's current meta.

3. Set the new meta's superclass slot to be the object's class. Note that even though we are technically modifying the object's 'class' we take care to preserve the semantics of the class method. So class will always return the class of which the receiver is logically an instance rather than just the contents of the classField. This makes the substitution transparent to all clients of the object except for a few VM primitives which directly access the classField of objects (e.g., allInstances).

4. Modify the object's classField to contain the new meta.

5. Add component(s) to the meta and thus override the defaults for the object.

Assuming an empty meta is used, the result of the substitution is an object which has identical behaviour to the original, but now has a unique place in which to hold and modify behaviour. The new meta is inserted in the class/superclass chain between the instance and the original class. As such, the standard VM method lookup technique will look in the meta before it looks in the original class. This allows us to modify the normal methods of the object but leave its class, superclass and

other instances of those classes untouched. If the behaviour to be added or changed via the meta is the form of new or changed methods, the change can be embedded directly in the meta and used directly by the VM. This eliminates all of the mechanical overhead.

Figure 6.5 shows an example of an object whose behaviour is defined by system-, class- and object-specific meta-components. The object's classField points to its meta which contains three meta-components. These components are logically specific to the object. The meta is implemented as a subclass (actually a sub-Behavior) of the object's real class. Searches for meta-components which are not satisfied in the meta, move on and look in the object's original class. In the figure, the object's class specifies a further two components. These are potentially shared by all instances of the class (and its subclasses). Finally, any meta-components not defined by the meta or the class (and its superclasses) are defined by Behavior. These are global to the entire system.

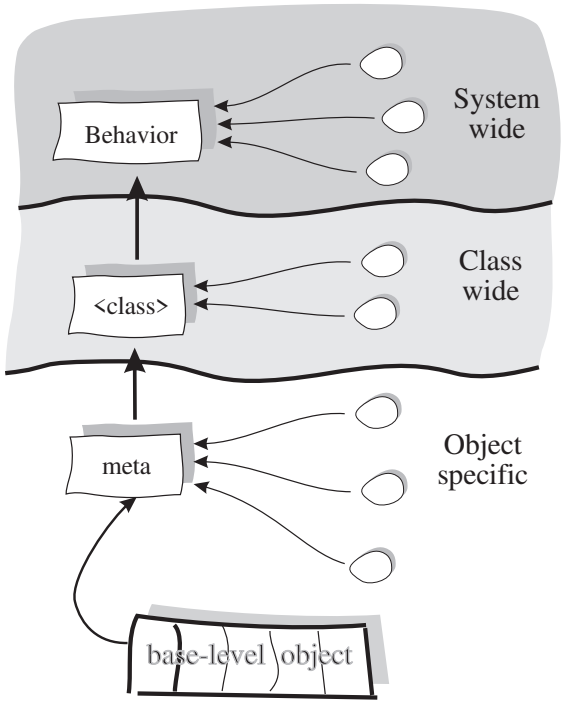


Figure 6.5: Implementation of the meta-level

6.3 Behaviours and meta-components

CodA defines an architecture for managing object behaviour descriptions. Object behaviours are described by meta-components. Components are generally, though not necessarily, subclasses of CodAMetacomponent. This abstract class supports a number of infrastructure methods for the installation, initialization and configuration of components. Behaviours are named and logically distinct though they may coincide in the implementation. That is, one meta-component can describe several different object behaviours. This logical distinction allows meta-level designers

more flexibility because meta-level programmers explicitly state the context in which they wish to compute.

Just as meta signifies a shift to the meta-level, specifying a behaviour (e.g., Queue) identifies the part of the meta-level which is of interest. In more concrete terms, the expression anObject meta queue answers an object which describes the Queue related behaviour of *anObject* and responds to all the messages expected of Queues (i.e., the Queue meta-component). It *may* happen that this object also describes some other behaviours (e.g., anObject's Receive), but this is in general neither apparent nor of interest to the base-level user.

An object's meta-components are accessed using a meta notation. For example, the following code accesses *anObject*'s State: anObject meta state. The method state is really just a convenience method and is equivalent to the more general: anObject meta componentAt: State for: anObject. Analogous methods exist for setting a particular meta-component of an object.

6.3.1 Changing behaviours

Since every object's default meta is its class, system wide default values for each component are set in the class Behavior[3]. Changing one of these components changes that behaviour for all CodA objects in the system which do not override the defaults. Individual classes override the system defaults by providing their own definitions. For example, CodA's Future objects redefine the default Accept component to be one which suspends the sender until a value is given to the receiver. To override the system default, a class redefines the component accessing method. How the method is overridden depends on how the default component is defined. The Future class, for example, defines a new class instance variable defaultAccept which is initialized with a FutureAccept component on system startup. The method Future class≫CodAAcceptComponent simply answers the value in defaultAccept.

Individual objects override their class and system defaults by instantiating an explicit meta-level and setting the appropriate components. As discussed above, the meta-level is instantiated using the Behavior≫asExtendibleMetaFor: method. This creates and installs an instance of the object's default explicit meta-level object. By default, this is an instance of CodAMeta. The object is then free to change or add any meta-component is chooses using the meta interface. For example, the following code installs myComponent as the definition of anObject's State behaviour:

```
anObject meta componentAt: State put: myComponent for: anObject
```

Components installed using this protocol are automatically (un)initialized and their shared resources properly handled.

There are relatively few restrictions on how meta-components are created and used. Definitions for particular behaviours are shared by arranging for the behaviour accessing method (e.g., CodAAcceptComponent) for several objects to return the same component. This is seen with the default components. Note that some components maintain a strict 1:1 relation with the base-level they represent and so cannot be shared. In other cases, components must be created dynamically for each reference (see Section 5.1.2 for an example). Programmers should be careful not to make assumptions about meta-component stability from one call to the next as the actual component accessed may be different in each case.

[3] All classes are instances of Behavior so Behavior's instance methods appear as normal class' class methods.

6.3.2 Adding behaviours

Adding entirely new behaviours to the framework is straightforward. First the user must create and install a name for the new behaviour. This is just a entry in a system table. Next they should create at least one object which will be the behaviour's description (i.e., a meta-component). Frequently two different definitions are required. One as the default for the general use and compatibility of all objects, and another which defines the user's new mechanisms.

If the behaviour is to be installed on all objects, then an accessing method (e.g., CodAMyBehaviourComponent) must be added to Behavior. As shown in Figure 6.5, this makes the component available to all objects. The method should return an instance of the behaviour's default description. Analogously, if the behaviour is class specific then this method should go in the appropriate places in the class hierarchy. Instance specific behaviours are installed directly on the meta of the object in question using a procedure similar to that for changing the value of existing behaviours.

6.4 Messaging

We have reified the meta in an integrated way but still have to enable its use. Since all Smalltalk execution is via message passing, we can hook the basic execution of Smalltalk objects via the message passing mechanisms. Message passing in Smalltalk is implicit. For CodA we would like to explicitly control both the sending and arrival of messages. The typical technique for doing this is *retro-reification* — Instrument receivers to trap incoming messages and then reinvoke a reified send operation from the original sender. Trapping is usually accomplished using the doesNotUnderstand: method in an object which is an instance of a subclass of nil (so it does not understand any/many messages). CodA supports this but in a slightly different way.

We introduce the notion of an *interceptor*. An interceptor is a Behavior which sits between the object and its real class and so is automatically searched by the VM on every incoming message. Note that interceptors are distinct from metas even though they use a similar mechanism (i.e., they are stored it the object's classField).

Using the interceptor we can transparently and individually trap messages. If the interceptor defines no methods and is a subclass of nil then all messages to the intercepted object are trapped or reified (see CodAMeta≫reifyAllMessages). Alternatively, we can install special trapping methods for individual selectors (see CodAMeta≫reifyMessage:to: and related methods). By default methods are *not* automatically trapped and programmers must explicitly state which incoming messages they want reified. Some object models (e.g., concurrent objects) provide infrastructure for declaratively stating which messages should be reified (see Behavior≫CodApublicMethods) but this is not true of all models.

6.4.1 Message sending reification

Retroactively reifying messages at the receiver is insufficient. It does not allow us to specify different forms of message sending or to reify messages to objects which have no installed interceptors. In this case we want to directly invoke the sending object's Send operations as detailed in Section 3.4.1.

For example, if message is some message we created then we can send it to anObject using the anObject meta send send: message for: anObject operation. There are a number of send operations from which to choose. Using this facility, message senders explicitly state the nature of the operation rather than depending on message receivers to ask how they would like the message transmitted.

This raw messaging syntax is not very convenient for users as it requires that they have an actual message object. Unfortunately, messages are not automatically reified, but are embedded in the system, making their creation somewhat cumbersome. In addition, the change in syntax of such a fundamental element of the underlying system, while functional, may have severe consequences. It is overly verbose, exposes too much of the underlying infrastructure, requires too much of the programmer and obscures the semantic content of the user's code with syntactic requirements of the environment. Furthermore, it makes CodA code look quite different from Smalltalk code.

Integrating the new operations into the language syntax is a better approach. Table 6.1 gives a set of ABCL-like equivalents for the CodA Send operations. Using this syntax the anObject meta send send: message for: anObject specification becomes anObject <- message. This is much clearer and more convenient.

Short-form	Long form
<-	send:for:
<>	sendSync:for:
||	sendAsync:for:
«	sendFuture:for:

Table 6.1: Message sending forms

There is still one problem, the creation of the message object which is the argument to the send operation. As we have said, messages are embedded in the underlying system, not automatically reified. We could modify the compiler to reify the messages specified after one of our special send operations but this involves the modification the underlying system, something we wish to avoid where possible. It is also not very extensible. New messaging concepts are not easily integrated as they will require further compiler modifications.

Instead, we use a dynamic technique involving *message builders*. A message builder is an object which dynamically creates and returns message objects. In fact, the VM itself acts as an implicit message builder when it invokes the doesNotUnderstand: method. The argument to this method is an explicit message object which represents the implicit message which was not understood. The VM has taken an implicit message and built a message object. Leveraging this, we can build message builders for our own purposes. A trivial message builder is just an object which understands only the doesNotUnderstand: message and implements it as shown below.

```
doesNotUnderstand: message
  ^message
```

If the message builder is called M then the expression M at: 1 put: 69 returns a message object whose selector is at:put: and arguments is #(1 69). This message can then be used in the typical

send operations. Due to a quirk in Smalltalk we cannot embed objects like M directly in methods (as would happen with this expression) because they do not respond to typical Object methods used by the system. So M is actually a message builder builder (i.e., class) and will return an instance of itself in response to the b message. Message reification now takes the form: M b at: 1 put: 69. Combining this with the sending short-forms we have:

```
anObject <- (M b at: 1 put: 69)
```

Not quite perfect but a big improvement over the fully manual version which looks like:

```
message := self meta send
    buildMessage: #at:put:
    withArguments: #(1 69)
    to: anObject
    from: self.
self meta send send: message for: self
```

Note that the message created by a builder can be further manipulated and stored before being used. For example, the message send strategy (e.g., asynchrony) and the notion of 'express' messages are independent. Any message can be made into an express message by setting it's express flag. So, anObject <- (M b at: 1 put: 69) asExpress will send an at:put: message to anObject and interrupt its execution (i.e., be express).

6.4.2 Message accumulators

The message builder concept can be extended to define new messaging concepts such as message *accumulation*. Accumulators collect up messages which are sent to them without actually sending them on to another object for processing. This allows sequences of messages to be grouped and *batch* processed. For example, the expression M a boss husband name at: 3 will result in the accumulation and grouping of the boss, husband, name and at: messages, in that order, into one message. Note that message accumulation is signified by M a rather than M b.

In addition to having a state vector for containing the accumulated messages, message accumulators also respond to one message, the m message. Sending m is the mechanism for escaping from infinite accumulation. It returns a message object containing all of the selectors and arguments as they were specified at their original execution. Note that the arguments are evaluated *prior* to message construction and so are embedded in the message structure.

Accumulated messages can be queried, modified and sent to a receiver just like any other message objects. For example, the expression self <- (M a boss husband name at: 3) m will answer the third character of the name of the husband of self's boss. This construct is useful for packaging a series of messages for remote execution, dynamically creating transactions or representing logical hypertext links.

6.4.3 Debugging with messages

Debugging in the face of message passing is easy in Smalltalk because all messages are sequential, synchronous and local. In CodA however, we are able to create object models where a message's sender and receiver are disjoint either in location or execution (e.g., distributed or concurrent objects) and unfortunately, the standard Smalltalk debugger is uniprocessor and uniprocessing. Cross-process(or) messages are not accommodated. If a user encounters a send to a concurrent or distributed object while stepping in the debugger, they will not be able to follow the execution chain into the receiver of the message. This can be quite frustrating.

We address this by having message objects (i.e., instances of CodAMessage) support a number of debug flags which can be set by the user. In particular, the *execute*, *accept* and *marshal* debug flags. If the accept flag is set then execution will halt when the message is accepted by the receiver. In a uniprocessor system this is not strictly necessary as conventional debuggers can step up to this point. In a distributed or multiprocessor system however, this gives users debugging breakpoint before the message is queued for the receiver. If the execute flag is set then the receiver will halt just before it executes the method which was resolved for that message. This allows users to send a message from one process and invoke the debugger on the receiver process and step through that execution. The marshal flag halts processing when the message itself is (un)marshaled during the send. This is convenient for investigating low-level problems and seeing how objects are being transmitted.

6.5 Programming with CodA

Programming with CodA is often an exercise in modification. Programmers use whatever language and tools they normally use and then apply meta-level behavioural changes to the developed objects. Examples of this are shown in Sections 7.1 and 7.2. Programmers may also choose to develop applications which explicitly depend on CodA's facilities. An example of this is given in the form of the Vibes behaviour analysis system detailed in Section 7.3.

Base-level programming with CodA is quite similar to programming with normal Smalltalk. Though CodA objects need not be class-based, programmers typically start by creating a Smalltalk class which gives the bulk of the desired base-level behaviour and then proceed to build mechanisms for modifying the behaviour of instances of these classes as desired/required.

Programming the meta-level is approached like programming any other application. Programmers create the objects (e.g,. metas and meta-components) which describe the desired behaviour and put them together to solve the problem. This is just like programming the base-level except the subject matter is object behaviour rather than some application domain.

6.5.1 Code changes

One of our goals in building CodA was to enable widespread programming by modification and thus the widespread reuse of class libraries in a variety of computational environments. Success or failure in this respect can be measured by looking at the amount of original base-level code which

needs to be modified, and the nature of the modifications required, to accommodate changes at the meta-level. We classify these changes into four categories:

Annotations are modifications which do not alter the base-level semantics of a method or the messages (contents or order) it sends. The alterations 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 annotational change in the receiver. Meta code too is often considered to be 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 a new behaviour (e.g., concurrency and distribution).

Structural changes are typically done to classes 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. In this case the code is restructured to have a separate method for the iteration section. One could argue that these changes are only required because the original designers somehow failed to adequately factor their code. In reality however, one cannot design for all possible uses 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 behaviours like concurrency or distribution. 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.

In later chapters and sections we measure the effectiveness of CodA in terms of the number of *required* and *optional* changes required in each of these categories. Required changes are, as the name implies, changes without which the system would not function. Optional changes are ones which improve readability, modularity or performance of the system.

Ideally we would like to make most required changes as annotations or additions as these have the least impact on the base-level. Structural changes can often be done with relatively little impact but may introduce programmer errors and may be incompatible 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.

6.5.2 Optimization

Optimization and behaviour change in general using CodA is quite straightforward. Because there are concrete objects at the meta-level which describe each operation of an object, users know where

a particular behaviour is defined and so what to change. The separation of the meta-level from the base-level and the components from each other isolates the effects of changes in one component.

From a meta-level implementor's point of view, optimization is a matter of implementing the meta-level itself rather than end-user applications. The decomposition into many fine-grained objects as proposed in CodA is very powerful but that power does not come for free. In general there is some overhead both in time and space. Having many different objects can cause state to be split or duplicated and methods to be broken into very small units. The physical representation of a fully reified meta-level can be quite large relative to the base-level object size.

The simplest optimization strategy is the substitution of less general but higher performance components. For example, by default, DistributedObjects use the fully general Marshaling component and have complete access to all its power. In some cases these capabilities may not be necessary. Users are free to substitute some other Marshaling component or to develop one of their own and then substitute it.

In addition, CodA makes a clear and strong distinction between behaviours and the meta-components which define them. From a meta-level user's view point, each behaviour is distinct as are the components which define them. From the implementation view point, this is not necessarily true. It is perfectly acceptable to have one physical component describe several behaviours. So, for example, in some situations there is a very strong correspondence between the Queue and Receive behaviours. Rather than maintaining two sets of queues and lists, we can implement a combined component which gives suitable definitions of both behaviours and has better performance. This is discussed in Section 3.7.

This does not require any changes to the base-level code or even to meta-level code which uses the correct meta-level accessing technique (e.g., anObject meta queue). It does however restrict somewhat our ability to combine meta-components because it increases the number of constraints which must be satisfied. So, we lose flexibility but gain time/space performance.

As future work we propose a technique whereby distinct meta-components are automatically *compressed* into a smaller number of composite meta-components. This compression would be accomplished by techniques related to partial evaluation and dynamic compilation. The compressed meta-components would have better performance characteristics but describe the same behaviours as their constituents.

By remembering something of their original form, we would be able to reconstruct the constituent components on demand. Using this, we can regain the flexibility and composability inherent in the fine-grained approach. When a change is desired we decompress the composite components, make the changes and then recover the performance improvements by recompressing the new constituents into a new composite.

6.6 Summary

In implementing CodA in Smalltalk we have found the operational decomposition of its meta-level to be quite useful. Since the architecture itself is free of high-level object notions (e.g., classes and inheritance), it is very easy to mold it to fit a particular environment. The implementation we have provided is completely integrated with the underlying Smalltalk system. It is so seamless

that CodA objects cannot, in general, be distinguished from Smalltalk objects. Furthermore, the models developed for CodA (e.g., DistributedObject) are universally applicable. They can even be applied to the objects which implement the CodA architecture itself! Chapter 7 presents several applications modified by or implemented with CodA. These are a demonstration of our success in enabling programming by modification and integration with the underlying system.

Our implementation also demonstrates a unique solution to the object-identity problem. Since the new information (e.g., the meta) is held within the object itself rather than externally in a table or encapsulator, every object has only one address and costly accessing overhead is eliminated.

Smalltalk has proven to be an excellent implementation environment but there are still areas for improvement. In particular, the Smalltalk virtual machine (VM) is not completely open. Some operations such as message sending and method lookup are embedded in the VM. Also, since Smalltalk is fundamentally a class-based language, the implementation of object-specific behaviour is somewhat costly with respect to memory requirements. We have fed back our observations on VM requirements to the virtual machine authors and look forward to reimplementing CodA in a system with more VM support for open implementations.

At a higher level, Smalltalk's untyped paradigm both helps and hinders CodA and its object models. The absence of strong typing allows the free substitution of components at the meta-level and the use of objects in completely unexpected ways. While the use of typing would not necessarily exclude these possibilities, it would significantly complicate their implementation. On the other hand, type information would be very useful for determining advance information on object use patterns. For example, if the type of method arguments was known at compile time, static, optimized marshaling code could be precompiled.

Chapter 7

Applications

The capabilities of the CodA framework have been demonstrated in the form of various object models described in Chapters 3, 4 and 5 (e.g., ConcurrentObjects, PortedObjects and DistributedObjects). We have not however, shown how these models and CodA in general are relevant to the real-world. Here we present a series of example applications which demonstrate how CodA is used on real problems and to show the usefulness of the models we have described.

The first two applications, the N-Body problem and an expert system tool, involve the ConcurrentObject and DistributedObject models — more generally, *Tj*. In the first we demonstrate the use of CodA as a runtime behaviour investigation tool. The N-Body application is small but exhibits quite different behaviours depending on how its object's meta-levels are defined. The expert system tool example is intended to show how CodA is used to change the computational domain of large, third party systems. Here we focus on the separation of the base- and meta-levels and the amount of base-level code change required to accommodate our meta-level changes.

The third example is Vibes, an object behaviour analysis system which uses the PortedObject model to build arbitrary analysis graphs and the CodA architecture to create an infrastructure for object behaviour monitoring. Whereas the first two applications are largely modifications of existing systems or algorithms, Vibes is a completely new application, explicitly designed with CodA in mind. This demonstrates CodA as a support environment for advanced application development and highlights the degree to which object behaviour can be manipulated. It is interesting to note that we actually use Vibes to monitor and analyze its own underlying implementation (i.e., CodA).

For each application there is a description of the basic problem and an overview of some experiments using CodA to implement the applications. This is followed by a summary of the concepts and mechanisms in CodA which were found to be particularly useful in the development of the application.

7.1 N-Body

7.1.1 Problem description

The N-Body problem is a relatively simple and deterministic application. This makes it ideal for experimenting with different computational models. The 2D N-Body problem is one of calculating

the forces exerted by, and movements of, a collection of bodies or *particles* in a 2D plane.

A popular force model is that of Newtonian gravity where the force exerted on particle A by particle B depends on the mass of B and the distance between A and B. The general solution requires each particle to consult all other particles in determining its force (acceleration) vector. Once all the forces are known, each particle is repositioned according to some time-slice factor and its current position, velocity and acceleration. In practice it has been found that since the force varies inversely to the cube of the distance, once the distance between particles reaches a certain distance, their effect on each other becomes minimal and can be estimated. Barnes and Hut [4] developed such a solution using quad trees.

Quad trees are spatially dependent structures which successively divide two dimensional areas into quadrants. Each quad tree node has an associated area and four children which represent the four quadrants of the node's area. Elements are stored in the tree keyed by their two dimensional Cartesian position. Since all the elements in a specific area will be in a known subtree, they can be easily manipulated. As elements are added, the tree is subdivided to ensure that each leaf node holds only one data element within its area.

To apply this to the N-Body problem we build a quad tree which holds all the particles and then traverse the tree when calculating forces. When a node is traversed, if the distance from the particle to the node's center of mass is beyond some threshold, we can stop our descent down that branch of the tree and use the subtree's center of mass data (i.e., position and mass) to estimate the force on the particle. This dramatically reduces the number of particle interactions. With that in mind, the algorithm can be summarized by the following four steps:

Tree construction All of the particles for the problem are inserted into a quad tree.

Tree preparation The tree is traverse and estimates of the center of mass' position and magnitude are made for each subregion (tree internal node).

Force calculation The tree is traversed for each particle and the forces acting on that particle are accumulated. Estimation is done as described above.

Particle movement Each particle's position is updated.

7.1.2 Adding concurrency and distribution

The sequential, uniprocessor implementation of the N-Body problem consists of three kinds (classes) of objects: Particle, QuadTree and Solver. The code for these objects is given in Appendix B. Particles and QuadTrees are quite generic. A particle is anything with position, mass and velocity attributes. QuadTrees can hold any object that has a position attribute. The Solver embodies the N-Body algorithm outlined above. It maintains a set of particles to be processed and any required data structures (e.g., the quad tree).

We are interested in the behaviour of this algorithm in a concurrent and distributed environment. Concurrency is introduced by making each node of the tree into an active object. The particles are treated as passive data but since they are only ever changed at one point in the algorithm (the final step), there are no consistency problems. The basic algorithm is not altered as a result of these changes.

The algorithm is distributed by partitioning the objects over the machine topology and managing their interaction. In this situation, interaction management takes three forms: positioning, replication and caching. Particle positioning is done by partitioning the particles into subsets which are worked by different processors. The subsets are not necessarily spatial in nature (i.e., they are random) and are interdependent during the force calculation phase. That is, the particles of one processor may exert forces on those of another. The main mechanism for coordinating this interaction is the quad tree. As such, we require one global quad tree which organizes all particles for the entire problem.

As a global, the tree is a bottleneck. It is traversed by all particles during tree construction and force calculation. Replicating at least its root on every processor can save many remote message sends. Returns on quad tree node replication costs decrease rapidly as we descend the tree since nodes are traversed fewer and fewer times. Rather than replicating deeper nodes, we can cache them. Caching differs from replication in that no consistency management is done. A cached object is a copied object which maintains an identity relationship with the original. This caching can only be done once the tree is complete as node values may be changed during element addition. During tree preparation (i.e., once the tree is complete), remote references to children are converted to cached references. When the tree is next traversed (i.e., during force calculation), accessed nodes are resolved locally and further remote messages eliminated.

An alternative is to explicitly cache the entire tree on all processors. This may be inefficient since the Barnes and Hut estimation scheme results in the traversal of only a limited number of the tree nodes. Using the on-demand caching technique outlined above copies only those nodes which are actually required.

We can also benefit from replication of the solver itself. By having a replica of the solver on each processor, we initiate processing in parallel rather than having one central control point. The replica need only synchronize periodically as the algorithm demands.

7.1.3 Changes to original code

Theoretically, caching and replication can be done without CodA or *Tj*. However, doing so requires significant changes to the base-level code. We must add caching and locality checks and somehow specify argument marshaling strategies. When experimenting with many different configurations, such changes require a great deal of effort and may introduce errors in the algorithm. On the other hand, using a meta-level architecture, we can make behaviour changes with either no changes to the base-level code or simple annotations.

Table 7.1 summarizes the changes required to add concurrency and distribution to a sequential implementation. Note that the types of changes listed in the figure are defined in Section 6.5.1. In the table the 'Total' row summarizes the original application while the 'Summary' row gives an overall view of the modifications and additions. The original and modified code is shown in Appendix B.

Both the summary and the code show that the changes required were quite minimal and non-intrusive. Only in our treatment of the tree update did we actually have to change the code in an intrusive way to accommodate the fact that the particles were distributed (see Solver≫addParticlesTo: in Appendix B.1.5). The rest of the changes are annotations or additions to handle the specification

of location information or control concurrency. Note that the Particle class remains unchanged.

Type	Required		Optional	
	Classes	Methods	Classes	Methods
Total	3	25	–	–
Annotation	2	5	0	0
Semantic	0	0	1	2
Addition	2	4	0	0
Structural	1	2	0	0
Summary	2	11	1	2

Table 7.1: Summary of N-Body code changes and additions

The location specification changes are required because the default of creating objects on the same processor as the creator is not always appropriate. It can lead to undue clustering. In particular, we need a way of distributing the quad tree nodes as the tree grows. This distribution determination is user-specified and can be anything from random to some function of particle or parent location.

Running parts of the problem on different processors introduces implicit concurrency. As such, synchronization may be required at various points. In this example, synchronization is required between each of the four main steps of the algorithm and during quad tree modification (e.g., particle insertion). There are also places which have unnecessary synchronization. While in general this does not harm the quality of the solution, it may restrict the amount of concurrency which can be realized. Synchronization is added or removed by declaratively changing the type of message send from the default future type to synchronous or asynchronous (respectively).

Replication of the solver and/or the tree root is done in a declarative fashion at problem creation time. For example, the code below creates a solver and replicates it on every processor. The (tree) annotation is a marshaling descriptor and indicates that only the value of the solver's tree slot is to be copied and consistency managed — all replicas use the same tree. The symbols in the (|| ...) annotation are message selectors which, when sent to the solver, will be simultaneously broadcast to all replicas and executed in parallel. Using these meta-level annotations we successfully hide the replication and parallelism of an object (the solver) from its clients (the objects sending it messages). For more on these and other annotations see Chapter 5.

```
solver := NBodySolver new.
solver meta replication
   replicateEverywhereUsing: #((tree) (|| createTree: calculateForces moveParticles:))
   for: solver.
```

There are a number of ways of adding caching. The easiest is by specifying it in the marshaling descriptor of some message. Simply by specifying the #cached descriptor the relevant object (argument) is passed by reference to the remote space. If the reference is ever sent a message, it is automatically resolved locally by fetching the remote object represented by the reference.

Alternately, existing remote references can be dynamically changed into caching references by modifying their meta-levels. Explicit caching of remotely referenced objects is done using one of the copy/localization operations of the reference's State meta-component (e.g., copyLocalUsing: descriptor or localizeUsing: descriptor).

7.1.4 Experiments

Figures 7.1 and 7.2 show messaging graphs for four different runs of the same N-Body problem. These particular runs were with 128 particles running on 32 nodes of a Fujitsu AP1000 [38]. All figures represent the number and type of messages sent by the objects on one processor of the AP1000 (processor 0). Alternative views and monitoring schemes are possible but this demonstrates the depth to which one can go in investigating object behaviour.

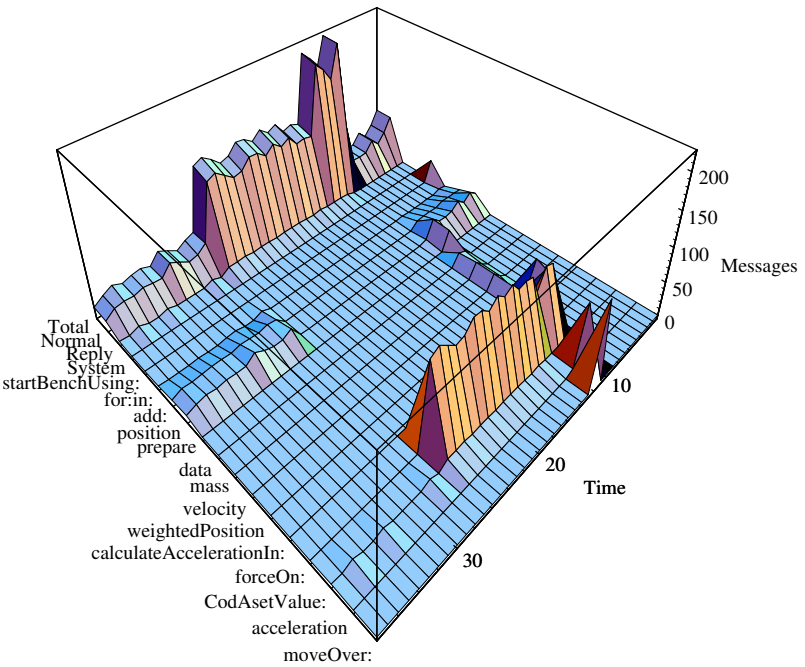
In the figures, the vertical dimension quantifies the number of messages sent to a remote processor. Time is sliced into uniform chunks of one second along the right-hand horizontal axis. The left-hand horizontal axis represents message *type*. There are two kinds of types; raw system and user. The system types are; normal, reply and system. These types are applied implicitly to all messages. Users can also specify their own types for each message. In our example we have simply mapped message selector onto a type. The symbols noted along the bottom left axis of each graph are the type identifiers. Note also that the left most type is *Total*. This is the total of all messages of any type.

The runs shown in each graph were varied by changing the use of replication and caching on the objects in the algorithm. The runs in Figure 7.1 have no replication while those in Figure 7.2 are with the root of the quad tree replicated on all processors. In both figures, the runs in the lower graph have particle caching enabled.

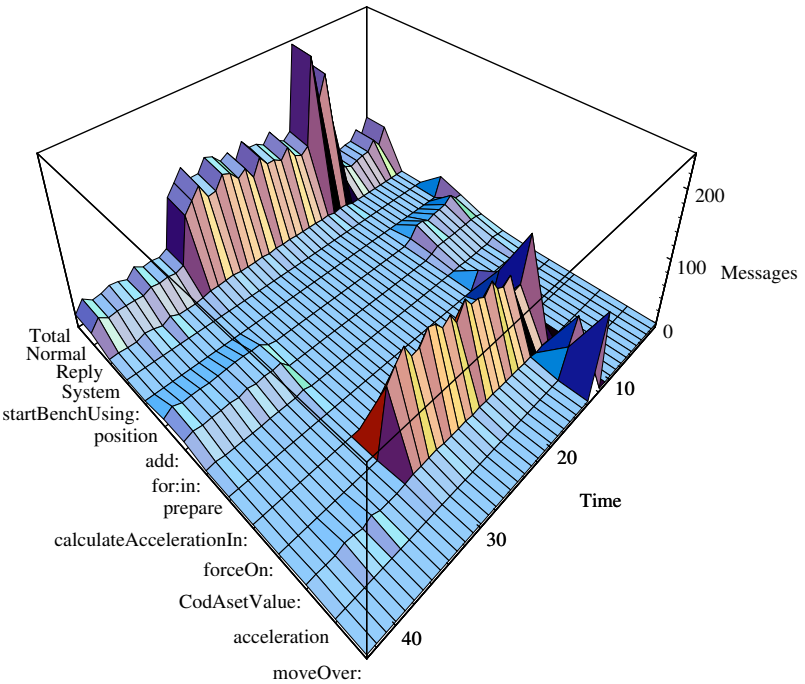
Looking at the figures there are several interesting points to note. First is that they all have roughly the same shape but that the time scales differ significantly between the replicated and non-replicated versions (Figures 7.1 and 7.2). We observe that the main peak in all the graphs is due to the forceOn: message. The forceOn: method traverses the quad tree to calculate the total force exerted on its argument. Comparing the replicated and non-replicated versions we see that the without replication the peak is lower but stretched out in time. This is in contrast to the replicated versions where the peaks are much higher but are compressed in time. This is the telltale sign of a bottleneck.

Without replication, all other processors send their forceOn: messages to processor 0 which happens to hold the root of the quad tree. Consequently, processor 0, in traversing the tree, will have to send many forceOn: messages. By replicating the root, many of these messages are distributed over the other processors. The level of concurrency is increased and the severity of the bottleneck is reduced.

We also noticed only a slight difference between the runs with and without caching (the lower and upper graphs of the figures respectively). This was somewhat surprising as we expected a significant savings due to eliminated remote message sends. Caching did however change the the number of different types of messages sent. We can see that almost all of the messages normally sent to Particle objects (e.g., data, mass, velocity) are eliminated by caching. Unfortunately, it appears that these messages do not represent very large portion of the overall message traffic and

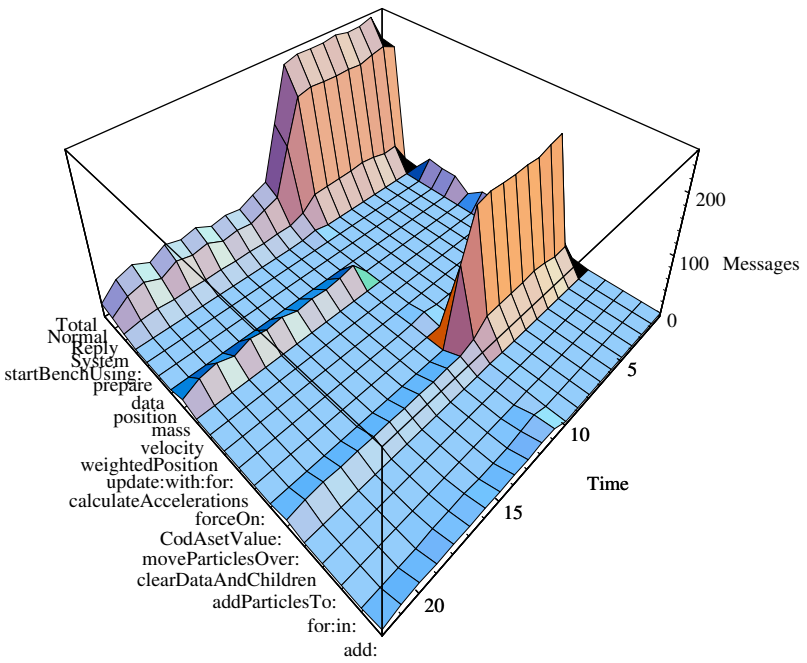


(a)

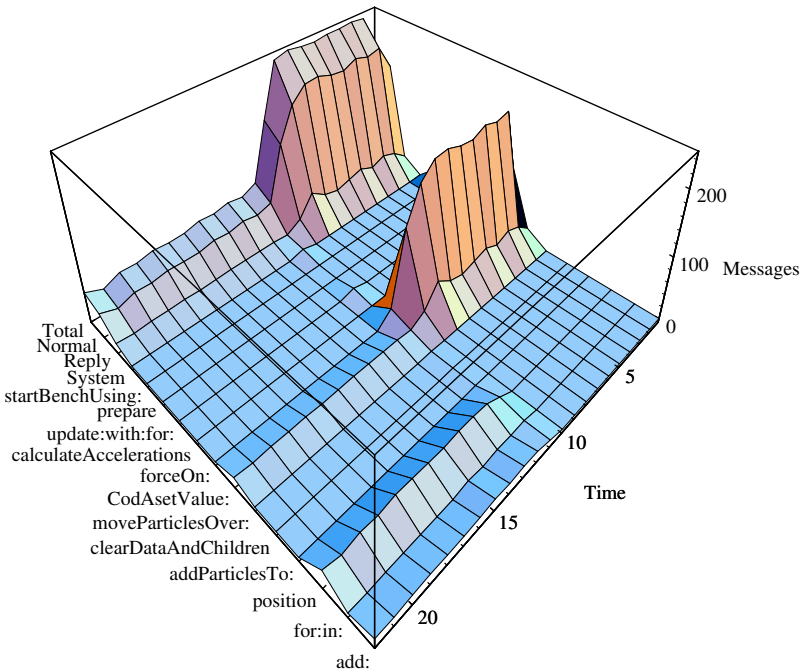


(b)

Figure 7.1: Messaging behaviour without replication



(a)



(b)

Figure 7.2: Messaging behaviour with replication

so our caching efforts are ineffective. While this is disappointing it does point out that not all *optimizations* are useful in all applications. It is important that we be able to easily identify those which are useful relative to those which are not. CodA allows us to do this with little effect on the base-level code and so greatly reduced programming effort.

7.1.5 Summary

Though we have done many more, these four tests are representative of the way in which users use the system. Appendix B.1.6 details the changes and invocation sequences required to run each of the test cases discussed here. They show the variety of things possible using simple modifications and without impact on the base-level code. The data shown in the figures is just one small portion of that which can be collected and analyzed by Vibes, CodA's monitoring system (see Section 7.3).

Using these techniques, programmers run their applications and gather performance data. This data highlights, for example, areas of the algorithm which result in inter-processor messages. The objects involved can then be modified (e.g., replicated, cached) and the number or characteristics of the inter-processor messages changed. As we have seen with caching, things that we think will improve performance do not always work. It is vitally important that designers and programmers be able to experiment with distribution configurations and examine their behaviour. Since they may want to try many configurations, it is also important that configuration specification be non-intrusive on the base-level application code. These are some of the primary motivations for creating and using CodA.

7.2 Expert system

7.2.1 Problem description

The N-Body problem discussed in the previous section is quite regular and predictable. The units of computation are distinct and disjoint as are the points of synchronization. To examine a more dynamic application we chose a forward chaining first-order production system based on the Rete pattern matcher[13]. The implementation we chose, **ENVY**[1]*/Expert*, is a commercially available product written by a third party. This 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 7.3. An *expert system* consists of a collection of objects or facts (*fact base*), a collection of rules (*rule base*) and an *inference engine* which processes this knowledge. The rule base is really a Rete pattern matching graph with input, output and internal nodes. The internal nodes represent individual tests corresponding to the clauses in the predicates of the rules. Each output node represents the complete set of predicates for one rule. The graph has a single input node which is the root of the pattern matcher. **ENVY/***Expert* is fully integrated with the Smalltalk environment and uses Smalltalk code to represent antecedent clauses and rule conclusions.

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

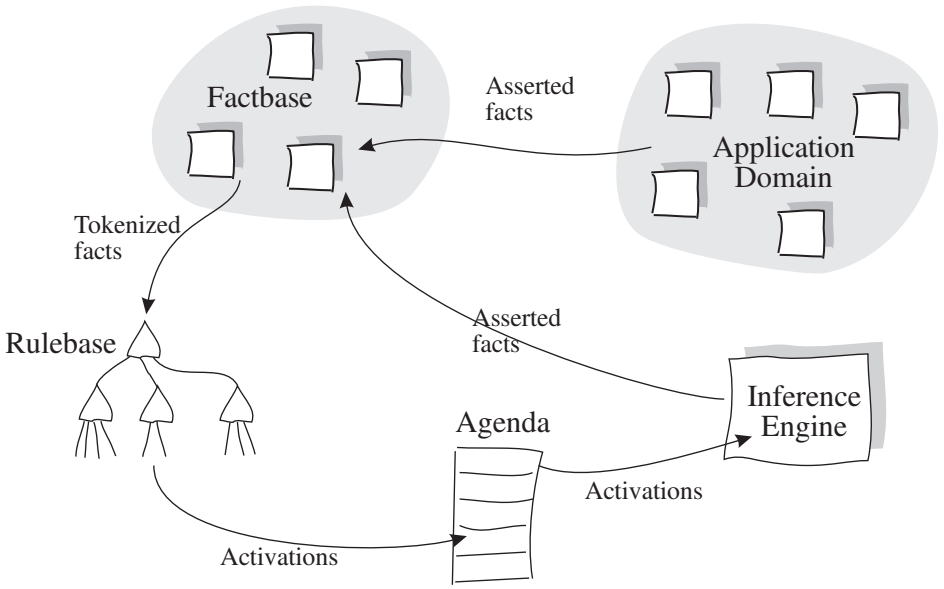


Figure 7.3: Overview of **ENVY**/*Expert*

When new *facts* are discovered, they are wrapped in *tokens* and injected into the root of the matching graph. At each node the token is optionally merged with other waiting tokens and tested. If it fails the test, that branch of matching is terminated. If it succeeds, it 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 a particular rule. That rule is activated (added to the *agenda*) with the satisfying context.

The inference engine is somewhat disjoint from the predicate matching process. 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, invoke the pattern matcher and thus activate or deactivate rules. Execution continues until there are no more activations to be processed.

7.2.2 Adding concurrency and distribution

The main goal of experimentation with this application was to investigate the addition of both concurrency and distribution to an *unsuspecting* application — one which was not designed with that in mind. To introduce these behaviours to the expert system (i.e., the facts and rules) and the inference engine itself, the first step is to enable the assertion of facts which are not local.

This does not distribute the inference engine directly but rather allows it to work on distributed objects. Since distribution in *Tj* is transparently added to objects, the application functions with no changes in the distributed environment. Facts can be freely distributed. Since rule antecedents are tested by executing methods on the facts being tested, rule satisfaction is partially distributed.

Unfortunately, most interesting facts are mutable objects and so are passed by reference. As such, there can be severe performance costs associated with this kind of distributed implementation. In particular, the pattern matcher sends many `class` messages to fact objects in the initial stage of

pattern matching. For remote facts this access is a costly remote message send.

Using a variation of the RemoteReference scheme discussed above, we cache the class of an object with its remote reference. Specifying this type of cached class reference as the marshaling default in a newly asserted fact's Marshaling component is a simple change. It eliminates all of these class messages but at the cost of 4 bytes per remote reference object (see EEShell≫addFactObject: in Section B.2.1). Though this change is technically optional, we view it as necessary for performance reasons.

The next step is to add distribution to the expert system's inference engine. Doing this effectively decouples the nodes of the pattern matching network from each other and the inference engine. This increases the available concurrency and enables behaviours such as migration.

The addition of distribution to the algorithm implicitly adds parallelism and requires that inference engine elements be protected from concurrent access. 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 (e.g., anObject meta installModel: CO for: anObject (see Section 3.5.1). Variations on this allow the specification of the object's protected (execution controlled) methods and default activity (see CodAActivityBlockFor: and CodApublicMethods in Section B.2.1).

Once the application elements are decoupled and protected, they can be freely distributed. But this is not the whole story. Distribution via remote referencing, as we have seen, can be ineffective. Replication and/or caching can be used to address some but not all cases of distributed mutable objects. In the expert system case, some nodes have memories or tests which involve local objects. Replication may be effective for accesses but at the cost of increased update costs. Selective replication of objects and object slots allows static (immutable) and slow changing objects parts to be duplicated on remote processors while keeping the large, frequently changing parts local. The declarative nature of *Tj* allows these factors to be changed easily and without affecting the base-level semantics.

7.2.3 Changes to original code

This example also demonstrates the usefulness of the CodA/*Tj* architecture in generic applications. The expert system used here is a commercially available product, **ENVY/***Expert*, implemented in Smalltalk. It was written strictly as a uniprocessor, sequential package taking no account of CodA or *Tj*, or the demands of distribution and concurrency. It consists of approximately 100 classes and 1500 methods in total. Of these, some 80 classes and 1000 methods are directly related to the execution and support of inferencing.

We approached the modification with a *black box* mentality and an eye to minimizing changes to the original code. The changes and additions made in the creation of the concurrent and distributed version are summarized in Table 7.2. As we can see from the table, the only changes were annotations and additions. Appendix B.2 shows most of the modified and added methods required for this application. The added methods generally relate to initialization and setup, and the establishment of proxies for remote objects.

The bulk of the annotations set meta-level structures used automatically by the CodA archi-

tecture. For example, the CodApublicMethods methods which declare a class' public interface and the TjmigrationDescriptorFor: which declares the shape of migrated objects (i.e., how they are migrated). Note that this shape is different from that used when objects are passed (i.e., marshaled). Although these changes are implemented as additional methods, they are not part of the normal call chain and are counted as annotations to the classes themselves.

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

Table 7.2: Summary of expert system changes and additions

7.2.4 Experiments

Our main goal in implementing this application was to investigate the software engineering aspects of CodA and *Tj* when applied to a real-world system. This was detailed in the previous section. Having applied our techniques to the algorithm itself, we also experimented with their application to several representative knowledge bases. In particular, the Waltz line labeling system [6].

Waltz is a system for aiding in the 3-dimensional interpretation of 2-dimensional line drawings. It contains 33 rules and operates over 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 junction objects representing the convergence of two to three lines. The knowledge for how these joints are constructed is embedded in the expert system's rules. 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 rule-based production systems [6].

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 fact base over the machine topology. The inference engine however is still centralized. Since it runs all the rule conclusions, which in turn create new facts, and new objects are by default created local to their creator, the distribution is less than even.

To smooth out the distribution of facts, we annotate the rules which create them. The annotations explicitly specify the location of new facts. 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 7.4 show an example of such a modification.

In the figure, the modified statements are indicated by a ->. The statement WaltzEdge in: (aLine meta state spaceFor: aLine) creates a new WaltzEdge object in the same space as aLine and so maintains locality with the fact it represents.

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

Figure 7.4: Example modified rule

We can also distribute the rule base's representation and execution. Rulebases are represented by Rete pattern matching networks. Networks for a particular rule base have a number of properties which can influence their distribution. For example, paths through the graph represent paths through individual rule antecedents. One rule may have many paths to satisfaction (e.g., if there are any OR operations). If the corresponding Rete nodes are clustered on the same processor, techniques such as caching are more effective. Depending on the rulebase itself, there may also be distinct subgraphs of related rule antecedents which should be clustered or distributed. Particular kinds (e.g., classes) of facts may only be relevant to a certain subset of rules encouraging those facts to migrate and new facts of those types to be created near the rules which will handle them.

7.2.5 Summary

This application is interesting because the original base-level program is large and not specifically designed to run in this environment (CodA and/or *Tj*). 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. This example also highlights the contagious nature of distribution. By

modifying only a few 'root' objects in the system (e.g., facts, nodes, inference engine), many other parts are distributed.

How the distributed version of an expert system is run depends on the situation. **ENVY/***Expert* is intended to be fully integrated with some other application. That is, it is not a stand-alone expert system shell. The fact objects over which it reasons are expected to be regular objects from some application domain (e.g., a banking application). As such, fact distribution depends largely on the application itself.

These strategies and others are made possible by using a uniform, transparent, non-invasive mechanism for introducing distribution and concurrency. It is outside the scope of our work to investigate these further but the effects of having a meta-level architecture such as CodA are clear. Sophisticated notions of distribution and concurrency can be added to 'unsuspecting' applications with relatively minor changes. The architecture is so completely integrated in to its implementation environment that it can be applied to tertiary systems which implement entirely different base-level computational models (e.g., rule-based systems).

7.3 Vibes

7.3.1 Problem description

Though object-oriented programming is much vaunted for its intuitiveness and similarly to the real world, many implementation behaviours bear little or no resemblance to any naturally occurring phenomena. Consequently, we have little intuition of how to observe and analyze them. Current monitoring and analysis tools (e.g,. Pablo [36] and Parasight [2]) help to a certain degree but are not completely applicable to object-oriented systems. They assume and require system models where code is fixed, types exist and are known, variables and storage are predeclared and system behaviour is *regular*. They cannot handle polymorphism, late binding or sophisticated symbolic processing in their analyses.

Further, their facilities are oriented towards measures of behaviour (e.g., numbers of messages sent, timestamps) rather than the behaviours themselves. In large complex object-oriented systems, very few of these premises and properties hold and the disparity only increases with the addition of diverse computational behaviours such as concurrency and distribution.

Since our work with CodA specifically targets these irregular, object-oriented systems with many different behaviours, this is a significant problem. If we want to provide a complete environment, we must supply tools for monitoring and analyzing object behaviour.

We propose a monitoring and analysis system called Vibes. Vibes draws on the good points of existing systems and adds support for object-oriented system analysis. In looking at available analysis systems from different domains we found that Pablo and scientific data visualizers like Explorer [39] and AVS [41] all use a *dataflow* model.

An analysis system consists of a series of interconnected nodes which describe data transformations. Observed data are input at various points in the graph and processed by the functions at the graph nodes. Each function performs some transformation or analysis of the data and passes it on to the next node for further processing.

The dataflow model is nice because it is clean and flexible. Graphs can be arbitrarily large and complex and the basic architecture supports extension. The paradigm is intuitive to users and easy to describe. For these reasons, we use the dataflow model in Vibes.

7.3.2 Analysis framework

An overview of the Vibes architecture is shown in Figure 7.5. The upper shaded area is the runtime instrumentation and data gathering. The irregular shaped objects are application domain entities while the uniform ovals are Vibes probes (see Section 7.3.3 for details) and analysis nodes. The unshaded (lower) area of the diagram depicts the off-line analysis portion of Vibes.

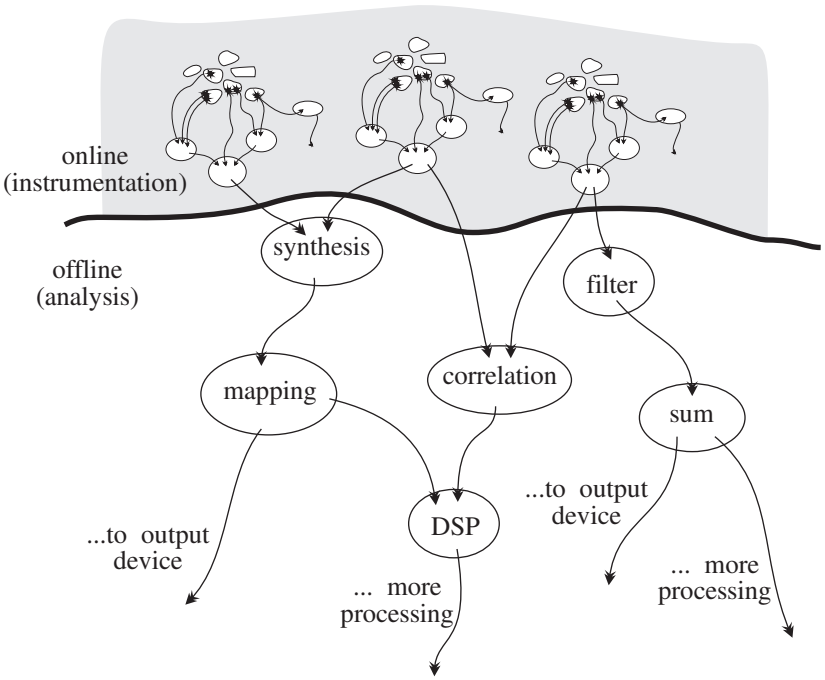


Figure 7.5: Overview of Vibes

From an implementation point of view, the off- and on-line analysis systems are the same. The difference is when and how they are used. The purpose of on-line analysis is data reduction. It is generally infeasible to collect all data from all entities in the system. By doing a certain amount of prefiltering and analysis we can eliminate extraneous data and compact or abstract other data.

Vibes improves on current analysis systems in two ways: by allowing any object to be processed and by allowing any object to be used as a processing node. The former is a consequence of using a pure object-oriented environment (e.g., Smalltalk) for the implementation of Vibes. The latter requires the power of CodA.

In current systems, processing nodes must be created explicitly as processing nodes. That is, existing class libraries cannot be directly reused in the building of analysis networks. Users wanting to extend the set of analysis functions have to, at best, program a dataflow wrapper for existing objects or, at worst, reprogram the objects in the analysis system's framework.

Users of Vibes, on the other hand, can directly use existing objects in the dataflow paradigm by modifying their meta-level. Readers of Chapter 4 on the PortedObject model will already have an idea of how we do this. In that chapter we gave an example of adding porting to a signal processing object to allow it to behave as a correlation PortedObject. All that was required was the identification of the ports and a specification of the coordination conditions and execution definition.

The power of this model is made evident by considering the construction of simple filters. A filter is a node which tests each incoming data value against some predicate. If the predicate succeeds then the object is passed to the next node in the graph. Otherwise it is rejected. Suppose for example that we are interested in all data objects whose foo attribute contains a magnitude greater than 20.

In Smalltalk, such a predicate would be represented by a Block containing the test code. Blocks are encapsulations of computation and environment which can be evaluated (i.e., run) at any time. Below is the Block representation of this predicate for use as a Vibes processing node. Note that when the predicate is run, anObject is the filter node itself.

```
aBlock := [:anObject |
  (anObject input foo > 20) ifTrue: [anObject result: anObject input]]
```

To install this block as a node in a Vibes analysis graph we add porting (as shown in the DSP example from Chapter 4) and connect its ports to other objects in the graph. The DSP example used declarative annotations to the object (i.e., methods on its class), to introduce porting. We can accomplish the same thing via parameters to the porting operation. The code below demonstrates how this is done. Note that aBlock is the predicate block described above.

```
po := PortedObject inputs: #(input) outputs: #(result) evaluator: aBlock.
anObject meta installModel: po for: anObject
```

Once the filter is ported, it is connected to the other nodes in the analysis graph using the technique described in Section 4.2.

In a non-Vibes system we do the same thing by either building a specific filter object for each kind of filter we want or by creating a generic filter object into which different predicates can be plugged. Clearly the latter option is best but it still requires a priori knowledge of the filter concept. That is, if the analysis system does not have a generic filter node then the user has to program one. Using CodA and Vibes, any object can be put into the analysis graph with little or no programming. Vibes is more extendible.

In addition to the PortedObject model's extensibility, we can use its capacity for creating compounds as a way of abstracting analysis subsystems. Rather than forcing users to know all the details of the potentially complex analysis techniques, we can group analysis nodes into modules with well-defined input and output ports. This gives us a mechanism for reuse and sharing of analysis techniques.

7.3.3 Monitoring

Vibes is designed to process behavioural data gathered from running systems. To collect that data we supply an instrumentation framework. Fundamentally this framework is the same as that used

for analysis. Monitoring graphs are constructed of PortedObjects which gather, massage and store data. The difference is that the monitoring system also contains instruments or *probes*. Probes are objects which are coupled directly to the objects being monitored and supply a feed of low-level physical events to the monitoring system.

These events are either used directly or synthesized into logical events by monitoring nodes. For example, to produce a logical duration event we must hook start and end events and count the time units which pass between them. Probes supply the events to monitors which track the time and construct the logical event.

The key goals in any monitoring system are to gather as much data as possible with as little disruption as possible. CodA's reification of meta-level behaviours into fine-grained objects supports both of these. All facets of object execution are reified as objects at the meta-level. The obvious approach, and that used by most examples of meta-level monitoring facilities, is to simply replace existing meta-components with ones which describe some sort of 'monitored' behaviour.

This is certainly feasible and possible in CodA but there are problems. Most significantly, this technique does not scale to handle the monitoring of behaviours in many different object models. The monitored version of a meta-component must do two things; monitor the component *and* emulate the behaviour description found in the original component. That is, we must duplicate the component's functionality. In Vibes we take the meta-meta-level approach.

Consider the monitoring of an object's message sending activity as shown in Figure 7.6. The object *A* is sending the message *M* to some other object. To do this, *A*'s Send's send:for: interface method is invoked. This is true for all Send operations regardless of their actual definition — The operational hooks are generic. The Send's Execution (i.e., the meta-meta-level) simply has to watch the invocations send:for: and perform the appropriate monitoring operations.

The benefits of this technique are that the monitoring mechanism is generic, it can be applied to any component regardless of the behaviour is defines, and that it is completely non-intrusive on the base- and meta-levels. This is not completely feasible in other systems as their behaviours are complex and not fully reified as objects. They do not present sufficient surface hooks.

With CodA however, meta-components are simple and fine-grained, and they reify the base-level object's operations which is typically the target of monitoring. So, there is generally no need to look inside the component, we can simply watch from its meta-level.

There may still be cases in which meta-component interface watching may not be sufficient. If so, whole meta-components must be replaced with monitored versions as described above. Again though, CodA's fine granularity decreases the impact and cost of doing this as only the operations being monitored are affected.

7.3.4 Summary

The Vibes application is an example of a system specifically designed for and with CodA. In contrast to the other examples which use CodA to add unanticipated behaviours 'after the fact', the basic Vibes architecture relies on an object model created in response to its requirements, the PortedObject model. Though we did not present it here, we also created concurrent and distributed versions of Vibes by combining the PortedObject model with the ConcurrentObject and ConcurrentObject models. This example demonstrates the usefulness of CodA both in the design of applications and

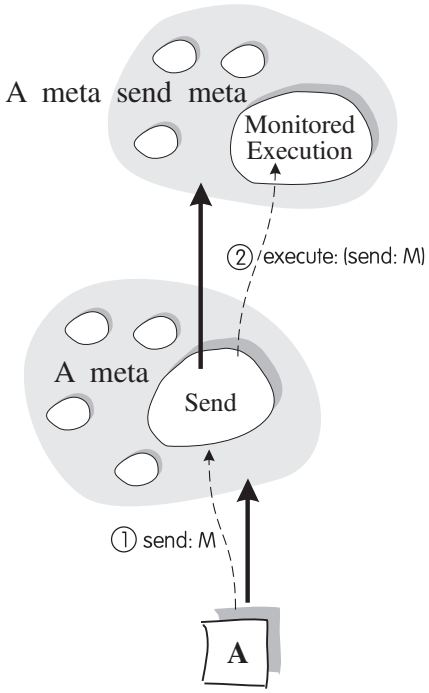


Figure 7.6: Monitoring Send activity

the prototyping of their behaviour.

Further, Vibes is a real and useful application. We use it to analyze the object models and applications we create using CodA and *Tj*. The performance data discussed in Section 7.1.4 was gathered and analyzed using Vibes. Though not explicitly discussed, it was also used while developing the distributed expert system tool in Section 7.2.

We have also done a number of ad hoc experiments with novel analysis techniques such as signal processing and using expert systems for abstracting gathered data. These are discussed in [26].

Chapter 8

Evaluation

Direct comparison between existing meta-level architectures is a tenuous proposition at best. Existing architectures typically focus on different aspects of object behaviour and so naturally have different capabilities. By abstracting out notions of capability we can at least try to position architectures relative to one another. Users can then match the capabilities of an architecture to their requirements. We also evaluate our system in terms of end-user performance in an effort to show that the system is usable as an environment for prototyping and experimentation.

8.1 Capability

Based on the goals outlined in Chapter 1, we identify a number of properties which we use to informally relate systems to each other. We show that CodA provides a reasonable level of support for these properties and so satisfies our overall goals for meta-level architecture design. Since most systems support a reasonable separation between the base- and meta-levels, we do not consider this aspect in the comparison. Support for the other aspects (i.e., expressiveness, extensibility and programmability) however varies widely.

The properties used in the analysis are enumerated and described below, and related to existing architectures in Table 8.1. There is one row in the table for each system and one column for each property. The meanings of the table's symbols are given in Table 8.2. We have not listed properties which we have found to be present or absent from all systems.

1. Is the programmer's interface clearly and well-defined? That is, does the architecture specifically set out a programmer's interface or is it left as an implementation detail? Such an interface is both in terms of objects and methods. Without a concrete structure, meta-level programmers are left on their own both to figure out how the system works and how to write their code so that it can be integrated with that of others. Of primary importance here is the granularity of the interface. Simply having a few high-level entry points may support meta-level use but does not support meta-level change. A meta-level object, like all objects, should present a rich, multi-level interface protocol.

2. Are the architecture's behaviour abstractions high-level? For the concepts that can be expressed, do the abstractions presented closely match the fundamental concepts of the system

Architecture	1	2	3	4	5	6	7	8	9
CodA	•	○	•	•	•	○	•	•	○
CLOS MOP	•	•	×	–	×	–	○	•	×
ABCL/R2	×	×	○	–	○	×	×	×	○
AL-1/D	•	○	•	×	○	×	○	×	•
RbCl	○	○	•	×	×	–	•	•	•
Apertos	•	○	○	○	•	○	•	•	•

Table 8.1: The relationship between meta-level architectures.

Symbol	Meaning
•	Fully and explicitly supported.
○	Partially (in scope or depth) supported.
×	Not explicitly supported.
–	Not applicable. Depends on unsupported property.

Table 8.2: Levels of support.

(e.g., message passing)? How many operations must a meta-level programmer invoke to effect some behaviour. Systems with good abstraction require relatively few since they have abstracted the details into a higher-level interface.

3. Does the architecture facilitate the description of object behaviour from more than one computational domain (e.g., sequential, concurrent, multi-threaded or distributed objects) or are all objects basically the same. This relates to the independence of the architecture from the object/language models it supports. Is the architecture a mechanism for reifying a particular language or computational structure, or is it more general. An open architecture will allow anything to be implemented but having a strong and clear separation between the architecture and the object models enables both the implementation of radically different computation styles and the inter-operation of objects using these styles.

4. Does the architecture have specific support for extension? The simple ability to add to the architecture is insufficient here. Without a framework for tracking and managing this change, user's will develop their own ad hoc styles which will clash when combined. Frameworks can range from simple structuring conventions to explicit mechanical support.

5. Does the architecture support the encapsulation of object models? This is closely related to the idea of abstraction but for structuring rather than interfaces and execution. Simple models are easily described by changes to, or the addition of, a few methods or objects in a restricted area of the meta-level. More complex behaviours affect many methods and objects

from various parts of the meta-level. We must be able to manipulate entire behaviours as a small number of atomic concepts regardless of their complexity.

6. Does the architecture support object model combination and configuration? Architectures with frameworks and encapsulation implicitly support model combination. However, factors relating to granularity and complexity also come into play when configuring meta-levels. Combination and configuration support enables the merging of two or more models whose changes and additions range over diverse parts of the meta-level. This is essential if the meta-level architecture is to scale to support many different models.

7. Is object behaviour reified to a reasonably fine granularity? All object-oriented meta-level architectures reify behaviour as objects at the meta-level. The important questions are; how many and how were they derived? The granularity of the decomposition has a profound effect on the nature of the use of the meta-level. Finer objects allow more flexibility but require more maintenance. The objects in a coarser decomposition are more easily used (i.e., they have a higher-level interface) but the behaviour they describe is more difficult to reuse.

8. Does the architecture support the reuse of behaviour descriptions (e.g., meta-components)? If an architecture has a sound framework including model encapsulation then it has implicit support for reuse. This is enhanced by a consistent and fine-grained decomposition of the meta-level as well as any explicit reuse mechanisms which may be provided. Decomposition into methods does not form a sound basis for reuse as individual methods do not generally support reuse.

9. Does the architecture reify low-level system related behaviours such as garbage collection, process scheduling and object location? Some of the systems considered have no relation to such operating system level concepts while others deal mainly with these issues.

The following sections highlight particular points of the considered systems with respect to these properties and form an explanation of the values in Table 8.1.

8.1.1 CodA

2 Since the basic architecture is independent of a particular language, the abstractions provided are not as high-level as those found in more language-specific architectures like CLOS. For example, the architecture has no explicit notion of class. However, implementations may 'borrow' concepts like classes from their host language. In the Smalltalk implementation, normal classes can be extended to define additional or redefine existing meta-components for its instances.

6 Though CodA's general architecture of fine-grained objects encourages and supports combination and reuse, the object model's configuration management mechanisms (see Section 3.5.1) are not sufficiently powerful to handle complex situations.

9 As shown with the ConcurrentObject and DistributedObject models, CodA reifies significant portions of an object's low-level behaviour. It does not however contain an explicit model of the underlying system (e.g., the memory or execution systems). These are left to the implementation platform.

8.1.2 CLOS MOP

The CLOS MOP's domain is somewhat different from that of CodA in that its aim is to unify various CLOS object models. This is accomplished by reifying at the meta-level, those components which describe differing behaviours. The reified behaviours tend to be more *structural* than operational as in CodA. As a side-effect of the CLOS object model, the definition of behaviour on a per-object basis does not fit naturally into the CLOS MOP design. The CLOS meta-level architecture is neither general purpose nor particularly extensible but it is quite powerful within its intended domain.

3, 4 The CLOS MOP's basic execution model is that of CLOS (i.e., method-oriented). It does not support things like object-specific method invocation reification in a scalable way. The base-level system (e.g., CLOS) does not generally support the addition of concurrency or distribution so these concepts cannot be introduced at the meta-level.

5, 6 The MOP is a fragmented reification of behaviour with no mechanism for grouping behaviours to form coherent wholes. CLOS classes are an insufficient mechanism for this as they are not independent of the base-level.

7 CLOS MOP metaobjects reify, in objects, the structural nature of CLOS (e.g., classes, methods, slots, etc.) rather than its operational aspects (e.g., sends, lookups). Operational behaviour is described in methods on the structural metaobjects. The meta-level is medium-grained.

9 The domain of the CLOS MOP is the CLOS language, not its implementation. As a result, very few of the underlying details of real CLOS systems (e.g., garbage collectors) are reified in the MOP.

8.1.3 ABCL/R2

The ABCL/R2 meta-level architecture is one of the few which deals with concurrency. It is quite open but does not explicitly abstract many object behaviours. Since the ABCL language is quite simple, this is not a too much of a problem. While it is in theory extendible, the architecture has no framework or infrastructure for extension. Creating new behaviour for an object implies writing code rather than changing parameters or plugging-in components as seen in other architectures. There are no facilities for configuring or structuring the meta-level itself.

1, 2 The ABCL/R2 meta-level is basically an interpreter whose design is left up to the implementor. The architecture itself says little about its structure or the abstractions it contains.

3, 4 ABCL/R2 objects are generally single threaded and concurrent. For implementation efficiency the system also provides *lightweight objects* which are essentially passive and stateless. There are no other facilities for extending this set of computational domains.

5, 6 Since the entire meta-level of an ABCL/R2 object is captured in its metaobject, the metaobject description implicitly encapsulates an object model. Metaobjects and thus object models are effectively interpreters, implemented as code and so do not readily support combination and configuration.

7, 8 The ABCL/R2 meta-level is made up of only a few objects. The internal structure of these objects is not explicitly set out by the architecture. Without this, the architecture cannot properly support scalability or reuse.

9 The group model of ABCL/R2 allows the description of some low-level execution behaviours such as scheduling.

8.1.4 AL-1/D

AL-1/D is one of the few architectures which deals with distribution. It focuses on a set of meta-level concepts directly related to distribution requirements. Parts of the AL-1/D system can be mapped onto those of CodA and *Tj* as shown in Table 8.3.

AL-1/D	**CodA/*Tj***
Operation	Standard CodA meta-components
Migration	*Tj* Marshaling and Migration
System	Smalltalk
DE+Resource	*Tj* infrastructure objects (topology, space, ...)
Statistics	*Tj* monitoring (largely meta-meta-level)

Table 8.3: Mapping from AL-1/D to CodA.

As noted in Section 2.6 and show in Table 8.3, the meta-level is somewhat unbalanced. The Operation model is equivalent to most of the basic CodA meta-components but the architecture contains no facilities for intra-model structuring — The behaviour defined and structured in CodA is unstructured in AL-1/D. Furthermore, the AL-1/D meta-model scheme does not extend to support the grouping of arbitrary metaobjects into manipulable wholes (i.e., object models).

3, 4 The portion of the meta-level which deals with object execution (i.e., the Operation model) is explicitly partitioned in AL-1/D. New computational domains are implemented by providing new Operation models. The models themselves however, contain no infrastructure to support this modification.

5, 6 The Operation model embodies entire and complete object model descriptions. The architecture does not support sub-models, model intersection, combination or other manipulation. Nor does it support the grouping of specific meta-models to form a coherent whole.

7, 8 The AL-1/D meta-level is decomposed into objects but the architecture says nothing in particular about their granularity or relation to one another. Reuse is provided via the meta-model concept but the models provided are of varying granularity and complexity.

8.1.5 RbCl

RbCl is another architecture with support for concurrent objects. The RbCl approach differs from ABCL/R2 and is similar to CodA in that it does not use the interpreter model of the meta-level. Rather, the meta-level is looked upon as a collection of objects which provide services to the base-level objects. Object behaviour is modified by replacing the objects at the meta-level on an individual basis. Since the meta-level objects are also objects having meta-levels, this can be seen as pushing the underlying machine further away from the base-level objects only in the areas of interest (i.e., those reified). CodA and RbCl differ in the level of infrastructure they support and their approach to meta-level decomposition.

1, 2 The programmer's interface for RbCl is well-defined in the sense that there are definitions for many (most) object behaviours. It is not well-defined in the sense that it is not clearly set out in the literature. Overall many of the interfaces are at quite a low-level and without higher-level abstractions to ease their use.

3, 4 The architecture is a reification of a specific object model, namely single threaded concurrent objects. Since the reification is at such a low level, there is some support for the implementation of other computational domains. However, this support is not explicit.

5, 6 There is no notion of overall object model encapsulation or behaviour grouping.

8.1.6 Apertos

Even though the domains of Apertos and CodA differ significantly, the basic architectures have quite a bit in common. Both reify the meta-level as objects and provide mechanisms for structuring and grouping these objects (i.e., Apertos metaspaces and CodA object models). It is difficult to directly compare the nature of their decompositions due to domain differences.

1, 2 Though not explicitly set out in the current literature, Apertos' programmer's interface is reasonably well-defined. This is a consequence of its role as an operating system which is used by various client programs. Though its reified structures closely match those of its domain, they are not particularly high-level or abstract.

3, 4 Apertos is an object-oriented operating system, not an object operating system. As such, it is not particularly concerned with the description of object behaviours at a level higher than its execution and resource requirements. It does not define how *object* behaviour is described or structured.

5, 6 The Apertos architecture supports a hierarchy of *metaspaces* which are used to encapsulate groups of meta-level objects. Metaspaces are used only as structuring concepts and do not

play any role at runtime. As such, they are somewhat limited in their support for behaviour combination and configuration.

7, 8 Though Apertos' domain (e.g, operating systems) is different from most of the other systems, its architecture does present a reasonably fine-grained reification of the behaviours it supports. The nature of the decomposition and meta-level framework explicitly supports reuse and combination.

9 Low-level execution details are the domain of Apertos and so are quite well treated.

8.2 Performance

Our goals for performance fall into two categories; execution and design. On the one hand, to have an effective system, one which people can actually use, we must have reasonable execution performance. Applications which utilize CodA should not suddenly take several orders of magnitude longer to run. In addition, the amount of overhead introduced should correspond to the meta-level reification and modification done. That is, users should only pay for what they use.

On the other hand, usability or *design performance* is related to how easy the system is to program, extend and apply. To improve performance in this area the system must support a wide range of behaviour descriptions and be extensible so as to facilitate completely new behaviours. It should also use and support typical software engineering practices such as encapsulation and reuse. Furthermore, when changes to the meta-level are applied, they must not require extensive changes to the base-level code.

And so a tension arises — A system which is very fast but horribly difficult to use is not very effective. The converse is also true. Beyond a certain point however, execution performance is largely an engineering issue. The architectural design issues fall away leaving just implementation and optimization details. Because of its fine granularity, CodA's design inherently facilitates incremental, deep and narrow implementation optimization. Far-reaching changes can be made in very precise regions of the system without affecting other, unrelated areas (see Section 6.5.2). Beyond this, the main focus of our work has been on design performance.

Design performance is difficult to show in quantitative terms. Chapter 7 is mostly concerned with demonstrating CodA's design performance by way of real-world examples. The most convincing evidence of CodA's design performance is its integration with the underlying implementation environment and the degree to which meta-level changes are isolated from the base-level code. We have shown that real applications can be quite radically altered with negligible changes to the original code. In some cases the changes required have no impact on the application semantics of the objects (see Section 7.2). This is very high performance from the design and use point of view.

In addition, we have designed numerous object models describing widely differing behaviours and have implemented these within the same architecture. Since the behaviours are encapsulated in concrete objects, they can be directly applied and reused in many different situations. We have also shown how the meta-level is completely extensible in that it facilitates the creation and attachment of completely new behaviours to objects. In addition, all of these behaviours and models can be freely combined.

8.2.1 Execution performance

CodA is intended as a platform for experimentation with object behaviour and object model design, and application implementation. Its main mode of use is that developers prototype their applications and object behaviours until they find those which meet their requirements. Then, if execution performance is found lacking, the implementation of individual or groups of objects and meta-components is optimized. In general, we have found that it is not appropriate to overly optimize the architecture itself as this would limit our expressiveness and ability to reuse prior work.

We must however, recognize the need for eliminating the excessive costs of using a meta-level architecture in the first instance (i.e., the prototyping phase). CodA approaches this in a number of ways. First, the overall architecture is one of individual behaviour replacement. Meta-level users only pay for that which they use. All non-reified and unmodified meta-level operations are implemented by the underlying language environment's native mechanisms.

Because of this, even without explicit optimizations, CodA and *Tj* perform well within acceptable bounds for experimental purposes. The worst case for *full* reification of a complete message send/receive cycle in the Smalltalk implementation is about an order of magnitude increase in runtime. While this may seem high at first, readers are reminded that not all message exchanges in a system need be reified and even those which are, need not be fully reified. In CodA you only pay for what you reify. As a result, the performance overhead seen in actual applications will vary substantially from this.

In real applications, while the actual performance ratios depend on the amount and type of behaviour reification done, we have found that the general trend holds across a number of applications. Table 8.4 gives the performance of applications done with CodA and *Tj* normalized to that of the pure Smalltalk version.

Application	**Smalltalk**	**CodA**	*Tj*
2D N-Body	1 (1.5s)	4	25
Waltz 1	1 (9.7s)	3	20
Waltz 2	1 (9.7s)	5	42

Table 8.4: CodA/*Tj* performance

The **Smalltalk** column represents the fastest, optimized implementation of the problem directly in Smalltalk. This column is the basis for the comparison and contains the normalized speed value (e.g., 1) and the actual runtime in seconds. The performance data in the other columns is relative to this plain Smalltalk performance. The **CodA** column is the same as the Smalltalk implementation but with certain objects fully reified and perhaps with new behaviours such as concurrency added. The difference between these two columns largely represents the overhead of introducing explicit meta-level components to the system. In cases where concurrency has been added there are also process switching and scheduling costs which are unrelated to the cost of using CodA's meta-level architecture. The *Tj* column depicts the performance of the same application and code but with certain objects physically distributed over a multiprocessor topology. Which objects are distributed

and how they are distributed depends on the problem itself.

We have done a number of these experiments with different applications and different configurations. It would be redundant and confusing to show them all, so we have selected these examples which are representative of the applications in general.

The N-Body (see Section 7.1) experiments were done with 1024 particles. The CodA runs were done with concurrent particles and quadtree nodes. *Tj* runs had particles randomly distributed randomly over 64 processors of a Fujitsu AP1000. Quadtree nodes were created on the processor of the first particle they contained as the tree was built.

Both Waltz examples (see Section 7.2) were runs of the standard Waltz line labeling rule-base on the same sets of lines. For Waltz 1, the CodA runs had fully reified facts. In the *Tj* runs the facts were both reified and distributed over 64 nodes of a Fujitsu AP1000. Waltz 2 follows suit but reifies and distributes the actual implementation of the expert system (i.e., the Rete pattern matching network).

Note that in the case of systems using *Tj*, by far the biggest factor affecting speed is inter-node message passing time. In the current implementation this is dominated by the highly sophisticated but somewhat costly object marshaling mechanism described in Section 5.2.2. For example, fully general message marshaling of in the N-Body application takes an average of 3ms per message with an average message length of 82 bytes.

We have implemented more efficient but less capable marshaling mechanisms, but in the end found that the difference in expressive power is well worth the execution overhead. This may not be true in all cases and users are free to substitute highly optimized marshaling components on a use-, object-, class- or system-wide basis. The values given in Table 8.4 are derived from experiments using the fully general marshaling mechanism.

The messaging times are also affected by the implementation problems pointed out in Section 5.5. Namely that there is some amount of unnecessary copying due to library layering and that message responsiveness is hampered by the lack of interrupts. It is expected that future versions of the AP1000 OS will eliminate some of these problems and that that platform's implementation will see commensurate performance increases.

8.2.2 Performance perspective

To put CodA and *Tj*'s performance in perspective with other systems we looked at RbCl and AL-1/D, two systems which were shown to be similar in domain and capability in the preceding section. In particular, we look at their speed with respect to distributed object operations to give a sense of performance in distributed end-user applications.

In [18] the runtime overhead introduced by modifying two system behaviours with RbCl code is discussed. It is shown that while the normal system takes 9.8ms to run a particular test, the modified version takes 600ms. A factor of 61 slower. While the nature of the changes is not given in detail, this is significantly more than we have seen in any of our experiments with remote messaging reification. Interestingly however, RbCl allows the user to replace the RbCl-based meta-level modifications with C++ based code and achieve a speed-up over the original unmodified meta-level. This same capability is available in *Tj* through specialization of an object's Marshaling component or the specification of optimized marshaling descriptors.

AL-1/D's remote messaging is quite fast. It introduces just 30% overhead to standard Unix stream data transfer. CodA and *Tj* implemented in Smalltalk cannot match this performance because they use a very general object marshaling scheme which trades runtime efficiency for flexibility. As outlined above, programmers can easily modify the marshaling technique used for the system, instances of a class, a particular instance or a particular use of an instance to regain the efficiency but at the cost of flexibility. To date, our experiments with real applications have not indicated that this is necessary.

8.3 Summary

In this chapter we show that CodA satisfies our goals for meta-level architecture design by rating its support for various concrete properties of meta-level design. We also use this set of properties as a basis for a comparison between CodA and existing architectures. While deriving absolute notions of 'better' or 'worse' is difficult in this situation, CodA is shown to be broader based than existing systems. This is due in part to the relatively narrow domain defined for many architectures.

We also show that CodA compares favorably to existing systems in terms of performance and that, more importantly, the performance realized by end-users (e.g., time to run a program) is reasonable. That is, the system as implemented in Smalltalk is usable for creating, testing and applying new object behaviours.

We also note that runtime performance is not the only measure of usability. CodA is completely integrated with the Smalltalk programming environment and so can draw on its powerful programming tools in support of the programmer's efforts. Other systems, while integrated with their implementation environment, are implemented in systems with relatively poor programmer and software engineering support.

CodA has been shown to provide a rich meta-level architecture which is as capable as existing systems when compared on specific points. This level of support is provided across a wide range of capabilities. It was also found to be usable both in run-time and program-time performance.

Chapter 9

Conclusions

We have identified an important problem in software engineering and complex system development. Namely that an object's base-level semantics and its computational behaviour are not well separated by current technology and techniques. This prevents users from building objects which have widely varying computational behaviour. It also prevents users of object (class) libraries from (re)using the libraries in paradigms for which they were not designed. These are severe restrictions in the current environment of complex, widely scoped systems.

We have traced the problem back to the implicit inclusion of base-level language concepts and constructs in the design of typical meta-level architecture. These architectures, while *open* within their domain, are not open to large-scale deviations from their original behaviour. There is no framework or infrastructure for supporting the addition of completely new concepts in an integrated way.

To address this we developed CodA, a meta-level architecture which is free of these language-based constructs and assumptions. CodA is based on an *operational* decomposition of object behaviour. This approach reifies into objects, the operations required for basic object execution. As a result, it is necessarily fine-grained and independent of base-level language semantics.

The decomposition is set in a generic framework which supports the composition and combination of the resultant components. Meta-levels, and thus object behaviour specifications, are constructed by composing definitions of the various operations (e.g., message sending and method lookup). The framework includes a powerful grouping and abstraction mechanism, *object models*, which are used to construct representations of higher-level object behaviours such as concurrency, distribution and classes.

We demonstrate that our approach is powerful in several ways. The design is implemented in, and completely integrated with, an industrial-grade software development environment, Smalltalk. Our ability to use the CodA features transparently and ubiquitously in the host environment is a strong indication of its independence from base-level semantics.

Using this implementation we designed and built several non-trivial models of object behaviour which vary widely in their computational domains. Models such as DistributedObjects and PortedObjects represent substantial deviations from normal object behaviour yet they are implemented in the same framework, can co-exist and can even be combined and applied to the same object. This is done with very little impact on the object's base-level code.

These models, and thus the framework as a whole, are shown to be useful in real-world situations by applying them to the implementation of several applications. Each application demonstrates a different aspect of CodA's features and capabilities. In particular, the N-Body solver system, highlights CodA as a good environment for algorithm and application prototyping. We show that new behaviours (e.g., distribution) are easily added to applications and demonstrate how these are monitored and varied to get different computational effects (e.g., remove bottlenecks).

Another very interesting result was obtained by applying CodA and some of its object models to a real, non-trivial, third-party, commercial application; an expert system tool called **ENVY/***Expert*. We showed that we were able to add significant concurrency and distribution throughout **ENVY/***Expert*'s implementation and further, that our techniques could be applied even within the language system (rule-based productions) defined by **ENVY/***Expert*. Analysis of the effort and modifications required to effect these changes revealed that only 34 of approximately 1,000 methods needed to be added or modified. Of these, half are shown to be *annotations* which have no effect on base-level semantics while the other half are *additional* methods required only for configuration and maintenance of the new behaviours.

A further non-trivial application, Vibes, a data analysis system, shows how CodA is useful in designing and integrating applications with completely new computing demands. Rather than applying our modifications to otherwise functioning applications as in the previous examples, Vibes is designed around a new model of object-to-object interaction, PortedObjects. PortedObjects are objects which interact via data flowing over channels and ports. By meta-level manipulation we allow programmers to adapt and use normal Smalltalk objects in this new and radically different domain. This style of programming is very useful to developers as it gives them immediate access to a vast array of classes for use in their computing domain.

9.1 Perspectives and future work

The benefits we have shown are clearly derived from the exclusion of base-level language constructs from the meta-level. That is, our *operational decomposition*. This decomposition technique enables the description of behaviours which are completely foreign to the underlying object system. It also allows us to construct a generic framework for organizing and managing these behaviours. We have shown that the *object model* concept is a superset of the organizational structures commonly found in other systems (e.g., classes). Detailed evaluation of CodA relative to other systems finds it to be at least as powerful in comparisons of individual points of capability and much broader ranging in its support for the key properties of meta-level architectures.

With the operational approach in mind, we feel that good progress can be made in the area of support for the automatic combination of meta-level concepts. This is, in general, an open problem but we note that CodA has a number of inherent features which ease the combination problem. In particular, the fine-grained encapsulation of potential points of conflict. This should be extended with additional support for object model requirements identification and meta-component capability specification. Using this as a basis, we can build systems to reason about and resolve conflicts automatically. This design is compatible with, and complementary to, the current state-of-the-art composition techniques such as those found in Moostrap [30].

In addition, we are particularly interested in applying techniques related to partial evaluation and dynamic compilation to the problem of meta-level combination and optimization. We envisage a system where distinct meta-components are used for prototyping object meta-levels but once the best meta-level design for a particular situation is determined, the components are *compressed* into one (or a small number of) objects by partial evaluation and dynamic compilation. By remembering something of their original form, compressed meta-levels could be *decompressed* back into their flexible, modifiable form.

Bibliography

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

[2] Z. Aral and I. Gertner. Non-intrusive and interactive profiling in Parasight. *Proceedings of the ACM/SIGPLAN PPEALS (Parallel Programming: Experience with Applications, Languages and Systems) 1988, published in ACM SIGPLAN NOTICES*, 23(9):21–30, September 1988.

[3] H. E. Bal and M. F. Kaashoek. Object distribution in orca using compile-time and run-time techniques. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 162–177, Sept. 1993. Published as ACM SIGPLAN Notices, volume 28, number 9.

[4] J. Barnes and P. Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324:446–449, 1986.

[5] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPOPP*, pages 168–176, Mar. 1990. Published as ACM SIGPLAN Notices, volume 25, number 3.

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

[7] J.-P. 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.

[8] J.-P. Briot and P. Cointe. Programming with explicit metaclasses in Smalltalk-80. In *Proceedings of OOPSLA '89*, pages 419–431, October 1989.

[9] S. Chiba. A metaobject protocol for real programmers. In *Proceedings of OOPSLA '95*, 1995. To appear.

[10] P. Cointe. CLOS and Smalltalk: A comparison. In A. Pæpcke, editor, *Object-oriented programming: The CLOS perspectives*, pages 215–250. MIT Press, 1993.

[11] J. des Rivieres and B. C. Smith. The implementation of procedurally reflective languages. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, 1984.

[12] P. Deutsch, 1992. Private communication.

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

[14] R. J. Fowler. *Decentralized object finding using forwarding addresses*. PhD thesis, University of Washington, 1985. Also available as Dept. of Computer Science Tech Report 85-12-1.

[15] B. Garbinato, R. Guerraoui, and K. R. Mazouni. Distributed programming in GARF. In *Proceedings of the ECOOP Workshop on Object-Based Distributed Programming*, LNCS 791, pages 225–239. Springer Verlag, July 1993.

[16] Hewlett-Packard. *Distributed Smalltalk User's Guide*.

[17] W. C. Hsieh, P. Wang, and W. E. Wiehl. Computation migration: Enhancing locality for distributed-memory parallel systems. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming PPOPP*, pages 239–248, July 1993. Published as ACM SIGPLAN Notices, volume 28, number 7.

[18] Y. Ichisugi. *A reflective object-oriented concurrent language for distributed environments*. PhD thesis, Department of Information Science, University of Tokyo, 1993.

[19] Y. Ichisugi, S. Matsuoka, and A. Yonezawa. RbCl: A reflective object-oriented concurrent language without a run-time kernel. In *Proceedings of the International Workshop on Reflection and Meta-level Architectures*, pages 24–35, November 1992. Tokyo, Japan.

[20] Y. Ishikawa. Reflection factilities and realistic programming. *SIGPlan Notices*, 26(8):101–110, 1992.

[21] Y. Ishikawa and H. Okamura. A new reflective architecture: AL-1 approach. In *Proceedings of the OOPSLA '91 Workshop on Reflection and Meta-level Architectures in Object-Oriented Programming*, October 1991.

[22] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, Feb. 1988.

[23] G. Kiczales, J. des Rivieres, and D. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, Massachusetts, 1991.

[24] P. Maes. Concepts and experiments in computational reflection. In *Proceedings of OOPSLA '87*, pages 147–155, October 1987. Orlando.

[25] H. Masuhara, S. Matsuoka, T. Watanabe, and A. 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.

[26] J. McAffer. On the monitoring and analysis of object-oriented systems. Position paper to the ACM SIGPlan OOPSLA Workshop on Visualization of OO systems, Oct. 1993.

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

[28] J. McAffer and J. Duimovich. Actra - An industrial strength concurrent object-oriented programming system. *ACM SIGPLAN OOPS Messenger*, 2(2):82–85, Apr. 1989. Proceedings of the ACM SIGPlan OOPSLA Workshop on Object-Based Concurrent Programming.

[29] R. Milner. The polyadic π-calculus: A tutorial. In *Logic and Algebra of Specification*. Springer Verlag, 1992.

[30] P. Mulet, J. Malenfant, and P. Cointe. Towards a methodology for explicit composition of metaobjects. In *Proceedings of OOPSLA '95*, October 1995. To appear.

[31] Object Management Group. *Object Management Architecture Guide*, 1992. OMG TC Document Number 92.12.1 Revision 2.0.

[32] The open implementations workshop proposal and responses. Available on internet at: http://www.parc.xerox.com/OI/.

[33] H. Okamura and Y. 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.

[34] H. Okamura, Y. Ishikawa, and M. Tokoro. AL-1/D: A distributed programming system with multi-model reflection framework. In *Proceedings of the International Workshop on Reflection and Meta-level Architectures*, pages 36–47, November 1992. Tokyo, Japan.

[35] G. A. Pascoe. Encapsulators: A new software paradigm in smalltalk-80. In *Proceedings OOPSLA '86*, pages 341–346, November 1986. Published as ACM SIGPLAN Notices, volume 21, number 11.

[36] D. A. Reed. *An overview of the Pablo performance analysis environment*. Department of Computer Science, University of Illinois, 1992.

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

[38] T. Shimizu, T. Horie, and H. Ishihata. Low-latency message communication for the AP1000. In *Proceedings of ISCA*, pages 288–297, 1992.

[39] Silicon Graphics Inc. *Explorer User's Guide*, 1992.

[40] B. C. Smith. Reflection and semantics in lisp. In *Proceedings of ACM POPL '84*, pages 23–35, 1984.

[41] Stardent Computer Inc. *Application Visualization System, User's Guide*, 1989.

[42] A. S. Tanenbaum, H. E. Bal, and M. F. Kaashock. Programming a distributed system using shared objects. In *Proceedings of the International Symposium on High Performance Distributed Computing*, pages 5–12, July 1993.

[43] Y. 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.

[44] Y. Yokote, F. Teraoka, and M. 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.

[45] M. W. Young. Exporting a user interface to memory management from a communication-oriented operating system. Technical Report CMU-CS-89-202, Carnegie Mellon University, 1989.

Appendix A

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
```

Appendix B

Example code

The following sections the code for two of the applications dicussed in Chapter 7, the N-Body problem and the expert system. Rather than giving all the code, we attempt to highlight the areas where changes were required during the addition of concurrency and distribution. Obvious and accessor (get and set) methods have been elided.

B.1 N-Body

Object
  Particle ('position mass velocity acceleration')
  QuadTree ('area children data')
  Solver ('tree particles')

Figure B.1: N-Body application class hierarchy

B.1.1 Particle

Instance methods

```
accelerationOn: pos
   "Answer the acceleration caused by the receiver exerting a Newtonian force
    something at pos."
   | distance |
   distance := self position dist: pos.
   ^(self position - pos) * self mass / (distance raisedTo: 3)

calculateAccelerationIn: root
   self acceleration: (root forceOn: self position)
```

```
mergeWith: particle
    | totalMass |
    position isNil ifTrue: [
        position := particle position.
        mass := particle mass.
        velocity := particle velocity.
        ^self].
    velocity := velocity + particle velocity.
    totalMass := mass + particle mass.
    position := particle weightedPosition + self weightedPosition / totalMass.
    mass := totalMass

moveOver: dt
    "move the receiver using acceleration over the time frame denoted by dt"
    | newVelocity dt2 |
    dt2 := dt / 2.0.
    newVelocity := velocity + (acceleration * dt).
    position := position + (velocity * dt2) + (newVelocity * dt2).
    velocity := newVelocity.
    acceleration := 0 @ 0

weightedPosition
    ^position * mass
```

B.1.2 QuadTree

Class methods

```
for: object in: area
    ^self new
        area: area;
        data: object
```

Instance methods

```
add: object
    | cPos oPos |
    (object isNil
        or: [(self area containsPoint: (oPos := object position)) not])
        ifTrue: [^self].
    self isEmpty ifTrue: [^self data: object].
    self isLeaf ifTrue: [
        (cPos := self data position) = oPos ifTrue: [^self].
        (self childFor: self data at: cPos.
        self data: nil].
    (self childFor: object at: oPos
```

```
addAll: objects
    objects do: [:object | self add: object]

childAreaFor: position
    | x y a |
    a := self area.
    x := a left + a right / 2.0.
    y := a bottom + a top / 2.0.
    ^position x < x
        ifTrue: [
            position y < y
                ifTrue: [^a origin corner: x @ y]
                ifFalse: [^a origin x @ y corner: x @ a corner y]]
        ifFalse: [
            position y < y
                ifTrue: [^x @ a origin y corner: a corner x @ y]
                ifFalse: [^x @ y corner: a corner]]

childCount
    ^4

childFor: object at: position
    "Answer a child to hold object at position. If such a child does not
     exist then create one and answer the new child."
    | child index |
    child := self children at: (index := self childIndexFor: position).
    child isNil ifFalse: [^child].
    child := self class for: object in: (self childAreaFor: position).
    ^self children at: index put: child

childIndexFor: position
    | x y a |
    a := self area.
    x := a left + a right / 2.0.
    y := a bottom + a top / 2.0.
    position x < x
        ifTrue: [position y < y ifTrue: [^1] ifFalse: [^4]]
        ifFalse: [position y < y ifTrue: [^2] ifFalse: [^3]]

containsPoint: point
    ^self area containsPoint: point

do: block
    | result |
    self isEmpty ifTrue: [^self].
```

```
    self isLeaf ifTrue: [^block value: self data].
    children do: [:c | c isNil ifFalse: [result := c leafDo: block]].
    ^result

forceOn: position
    | result |
    self isEmpty ifTrue: [^0 @ 0].
    self isLeaf ifTrue: [
        ^(self containsPoint: position)
            ifTrue: [0 @ 0]
            ifFalse: [self data accelerationOn: position]].
    ((self containsPoint: position) not and: [self isFarEnough: position])
        ifTrue: [^self data accelerationOn: position].
    result := 0 @ 0.
    self children do: [:c |
        c isNil ifFalse: [result := result + (c forceOn: position)]].
    ^result

isEmpty
    ^data isNil and: [self isLeaf]

isFarEnough: position
    ^self area width < ((position dist: self data position) * 0.5)

isLeaf
    self children do: [:c | c isNil ifFalse: [^false]].
    ^true

prepare
    self isLeaf ifTrue: [^self].
    self data: (
        VirtualParticleClass new mass: 0.0 position: nil velocity: 0.0).
    self children do: [:c |
        c isNil ifFalse: [
            c prepare.
            self data mergeWith: c data]]
```

B.1.3 Solver

```
addParticle: p
    self particles add: p.
    ^self tree add: p

iterate: count
    count timesRepeat: [
        self stepOver: 0.3.
```

```
        self updateTree]

calculateAccelerations
    self particles do: [:particle |
        particle calculateAccelerationIn: self tree]

moveParticlesOver: dt
    self particles do: [:particle | particle moveOver: dt]

stepOver: dt
    self tree prepare.
    self calculateAccelerations.
    self moveParticlesOver: dt

updateTree
    self tree: (self tree class for: nil in: self tree area).
    self tree addAll: self particles
```

B.1.4 Distributed QuadTree

The following are the methods which were changed or added to introduce distribution and concurrency to the application. Each method is marked with its change classification (see Section 6.5.1) and the actual changes (if any) are noted by a -> in the left margin.

Class methods

```
"Required Annotation"
for: object in: area
->  ^self newForkConcurrent
        area: area;
        data: object

"Required Addition"
TjreplicationDescriptorFor: anObject
    "Answer a descriptor which copies and replicates the default slots."
    ^#((true rep true) (|| clearDataAndChildren) (-> add:))
```

Instance methods

```
"Required Annotation"
add: object
    | cPos oPos d |
    (object isNil or: [
            (self area containsPoint: (oPos := object position)) not])
        ifTrue: [^self].
```

```
    self isEmpty ifTrue: [^self data: object].
    self isLeaf ifTrue: [
        d := self data.
        (cPos := d position) = oPos ifTrue: [^self].
->      self childFor: d at: cPos in: (d meta state spaceFor: d).
        self data: nil].
->  (self childFor: nil at: oPos in: (object meta state spaceFor: object))
->      <> (M b add: object)

"Required Addition"
childFor: object at: position in: space
    "Answer a child to hold object at position. If such a child does not
     exist then create one in space and answer the new child."
    | child index |
    child := self children at: (index := self childIndexFor: position).
    child isNil ifFalse: [^child].
    child :=
->      (self class in: space) for: object in: (self childAreaFor: position).
    ^self children at: index put: child

"Optional Semantic.  Enable concurrency using Fork/Join iteration"
forceOn: position
    | accels |
    self isEmpty ifTrue: [^0 @ 0].
    self isLeaf ifTrue: [
        ^(self containsPoint: position)
            ifTrue: [0 @ 0]
            ifFalse: [self data accelerationOn: position]].
    ((self containsPoint: position) not and: [self isFarEnough: position])
        ifTrue: [^self data accelerationOn: position].
->  accels := OrderedCollection new.
->  self children
->      do: [:c | c isNil ifFalse: [accels add: (c forceOn: position)]].
->  ^accels inject: 0 @ 0 into: [:result :a | result + a]

"Optional Semantic.  Enable concurrency using Fork/Join iteration"
prepare
    | kids |
    self isLeaf ifTrue: [^self].
    self data: (
        VirtualParticleClass new mass: 0.0 position: nil velocity: 0.0).
->  kids := OrderedCollection new.
->  self children do: [:c | c isNil ifFalse: [kids add: c prepare]].
->  kids do: [:kid | self data mergeWith: kid data]
```

B.1.5 **Distributed** Solver

Class methods

```
"Required Addition"
TjreplicationDescriptorFor: anObject
   "Answer a descriptor which copies only the tree slot.  The default."
   ^#((true nil)
     (|| calculateAccelerations moveParticlesOver: addParticlesTo:))

"Required Addition"
TjreplicatingReplicationDescriptorFor: anObject
   "Answer a descriptor which replicates the tree slot."
   ^#((rep nil)
     (|| calculateAccelerations moveParticlesOver: addParticlesTo:))

TjClassArgUseSpecs
   "Answer the known argument usage for the receiver's methods.  This
    method can be auto- or hand-generated."
   ^#((initialize:using: (ref (nil nil nil true true deep nil true deep))))
```

Instance methods

```
"Required Structural.  Give a hook for replica multicasting."
addParticlesTo: t
   particles synchronousDo: [:p | t add: p]

"Required Annotation.  Make into a synchronous operation."
->  self particles synchronousDo: [:particle |
       particle calculateAccelerationIn: self tree]

"Required Annotation"
stepOver: dt
->  self tree <> (M b prepare).
->  self <> (M b calculateAccelerations).
->  self <> (M b moveParticlesOver: dt)

"Required Structural"
updateTree
   self tree: (self tree class for: nil in: self tree area).
->  self addParticlesTo: self tree
```

B.1.6 Invocations

Given the above classes and methods we can create various configurations of the N-Body problems. Below are four helper methods used for creating and initializing various objects. Following those

are four methods which create different problem topologies. Note that the spec argument is an instance of ProblemDescription which is just a convenient repository for configuration state.

Solver class **helper methods**

```
instantiate: spec
   "Make a solver which is replicated in space and the root of the tree is
    replicated in spaces as well."
   | selector problem |
   selector := self problemSelectorFor: spec.
   problem := self perform: selector with: spec.
   self initialize: problem using: spec.
   ^problem

createParticle: particleClass x: x y: y
   | p |
   p := particleClass new.
   p
      mass: Random next * 100
      position: (Random next * x) @ (Random next * y)
      velocity: (Random next * 6 - 3) @ (Random next * 6 - 3).
   ^p

initialize: problem using: spec
   | x y p |
   x := spec dimensions width.
   y := spec dimensions height.
   spec particleCount timesRepeat: [
      p := self createParticle: spec particleClass x: x y: y.
      problem <> (M b addParticle: p)]

problemSelectorFor: spec
   ^('createProblem', spec name, ':') asSymbol
```

Problem creation methods

```
createProblem1: spec
   "Make a solver which only distributes the tree and does no caching or
    replication."
   | solver |
   solver := spec solverClass newForkConcurrent.
   solver tree: (spec treeClass for: nil in: spec dimensions).
   ^solver

createProblem2: spec
   "Make a solver which replicates the tree root in spaces."
```

```
    | solver tree |
    solver := spec solverClass newForkConcurrent.
    solver tree: (tree := spec treeClass for: nil in: spec dimensions).
    tree meta replication
        replicateInAll: spec spaces using: #default for: tree.
    ^solver

createProblem3: spec
    "Make a solver which is replicated in spaces but the tree is not."
    | solver |
    solver := spec solverClass newForkConcurrent.
    solver tree: (spec treeClass for: nil in: spec dimensions).
    solver meta replication
        replicateInAll: spec spaces using: #default for: solver.
    ^solver

createProblem4: spec
    "Make a solver which is replicated in space and the root of the tree is
     replicated in spaces.  Note the #replicating in the replicateInAll:
     message."
    | solver |
    solver := spec solverClass newForkConcurrent.
    solver tree: (spec treeClass for: nil in: spec dimensions).
    solver meta replication
        replicateInAll: spec spaces using: #replicating for: solver.
    ^solver
```

B.2 Expert system

This section outlines the new and annotated code required to convert the original sequential uniprocessor expert system to a concurrent distributed system (see Section 7.2 for more details on the application itself).

B.2.1 Annotations

Concurrency control

The default activity for an active object is to simply loop getting messages and processing them. Inference engines are slightly different in that they are essentially compute servers. Ideally they would just continually process activated rules but they must also admit external control in a graceful way. Here we describe an engine which gives priority to external messages before running any available activations.

```
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]]]
```

Public interface specification

In creating an active object we need to specify which of its methods are to be made available for public/concurrent use. We can specify all methods, no methods or a subset of the object's existing methods. In many senses, the methods specified as public here are equivalent to Emerald's monitored methods. Senders of these messages are guaranteed that the methods will be executed in a concurrency controlled and safe way.

```
RuleAgenda class»CodApublicMethods
   "Declare the public interface for instances of the recevier"
   ^super CodApublicMethods
      addAll: #(addActivation: removeActivation: remove: selectAndRemove:
         removeSelection);
      yourself

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

GeneralNode class»CodApublicMethods
   "Declare the public interface for instances of the recevier"
   ^super CodApublicMethods
      addAll: #(clearWithEngine: acceptNewToken:
         acceptNewObject:withClass:fromEngine: removeObject:withClass:);
      yourself

LocalTrigNoVarNode class»CodApublicMethods
   "Declare the public interface for instances of the recevier"
```

```
    ^super CodApublicMethods
        addAll: #(acceptNewObjectModified:withClass:);
        yourself

LocalTrigOneVarNode class»CodApublicMethods
    "Declare the public interface for instances of the recevier"
    ^super CodApublicMethods
        addAll: #(acceptNewObjectModified:withClass: removeObject:withClass:);
        yourself

LocalTrigVarNode class»CodApublicMethods
    "Declare the public interface for instances of the recevier"
    ^super CodApublicMethods
        addAll: #(acceptNewObjectModified:withClass:);
        yourself

NoVarNode class»CodApublicMethods
    "Declare the public interface for instances of the recevier"
    ^super CodApublicMethods
        addAll: #(removeObject:withClass:);
        yourself
```

Marshaling, Replication and Migration descriptors

Many objects have special requirements when they are communicated between processors. These requirements vary depending on the use-case. For example, an object's default marshaled representation may not be effective as a migration representation. To cope with this we allow classes to be annotated with default descriptors for various operations such as marshaling and migration. The methods below set out such defaults.

```
Shell»addFactObject: anObject
    "Before asserting anObject, modify its marshaling descriptor to cache
     its class when passed as a reference."
->  anObject meta marshaling descriptor: #classed.
    engine acceptNewObject: anObject.
    engin resume

Configuration class»TjmigrationDescriptorFor: anObject
    "Define the default `shape' of receiver instances when they are migrated"
    ^TjMarshalDescriptor reg: #(true nil true nil nil true true)

Implementation class»TjmigrationDescriptorFor: anObject
    "Define the default `shape' of receiver instances when they are migrated"
    ^TjMarshalDescriptor reg: #(true true true)

FactBaseImplementation class»TjmigrationDescriptorFor: anObject
```

```
    "Define the default `shape' of receiver instances when they are migrated"
    ^TjMarshalDescriptor reg: #(true -2 -2)

RuleBaseImplementation class»TjmigrationDescriptorFor: anObject
    "Define the default `shape' of receiver instances when they are migrated"
    ^TjMarshalDescriptor reg: #(true deep true -2 shallow)

InferenceEngine class»TjmigrationDescriptorFor: anObject
    "Define the default `shape' of receiver instances when they are migrated"
    ^TjMarshalDescriptor reg: #(
        true true true shallow nil true true shallow true deep true deep)

Activation class»TjmigrationDescriptorFor: anObject
    "Define the default `shape' of receiver instances when they are migrated"
    ^TjMarshalDescriptor reg: #(true true true true)

GeneralNode class»TjmigrationDescriptorFor: anObject
    "Define the default `shape' of receiver instances when they are migrated"
    ^TjMarshalDescriptor
        reg: #(true true true shallow -2 shallow -2 true true shallow true)

Token class»TjisImmutable: object
    "Declare whether or not instances of the receiver are immutable"
    ^true

Token class»TjmarshalValueOnly
    "Declare if receiver instances must only be marshaled as values"
    ^true
```

B.2.2 Additions

Initialization

The following methods are used to traverse an expert system's implementation after it has been created and install or remove concurrency. Despite the name 'initialize', these methods can actually be used after the system is initially created to dynamically change the way it runs.

```
Shell»initializeActive
    "Initialize the receiver (and its depends) to function actively"
    self configuration inferenceEngine initializeActive

InferenceEngine»initializeActive
    "Initialize the receiver (and its depends) to function actively"
    self context initializeActive.
    self ruleAgenda initializeActive
```

```
RuleAgenda»initializeActive
   "Initialize the receiver (and its depends) to function actively"
   (self meta concurrentFor: self) ifTrue: [self meta execution resume]

Implementation»initializeActive
   "Initialize the receiver (and its depends) to function actively"
   self ruleBaseImplementation initializeActive

RuleBaseImplementation»initializeActive
   "Initialize the receiver (and its depends) to function actively"
   self implementation do: [:node | node initializeActive]

GeneralNode»initializeActive
   "Initialize the receiver (and its depends) to function actively"
   self followingNode notNil ifTrue: [self followingNode initializeActive].
   (self meta concurrentFor: self) ifTrue: [self meta execution resume]

Shell»initializePassive
   "Initialize the receiver (and its depends) to function passively"
   self configuration inferenceEngine initializePassive

InferenceEngine»initializePassive
   "Initialize the receiver (and its depends) to function passively"
   self context initializePassive.
   self ruleAgenda initializePassive

RuleAgenda»initializePassive
   "Initialize the receiver (and its depends) to function passively"
   self meta concurrentUninitializationFor: self

Implementation»initializePassive
   "Initialize the receiver (and its depends) to function passively"
   self ruleBaseImplementation initializePassive

RuleBaseImplementation»initializePassive
   "Initialize the receiver (and its depends) to function passively"
   self implementation do: [:node | node initializePassive]

GeneralNode»initializePassive
   "Initialize the receiver (and its depends) to function passively"
   self followingNode notNil ifTrue: [self followingNode initializePassive].
   self meta concurrentUninitializationFor: self
```

Local remote object processing

The following methods enable the local processing of some generic methods available to all objects. These methods are copies of the ones found in class Object in the expert system's implementation.

```
TjRemoteReference»exist
    "By default an object exists and an UndefinedObject does not exist.
    Since the receiver is a remote object and nil cannot be remote we can
    answer this question immediately."
    ^true

TjRemoteReference»goFor: anInferenceEngine
    "Add the receiver as a new fact in the context of anInferenceEngine."
    anInferenceEngine acceptNewObject: self

TjRemoteReference»modifiedFor: anInferenceEngine
    "The receiver is a modified fact in the context of anInferenceEngine."
    anInferenceEngine modifyObject: self

TjRemoteReference»removeFor: anInferenceEngine
    "Remove the receiver from the context of anInferenceEngine."
    anInferenceEngine removeObject: self
```