TR-0244-12-93-165

MetaObject Protocol Concepts for a "RISC" Object Model

Frank Manola

December 30, 1993

GTE LABORATORIES INCORPORATED
40 Sylvan Road
Waltham, MA 02254

TR-0244-12-93-165

This document is a GTE Laboratories Technical Report. It describes results and conclusions reached upon completion of a major phase of a research project. The ideas and views put forth by the author have been reviewed and accepted by the appropriate Laboratory Director.

Author	Date
Manager Intelligent Database Systems	Date
Laboratory Director Computer and Intelligent Systems Laboratory	Date

Abstract

Future information processing environments will consist of a vast network of *heterogeneous*, *autonomous*, and *distributed* (HAD) computing resources (hardware, programs, and data). There is increasing agreement that modeling such a system as a distributed collection of *objects* provides the appropriate framework for integrating HAD resources, in both distributed computing and telecommunications environments. This is illustrated by the increasing number of standards activities related to HAD systems, including communications and distribution standards, database and programming language standards, and repository standards, that are moving toward adopting, or have already adopted, an object-oriented approach. These activities include not only those of official standards bodies, such as ISO and ANSI, but also those of industry consortia, such as the Open Software Foundation (OSF) and the Object Management Group (OMG). The use of the object abstraction in integrating heterogeneous and autonomous components is also characteristic of recent developments in *application integration architectures*. These include facilities for personal computers, as well as facilities for integrating distributed software on a large scale, such as the OMG's Object Management Architecture (OMA) and the OSF's Distributed Management Environment (DME).

A typical feature of such architectures is the use of a *common object model* to provide a shared set of abstractions understood and supported by all components, to support component interoperability. A number of candidates exist that could serve as such a common object model, including models developed explicitly for this purpose. However, while interoperability may be achieved by the universal adoption of one of these models, it is possible that no one model will achieve universal adoption. In this case, the next best situation would be for there to be straightforward mappings between the features of the various object models used in the network. In addition, new object models will continue to be developed, and existing object models will evolve to incorporate new features. In the face of such change, it is important that an object model be *adaptable*, in order to accommodate new features in a way that is modular, and does not unduly interfere with existing software. To address this problem, an earlier report suggested defining what might be termed a "reduced instruction set" or "RISC" object model, consisting of a few basic, but in combination very powerful, facilities to allow the features found in many different object models to be defined as combinations of these basic facilities.

This report describes further progress in identifying the key features of the RISC model. After presenting the background behind the RISC model concept, the report briefly reviews the RISC object model components identified in earlier work, and compares them for consistency with a set of basic object model concepts being developed by the X3H7 object standards committee. The report then describes a number of different object-oriented languages or systems that either explicitly support, or contain aspects related to, a key aspect of the RISC model called *computational reflection* (the ability of a model to represent its own concepts, and to perform computation that affects its own behavior). In the RISC model, this involves the definition of *metaobjects* (objects that represent components of object models), and their operations, called a *metaobject protocol*. The intent of studying these reflective object systems is to look at what designers who are explicitly trying to provide reflection (and thus are explicitly trying to identify the key primitives in their own languages) are actually identifying as key concepts, in order to contribute to a better understanding of the concepts that the RISC model should contain. The descriptions of reflective systems in this report illustrate a wide range of applications for adding extended object model capabilities using reflection. A number of these applications illustrate capabilities directly related to object model interoperability and applications within heterogeneous distributed object systems. The report concludes by describing some conclusions, and planned future work based on these ideas.

Table of Contents

1. Introduction and Background

Future information processing environments will consist of a vast network of *heterogeneous*, *autonomous*, and *distributed* (HAD) computing resources, including computers (from mainframe to personal), applications, and information systems, and data (files and databases). With the appearance of increasing numbers of such HAD computing resources, there is an increasing need for technology allowing such resources to be flexibly combined, and their activities coordinated, to create network-wide integrated *distributed systems* that address challenging new information processing requirements. In some cases, these systems may combine resources belonging to *different* individuals or organizations, in support of various social, governmental, or business activities (e.g., connections between customer inventory systems and supplier order-entry systems).

There is increasing agreement that modeling a distributed system as a distributed collection of interacting *objects* (in the sense of object-oriented programming) provides the appropriate framework for use in integrating HAD resources, in both distributed computing and telecommunications environments [WA91]. This is illustrated by the increasing number of standards activities related to HAD systems, including communications and distribution standards, database and programming language standards, and repository standards, that are moving toward adopting, or have already adopted, an object-oriented approach [FKMT91]. These activities include not only those of official standards bodies, such as ISO and ANSI, but also those of industry consortia, such as the Open Software Foundation (OSF) and the Object Management Group [OMG91a].

Objects form a natural model for a distributed system because distributed components can only communicate with each other using messages addressed to well-defined interfaces, and components are assumed to have their own locally-defined procedures enabling them to respond to messages sent them. The use of objects also naturally accommodates many of the heterogeneous and autonomous aspects of HAD systems. It accommodates heterogeneous aspects because messages sent to distributed components depend only on the component interfaces, not on the internals of the components. It accommodates autonomous aspects because components may change independently and transparently, provided their interfaces are maintained. Object-oriented programming, distributed or not, simply carries this model down to the level at which the "components" are individual application abstractions, or even individual data items, rather than entire systems or nodes in a more conventional distributed network.

The use of the object abstraction in integrating heterogeneous and autonomous components is also characteristic of recent developments in *application integration architectures* [HT93]. These include facilities for personal computers, such as Object Linking and Embedding (OLE) in Microsoft® Windows™, as well as facilities for integrating distributed software on a large scale, such as the OMG's Object Management Architecture (OMA) [OMG92a] and the OSF's Distributed Management Environment (DME). In this case the components include application programs and their associated data, as well as general services such as those provided by database management systems.

1.1 Object Model Interoperability and Flexibility Requirements

The *distributed object management* technology required to support HAD systems and application integration architectures continues trends in object-oriented programming, distributed system models, application integration environments, and database systems, with the goal of allowing the resources available in a distributed network to be treated as a commonly-accessible collection of objects which can be combined in arbitrary ways to provide new information processing capabilities. These objects can range in granularity from entire systems to individual application objects (text strings, images, etc.). [MHGH+92] describes general aspects of this technology in more detail. Distributed object management describes the task of managing a distributed collection of objects in the same sense as distributed database management describes the task of managing a distributed database.

We refer to a system performing this distributed object management task as a *Distributed Object Management System* (*DOMS*). A number of proposals for generic DOMS architectures have been developed, for example, the Common Object Request Broker Architecture (CORBA) defined by the OMG [OMG91a]. Distributed object management is an essential component of most application integration architectures (and in fact, as we use the concept, the architecture of a DOMS is that of an object-oriented application integration architecture.)

In an application integration architecture, abstractions must be provided in application programming interfaces (APIs) that allow one object[1] to invoke operations provided by other objects, even when they are written in other languages or exist in distributed network components. A typical feature of such architectures is the use of a *common object model* to provide a *shared* set of abstractions understood and supported by all components[2]. This is similar to the use of a common global *data* model in a heterogeneous DBMS, except that in a DOMS an *object* model is required to model and provide access to the behavior (operations) supported by the various components in the system. ([MH93] provided additional details on the role played by a common object model in an application integration architecture.) General acceptance (or standardization) of the object model and interfaces of such an architecture would facilitate independent development of interoperable objects and supporting software in an application integration architecture (which is, e.g., the goal of the OMG in specifying and promoting such an architecture). For example, use of a common object model means that the developer of a new object in the architecture must only support interfaces defined in terms of a *single* set of abstractions (those of the common model), no matter what implementation language is used for that object. Use of a common object model also means that an object may access other objects in the architecture via interfaces defined in terms of that same set of abstractions, no matter what other languages may have been used to define those objects.

Figure 1.1 shows the situation that exists when no common object model is available. In this situation, a system or application using one object model that needs to use objects based on other object models requires a separate mapping for each of those object models. In general, mapping between N different object models requires N(N - 1) mappings. Moreover, adding a new object model is complex, since mappings to all the existing models must be defined for each new model.

---

[1] The object may be a complete application, which is implemented as an object to allow its reuse.

[2] The term *object model*, as used throughout this report, refers to the collection of concepts used to describe objects in a particular object-oriented language, specification, or analysis and design methodology, and corresponds closely to the use of the term *data model* in "the relational data model". Thus, we speak of "the Smalltalk object model" or "the OMG object model". This is in contrast to use of *object model* to describe the collection of objects created to model a particular system or application, as in "the Automatic Teller Machine object model" or "the object model of a windowing system" [RBPE+91]. From our point of view, [RBPE+91] defines a particular object model (our sense), which includes concepts like *object, inheritance*, *attribute*, and so on, and uses it to define the object models (other sense) of various applications. This dual usage is unfortunate, but is common in the literature.

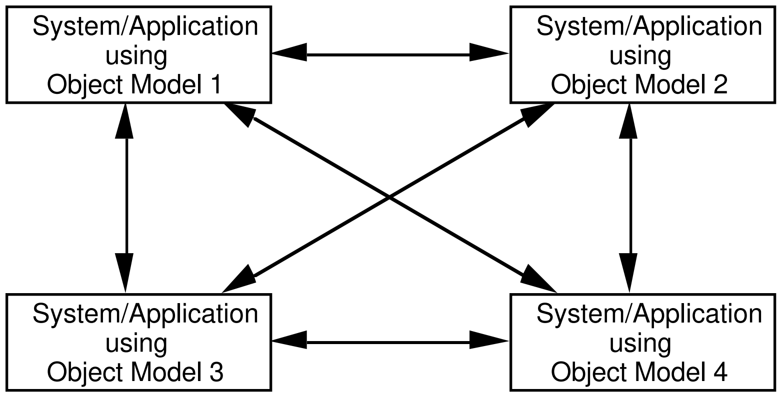

Figure 1.1. Object Model Mappings Without a Common Object Model

Figure 1.2 shows how a common object model simplifies the situation. In this case, all that is necessary is to provide a mapping between each system or application object model and the common object model. In this case, mapping between N different object models requires only N mappings. Moreover, adding a new object model requires defining only a single new mapping, between the new model and the common object model.

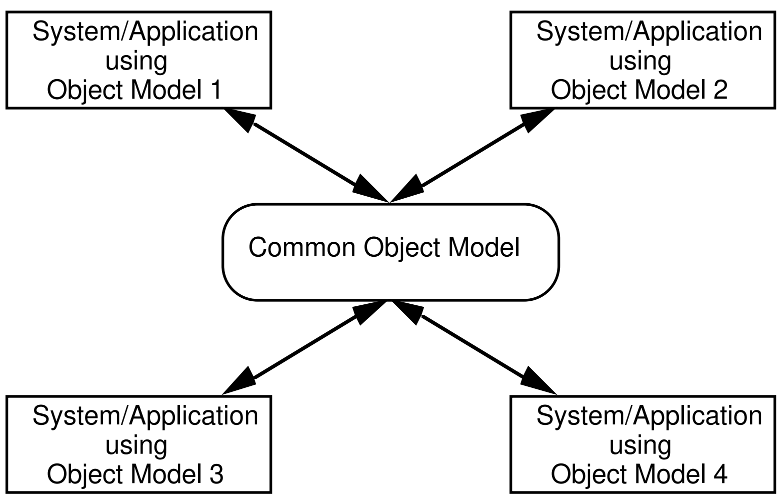

Figure 1.2. Object Model Mappings Using a Common Object Model

Many different object models have been and are currently being developed to serve various application areas. In addition, object models have been developed explicitly to serve as the common object model in application integration architectures, such as the OMG Core Object Model described in [OMG92a]. Any of these object models could theoretically serve as the common object model in a generalized application integration architecture. However, while interoperability may be achieved by the universal adoption of one of these models, it is possible (even likely) that no one model will achieve universal adoption. The recognition that this is a possibility, and of the complications that this could cause, is becoming increasingly well-known in the computer trade press [Als93, Way94], in connection with the different common object models of competing application integration architectures. If no one of these models achieves universal adoption, the next best situation would be for there to be well-understood mappings between the features of the various object models used in the network.

The situation is further complicated by the fact that, even in a single application integration architecture with a well-defined common object model, such as OMG's Object Management Architecture [OMG92a], the object models (or type systems, if the languages are not object-oriented) of the different programming languages used to develop objects in the architecture must be mapped to the common object model of the architecture. This is typically done through *language bindings*, one per programming language, to a language based on the common model. An example is the C language binding to OMG's Interface Definition Language (IDL) defined in [OMG91a].

Defining such mappings is complicated by the variety of ways in which object models may differ from each other. Some of these differences were described in previous reports [Man89, MH93]. In some cases, these variations are arbitrary, and could conceivably be eliminated by changing individual object models to conform to some common standard. However, in other cases the variations reflect important differences in application requirements, and could not be eliminated without creating difficulties in developing those applications (at least as perceived by those who have to develop the applications). For example, the "managed object model" used in telecommunications [Ash93, QP91] relies heavily on the concept of an *event*, which is not contained in many conventional object models. Even though events could be defined in terms of other concepts, doing so might make developing applications involving managed objects more difficult, since a translation would always be required between the actual concepts used to develop the applications, and the target (managed object) model. This example, and other similar examples that could be cited, illustrates why variations in the object models used within applications in a given application integration architecture will almost certainly continue to exist. As a result, there is a need to investigate mechanisms for achieving interoperability among these various object models.

In addition, object-oriented software technology is relatively young, and rapidly developing. This means that object models will continue to evolve. Specifically, new object models will continue to be developed, such as the relatively-recent Component Object Model of OLE 2 [Bro94], and existing object models will evolve to incorporate new features. In the face of such change, it is important that an object model be *adaptable*, in order to accommodate new features in a way that is modular, and does not unduly interfere with existing software. For example, the Object Management Group has already identified a number of situations in which extensions to its object model may be necessary. As a result, there is a need to also investigate mechanisms by which an object model may be made more adaptable, allowing it to be modified as necessary to support changing requirements. The subject of adaptability is discussed more fully in Section 1.3.

1.2 The Concept of a RISC Object Model

An object model is a description of the general characteristics of the objects that can be defined in a particular object-oriented language or system. Ideally, this description is precisely defined. The capabilities defined by the object model must then be supported by the various languages and run-time facilities associated with the system. As previous reports have observed [Man89, MH93], there are differing notions of "object" among the models found in various object-oriented systems, and many different ways in which these object models are defined. For example, the intraobject computation style may be based on any of the standard computational paradigms, such as imperative, functional, or rule-based. An object may or may not be thought of as having an inherent process associated with it. An object method may appear to be invoked by sending a message to an object, or applying a function with the object as a parameter. Objects may or may not have explicitly-defined instance variables to hold state information. Messages may be passed synchronously or asynchronously. Moreover, object models may have different mathematical foundations (or none at all), raising issues as to the completeness and consistency of the models.

To address the problem of defining mappings among object models with different characteristics, [MH92,93] suggested defining a "reduced instruction set" or "RISC" object model[1], based on work on the formal foundations of object models such as [Bee90, DT88, Agr91]. The model would consist of a few basic, but in combination very powerful, concepts to allow the features of many individual object models to be defined as combinations of the basic concepts. An example of this idea is the way in which some object models define the concept of an *attribute* `X` in terms of a pair of operations `getX` and `setX` [SCBK+86]. More complex examples are the use of combinations of *rules* to define the semantics of various object relationships [DMBC+87], or to define the semantics of delegation and inheritance [AT88]. The RISC object model concept involves extending this idea to include all parts of an object model definition, with the intent of providing a common framework for understanding heterogeneous object models by allowing their semantics to be defined in terms of a single set of concepts.

The common framework provided by a RISC object model could serve a number of different roles in addressing the problem of object model interoperability. Serving in a purely *descriptive* role, the RISC object model could be the basis for understanding differences among object models, for defining mappings between different object models, and also for defining new, application-specific models. Work on such a framework, and on comparing and defining mappings between specific object models, is part of the standards activity recommended in [FKMT91], and being pursued by the X3H7 (Object Information Management) standards committee. For example, X3H7 is currently preparing a "Features Matrix" which attempts to describe many different object models in terms of a common set of concepts [X3H793a]. It is expected that describing object model features in terms of a common set of concepts will assist in minimizing differences among object models, and also in defining mappings between specific object models.

The RISC object model could also be *implemented*, by defining *metaobject types* for each of its basic concepts, and allowing them to be used in different combinations as the basis for mapping between different object models, and for modeling both existing and new object models. Both the CLOS metaobject protocol [KRB91] and IBM's System Object Model (SOM) [OH92] illustrate aspects of this approach to object model interoperability. In addition, due to the potentially greater flexibility of the RISC model, it could provide the basis for improving the flexibility with which existing components (procedural and data), treated as objects, could be combined into new objects at higher levels of abstraction, by providing model concepts equally applicable to objects of arbitrary granularity. This is an important issue in a HAD environment, which must deal with both "legacy" code and data, which may exist as fairly large aggregates, as well as new objects constructed from them.

If implemented, the RISC object model could play the role of a form of object model *microkernel*. The use of microkernels is a key element in the design of such new operating systems as Windows NT, IBM's Workplace OS, and OSF's OSF/1, due to the flexibility and design modularity they provide. In a recent paper [Tan93], Tanenbaum described the state-of-the-art in distributed operating systems, and observed that microkernel implementations were a good idea. In operating systems, a *microkernel* is a relatively small operating system kernel with a limited functionality. It is intended to make it possible for system programmers to use it as a base for building various operating systems on top of it. Tanenbaum went on to say:

---

[1] by analogy with "Reduced Instruction Set Computer".

> There is a clear analogy between microkernels and RISC machines...In short, microkernels can be thought of as RISC operating systems. The advantages are simplicity, modularity and flexibility. The primary disadvantage is performance...I do not believe that this small difference in performance even comes close to outweighing the advantages of simplicity, modularity, and flexibility.

The RISC object model embodies aspects of the same basic idea (together with facilities for extending itself to be discussed below).

The RISC object model concept is based on the observation that descriptions of object models tend to use a small set of common concepts, even though the object models themselves may be very different. Examples of such concepts used to describe conventional object models include *identifier*, *method*, *interface*, *class*, and *state*. The use of a small set of common concepts to describe object models tends to characterize not only informal descriptions of object models, such as the current work of the X3H7 standards committee [X3H793a], and the descriptions of many object models in the literature, but also more formal work on the foundations of object models such as [Bee90, DT88, Agr91], although sometimes the concepts used in this latter work are even more primitive. A related observation is that new object models are frequently developed by generalizing one or more of these common concepts, for example, by allowing an object to have more than one interface [RS91a] rather than restricting it to only one. (Additional background was provided by prior work on defining object models with *reflective* (self-descriptive) capabilities [MD89, HZ90, MB90], and in defining object model capabilities required for use in distributed applications [Man89].)

The basic approach of the RISC object model is to identify the primitive concepts that represent the basic components of objects in any object model (those that are necessary to "talk about" those object models) and to define them as (more primitive) objects. The idea is for the RISC model to include explicit primitives for describing the basic components of objects in any object model, such as methods (procedural or computational objects), state, object identifiers, and interfaces, together with other necessary primitives to allow object representations and types to be definable in the RISC object model itself. The RISC model would also include explicit primitives for modeling messages (invocation requests), expressions, and the invocation of methods. A given object model can then be described in terms of the way that object model combines these RISC primitives to form objects and other constructs, and in terms of the behavior of these primitives and their interactions, within that particular model. Moreover, variations among object models can be described in terms of variations in the composition and behavior of these primitive objects.

To illustrate the *structural* aspects of this approach (the *behavioral* aspects are discussed later), consider an individual object in a simple object model, such as the `3Dsolid` object shown in Figure 1.3. From the point of view of the RISC model, this object can be considered as a *structure* consisting of a number of basic components, and the relationships between them. The components of this object include:

- a collection of variables (and their assignments) that defines the object's *state*. In this simple object, or in a Smalltalk object, this collection is essentially a tuple, and in general can be considered an internal object accessible only to the containing object's methods.

- a collection of programs (the object's *methods*) that can operate on the state, and which implement the operations that the object can perform when a message is sent to it. The methods can be considered as independent pieces of program text that are explicitly associated with the object. (In this object the methods are contained within the object itself, as opposed to being contained in a separate class object as in the Smalltalk model).

- an *interface* or object boundary that separates the "inside" of the object from the "outside". The interface specification determines which aspects of the object are visible to clients of the object, and which aspects are invisible. The interface may make visible to clients some or all of the object's operations, or some or all of the object's state. The methods, being within the object boundary, are generally considered to have direct access to the object's state.

- an explicit value (or object) that serves as an *object identifier*. The object identifier serves as a reference to the object, and internally supports some form of dereferencing operation via some form of addressing mechanism. The object identifier is not necessarily stored within the object, and is thus not explicitly shown here. However, object identifiers of other objects are stored in the object state and passed as parameters in messages as references to these other objects. For example, @myparts in Figure 1.1 represents the object identifier of another object representing the `3Dsolid` object's subparts.

Other aspects of this figure are also pertinent:

- The figure shows the relationships between the object's methods and its state. These relationships are based on the fact that the methods have special access to the state, and are dependent on the state.

- The figure shows relationships between the operations defined at the object's interface (that seen by client objects) and the object's methods. These are shown as dashed lines, because the relationship is not strictly structural, but is actually determined by a *dispatcher*, which determines which of the object's methods to invoke in response to a message directed to the object. (In this case, the relationship between operations and methods is one-to-one, but this need not be the case in general).

As suggested by the figure, the primitive RISC objects that make up a conventional object form a structure: a *graph*, in which the nodes are state, methods, and other primitive objects, and the edges are the relationships between these primitive objects defined by the object model being represented (in Figure 1.1, the graph consists of the methods, state, and the lines representing their relationships). This graph is traversed during dispatching (the process of translating a message to the invocation of a specific method or methods) or other "internal" operations of the object model. [MH93] showed how, using these same primitives for representing objects, variants in the representation of objects in different object models could also be described.

In addition to modeling the *structural* aspects of different object models, the RISC object model must also be able to model the *behavioral* aspects of different object models. For example, many variations exist in object models as to how messages are handled by receiving objects, e.g., synchronous vs. asynchronous message passing, and message passing in class-based vs. prototype-based models. Since message passing is such a basic aspect of any object-oriented system, the ability to explicitly model the message passing (or dispatching) process is a key aspect of modeling the behavior of different object models. (For this reason, the capability of explicitly specifying aspects of dispatching is supported by both the CLOS metaobject protocol [KRB91] and IBM's SOM [OH92]). Support for this capability also requires the ability to explicitly represent messages.

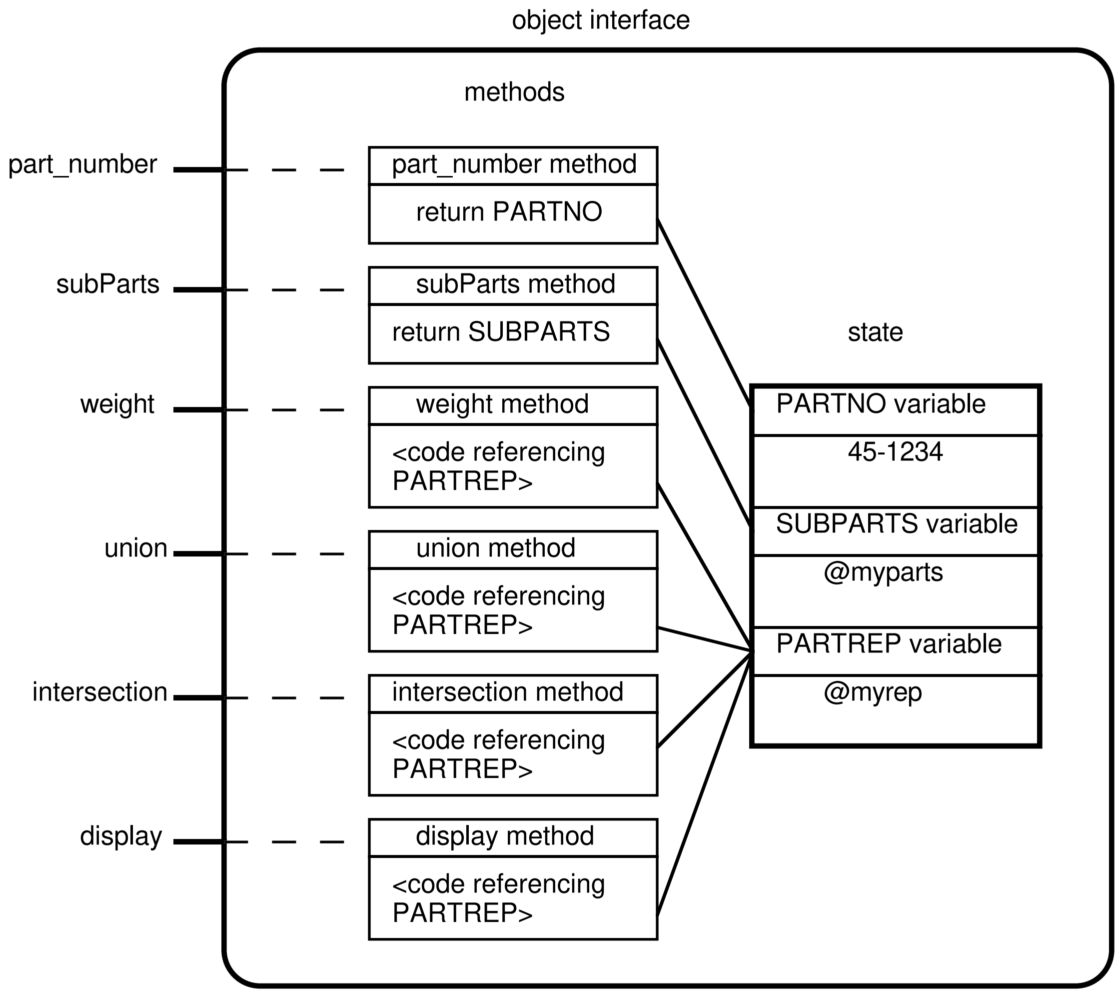

Figure 1.3. A 3Dsolid Object

The ability of an object model to explicitly represent and manipulate its own structural and behavioral aspects is called *computational reflection* and, hence, the RISC model is a form of *reflective* object model[1]. Section 3 describes a number of reflective object models, in an attempt to understand the key concepts and mechanisms provided in such models, compare them with those currently included in the RISC model, and fill in any gaps that may currently exist in RISC model concepts.

---

[1] Technically, the goal of the RISC object model is to describe *other* object models, rather than itself. Hence, it is not reflective in the same sense as these other reflective models. However, reflective capabilities are nevertheless necessary in the RISC model. This issue is discussed more fully in Section 3.

Since the RISC object model takes the approach of representing the primitive components of object models as primitive objects, which like other objects have both state and behavior, the behavioral aspects of different object models can be represented in terms of the behavior (operations) of these primitive objects. Hence, the approach taken in the RISC object model is to define an appropriate set of operations for each of the types of primitive objects, together with an overall flow of control that ties these operations together. The combination of the operations of the individual primitives, together with the steps in the overall flow of control, can then be used to describe the behavior of various object models. The overall flow of control could be expressed procedurally, or as rules, and could be message-based or event based, as discussed in [MH93]. The primitive objects defined by the RISC object model constitute *metaobjects* (objects that describe objects), and the operations defined for these objects, together with the overall flow of control, constitute a *metaobject protocol*.

This concept of a metaobject protocol is similar to that of the CLOS metaobject protocol (which introduced the term), except that the RISC object model is not tied specifically to Common Lisp concepts. The inclusion of explicit metaobjects, and the ability to specify their behavior, provides the basis of an adaptable (actually, a *programmable*) object model. As described in [KRB91], a metaobject protocol is an interface to a model or language that provides the ability to describe and incrementally modify the model or language's behavior and implementation. Languages that incorporate metaobject protocols blur the distinction between language designer and language user. In a language based on a metaobject protocol, the basic elements of the language (classes, methods, and generic functions in the case of CLOS) are first made accessible as objects. Because these objects describe other (application-level) objects in an object-oriented program, they are called *metaobjects*. Individual decisions about the behavior of the language are then encoded as operations (protocol) defined for these metaobjects, which is why these operations are called a *metaobject protocol*. Finally, for each kind of metaobject, a default class or type is created, which lays down the behavior of the default language in the form of methods in the protocol. In this way, metaobject protocols, by supplementing the base language, provide control over the language's behavior, and provide the user with the ability to selectively modify key aspects of the language's behavior controlled by the default classes. The motivation for providing built-in reflective capabilities in object-oriented languages and systems is to provide a general framework for programmers to extend or tailor the language or system to specific application requirements in a modular way. Section 3 describes a number of examples of how metaobject protocols and reflection can be used to extend the capabilities of objects in a given language, such as by adding support for distributed objects or concurrency control.

In the CLOS metaobject protocol, for example, the rules used to determine the representation of individual objects are controlled by a small number of generic functions. This makes it possible to change those rules by defining a new kind of class, as a subclass of the default, and by giving it specialized methods on those generic functions. By doing this, the user can make an *incremental* adjustment in the language. Most aspects of both the behavior and implementation remain unchanged, with just the instance representation strategy being adjusted. The topic of metaobject protocols is also discussed further in the rest of this report.

Because the RISC object model is primarily descriptive, it need not necessarily define actual methods specifying default behavior for the operations of each of its primitive types. Instead, the methods of these types could be left undefined (such types are called *abstract* in some object models), allowing them to be actually specified only in subtypes used to model a specific object model. On the other hand, an implementation of the RISC object model could also use an extremely primitive definition for the default behavior of the primitive types, which could then be overridden to model other object models. The current DOM prototype developed at GTE Laboratories [NH93a,b] implements such primitive behavior for its objects, and is being used as the basis for an experimental implementation of the RISC model. [NH93a,b] also demonstrate the adaptability provided by concepts such as those being developed in the RISC model.

We imagine the RISC object model being defined in several stages. The initial stage in defining the RISC model is to identify a *vocabulary*, a common set of concepts, for describing different object models. The X3H7 (Object Information Management) standards committee is currently pursuing this approach of describing different object models in terms of a common set of concepts. This is being done in order to understand the different object models, and in particular to understand the *differences* among these object models, with the goal of either reducing those differences or identifying mappings between them in order to improve interoperability among the different models. Variations in different object models are being described in terms of variations in these concepts, their behaviors, and relationships between them. However, all these descriptions are currently informal statements in English. As a result, *this vocabulary is not itself an object model*; it is just a set of concepts.

The second stage in defining an actual RISC *object model* is to define the set of more primitive types of objects, or *metaobjects*, that capture the concepts in the common vocabulary, together with the corresponding *metaobject protocol*. Use of these metaobjects would allow the definition of an object-oriented implementation of a given object model in terms of the metaobjects (using various combinations of them, and specializing the metaobject protocol).

It is also possible to imagine a third stage, that of defining the RISC primitives themselves using some formal specification technique. For example, the X3H4 (Information Resources Dictionary System) standards committee is currently investigating a formal basis for interpreting all (object and data) models in terms of statements in predicate logic. X3H4 is proposing that the logic for this purpose be based on the common semantics expressed by two existing systems of predicate logic. One of these is the Knowledge Interchange Format (KIF), which has been developed under DARPA sponsorship by the AI knowledge representation community. The other is Conceptual Graphs, a graphical notation for logic that has been applied in the interpretation of natural language. Work on defining object model concepts in terms of Conceptual Graphs is described in [Sow93]. X3H4 and X3H7 are jointly exploring the use of these logics in describing object model concepts, and X3H7 is considering other formal specification techniques as well, such as Z [Wor92].

1.3 The Need for Adaptable Object Models

[MH93] described the requirements that may be imposed on the common object model of a distributed application integration architecture. In particular, the object model must support the requirements of architecture services [OMG92b], some of which may require specific capabilities in the model (e.g., the ability to specify emitted events in an object's interface to support network management applications [Ash93, QP91] or OMG's event service [OMG93]). In addition, the model may need to support interoperability among the object models used in attached components. Supporting both services and interoperability potentially requires a highly *adaptable* object model, since the model must be capable of supporting new services as they are added, and new object models in attached components, with as little change as possible.

Two frequently-cited approaches do not address these problems of adaptable interoperability and service support. Defining a "union of all object models" by simply combining features from all the object models that need to interoperate, and all the services that must be supported, would create a complex and unwieldy model that would be both difficult to use (since it could contain a great deal of redundancy) and difficult to implement. Defining a "least common denominator model" which contains only those basic features found in most or all object models or services does not address the problem of supporting or mapping between the *differences* found in the models and services, and would not itself provide sufficient support for application development involving these differences. Moreover, neither of these approaches is particularly easy to extend as new object technology is developed, or new requirements are identified.

The Object Management Group has recognized these problems in defining a common object model to support these requirements, and has tried to address them by defining a way of *structuring* their common object model into a "Core + Components". This approach can be described as follows:

1. The *Core* Object Model is a consensus of the least common denominator of atomic features that the OMG membership feels must be supported by any system that calls itself "object technology." Those features are identity, typing, operations, and subtyping/inheritance. The features are considered "atomic" in the sense that they cannot be further broken down. For example, subtyping must be included in its entirety as defined in the Core in any OMG-compliant object model.

2. *Components* are compatible extensions to the Core that may be needed in some application domains, but not others. Examples of components are exception handling, attributes, specific sets of built-in types, and relationships. (For example, exception handling and a specific set of built-in types are required to extend the OMG Core Model to support the object model defined in the CORBA architecture [OMG91a].) Components provide a way to define new extensions to the Core. For example, the exceptions component extends the signatures of operations. Components can also be used to define new types to be added to the model without extending the Core Object Model semantics.

3. *Profiles* are collections, consisting of the Core plus one or more Components, that together make up a useful object model for a specific domain. These domains can be technology-specific (DBMS, GUI, programming language, etc.) or application-specific (manufacturing, network management, finance, etc.).

4. Only Profiles make up complete object models, and compliance will be measured against a Profile. Compliance to just the Core or individual Components that do not make up a Profile is irrelevant.

As noted above, components must be compatible with the Core. They can extend the semantics of the Core but they cannot restrict the functionality defined in the Core. Also, components cannot be incompatible with each other. However, a component can be a refinement or extension of another. This appears to indicate that there can be alternative components providing the same facility (e.g., different versions of exception handling) *only* as long as one extends the other; they cannot be fundamentally incompatible. [OMG92c] contains very early drafts of a number of possible components, including the exception handling, attributes, and relationships components mentioned above, and also others such as non-objects (value types), metadata, collection types, constraints, and extended operation semantics. The OMG has also indicated that, at some later time, features contained in Components could be moved into the Core, if the features become so generally available in object models that the "Least Common Denominator" effectively changes.

The Object Management Architecture Guide [OMG92a] explicitly states that the Core Object Model is not a metamodel (a model for describing or deriving other object models), and thus components defined as extensions to the Core do not have to be derived from Core concepts. In fact the Core does not contain the necessary facilities to allow this. Moreover, the Core deliberately says nothing about some aspects of the model, in order to provide for maximum flexibility in adding Components. However, the "Core + Components" approach provides a disciplined way of adapting the model to support specific application domains, while maintaining compatibility among added features.

Another object model designed with adaptability in mind is IBM's System Object Model (SOM), developed as part of OS/2 2.0 [OH92, SC92]. SOM defines a "language-neutral" object model (i.e., one different from those in programming languages), together with technology that allows different languages to share class libraries (e.g., to allow a class library developed in C++ to be used in a Smalltalk or COBOL program; this includes subclassing in the new language). A stated goal of SOM is to support features considered important in the packaging of "industrial strength" class libraries. These include support for the dynamic linking and loading of classes, the ability to update class libraries without requiring that client applications be recompiled, the ability to ship class libraries in binary form (without source code), and the ability to use object-oriented technology from within traditional procedural programming languages. In some respects, the approach used in SOM resembles that of OMG's CORBA in defining a language-neutral object model, and an Interface Definition Language (IDL) to allow multiple languages to be used in implementing SOM objects (in fact, SOM now supports OMG's CORBA IDL).

In order to provide the adaptability to establish interfaces with object-oriented languages having widely varying characteristics, SOM supports a number of features that are effectively part of a metaobject protocol (although SOM does not use that term). For example, SOM supports explicit metaclass objects (a metaclass is the class of a class), and allows the standard metaclass, `SOMClass`, to be redefined by users. This is useful, for example, to add special instance construction methods, or to modify how memory is allocated for objects [CS92]. In addition, SOM allows class definers to specify the dispatching mechanism that will be used for objects of the class. Several dispatching mechanisms are predefined. In the most general case, the class definer can specify an arbitrary function which will be called at run-time to do the dispatching.

Evaluating the adequacy of these approaches in satisfying the adaptability requirements of application integration architectures is difficult, since both the OMG approach and SOM are relatively new, and the requirements are not fully understood. In discussion at a recent workshop on application integration architectures [HT93], a range of views on the OMG approach was expressed. Some felt that the OMG Core was more than they would consider a "Least Common Denominator" model. It was also noted that a model in which "components" were required to be atomic, and there was only one way of expressing a particular feature (e.g., exceptions), as in the OMG approach, is less flexible than a model in which "components" could express, e.g., something to be *subtracted* from another component ("negative components"), or in which there might be alternative components for the same feature. On the other hand, while the OMG "Core + Components" approach might be more flexible if it allowed "negative components", as opposed to just atomic, nonoverlapping components, it might not then serve its intended purpose in facilitating interoperability. However, there was a feeling that richer object models than were available now would be needed to do component integration in application integration architectures. It was observed that the OMG approach had not really been tested yet, and it is not clear how robust it is. However, there was general support for trying some approach like it.

SOM illustrates the advantages for adaptability of including explicit metaobject concepts in the model. In particular, SOM illustrates the importance of providing dispatching as an explicit capability to be specified and controlled in the model (rather than being predefined by the model). However, SOM as yet does not describe a complete metaobject protocol (although this could easily change).

Unlike both these approaches, the RISC object model *is* intended to be a metamodel. As such, it would provide not only a core set of features, but also a built-in extension mechanism, namely, the ability to define new features in terms of its metaobjects, together with its metaobject protocol. Even if the RISC object model itself would be considered too primitive an object model to directly serve as the "Core" in the OMG sense, the RISC model could be directly useful in a "Core + Components" approach in providing the basis for more flexible Core and extension capabilities, as well as in helping to define specific extensions. In addition, the RISC model could be helpful in defining how features of the object models of programming languages used to define applications or services in the OMG architecture should be translated to features of the Core object model or one of the Components. That metaobjects might be useful in the OMG object model is illustrated by the inclusion of a metadata component in [OMG92c]. Such a component, if fully developed, could possibly provide the basis for RISC-like extension facilities within the "Core + Components" approach. Experiments investigating the use of RISC object model facilities within the framework of the OMG "Core + Components" approach are currently planned, based on GTE Laboratories' DOM prototype [NH93a,b].

The particular importance of being able to explicitly model the message passing (or dispatching) process in describing an object model has been mentioned already. This is also a key theme of the reflective object models discussed in Section 3. Like SOM, the RISC object model is intended to include the ability to explicitly specify different dispatching mechanisms for different models. The RISC model will also provide a complete metaobject protocol. Experiments in integrating object models with different dispatching mechanisms using the DOM 3.0 prototype, and illustrating the adaptability that is available with explicit control over the dispatching mechanism, are described in [NH93a,b].

Finally, it is interesting to note that the basic idea behind reflective object-oriented processing, and in particular the ability to explicitly represent and modify the message passing mechanism of an object-oriented system, is similar in many respects to the basic idea behind *intelligent networks* in telephony [GMS88, HH87, Man88]. In an intelligent network, some call-control functions for new services are moved from the switching systems to specialized control points. When a call is processed that requires special services, the switching systems will then, under certain conditions, seek instructions from these control points over special-purpose signaling paths. Intelligent networks have a number of potential advantages, including the following:

- Services can be introduced more rapidly, because only the control points must be changed in order to introduce the service.

- The generality of services that can be provided can also be increased, because the control points may potentially supply more complex processing capabilities than might be feasible at a given switch.

- Services can be introduced in a way that is independent of vendor-specific switch hardware and software, because individual switches need not be reprogrammed or otherwise modified to introduce the service.

In an object-oriented language or system, the built-in message passing mechanism plays the role of the switch in this analogy. Once this is modified so that it can interrupt normal message processing to invoke additional services, the services themselves can be added elsewhere (as additional objects). The advantages cited above for intelligent networks are mirrored by the potential advantages of object-oriented reflective processing, as shown by the (bracketed) wording in the following repetition of the above advantages of intelligent networks:

- Services [computational services] can be introduced more rapidly, because only the control points [objects providing services] must be changed in order to introduce the service.

- The generality of services that can be provided can also be increased, because the control points [objects providing services] may potentially supply more complex processing capabilities than might be feasible at a given switch [in the message passing mechanism of a particular language or system].

- Services can be introduced in a way that is independent of vendor-specific [language-specific] switch hardware and software [message passing mechanisms], because individual switches [message passing mechanisms] need not be reprogrammed or otherwise modified to introduce the service.

1.4 Report Overview

The primary purpose of this report is to describe progress made toward identifying the basic concepts of a "RISC" object model. The rest of this report is organized as follows. Section 2 briefly reviews the RISC object model components identified earlier in [MH93], and compares them for consistency with a set of basic object model concepts being developed by the X3H7 standards committee for its Object Model Features Matrix (mentioned above). Section 3 then describes a number of different object-oriented languages or systems that either explicitly support reflection (metalevel concepts), or that contain aspects related to reflection that address issues from [MH93] that required additional work. The intent is to look at what designers who are explicitly trying to provide reflection (and thus are explicitly trying to identify the key primitives in their own languages) are actually identifying as key concepts, in order to contribute to a better understanding of the concepts that the RISC model should contain. In addition to material describing each language or system, commentary is provided on the various features identified. Section 4 then discusses some conclusions, and planned future work based on these ideas.

As with [MH93] this report is *not* a full specification of a RISC model. However, it does constitute progress and greater understanding of what the components of such a model should be, and of how to use them to address issues in object model interoperability and adaptability. Additional research, described in Section 4, will be necessary to validate this concept, and to provide a full specification of a complete set of RISC object model components.

2. A Review of Previously-Identified RISC Components

As noted in Section 1, the idea behind the RISC object model is that different object models can be described in terms of different combinations of a common set of more primitive concepts. The goal of our research on the RISC object model is to identify the right set of primitives to serve this purpose. [MH93] described a strawman set of RISC object model primitives. These primitives were based a survey of a wide variety of object models, including not only models included in the X3H7 Object Model Features Matrix, but also others, e.g., those cited in [MHGH+92, MH93]. This section briefly reviews those primitives, in order to provide a basis for comparison with the concepts defined by a set of reflective object models surveyed in Section 3. Commentary in Sections 3 and 4 will indicate similarities and differences between the RISC concepts and those of the reflective models. Each RISC concept is described in a separate subsection. These concepts are described in general terms because the details will be the result of further work on the RISC object model.

2.1 Object State

An object's *state* is essentially the object's private memory, which maintains information between execution of the object's operations (i.e., what the object "remembers" about operations performed on it). An object's state is typically represented by some data structure internal to the object (i.e., not directly accessible to clients). In many object models, such as that of Smalltalk, this is effectively a "flat" tuple of instance variables, each of which contains a reference to another object (in some models values may be contained in instance variables as well). These references allow the state of an object to refer to (and thus be dependent on) other objects, so that arbitrarily-complex states may be defined.

The RISC object model must provide an explicit mechanism for modeling object state, so that it can describe the variations in how state is handled in different object models. Some of these variations involve different ways of modeling state. For example, some object systems, particularly object-oriented database systems, support internal structures that are more complex than tuples or records, such as arrays or nested structures. Other variations involve differences in how state is grouped or associated with other parts of objects (such as methods).

In the approach described in [MH93] for modeling state in the RISC object model, a state variable is considered as simply a pair of `get` and `set` operations referencing a common name, together with a constraint that the last value assigned by `set` is the value obtained by the next `get`. For example, the `partno` variable of the `3Dsolid` object whose object identifier was `oid` would be represented by the pair of operations `getpartno(oid)` and `setpartno(oid, newvalue)`. This models the abstract behavior of a state variable or memory address entirely in terms of operations. The advantage of this approach is representation independence; the implementation of the referenced value can be changed from stored to computed (or vice-versa) without affecting methods that reference the variable. Representing state by pairs of `get` and `set` operations can then be viewed as a variant of representing state as a tuple in which the tuple is itself treated as an (internal) object providing the appropriate collection of `get` and `set` operations. In this case, there is no distinction between the *form* of internal access to the state and external access to the object; all access is via operations applied to some object (in the case of access to state, these are the `get` and `set` operations applied to the internal tuple mentioned above). The only difference is that some operations are internal to the object, while other operations are visible to clients of the object.

Also, in a generalization of the usual notion of "state", *any kind of object* (not just a record or tuple) may be used as the internal state of another object, meaning that an object's state is represented simply by the set of operations made visible to the object's methods by this internal object. In this approach, the only difference between an object representing state and any other object is that access to a state object is restricted by the definition of encapsulation used by the object model being modeled (see Section 2.3; typically this means that access to state is restricted to methods of the object whose state is being represented[1]). This approach extends the unification of "object" and "state" provided by expressing all access in terms of operations by making the distinction between "object" and "state" one of access rights (*visibility* or *scope*), rather than one of *form*. Of course, an object used as "state" would, in turn, have its own state object, and so on. This "recursion" would end ("bottom out") when the state object is of some primitive type supported by the object model (or is not further decomposable for some other reason). Object models which represented state as tuples would be represented in this approach by using only tuples or records as internal objects representing state.

The advantage of this approach is its generality, which allows it to represent the forms of states used in existing object models. This generality can also be useful if the RISC model is used directly, or to define new object models. For example, it allows existing aggregate structures of virtually any form to be used as the states of objects. This is a considerable advantage in a heterogeneous environment, in which it may be necessary to use legacy data to represent the states of objects in a system.

2.2 Object Methods (Operation Implementations)

An object's *methods* are the programs which implement the operations that the object can perform when a message is sent to it. The methods can be considered as independent pieces of program text that are associated with the object in some way. The RISC model must provide an explicit mechanism for modeling object methods, so that it can describe the variations in how methods are handled in different object models. Most of these variations involve differences in how methods are associated with related methods, or with other parts of objects (such as object state). For example, in the *classical* object models, the abstraction of a request is that of a message being sent to exactly one object requesting that the object execute one of its operations. The message may include other objects as parameters, but the object acting as the *receiver* of the request is syntactically distinguished in some way, and the operation is considered as being defined as part of the receiver, rather than as part of any of the parameter objects. In the *generalized* object models, the abstraction of a request is that of an operation invocation having a number of objects as parameters, with *none* of the parameters being more significant than the others in defining the operation. The operation is considered as being defined either independently of its argument objects or classes, or as part of *all* of its argument objects or classes [FKMT91]. Other variations could also involve different ways of modeling the methods themselves.

The approach described in [MH93] for modeling methods in the RISC model is to model a method as a distinct object (e.g., of type `method`) having an `invoke` or `apply` operation is its basic behavior. The `apply` operation takes a set of input parameters, and denotes a request to execute the method. Other operations would also be specified for method objects to define the other aspects of methods required in different object models. For example,

---

[1] A further generalization would be possible by allowing *multiple* internal objects to represent object state, provided that some way of associating these multiple objects with the single object whose state is being represented were defined.

attributes defined for method objects (specified as `get` and `set` operations), together with attributes defined for other RISC primitives, would be used to define the relationships between method objects and other primitive components that together form objects in a given model. (As noted earlier, the primitive RISC objects that make up a conventional object form a *graph* in which the nodes are state, methods, and other primitive objects, and the edges are the relationships between these primitive objects defined by the object model being represented.) The generalized operations found in some object models could be defined as aggregates of these method objects. The CLOS metaobjects representing generic functions and methods described in [KRB91] illustrate these ideas.

Defining methods as objects with specified interfaces allows flexibility in how methods are associated with other RISC primitives to form objects in a given object model. For example, [MH93] showed how method objects could be combined with other primitives to represent objects in both non-class-based and class-based classical object models, as well as objects in an object model based on generalized operations.

The inclusion of explicit method objects in the RISC model also has other advantages:

- it allows representation of existing programs as objects, either as methods of specific objects, as generalized operations, or as first class objects in their own right[1], which helps in reusing legacy code

- it allows flexibility in how methods are actually implemented, e.g., in a procedural fashion as in C++, in a declarative fashion as in logic programming [KL89a], or a query language, or as an extensional table [MD89]

- it allows flexibility in how programs can be composed to form higher-level behavioral aggregates, such as generalized operations, extended transactions, or workflows [GH92, HG93].

2.3 Object Interfaces

An object's *interface* defines the object boundary that separates the "inside" of the object from the "outside". Conventionally, the interface specification determines which aspects of the object are visible to clients of the object, and which aspects are invisible. The technique of minimizing dependencies among objects by defining strict external interfaces is often referred to as *encapsulation*. The external interface of the object serves as a contract between the object and its clients. An object is encapsulated if clients are restricted by the definition of the language or model to access the object only via its defined external interfaces. There are various forms of object encapsulation. For example, in some object models, all state is private, while in others some is public and some is private. Similarly, in some object models all methods are public, while in others some are public and some private.

The RISC model must provide a mechanism for modeling object interfaces, so that it can describe variations in how interfaces are handled in different object models. These variations are of three general types:

[1] The full development of this facility in the RISC model will involve further work on defining method *types*, in order to separately represent both methods, and the *operations* of which they are implementations.

- what interfaces exist within the model (e.g., whether objects have one interface, or multiple interfaces)

- how different interfaces are associated with different clients (or other objects operating in specific roles, such as class objects), if multiple interfaces exist

- what the characteristics of an interface are (e.g., whether the interface consists entirely of operations, or also includes concepts such as attributes or relationships).

The approach described in [MH93] for modeling interfaces in the RISC model involves two mechanisms. Some interface specifications, such as user-defined type or class specifications, or the multiple interfaces defined in models such as OLE 2 [Bro94], would be modeled as explicit RISC primitives. Other interface specifications, such as the internal interfaces provided by class objects to subclass methods, would be represented in the interfaces and behavior of the corresponding RISC primitives, as well as the relationships defined among them. What a given interface in a model conceals or makes visible, and to whom, is defined by scope rules in the language or model. These rules are also associated with specific language/model constructs (boundaries), such as blocks in block-structured languages, or object boundaries in object-oriented languages. The behavior defined for the RISC primitives would reflect these rules.

2.4 Identity and Object Identifiers

An object's *identity* is that property of an object which distinguishes it from all other objects. In the ideal *consistent object system*, as defined in [KC86], no two district objects in the system can have the same identifier (each identifier is unique) and for each identifier in the system there is an object with this identifier (there are no dangling identifiers). Generally, object identity is implemented by means of *object identifiers (oids)* assigned explicitly by the system. These values (or objects) serve as references to objects, and support some form of dereferencing operation in an addressing mechanism. Object identifiers are used in messages to indicate the object to which the message is being sent. Object identifiers are also passed as parameters in messages as references to argument objects and, within the state of an object, object identifiers of other objects serve to represent relationships with those other objects.

The RISC model must provide a mechanism for modeling object identifiers, so that it can describe the variations in how they are handled in different object models. Examples of variations include:

- whether referential integrity is maintained for object identifiers (whether objects can be explicitly deleted, or whether automatic garbage collection is implemented)

- whether object identifiers are distinct from the objects they reference

- whether identifiers are associated only with objects, only with other constructs in the language (such as interfaces), or with many different constructs.

The approach described in [MH93] for modeling object identifiers in the RISC model is to model them as distinct objects (having distinct types). This approach allows the association of identifiers with objects or other constructs to be explicitly modeled. It also allows the definition of specialized operations to construct identifiers, and other related operations (such as equality predicates) that represent the semantics of identifiers in different object models.

2.5 Messages (Requests) and Execution Model

In addition to modeling the *structural* aspects of different object models, the RISC object model must also be able to model the *behavioral* aspects of different object models. In a conventional object model, most of the behavior of interest involves how messages are handled. However, many variations exist in how the messages are handled, e.g., how the object determines what to do, and when (and whether) the object responds, in different object models. For example, object models differ in their support for synchronous or asynchronous messages and object execution. Similarly, conventional class-based object systems include built-in mechanisms for dispatching which describe, among other things, how the object model's inheritance hierarchy is traversed to find the proper method to respond to a particular message, and which differ considerably among object models. A different form of dispatching is used in the prototype-based object models that support delegation. An object model's *execution model* describes these and other aspects of computation within the object model. The RISC model must be capable of describing these variations in the execution models of different object models.

The approach described in [MH93] for addressing this is to define an appropriate set of operations for each of the types of primitive objects, together with an overall flow of control that ties these operations together. The combination of the operations of the individual primitives, together with the steps in the overall flow of control, can then be used to describe the execution models of various object models. The primitive objects defined by the RISC object model constitute *metaobjects* (objects that describe objects), and the operations defined for these objects, together with the overall flow of control, constitute a *metaobject protocol*. In order to be able to describe the execution models of various object models, it is necessary to have explicit representations of messages, any aspects of objects that have to do with message handling, and other aspects of the execution model.

The approach described in [MH93] for modeling messages themselves is to model them as explicit objects. Message queues and other special aspects of active objects can be added as additional attributes to subtypes of the pre-defined metaobjects that specialize the standard ones to model the variations found in different object models.

[MH93] also noted the potential need for a basic *activation* primitive (a primitive process or thread concept). A single such activation could be used to model synchronous models, and activations could be associated with individual objects to model active objects (or *actors* [Hew77, AH87]). Alternatively, an activation could be associated with each object as the default, with a specialization having no activation (the idea being that the active object is the general case). This issue is discussed further in Section 2.8. Several object models that include such concepts are described in Sections 3.4 and 3.5.

[MH93] also noted that, for modeling the execution model, both procedural and event-based approaches are being considered in the RISC model. In either approach, key events (e.g., such basic elements in the control flow as the start of a method execution) would be modeled explicitly, either as events, or as invocations of operations in the metaobject protocol. The specific approach will be chosen on the basis of further study (keeping in mind that modeling both is necessary). The various models offering explicit metalevel control are similar in offering the opportunity to alter the control (and therefore fundamental aspects of the model) by altering the explicit representation of the control definition. The difference between the event-based models and those based on a procedure are primarily notational. An event based model can be simulated using a procedure-based one (the procedure contains steps for dealing with messages representing "events"; the model contains constructs such as queues for holding them, etc.) and vice-versa (as illustrated by the model of [JS92], which is described briefly in Section 3.6.4).

Metaobject protocol concepts, and related issues of how to model messages and message passing, are a major subject of Section 3 of this report. Experiments performed with GTE Laboratories' DOM 3.0 prototype [NH93a,b], which implements some early variants of RISC concepts, illustrate the flexibility and power afforded by the ability to explicitly represent messages and message passing (dispatching).

2.6 Types and Type Construction

In object models, the concept of a *type* or *class* plays some, if not all, of the following roles:

a. it defines a collection of objects having exactly the same external behavior, in terms of an external interface specification that can be used for type-checking.

b. it defines a collection of objects that have exactly the same implementation, in terms of a specification of the internal data structures and code that implements the operations.

c. it defines an object existing at run-time that acts as a "factory" for creating new objects, and that holds the operation implementations shared by the collection.

The concept of class or type in many object models combines all these roles. On the other hand, there is an increasing tendency for object models to provide separate facilities for the roles identified above. In such models, a *type* is only a *specification* of the external interface to be supported by a collection of objects (role a), typically in terms of a set of operation signatures. Separate implementation specifications are then used to define particular implementations of objects providing that interface (roles b and c). (Sometimes these implementation specifications are referred to, confusingly, as *classes*). This separation of type and implementation allows heterogeneous implementations of objects providing the same capabilities to co-exist within the system, and allows objects, provided they maintain existing interfaces, to add new capabilities, or alter existing implementation decisions, without affecting clients. This is an important advantage in a DOMS environment.

The RISC object model must provide explicit mechanisms for modeling object classes or types, in order to represent the many variations in object type and implementation (class) mechanisms found in different object models. These variations involve differences in how types or classes are defined, differences in how objects are associated with types or classes, and differences in relationships between types or classes (see [MH93] for a discussion of these differences). The ability to describe the type systems of different object models is critical in using the RISC model to address certain interoperability problems.

The approach described in [MH93] for modeling the *specification* aspects of types in the RISC model is to model types as distinct objects (of type `type`). The specification of object implementation is considered separately (see Section 2.7). Basic behavior supported by `type` objects would include an operation that returns the signature (specification) of the type (e.g., the operation signatures), and a type-checking operation that determines whether a given object satisfied the type. Examples of these ideas are found, e.g., in the Emerald language [RTLB+91]. Specializations of `type` objects could also define other operations, e.g.:

- operations that return a set of the type's subtypes or supertypes
- operations that determine whether another type is a subtype or supertype of the type
- operations that return the set of existing instances of the type (this is sometimes called the type's *extent*, and is frequently maintained in object database systems)
- operations that return the type's name or other attributes

Differences in relationships between objects and types, such as conformance [RTLB+91] versus explicitly creating objects as instances of specific types, can be represented in different versions of the type-checking operation, as well as differences in the state maintained by individual objects (e.g., recording what its type is). Similarly, differences in relationships between types, such as conformance-based versus explicit subtyping, can be represented in different versions of the type-checking operation, and the operations that determine whether another type is a subtype or supertype of the type.

Operations would also be provided to serve as object *type constructors*. These operations would take a set of signatures (and possibly other specifications), and do whatever is necessary to produce a `type` object. Such operations would simply be a special kind of object constructor, as discussed in the next section.

[MH93] noted that the RISC model also needs some way of describing when type checking takes place. This requires some explicit modeling of the translation of expressions, and of what kinds of errors objects are prepared to deal with. The distinction between compile-time and run-time may translate to distinct scopes described by rules about when such things as types and expressions are available for checking, and special behavior (such as type checking) that is available within some scopes and not within others. For example, [NN92] describes a formal framework in which the type notation of a functional language is extended to include a notation for describing different binding times (run-time and compile-time).

The approach described in [MH93] for modeling object construction (including implementation specification) in the RISC model is described in the next section. It is conceptually separate from the way types are modeled. This allows the direct modeling of object models that separate specification of object types and specification of object implementation. It also allows the modeling of object models that combine aspects of types and implementation by associating specific object implementation mechanisms with object construction operations defined for types, and basing those implementation mechanisms on the type specifications where appropriate. For example, the representations of class hierarchies can be constructed using the object construction techniques described in the next section. These relationships can be declared using type specifications, with the constructors using that information to build the appropriate implementations.

2.7 Object Construction

An object model generally provides a way to construct new objects. As a result, the RISC model should include general mechanisms for describing object construction, in order to model the variations in object construction found in different object models. Object construction in RISC should cover not only the construction of objects visible in a given object model, but also construction of the RISC primitives making up those objects, e.g., objects representing methods and identifiers.

In *class-based* object-oriented languages, an object is created by invoking an operation (often called `new`) defined for a class or type object. The operation uses the class or type specification as a template in constructing the new object (if the model supports inheritance, this template is partially derived from superclasses or supertypes). Smalltalk and Eiffel are examples of this approach. Some object models separate type and class (implementation) specifications. In this case, the `new` operation must either be defined for the implementation specification or, if defined for the type specification, must take the identity of the particular implementation to be used as a parameter, in order to identify which implementation is to be used in constructing the new object. On the other hand, in *prototype-based* languages such as Self, objects are "cloned" from existing objects. In this case, an existing object is specified which acts as the template for constructing a new object. Other object construction mechanisms are also used. For example, object creation in the Emerald language [RTLB+91] is done using an expression called an *object constructor* which, when evaluated, creates a new object.

The approach described in [MH93] is to define `new` operations to serve as object constructors for each of the RISC primitive object types, such as state, method, and other types (possibly based on a primitive, Emerald-like mechanism). Object construction operations in different object models would then be defined as different patterns of these primitive `new` operations creating the appropriate structures of RISC primitives to represent those objects.

For example, using the RISC approach, Emerald object constructors could be represented directly, in terms of specific collections of RISC primitives. The more complex types of object construction mechanisms found in other object models could be modeled by using RISC `new` operations in combination with other features. For example, a class-based implementation could be represented by defining a `new` operation for the class. This operation would invoke `new` operations for the appropriate collection of RISC primitives to form the instances, using the class definition itself as a template. In class-based models, this involves constructing the appropriate relationships between the instances and the class object. Object construction in prototype-based languages could also be represented using this approach by defining the `new` operation so that it uses the structure of the object itself as a template to clone new instances. This would also involve constructing a relationship between the new object and the object from which it was cloned. The built-in type constructors found in various models, such as tuples and sets, could also be defined as specific patterns of predefined RISC objects (e.g., the `get` and `set` methods required to access a variable in a tuple). Finally, the various forms of inheritance found in different object models could be represented by differences in the RISC-defined construction of the class objects themselves, together with the behavior of the `new` operations defined for them.

2.8. Other Object Model Facilities

[MH93] also discussed the need to model other object model facilities, such as the queries and transactions found in object-oriented DBMSs. Transactions are of particular interest here because they introduce the general issue of how to describe any *concurrency* that may be provided in the object model.

[MH93] noted that RISC concepts, particularly those related to the description of object execution models, can provide the basis for describing (and possibly defining) object transaction mechanisms. In conventional DBMSs, data sharing is controlled through the DBMS's concurrency control mechanism, which provides facilities to ensure that users do not interfere with each other while reading and writing data. In these systems, the atomic units of computation are *transactions*. The system may interleave the execution of several transactions (to improve resource utilization and throughput), but it guarantees that the interleaved execution is *serializable*, i.e., equivalent to a non-interleaved execution of the transactions. Various mechanisms, based on locking and timestamping, have been devised to synchronize accesses to shared data in order to guarantee this property [BG81]. The language syntax used to access the database allows the DBMS to determine the beginning and end of transactions (either by defining any given command as a transaction, or through the use of explicit commands (e.g., `commit`) indicating the beginning and ending of the transaction, and what data is read or written by the transaction (although the latter often cannot be determined prior to run time).

The RISC-based description of primitive object events in a given object model's execution model (including such events as invocation of object operations and transaction control operations such as `commit`), and compositions of such events, together with RISC object construction facilities, could provide the basis of a merger of the RISC model and similar (in some respects) approaches, such as ACTA [RC92] and the TSME of [GH92, HG93], for describing different *transaction* models (including both conventional serializability-based models and more advanced transaction models that apply, e.g., to workflows) in terms of a few basic primitives. Section 3 will describe how reflective facilities in object models, such as those proposed in the RISC model, can be used to perform this integration of object model and transaction concepts.

Conventional transaction facilities effectively conceal the existence of concurrency from the object model itself (although some of the extended transaction models considered in [GH92, HG93] begin to change this situation). Dealing with concurrency explicitly in the object model changes the situation considerably, as far as describing the object model is concerned. The object models of conventional object-oriented languages and systems present a fairly conventional view of object operations. In such models, a computation is invoked by some individual command (such as the `doIt` menu item provided in the Smalltalk user interface) to execute an object method. This method in turn may send messages to other objects to execute their methods. These messages correspond to subroutine calls, in that they are executed *synchronously* (the sending object waits for a return from the message before it continues). These calls and responses continue until the original method has completed executing. Moreover, the objects in such systems are *passive*: that is, they must receive a message before performing any action, just as a subroutine must receive a call before executing.

Other object-oriented languages and systems attempt to extend this view of object operations to include *active* objects [Nie87, EG89]. These are objects that can execute methods concurrently, and that can (from the point of view of other objects) initiate actions asynchronously, without having received a message. The basic motivation for modeling concurrent object execution is that the real world consists of activities that take place in parallel, and of objects that can, in some circumstances, appear to be "active" in the sense used above. Thus, being able to model these aspects of behavior allows a more natural and modular system design. Such a model also allows the use of parallelism among components in order to gain performance, and the use of shared information to reduce redundancy. In particular, a major issue in distributed object management is the development of techniques to support efficient coordination, sharing, and control of concurrent (parallel) processing activities within the system, since a distributed system invariably introduces the potential for such activities. Such techniques include both explicit concurrency in the languages used to develop objects, and transaction facilities that can be used to synchronize the operations of objects developed in non-concurrent languages.

Programming languages that can express concurrent activities, or processes, and communication among them, are referred to as *concurrent* (or sometimes *parallel*) *programming languages*. *Processes* are abstract entities that generalize the concept of the execution flow of control of sequential programs. The actions a process can take include interprocess communication, change of state, creation of new processes, and termination. Thus, object-oriented programming languages that support concurrent object execution are frequently referred to as *object-based* (or object-oriented) *concurrent programming languages* [AWY89]. One of the goals of this report is to consider any additional RISC concepts that may be needed to model such concurrent object models. For example, Section 2.5 noted that a basic activation primitive, such as a primitive *process* or *thread* concept, was under consideration.

There are two classes of issues to be addressed in describing the additional characteristics of active objects. One class of issues is the *external* characteristics of active objects (how they appear to other objects, and what facilities are available to other objects to affect processing taking place within the active objects). The other is what they look like *internally* (what structures, such as queues, have to be provided to support them, what sort of process management takes place both within and between objects, etc.). The need to explicitly describe these various aspects of active objects, particularly their execution aspects, can create the need for additional RISC components beyond those identified already. These issues are explored further in Section 3.

2.9. Comparison with X3H7 Object Model Features Matrix

As noted in Section 1, the X3H7 (Object Information Management) standards committee is preparing a "Features Matrix" which describes many different object models in terms of a common set of concepts [X3H793a]. It is expected that describing object model features in terms of a common set of concepts will assist in reducing differences among object models, and also in defining mappings between specific object models. It is natural to compare the set of concepts being defined by X3H7 and those of the RISC model, because many of the reasons for defining the features matrix are the same as those for defining the RISC model. In this case, the "features" of the Features Matrix correspond roughly to "primitives" in the RISC model. At the moment, the "features" are expressed informally, and there is considerable overlap among them (i.e., they do not constitute a minimal set of concepts required to describe object models), a situation explicitly recognized by X3H7. However, it is intended that this overlap be largely removed and, as noted in Section 1, ultimately these features may be described formally. In fact, it is anticipated that work on the RISC model will contribute to X3H7's work in formally describing object models. Also, the current set of features may change, as additional object models (such as the concurrent models described in Section 3 of this report) are considered.

Table 2.1 shows the current set of "features" from the X3H7 Features Matrix. It should be noted that some of these, such as "intended use" are not object model primitives, and thus would not correspond to any RISC model primitive.

0. Intended Use
1. Basic Concepts
2. Objects
    - 2.1 operations
    - 2.2 requests
    - 2.3 messages
    - 2.4 specification of behavioral semantics
    - 2.5 methods
    - 2.6 state
    - 2.7 object lifetime
    - 2.8 behavior/state grouping
    - 2.9 communication model
    - 2.10 events
3. Binding
4. Polymorphism
5. Encapsulation
6. Identity, Equality, Copy
7. Types and Classes
8. Inheritance and Delegation
9. Noteworthy Objects
    - 9.1 relationships
    - 9.2 attributes
    - 9.3 literals
    - 9.4 containment
    - 9.5 aggregates
10. Extensibility
    - 10.1 Dynamic
    - 10.2 Metaclasses/Metaobject Protocol
    - 10.3 Introspection
11. Object Languages
12. Semantics of Base Classes (+ type constructors)
13. Background and References

Table 2.1. X3H7 Object Model Features

Table 2.2 shows the basic features from the features matrix, and the corresponding RISC components. The table illustrates that there is substantial agreement on the basic concepts. Of course, considerable work remains to be done on the details of these general concepts. There is also substantial agreement between these concepts and the basic concepts described in the ISO reference model for Open Distributed Processing (ODP) [ISO93]. The ODP model is somewhat more abstract, at least as currently defined, than either the RISC model or the X3H7 features. However, it is likely that these models will converge, at least to some extent. In particular, the ODP model includes a number of valuable concepts for describing different concrete object models at a high level of abstraction, such as type concepts, which have already contributed to ideas in the RISC model.

Features Matrix Entry	RISC Component
identity, objects	identity/identifiers
methods	methods
messages, requests, operations	messages
state	state
encapsulation	interfaces
types and classes	object construction, types
communication model	messages, other facilities
extensibility	execution model, metaobject protocol
object lifetime	object construction

Table 2.2. X3H7 Features Compared with RISC Components

Section 2 has reviewed the primitives identified for the RISC object model in [MH93], and briefly compared them with object model features identified by X3H7. Section 3 will describe relevant work on *reflective* object models, in which object model primitives are also identified. These primitives provide another source of ideas for the components that should be contained in the RISC model.

3. EXAMPLES OF REFLECTION AND METALEVEL CONCEPTS IN OBJECT MODELS

Section 2 described and motivated the basic concepts previously identified for the RISC model, as described in an earlier report [MH93]. Those basic concepts had been identified based on a survey of a wide range of different object models, with the goal of obtaining a set of primitives capable of representing these different object models. This section reports on a survey of object models and languages from a different direction. Specifically, this section describes a number of object-oriented languages or systems that either explicitly support reflection (meta-level concepts), or that contain aspects related to reflection that address issues from the previous report that required additional work. The material describing each language or system is, for the most part, a paraphrase of material from the cited papers. In addition to material from the cited papers, commentary is provided on the various features identified to relate specific features to issues of practical interest. The intent of this survey is to look at what designers who are explicitly trying to provide reflection (and thus are explicitly trying to identify the key primitives in their own languages) are actually identifying as key concepts. This is not intended to be an exhaustive survey, but it is intended to cover a sufficient range of different models to get a general idea of some of the key issues and primitives, in order to contribute to a better understanding of the concepts that the RISC model should contain. The papers reviewed here, together with workshops such as [Ibr91], should also suggest the increasing amount of research being done on reflection and metalevel concepts in object models.

The rest of this section is organized as follows. Section 3.1 describes the basic principles of computational reflection, principles which will be illustrated by the different languages described in the rest of Section 3. Section 3.2 describes reflection in *class-based* languages, a category that includes some of the more familiar object-oriented languages, such as C++ and Smalltalk. Section 3.3 then describes reflection in *prototype-based* languages, which use different approaches for object creation and inheritance then those used in class-based languages. The goal of Sections 3.2 and 3.3 is to examine reflection in important categories of non-concurrent object-oriented languages. Section 3.4 describes reflection in languages based on the actor model. These languages introduce explicit concurrency and asynchronous message passing. Section 3.5 then describes reflection in several concurrent object-oriented languages that are not based on the actor model. The goal of Sections 3.4 and 3.5 is to examine how reflection is handled in different kinds of concurrent languages. Finally, Section 3.6 describes some additional examples of reflective concepts, or concepts related to reflection, that do not necessarily fit cleanly into one of the earlier categories.

3.1 An Overview of Computational Reflection

In a conventional object-oriented system, computation is performed by objects that represent (or model) entities which are *external* to the system (its domain). The system is *causally connected* to its domain if the objects and the domain entities are connected such that changes to one side lead to corresponding changes to the other [Mae87]. For example, a network management application (the system) might contain objects that represent the various network components, such as switches and trunks, to be managed (the domain). The network management system is causally connected to its domain when:

- changes in the status of the physical network components trigger corresponding changes in the states of the objects representing the components in the system, and

- changes made to objects in the system (e.g., by management applications) cause corresponding changes in the states of the actual network components represented by those objects.

Computational reflection is the activity performed by a system when doing computation about (and possibly *affecting*) its own computation. Most systems, in addition to computation about the external problem domain, include some instances of reflective computation. Examples include recording performance statistics, recording information for debugging purposes, interface-related computation (e.g., tracking mouse position), control-related computation (determining what process to execute next), self-optimization, self-modification (e.g., response to failures by bypassing failed components), and self-activation (monitors or daemons). However, in a conventional computational system much of the reflective computation that is done is hard-wired into the system.

On the other hand, a *reflective* (object-oriented) *computational system* contains objects which represent (or model) its own structural and behavioral aspects (often referred to as *metalevel* objects). The objects making up the system's self-representation must be accessible and modifiable within the system itself, and more importantly, changes made to such objects must be causally connected to the actual computations being performed. More specifically, a reflective object-oriented computational system is an executing system of programmed objects that makes aspects of the system such as method invocation, interfaces, inheritance, and object implementations explicitly available as objects that can be the subjects of computation.

The motivation for providing built-in reflective capabilities in object-oriented languages and systems is to provide a general framework for programmers to extend or tailor the language or system to specific application requirements in a modular way. Distributed computing provides a simple example of a requirement for language extension. In most imperative languages for distributed computing, procedural calls are extended to support remote communications. These remote procedure calls provide extended functionality for marshaling and unmarshaling arguments and results on client and server machines, dealing with network communications, invoking the target procedure at the remote machine, etc. However, such extensions are typically provided as pre-defined packages. Reflective or metalevel techniques can be used to allow programmers to provide these and other extensions to method calls (and other aspects of the system) for their own purposes.

Another example of the use of reflection in a distributed context is the task of monitoring the behavior of a running object. It might be desired that, each time a given object accepts a message, this fact is reported to a monitor object. One approach to doing this would be to modify each method of the running object. However, a reflective system would allow this to be done by modifying the single metalevel script that controls the handling of all messages by that object.

[Fer89] notes that some of the key features of reflective languages are their ability to:

- describe aspects of their own implementation, such as allocation of objects and garbage collection, and support incremental extensions, or even re-implementation of the whole system

- monitor aspects of the system, including inspection, analysis, and dynamic modification of the code within the language itself

- support interfacing and debugging, including implementation of programming tools within the language

- reorganize themselves, including the ability to learn about the system's behavior, and make adjustments to improve efficiency or consistency.

As described in [Fer89], defining a reflective system involves relating the entities of a computational domain D_n, called the *base level*, with those of another computational domain, called the *metalevel*, which describes the base level. Each domain can serve both as a base domain for a higher level, or as a metadomain for a lower level, except for one domain D_0, which can only be used as a base level.

Models of reflection are defined using *reflection equations*, which express how entities and expressions of the base level are described at the metalevel. As a result, reflection equations act as *semantic* equations, i.e., they define the semantics of the lower levels in terms of upper levels. This explains the relationship between the RISC model and computational reflection, since the RISC model is intended to describe the semantics of concepts in various object models in terms of a different set of (more primitive) concepts. This also brings up an important point about a source of possible confusion. A metalevel is usually considered to be a *higher* level of abstraction than the base level it describes. However, in a reflective system the metalevel description is often expressed operationally, in terms of an *implementation* of the base level, and an implementation is usually considered a *lower* level of abstraction. In the discussion below, lower levels are considered to be described by, and implemented by, higher (meta) levels, even though the metalevels may be defined using what appear to be implementation-level constructs.

The traditional model of reflection is based on *interpretation* (although in many cases it need not be actually implemented that way). The domain D_n consists of a set of entities and expressions written in a language L_n, and an interpreter I_n which interprets those expressions. The interpreter is written in a language L_{n+1} and is interpreted by an I_{n+1} interpreter. In a *non-reflective* system, the language L_{n+1} is *different* from L_n. For example, if L_1 is Lisp, L_2 (the language in which the Lisp compiler or interpreter is written) might be a language like C, L_3 the machine language instruction set, and so on (e.g., if the machine instructions are microcoded).

In *reflective* architectures, on the other hand, there is a virtually infinite chain of *identical* languages L_i, often called the *reflective tower*. This is possible because the interpreter I_i of L_i is defined as a *meta-circular interpreter*. That is, I_i is itself written in L_i, is represented within L_i, and is used to interpret itself. The representation of I_i within L_i minimally consists of a name for the interpreter (such as `eval` in Lisp) plus some interpreter data (such as the list of bindings and the continuation in Lisp). This requires that the language allow for programs to be viewed as data structures in the language [Mae87]. In addition, there is one non-reflective version of the interpreter, say I', written in a language L' different from L_i, which is used to end the regression and replace the interpreter I_n at the highest level of computation needed. In a language based on this approach, when the system encounters a reflective operation at level D_n, it shifts up one level to D_{n+1} and starts another computation in order to interpret the lower level operation. If there is a call to a reflective operation at that level, it goes one level higher, to level D_{n+2}, and so on, until it finds a reflective operation which does not use any other reflective operations. All non-reflective operations are interpreted by I', written in another language (technically, this "other language" could also be a non-reflective version of the *same* language; the goal is to ensure that there is one level at which reflection is guaranteed *not* to occur). This ensures that reflection eventually terminates, because no infinite loop exists in the definition of reflective functions. An approach often taken to make reflective implementations more efficient is to drastically limit the reflective capabilities to simple ones, such as message dispatching, that can be efficiently implemented. Another approach used in some languages is to allow the definition of objects which lack individual reflective capabilities, and thus can be implemented using conventional techniques. In addition, in most practical cases relatively few levels of reflection are actually used.

The first step in reflection is the representation of specific aspects of the object-oriented program or system as objects within that program, a process referred to as *reification*. The resulting objects are referred to as *metalevel objects* (or *metaobjects*), because they represent instances of concepts describing the program, rather than concepts of the program itself. Decisions regarding the aspects of the system to be reified as metalevel objects constitute key design decisions in a reflective system. These aspects include such things as the classes of objects, the inheritance hierarchy, and the internal structure of objects.

The metaobjects representing reified aspects of the system must be defined so as to provide operations that support *reflective computation* and *reflective update*. The operations defined for these metaobjects constitute the system's *metaobject protocol*. Invoking an operation of a metalevel object (such as the `send` operation on a `message` object) constitutes reflective computation. If the reflective computation updates reified objects, this constitutes reflective update (i.e., the result of the reflective operation is reflected back into the base-level computation). The metaobject has a causal connection with the aspect of the system it represents; a metalevel operation performing a reflective update will modify that aspect of the system. Other key design decisions in a reflective system are how the causal connection (the relationship between a base level and its metalevel) should be implemented, and when the shift to a metalevel should take place. In an object-oriented language, where the basic concepts are objects and messages, these issues become, respectively, how the handling of messages and lookup of methods described at the metalevel should be related to base-level message passing, and when the system should use metaobjects and/or metalevel communication [Fer89][1].

To sum up, in an object-oriented reflective language, a running program consists of a set of objects. Some of these are ordinary end-user objects, while others are metaobjects used by the system to actually run the program, such as method objects, selectors reified as symbol objects, object vectors used to pass arguments to methods, and other metaobjects that can be created by users to modify the behavior of base-level objects. In some cases, these other objects need not actually be materialized, or may be materialized only on demand. However, the language or system must operate as if they explicitly existed[2].

Object-oriented languages are particularly well-suited to the implementation of reflection. This is not only because of the distributed nature of data and procedures in objects, but also because many of them implement aspects of reflection (such as class information) in objects already, even in languages that are not explicitly reflective. Reflection is increasingly being investigated in the context of object-oriented systems as a way of constructing flexible, large-scale systems, because it offers a built-in mechanism for incorporating user-defined customization into the system. In this sense, reflection resembles inheritance, which also provides a way for users to incrementally customize existing objects, and which is widely viewed as a key facility in object-oriented systems.

---

[1] Some of the points being made here may seem fairly abstract. Hopefully, they will become clearer during the discussion of specific examples of reflective systems in the subsequent sections.

[2] Messages in Smalltalk are an example of such "objects". They are defined as if they were objects, but are only explicitly materialized as objects in special circumstances, in order to reduce overhead.

In addition, object-oriented techniques provide a way to address an issue that is frequently noted in reflective programming, namely how to simultaneously balance the objectives of:

- providing sufficient freedom for programmers to make useful customizations of the language or system, and
- restricting the generality of the customizations allowed so that:
    - the compiler and other system components can have enough information about implementations so that useful optimization can be done, and
    - the implementation is protected from errors in customized end-user programming.

This requires careful design of the behavior defined for the various metaobjects (the metaobject protocol) so that aspects of the implementation are selectively revealed and made changeable, while other aspects are held constant. The need for careful design of the metaobject protocol is emphasized in [KRB91].

As noted above, choosing what aspects of the system should be reified as metalevel objects constitute key decisions to be made in a reflective system. This point establishes another connection between work on computational reflection and the RISC object model. Investigations of computational reflection in different object-oriented programming languages invariably involve identifying the key concepts of the object models of those languages that must be reified as metaobjects. Moreover, this reification requires identifying the behavior of those metaobjects as well. These aspects are exactly those that would be required in a RISC model implementation of those object models. In fact, the RISC model is intended to be a reflective representation of different object models.

There is, however, a difference between the RISC object model concept and the instances of object model reflection to be described below. In these models, the idea is to take a *single* object model and implementation, and extend it by creating runtime metaobjects that represent pieces of the implementation (and so can be altered by the user/programmer). In the case of the RISC object model, there is no single object model that is to be implemented using metalevel facilities. Instead, what is to be provided is a sufficiently-general set of metaobjects so that a very wide *range* of object models can be defined in terms of them. This is why the selection of the individual primitives is less immediately obvious than it might be if only a single object model were involved, and why a range of object models must be examined.

It should also be noted that the difference between the RISC model and a particular reflective model is *relative*, especially in the case of a fairly primitive object model such as the actor model described in Section 3.4. This is because, even in the RISC model, *some* minimal set of assumptions that define a basic computational model must be made (e.g., assuming a basically synchronous or asynchronous one), with the result that a particular (even though primitive) object model has been defined. It is possible to imagine multiple RISC models based on different assumptions at this level. At the appropriate time, one of these will be chosen, with the understanding that the decision would necessarily be somewhat arbitrary, and could have been done differently.

3.2 Examples of Reflection in Class-Based Languages

This section describes reflection in class-based languages, a category that includes some of the more familiar object-oriented languages, such as C++ and Smalltalk. In such languages, objects are defined as members of one or more classes or types[1]. The class defines the internal form of every object of the class, as well as defining the operations (protocol) associated with objects of the class. In some of these languages (Smalltalk is an example), an object class is itself represented by a *class object*. The abstraction provided by an object-oriented language is that each object is a self-contained entity having its own operations. However, in a class-based language, the class object stores, among other things, the code for the methods of all objects of the class, so they need not be duplicated in each individual object. (C++ also provides for the sharing of method code among the objects of a given class, although an explicit class object is not used for this purpose.)

3.2.1 Open C++

[CM93] describes Open C++, a C++ implementation which uses reflective techniques to allow programmers to extend the C++ method call mechanism. Many different types of language extensions can then be made using these extended method calls. In Open C++, the implementation of the method call (dispatching) mechanism is exposed to programmers as a *metaobject,* which acts as an explicit representation of the object's method call mechanism. This metaobject is almost the same as a normal object, but its behavior defines how the system responds to a user-level method call. An object at the *base level* (i.e., an ordinary programmer-defined C++ object) has a metaobject at the *metalevel*, and the execution of the base object's methods is controlled by its metaobject. If a method of the base object is invoked, a specific method of the metaobject is used to process the method invocation, instead of always invoking a default implementation defined by the compiler. Since the metaobject is also defined as a C++ object, the programmer can alter the implementation of the method call mechanism by defining another metaobject and substituting it for the one that would normally be generated. Thus, alterations to the language's method call mechanism can be made within a user program, rather than requiring alterations to the compiler.

Open C++ provides a very simple *metaobject protocol* to make the method call mechanism extensible. Base level objects controlled by metaobjects are called *reflective objects*. Because control by a metaobject imposes some performance and memory overhead, the programmer can specify whether or not an object is reflective. A nonreflective object is compiled as a normal C++ object, without a metaobject. To distinguish reflective and non-reflective objects, a reflective object is identified by a different class name. If the class of an object that may be reflective is `X`, then the class of a corresponding reflective object is `refl_X`, which is defined to be a subclass of `X`.

To create a reflective object, the class of the object and its metaobject must be declared using special directives, which are specified as C++ comments that start with `//MOP`. This allows programs containing Open C++ directives to remain valid C++ programs. The declaration of a reflective object takes the form:

```
//MOP reflect class X : M;
```

[1] The distinction between "class" and "type", where there is one, is irrelevant for purposes of this discussion.

This specifies that an object of class `refl_X` is a reflective object controlled by a metaobject of class `M`. Class `M` is a normal C++ class that inherits from class `MetaObj`. To further extend its implementation of a method call mechanism, a metaobject can itself be a reflective object controlled by its own (higher level) metaobject.

Not all methods of a reflective object need to be controlled by its metaobject. Those that are are explicitly specified as *reflect* methods; the other methods are invoked according to the normal C++ method call semantics. An example of specifying a reflect method is:

```
class X {
public:
     X();
//MOP reflect:
     int func(int);
private:
     int p;
};
```

Reflect methods may have a *category name* (`int` in the example) to enable the associated metaobject to distinguish between methods that play different *roles*. For example, the metaobject may alter the execution of a method call depending on the category name.

As noted above, when a reflect method is called, its execution is controlled by its metaobject. A metaobject must inherit from class `MetaObj`, which defines the following two methods:

```
Meta_MethodCall(Id method, Id category, ArgPac& args, ArgPac& reply)

Meta_HandleMethodCall(Id method, ArgPac& args, ArgPac& reply)
```

`Meta_MethodCall()` defines how a base-level method call is processed, and is called whenever a reflect method is called. `Meta_HandleMethodCall` is used to actually execute a reflect method (this is the way the metalevel reaches down into the base level to invoke the code defined as the base-level method). To alter the implementation of a method call mechanism, the programmer defines a subclass of `MetaObj` in which these methods are redefined so that the metaobject acts in the intended way.

For example, Figure 3.1 shows a reflective object whose class declaration would have the form:

```
//MOP reflect class Foo : ExtendedMetaObj;
```

An object of this class has class `refl_Foo`, and a metaobject of class `ExtendedMetaObj` (a subclass of class `MetaObj`). In this case, the base object implements a reflect method `someop()`. The metaobject class `ExtendedMetaObj` redefines the `Meta_MethodCall()` method to perform some extended computations in addition to the normal method call processing, extensions such as printing or logging a message each time a reflect method is executed.

Suppose the reflect method `someop()` is called as part of the base level computation. At the base level, this appears to result only in the base-level call and return shown by the plain horizontal arrows in Figure 3.1. However, what actually happens is that the method `Meta_MethodCall()` is invoked at the metalevel (a *reification* of the base-level method call), as shown by the bold vertical arrows. The first argument of `Meta_MethodCall()` is bound to the integer identifier of the called method `someop()`, while the second argument represents the category name of the method `someop()`. The actual arguments of the call to `someop()` are passed as the third argument, `args`. Note that, at the metalevel, the actual argument list of a method call is a first-class C++ construct, since `args` is a normal C++ object (of class `ArgPac`). The argument `args` has the same interface as a stack, so that the programmer can access any actual argument stored in `args`. The programmer can also transfer `args` to another metaobject. Converting the actual arguments to an `ArgPac` object is a *reification* of those arguments.

`Meta_MethodCall()` then carries out its computations. The computations performed by `Meta_MethodCall()` include whatever extensions have been defined, together with a call to `Meta_HandleMethodCall`, which calls the base-level method to compute the actual return value requested by the base-level call (shown by the bold vertical arrows in Figure 3.1). `Meta_HandleMethodCall` takes a method identifier and an actual argument list, and returns the result value of the specified method. `Meta_MethodCall()` then stores the result of the base-level call into its reply argument (thus *reflecting* the meta-level computation into the base level). This result is then provided as a return value to the original caller of `someop()`.

[CM93] describes an example of defining a new metaobject that prints a message each time a reflect method of an associated object is called. This extension is implemented by embedding a `print` statement in the implementation of `Meta_MethodCall()` for that metaobject, followed by the call to `Meta_HandleMethodCall()` to actually execute the invoked method.

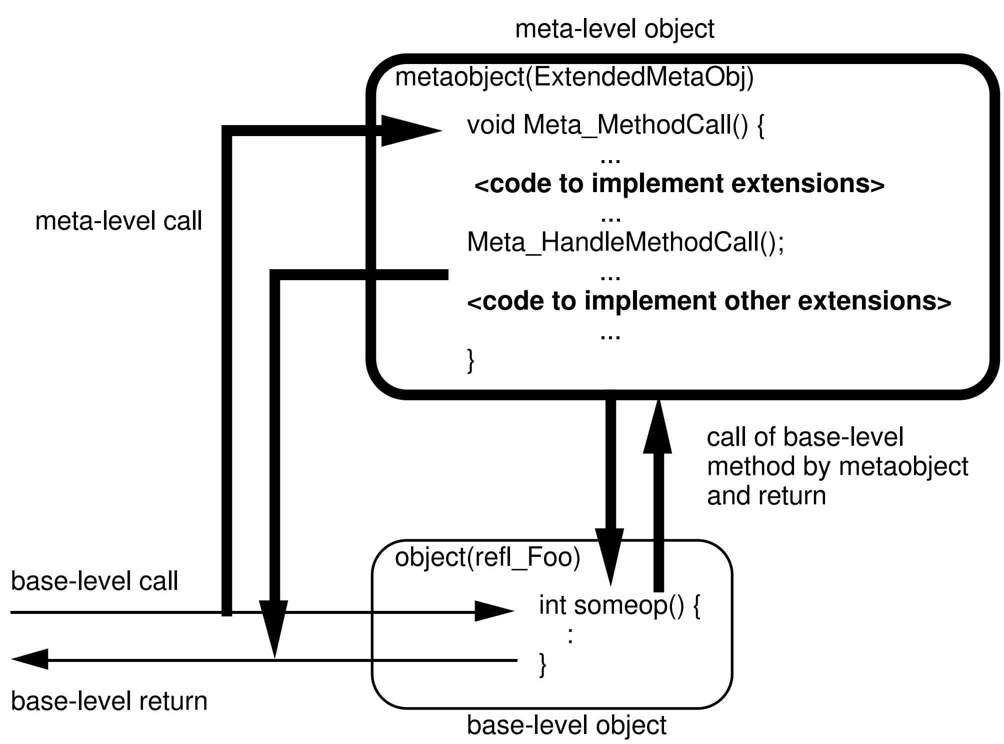

Figure 3.1. Method Call Extension in Open C++

Converting the actual arguments of a base-level method call to an `ArgPac` object is similar to the marshaling and unmarshaling of arguments in remote procedure calls. The class `refl_X` generated by the Open C++ compiler automatically redefines a reflect method so that it carries out this conversion to an `ArgPac` object, and then invokes the `Meta_MethodCall()` method of its metaobject.

As described above, Open C++ does not expose the implementation of a method call directly, but instead uses the abstract interface defined by class `MetaObj` and its associated `Meta_MethodCall()` and `Meta_HandleMethodCall()` methods. The default implementation of the method call mechanism, which is defined by the compiler, is described in assembly code. However, the programmer who wants to extend the method call mechanism can do so by providing a new implementation (in C++) of methods such as `Meta_MethodCall()` instead of writing assembly code (and instead of changing the compiler). When a method is invoked, the computation of the method call is *reified* to constructs available in the C++ program, and operations on those constructs are *reflected* to the actual base-level computation. This distinguishes systems based on metaobject protocols from open systems that directly expose their internal structure in order to provide extensibility. In Smalltalk, for example, the whole source code of the runtime system is available for modification, but such modifications are complicated, and can lead to the breakdown of the entire system if care is not used. There is a performance penalty for the reflective operations. This penalty is negligible in many cases, e.g., when the technique is used for distributed computing, since in this case there is overhead on method calls anyway.

[CM93] notes that method call mechanisms can be extended using reflective techniques to provide not only ordinary remote method calls, but also asynchronous message passing and message broadcasting (e.g., the method call mechanism could be redefined to append the message to a message queue of the receiver, and return control immediately to the caller). Method calls can also be extended to provide a synchronization mechanism, such as a rendezvous or a distributed semaphore.

[CM93] also describes how to implement distributed facilities using the Open C++ metaobject protocol, through a framework called *Object Communities*. This is a class library of metaobjects. The fundamental functionality of Object Communities is the management of a group of objects distributed in different machines. The definition of individual base-level objects include no distributed capabilities; these appear only in the metaobjects. In Object Communities, a metaobject has the following additional capabilities:

- concurrency control—A metaobject controls the internal consistency of its associated base-level object. It can delay execution of a called method of the object until some condition is satisfied. A metaobject can also execute multiple methods of its object concurrently, or execute a method of its object when other metaobjects request that execution.

- communication—A metaobject can implement remote method calls and message broadcasting.

Although a metaobject controls the internal consistency of its base object, there is no default internal concurrency of the metaobject itself. If internal concurrency of a metaobject is necessary, the metaobject itself must be controlled by an explicitly specified higher-level metaobject.

[CM93] describes a number of other method call extensions using metaobjects. Examples include: remote object pointers enabling an object to transparently call a method of an object on another machine, persistent objects (using the ability of the metaobject to deal with object instance variables whose values are stored persistently elsewhere, such as in a file or database), and replicated objects that notify copies when they are updated, in order to maintain consistency. This latter example uses method category names to indicate whether methods are "read" or "write" methods to enable the metaobjects to correctly synchronize operations on the replicated objects.

Another example discusses how method call extensions and Object Communities can be used to implement atomic transactions (using locking). In this approach, a special class `OcLockMetaObj` of metaobjects containing lock variables is defined. The method `Meta_MethodCall()` of this class of metaobject is specialized to coordinate its activities with the other metaobjects in the same object community, blocking or performing execution of methods based on the status of the locks on the objects in the community.

The paper also describes the performance aspects of the Open C++ approach, and compares it with various other reflective and RPC mechanisms. Several systems are described as supporting "compile time" reflection to reduce costs. These mechanisms modify the compilers to compile the program using a different approach; in this case the code substituted by users replaces the code that would be generated by the compiler. This reduces the reflective overhead, but results in less-straightforward code at the meta-level.

3.2.2 Choices (C++)

[MKIC92] describes reflective C++ capabilities for use within the Choices operating system. Choices takes a much more conservative approach to the use of reflection than does Open C++ (or the Apertos reflective operating system described in Section 3.5). Rather than using reflection throughout, as these systems do, Choices uses conventional C++ for most computation, resorting to reflection only in key places where it was felt to be particularly useful. In particular, Choices uses reflection to support the following facilities:

- storage—concurrency and performance of memory allocation and deallocation
- existence—mechanisms for automating the deletion of objects
- class—the relationship between an object and its class
- inheritance—the relationships between the classes in a class hierarchy
- persistence—mechanisms for persistent object activation and deactivation
- encapsulation—hardware-enforced encapsulation of instance data

C++ directly supports reflection on heap management by allowing overloading of the `new` and `delete` operators for both the entire system and for individual classes. An `Allocator` object reifies the heap manager by defining `allocate` and `free` functions, which are called by the C++ `new` and `delete` operators. The heap can be replaced in a running system by calling a global `SetNew Allocator` function with the address of a new `Allocator` object. Several subclasses of `Allocator` are defined that implement heaps with different properties.

Choices also implements automatic storage management. The `ReferenceCount` class reifies the number of pointers to an object. The `Object` class defines operations `reference`, `unreference`, and `noRemainingReferences` that perform reflective computation by incrementing and decrementing object reference counts and calling an object's destructor when the reference count reaches zero. In addition to reifying reference counts, pointers to objects are also reified, by defining a class called `ObjectStar`. Instances of `ObjectStar` automatically call `reference` when assigned the address of an object. `ObjectStar` instances are used instead of conventional C++ pointers.

The Choices class `Class` reifies C++ classes as objects. At runtime, each object in Choices refers to a specific `Class` object. These objects have reflective computation functions `isMemberOf` and `isKindOf` to allow programmers to test whether the objects belong to particular concrete or abstract classes at runtime. Instances of class `Class` also reify the class hierarchy. Constructors are reified as the constructor member of class `Class`. This allows programs to specify objects as instances of abstract classes whose concrete subclass implementations can be changed at runtime.

[MKIC92] also describes the implementation of persistence using overloaded pointer references, as in C++-based object DBMSs, using class `PersistentObject`. When a process references a persistent object, the object is automatically retrieved. A persistent object has a global lifetime that is independent of the process that creates it. It persists as long as there is at least one reference to it from another persistent object. Applications can dynamically identify a set of "root" objects that are by definition persistent, whether or not any other object refers to them. The `persist` operation on a `PersistentObject` adds it to the set of root objects; the `desist` operation removes it. Non-root objects persist as long as they are reachable from a root object.

Instances of the `ObjectProxy` class reify runtime encapsulation of system objects. An `ObjectProxy` represents an object that is encapsulated in a different hardware protection region.

In addition, the `Object` class reifies human readable information about an object. This class defines the reflective computation operations `writeName` and `inspect` that display the class, address, and an optional symbolic name to identify an object when debugging.

3.2.3 General Techniques for Class-Based Reflection

[Fer89] provides a general overview of the basic mechanisms that can be used to provide computational reflection in class-based languages[1]. The paper observes that two different models of reflection have been developed. The first model involves the reflective use of classes and metaclasses in implementing objects. In a typical class-based model, each object is an instance of a single class. Classes are, in turn, instances of other classes, called *metaclasses*. This model allows for reflection on the structural aspects of objects, since these aspects are represented explicitly in the classes (hence, this form of reflection is sometimes called *structural reflection*). The second model, described in [Mae87], is directed toward full computational reflection as described in Section 3.1. This model is based on the idea that each object has its own *metaobject* that represents information that would otherwise be implicit, including both its structure and its way of handling messages. This model allows for reflection not only on the structural aspects of objects, but also on

[1] The discussion of this paper has been put here, rather than at the beginning of Section 3.2, because it was felt its points might be better understood after one or more examples of reflective systems.

their behavioral aspects such as message handling (this additional aspect is sometimes called *behavioral* reflection; *computational reflection* refers to both structural and behavioral reflection).

The paper goes on to argue that there are three basic models for supporting computational reflection that can be used in class-based languages. The three models are[1]:

- the *metaclass model*, which is based on considering object classes (and their metaclasses) as the metaobjects for both structural and behavioral reflection.
- the *specific metaobject* model, in which metaobjects used for computational reflection are instances of a special class, say `Metaobject`, or one of its subclasses.
- the *metacommuncation* model, which is based on the reification of messages.

Ordinarily the semantics of message passing can be considered to be defined in terms of two more primitive operations, `default-lookup`, and `apply`. `default-lookup` is a built-in operator that finds a method using the selector (operation name) given in the original message. `apply` then invokes that method, with the receiving object and any arguments passed in the message as parameters. In other words, the expression:

```
obj <= op(arg1: a1, arg2: a2, arg3: a3)
```

that sends the message `op(arg1: a1, arg2: a2, arg3: a3)` to object `obj` is implemented by the metalevel expression:

```
(obj <= default-lookup(op)) <= apply(obj, arglist)
```

In both the metaclass and specific metaobject models, the semantics of message passing is modified by defining them in terms of the sending of a message, called `handlemsg` in the paper, to the metaobject. The only difference between the two models is the identity of the metaobject. In other words, the message sending expression:

```
obj <= op(arg1: a1, arg2: a2, arg3: a3)
```

is, in these reflective models, implemented by the metalevel expression:

```
(obj <= meta-of) <= handlemsg(obj, op, list(a1, a2, a3))
```

`handlemsg` in turn, is defined in terms of variants of the `lookup` and `apply` operations. (The `handlemsg` message defined here is not the same as the `Meta_HandleMethodCall` operation of Open C++. In fact, `handlemsg` has roughly the same semantics as Open C++'s `Meta_MethodCall` operation, while `apply` has roughly the semantics of the `Meta_HandleMethodCall` operation. The notation used here is also somewhat different from the Lisp-like syntax used in the paper).

In the *metaclass model*, the class of an object is considered as its metaobject, due to the natural structural reflection of the model. In this case, the `meta-of` operation described

---

[1] The description of these models is somewhat simplified here compared to that in the paper. However, an attempt has been made to preserve the basic ideas.

above is equivalent to a `class-of` operation which returns the class of an object. The `handlemsg` method is then defined in the metaclass of the class object, together with the `lookup` operation. The `apply` operation would be defined either as a separate function (as in Lisp) or as part of the virtual machine (as in Smalltalk). The semantics of message handling for specific classes can then be modified by substituting or altering the metaclass of the receiver's class. The paper identifies a number of problems with this approach:

- Every instance of a class shares the same message interpreter. Thus, there is no way to specialize message handling for a particular object.

- Modification of the message interpreter requires substituting a new metaclass. However, this must be done carefully, since otherwise system inconsistencies can result.

- It is difficult to record characteristics of individual objects (e.g., statistics or message history information) in the metaobject (since it is shared by many instances).

In the *specific metaobject model*, metaobjects are defined as instances of a special class, here called `metaobject` (this corresponds to class `MetaObj` in Open C++). These metaobjects differ from the class of the receiver. They include only operational and control information about their base objects, leaving the structural information about the receiver in the class object. In this case, each object can have its own metaobject; also, class objects, including class `metaobject`, can have their own metaobjects. The main distinction between this and the metaclass model described earlier is that structural and behavioral reflection are separated. Behavioral reflection (how a message is interpreted and how a method is applied) is handled by metaobjects, while structural reflection is handled by the class objects.

Implementing this model involves each object having an additional slot referencing its metaobject. `handlemsg` is defined in class `metaobject` or a specialized subclass of `metaobject`. `lookup` could be defined either as before, or in class `metaobject` (or one of its subclasses), using the class `lookup` operation for basic access to methods defined in classes. The paper gives examples of using this approach to define metaobjects that record statistical information about method application, and that keep a log of messages applied. The advantages of the specific metaobject model are:

- The metaobject of an object can be modified relatively easily, and without affecting other objects.

- An object can be monitored by its metaobject, which can act in various ways to control the methods actually applied in response to messages.

- New ways of handling messages can be introduced by creating new subclasses of `metaobject`.

The *metacommunication model* provides features similar to those of the specific metaobject model, but is based on the reification of the communication itself. In this approach, rather than defining a metaobject for each base-level object, messages are represented as instances of class `message`, which respond to the message `send`. The message send:

```
obj <= op(arg1: a1, arg2: a2, arg3: a3)
```

is implemented by:

```
(message <= new(obj, op, list(a1, a2, a3))) <= send
```

It is the responsibility of a message to interpret itself, and class `message` contains all the information needed to do this. Compared to the previous two models, this one has the following advantages:

- It is easy to differentiate between different types of messages. Specializations of message handling can be incorporated by defining specialized subclasses of `message`.
- The use of different types of messages allows for incremental extensions such as concurrent message passing, etc. Moreover, the same object can support different types of messages.

The main disadvantage of this model is that it does not directly model information associated with specific objects. It is impossible to monitor objects, to represent specific information about receivers, or to represent the behavior of a single object. However, it is possible to use this model in conjunction with the specific metaobject model. In this approach, the `send` message defined for class `message` could be defined to cooperate with per-object metaobjects in handling messages.

3.2.4 Commentary

The descriptions of reflective, class-based, object-oriented languages in the sections above illustrate the basic themes that will be repeated over and over, in various forms, in the sections to follow. In each case, there is a desire to be able to extend the language or system (or its behavior) in a modular way, without having to modify the individual application objects. Since the basic operation in any object-oriented language or system is message passing, an obvious approach to providing the extensions is by extending what happens during message passing (dispatching). This involves detecting the act of message passing, and inserting additional computation (either in addition to or instead of the invocation of the base-level method requested) to implement the desired extensions. Also, in each case, there is a desire to be able to restrict the use of reflection to those cases where it is strictly necessary, due to its overhead.

The basic mechanism of reflecting on message passing involves:

1. Defining (making first class metaobjects of) messages and other relevant parts of the message passing mechanism, such as methods, and classes (and the message queues used in concurrent languages discussed in later sections)

2. Defining the basic elements of the message passing mechanism as operations of explicitly-defined metaobjects which are called in the message passing process (i.e., defining a *metaobject protocol*), and providing default implementations for these operations. Typically, these operations include at least the operations:

    - send a message to a base object (`Meta_MethodCall()` in Open C++, or `handlemsg` in [Fer89])

- invoke a base-level method (`Meta_HandleMethodCall()` in Open C++, or `apply` in [Fer89])

3. Extending the semantics of message passing, as necessary, by substituting versions of these operations that include the extended behavior for the default versions (usually by defining subclasses of the metaobjects that include the operations).

[Fer89] illustrates that different mechanisms can be used in providing reflection. Open C++ is an instance of the *specific metaobject model* described in [Fer89], as are many of the reflective object models discussed in the subsequent sections. However, variations of these models also exist. For example, unlike Open C++, Choices does not define explicit metaobjects for each base-level object. Instead, in those cases where reflection is required, such as for maintaining reference counts, or for accessing class information, reflective operations are defined for the *base-level* objects, even though some specialized metaobjects (e.g., instances of class `ObjectStar` or `Class`) are also required. A similar approach is illustrated by the prototype object model described in Section 3.3, and in the actor model variant described in Section 3.4.5.

[Fer89] also demonstrates a way to deal with the issue of introducing reflection into languages where objects already exist that play both base-level and metalevel roles, such as class objects, which provide a structural reflection capability. The approach is to continue to allow such objects to play their current roles, but also to introduce additional metalevel objects as necessary to complete the reflective capabilities. These additional metalevel objects would not necessarily be visible at the base level. Choices and Open C++ show how this same approach would be applied in a language that does not already support any reflection, such as C++. In this case, capabilities for both structural reflection (e.g., class objects in Choices) and behavioral reflection (e.g., individual metaobjects in Open C++) must be introduced. A similar approach could be followed in using the RISC model to model both languages that already support explicit class or type objects, and languages that do not.

Open C++ illustrates the principle, also used in many of the languages discussed below, of using the abstract interfaces defined by metaobjects (i.e., a metaobject protocol) to structure how extensions are added, rather than simply opening up the implementation of the language to arbitrary modification. A less-structured way of performing reflection on message passing (although in a more primitive model) is illustrated by the *dispatchers* used in GTE Laboratories' DOM 3.0 prototype (Section 3.6.3).

The sections above also describe a *range* of applications for adding extended capabilities using reflection. Some of these applications involve extensions on a *per-object* basis, such as changing an object so that it prints when a method is invoked, or automatically deletes itself when it is no longer referenced by another object, as well as extensions for remote or asynchronous messaging. Other applications involve extensions involving multiple objects, or on a more global basis, such as message broadcasting, object synchronization, transaction management, or access to other shared system services, such as persistent storage (in general, these services are represented as objects as well, as illustrated in Choices). The ability to integrate general system services with individual application objects is a key element in building distributed, object-oriented application architectures, such as the OMG's Object Management Architecture [OMG92a,b], and the Telops Application Framework. Details concerning the relationship of system services to the object model of an application architecture were given in [MH93].

The Open C++ and Choices sections show how both distributed object and persistent object capabilities can be added using reflection. This is done by changing per-object metaobjects (Open C++) and by overloading object references (the subclasses of the `ObjectStar` class in Choices). A primitive form of changing per-object metaobjects to add distribution and persistence was illustrated in [NH93a,b] using the DOM prototype. The approach of overloading object reference classes is used frequently in object DBMS products to add persistence and other capabilities.

The Choices section also illustrates the use of reflection to add automatic storage management (garbage collection) to C++, a language that lacks this as a built in capability. Whether a language supports or does not support automatic storage management is one of the differences identified by the X3H7 standards committee as potentially creating interoperability problems among object models [MH93]. The fact that, using reflection, this capability can be added to a language that does not have it suggests that if languages had built-in reflective capabilities (or were based on a core containing them) it would provide a general means for extending them as required to deal with their integration with other languages having different facilities.

The use of reflection to add facilities for concurrency and transactions illustrates another important theme. The basis of any of these mechanisms is that an object wait to perform some operation until it is safe to do so, according to some condition or criterion. In some cases, this condition is based on the condition of a number of other objects. Reflection allows the invocation of an operation to be extended so that the appropriate conditions can be checked before proceeding. A number of examples in the following sections illustrate variations of this approach to concurrency control, i.e., altering the reflective computation to interrupt the flow of method invocation to perform concurrency-related communication, and delaying the operation execution until a condition is satisfied that indicates that it is safe to proceed with the operation. In some cases, the checking to be performed is simple consultation of locks or semaphores. In more general or complex cases, the operation invocation can be referred to a Transaction Processor or other transaction management executive. Such a component could be considered as a metaobject providing a specific service (e.g., a member of an Open C++ Object Community). [HG93] describes ongoing work at GTE Laboratories showing how this approach of interrupting message dispatching to consult generalized transaction management components can used to integrate extended transaction facilities in a distributed object environment. It is important to note that the details of the various concurrency control schemes using metaobjects that will be described are not as important as the basic pattern, which is that this can be done at the metalevel, by reflection on the global state of the computation. This makes it possible for different concurrency control policies to be defined in a modular fashion, largely independently of the definitions of the individual application objects.

Finally, as noted in Section 1, the basic idea behind reflective object-oriented processing is similar in many respects to the basic idea behind *intelligent networks* in telephony [GMS88, HH87, Man88]. In an intelligent network, some call-control functions for new services are moved from the switching systems to specialized control points. When a call is processed that requires special services, the switching systems will then, under certain conditions, seek instructions from these control points over special-purpose signaling paths. Intelligent networks have a number of potential advantages, including the following:

- Services can be introduced more rapidly, because only the control points must be changed in order to introduce the service.

- The generality of services that can be provided can also be increased, because the control points may potentially supply more complex processing capabilities than might be feasible at a given switch.

- Services can be introduced in a way that is independent of vendor-specific switch hardware and software, because individual switches need not be reprogrammed or otherwise modified to introduce the service.

In an object-oriented language or system, the built-in message passing mechanism plays the role of the switch in this analogy. Once this is modified so that it can interrupt normal message processing to invoke additional services, the services themselves can be added elsewhere (as additional objects). The advantages cited above for intelligent networks are mirrored by the potential advantages of object-oriented reflective processing, as shown by the (bracketed) wording in the following repetition of the above advantages of intelligent networks:

- Services [computational services] can be introduced more rapidly, because only the control points [objects providing services] must be changed in order to introduce the service.

- The generality of services that can be provided can also be increased, because the control points [objects providing services] may potentially supply more complex processing capabilities than might be feasible at a given switch [in the message passing mechanism of a particular language or system].

- Services can be introduced in a way that is independent of vendor-specific [language-specific] switch hardware and software [message passing mechanisms], because individual switches [message passing mechanisms] need not be reprogrammed or otherwise modified to introduce the service.

The descriptions of the class-based reflective languages above, and the descriptions of the other reflective languages in the following sections, hopefully illustrate these points.

3.3 An Example of Reflection in a Prototype-Based Language

Section 3.2 described the use of reflection in several *class-based* object-oriented languages. [MDC92] describes the use of reflective techniques in a *prototype-based* language. Prototype-based languages such as SELF [US87] and Cecil [Cha92,92] have no classes. Instead, they are based on the use of *prototypes*. In this approach, new objects are created by *cloning* (shallow-copying) existing objects (the prototypes). The state of the new objects can then be set to instance-specific values. Each object conceptually defines its own object-specific behavior, and may borrow behavior from other objects (called parents) using *delegation* (instance-level inheritance). A detailed description of the commonalities and differences of various prototype-based languages can be found in [DMC92]. The reflection approach described in [MDC92] is based on the authors' work in analyzing many different prototype-based languages from [DMC92], and hence is generally applicable to a wide range of such languages.

In class-based languages, classes and metaclasses provide an explicit mechanism for *structural reflection* (the representation of the structure of base-level objects within the language). However, in prototype-based languages there are no classes, so an alternative way of providing structural reflection must be defined. In [MDC92], structural reflection is provided by having individual objects support predefined methods that provide access to object structure, and by reifying methods as objects. The primitive access methods provided are `size(oid)`, `name(oid, slotid)`, `isMethodAt(oid, slotid)`, `contentsAt(oid, slotid)`, and `contentsAtPut(oid, slotid)`. Each of these methods is then defined as an object. Thus, instead of accessing an object's class to obtain structural information, these methods are defined for individual objects to provide the same information.

A reflective language should also support *behavioral reflection* (the representation of the behavior of base-level objects within the language). There are two main aspects to modeling behavioral reflection. First, the model must describe the behavior of objects using other objects. Second, the model must provide a method invocation protocol that allows the user to intervene in the current execution in order to modify the system's behavior.

To describe the behavior of objects, [MDC92] associates a metaobject with each base-level object. The job of this metaobject is to make explicit how the object reacts when it receives a message. [MDC92] notes that, since the object-oriented model of computation uses message passing as its fundamental mechanism, it is usual to make message passing the point at which programs can shift into a reflective phase. The standard way to achieve this is by making visible the two main operations done by the evaluator when a message is sent: the lookup and the application of the method, i.e.:

message execution = lookup + apply

These operations are implemented as methods, themselves reified as objects in the language. Making these methods objects allows the user to redefine them in order to define customized message passing behavior. Essentially, each message passing operation `obj <= op(arg1: a1, arg2: a2, arg3: a3)` is replaced by a reflective "introspection" which has three phases:

1. find the metaobject of the receiver `obj` (by sending it a `metaObject` message asking for the value of its `metaObject` slot)

2. send the metaobject a message `lookup(name: op, in: obj)` that returns a method object,

3. send the method object a message:
   `apply(to: obj, arglist: #(arg1 arg2 arg3).`

In other words, the reflective rewriting rule is:

```
obj <= name(arg1: a1, arg2: a2, arg3: a3) =
(((obj <= metaObject) <=
   lookup(opname: name, in: obj)) <=
   apply(to: obj, arglist: #(arg1 arg2 arg3))
```

(In this notation, message sending is left-associative, so the result of each message is the receiver of the next one to the right, etc. The rule makes no assumptions about the mechanism used to evaluate the receiver, the selector, and the arguments, except that they are reified into objects. The notation used here is slightly different from that used in the paper.)

In short, every object has a metaobject that is able to answer `lookup` messages; metaobjects can be shared among several objects and behave like local interpreters for them. Metaobjects answer `lookup` messages by returning method objects that are able to answer `apply` messages. In a typical prototype-based language, data slots and method slots are both accessed using messages. Because of this, the protocol must cope with data as well as methods. In both cases, the lookup phase must return an object that is able to answer an `apply` message in order to make the reflective rewriting rule work properly. This is handled by having data values return themselves in response to those messages.

The problem of potential infinite regression (having metaobjects which in turn have their own metaobjects, and so on, and never actually invoking the method originally requested) is handled by defining a basic metaobject `BasicMetaObject`, which is its own metaobject, and which defines the default behavior for objects in the system. The circularity of the `metaof` link for this object closes the metaregression on `BasicMetaObject` (in the same way that the instance-of link is closed over class `Class` in some class-based languages). `BasicMetaObject`, like any other metaobject, must be able to respond to `lookup` messages. Its lookup method is a primitive `lookup` function reified as an object in the language called `BasicLookup`. Reifying this primitive requires the introduction of its `apply` method to reflect the method invocation protocol. This apply method is defined as the primitive `apply` function, reified as an object `BasicApply`, which is constructed to have itself as its own `apply` method. This can be summarized as:

- `BasicMetaObject` is the first metaobject in the system; it has itself as its own metaobject.

- `BasicLookup` is the first lookup method in the system; it represents the primitive lookup function reified as a method object.

- `BasicApply` is the first apply method in the system; it represents the primitive apply function reified as a method object.

The kernel of the model is constructed from these three primitive objects, plus `ROOT`, the origin of parent-of delegation hierarchies (`ROOT` is defined to be its own parent, to have `BasicMetaObject` as its metaobject, and to have the above primitive methods as its methods). Figure 3.2 illustrates this structure.

The paper goes into detail on validating the correctness of these semantics, explicitly specifying the details of how the system would operate using variations of the rewriting rule described above. The paper also describes how partial evaluation, semantics-based program transformations, and advanced compilation techniques could be used to reduce in practice the inefficiencies that would be created by an implementation of reflection using multiple levels of interpreters (the "reflective tower" described earlier) in a literal way.

3.3.1 Commentary

[MDC92] does not describe any particular applications. Its importance is as an illustration of how to handle reflection in a prototype-based model, and the similarities it turns out to have with the class-based approaches described in Section 3.2. These similarities suggest that a common set of reflective principles, such as those we hope to provide in the RISC model, may be generally applicable to a wide range of object models, including both class-based and prototype-based models.

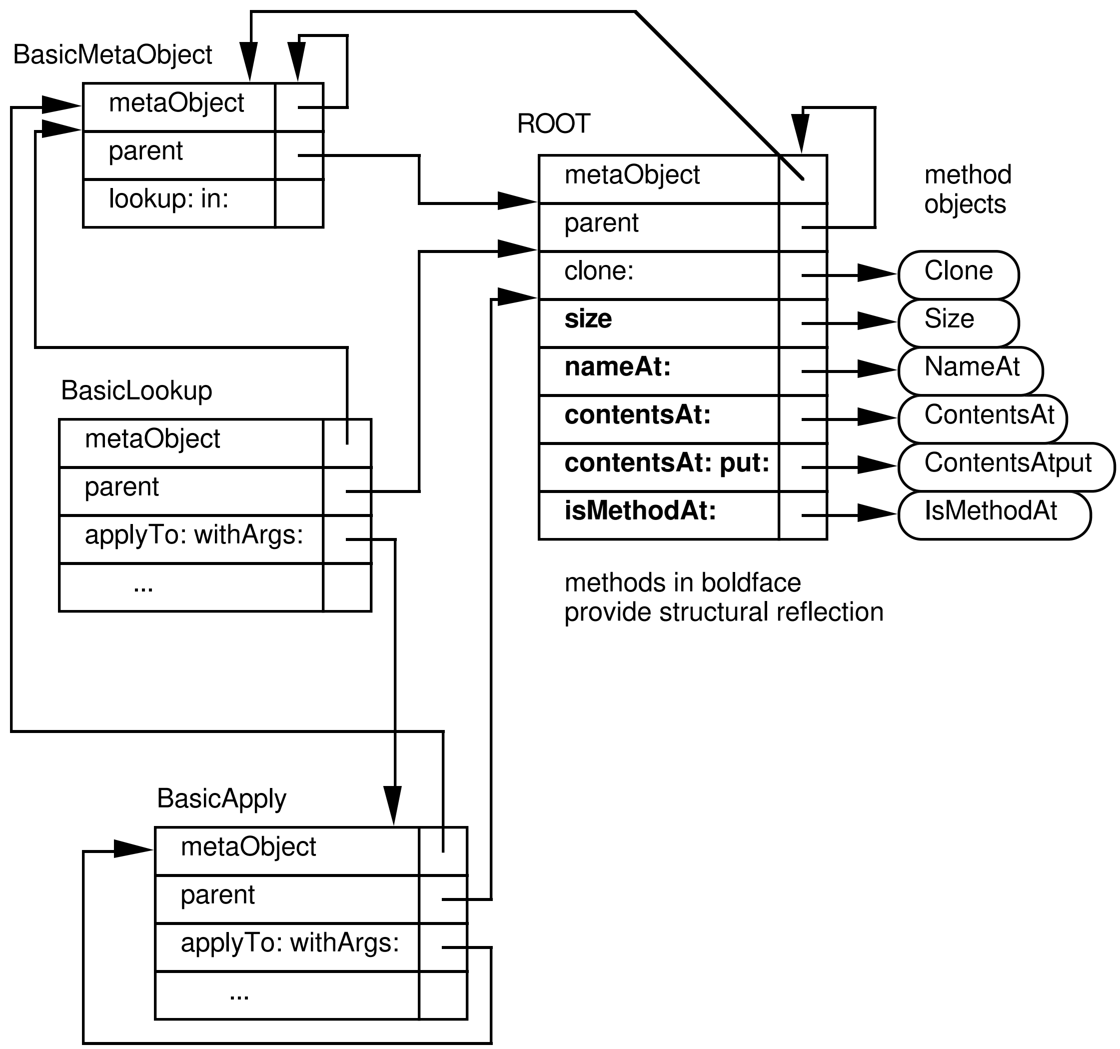

Figure 3.2. Basic Prototype Objects and their Relationships

In particular, the basic structure of how reflection is handled in [MDC92] is similar in many respects to the way it is handled in Open C++, with adjustments for the differences between prototype- and class-based languages. For example, the two metamethods of Open C++ metaobjects, `Meta_MethodCall()` and `Meta_HandleMethodCall`, and their use in behavioral reflection, correspond roughly to the way the prototype language accesses an object's metaobject, and the `lookup` and `apply` methods defined for it. The role of metaobjects in the system is also somewhat similar. The way in which such concepts as methods and `ROOT` (which corresponds to type `Object` in some other reflective systems) are defined as primitive objects based on themselves is also fairly typical of a number of reflective mechanisms, and resembles the way similar concepts (e.g., methods) are intended to work in the RISC model.

A difference from some reflective mechanisms is the way structural reflection is provided by having individual objects support predefined methods that provide access to object structure, instead of having the objects' *metaobjects* support this information. In other words, in this case, reflection is supported partly by having special reflective operations at the base level, rather than by having all reflective operations defined at the metalevel. One of the reflective actor languages to be described in the next section uses this same approach to support behavioral reflection as well. That is, instead of implementing reflection by having a metaobject for each object, it uses reflective operations defined on base objects to accomplish the same goals. The management interfaces defined for managed objects in telecommunications applications [QP91] provide operations that can be considered examples of reflective operations of this sort.

Neither the class-based languages described in Section 3.2, nor the prototype language described here, are concurrent programming languages in the sense described in Section 2.8. However, as discussed in Section 3.2.4, some aspects of using these languages in concurrent processing (e.g., synchronization and transactions) can be addressed using reflection. Sections 3.4 and 3.5 will show how reflection can be incorporated in languages that are explicitly concurrent, and will illustrate the basic similarities of the reflective mechanisms involved.

3.4 Examples of Reflection in Actor Languages

The *actor model* [Hew77,AH87] unifies object-oriented and concurrent programming. The intent of the actor model is to provide a basis for describing all sorts of object-oriented concurrent behavior. Since languages based on the actor model are explicitly concurrent, reflective versions of these languages are of interest as providing examples of the primitives required to model concurrency.

Actors are autonomous, distributed, concurrently executing objects that can send each other messages asynchronously. An actor sending a message need not block its execution until the recipient is ready to receive or process a message (and synchronous processing tends to be avoided in actor programming, in order to maximize concurrency). Actors do not prescribe a particular organization for object-oriented programming, and actors can play the roles of individual objects, object prototypes, object classes, etc., all within the same program.

Each actor has a conceptual location, its mail address, and a *behavior*. The behavior consists of a script and a *set of acquaintances* (other actors known to the actor). The script is essentially a code body that defines the response of the object when a message is received. The actor model provides for buffered communication; each actor has a *mailbox* (identified by the mail address) which is always available to receive messages. In actors, mail addresses serve as identifiers and, like object identifiers in conventional object models, can be freely communicated between actors. In the conventional actor model, neither the mailbox nor the behavior are treated as objects.

As shown in Figure 3.3, in response to a message, an actor can:

1. Send a message asynchronously to a specified actor (one of its acquaintances, an actor referenced in the message, or an actor created in the script)

2. Create an actor with a specified behavior, or

3. "Become" a new actor, assuming a new behavior for responding to the next message.

The *send* primitive is the asynchronous analog of procedure invocation. It is the basic communication primitive, causing a message to be put in an actor's mailbox. To send a message, the recipient's identity (its mail address) must be specified. Although the arrival order of messages is nondeterministic, every message sent to an actor is guaranteed to be delivered eventually.

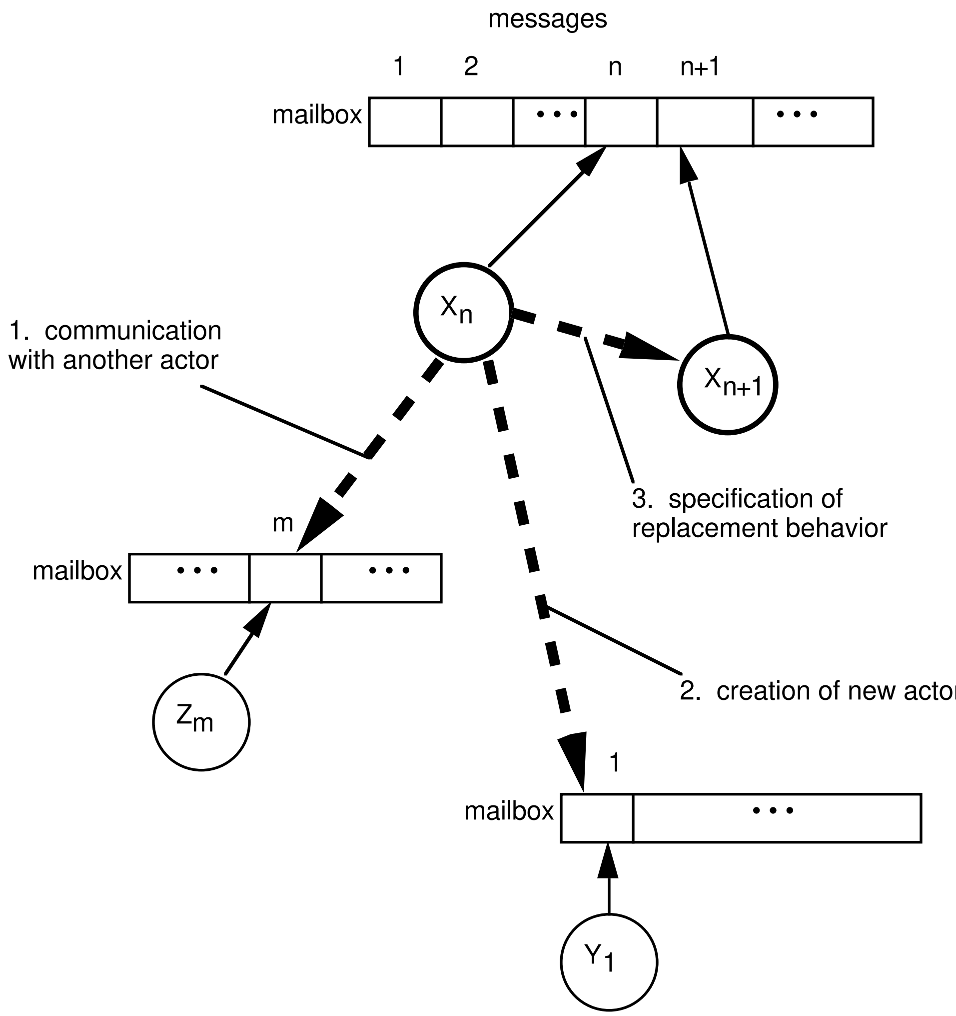

Figure 3.3. Actor Computation Model

The *create* primitive creates new actors. Newly-created actors are autonomous and have a unique mail address.

The *become* primitive gives actors a way of providing the history-sensitive behavior necessary for shared, mutable data objects (in contrast to a purely functional programming model). As shown in Figure 3.3, when actor X processes the nth message, it determines the *replacement behavior* which will process the n+1st message. The replacement behavior becomes a new actor. The mail address of the actor remains invariant. An actor computes a replacement behavior in order to account for changes in its responses following the communication it has just processed. A new behavior is the externally-visible effect of a state change, and thus corresponds to a state change in a more conventional model.

A key concept in the actor model is that an object becomes locked when it begins executing a message [TS89b]. The *become* primitive not only specifies a replacement behavior, but also has the effect of unlocking the object and permitting the next message to be selected for execution. If the replacement behavior is specified prior to the completion of the execution of the script, then it is possible for a second thread of execution, operating on the new version of the behavior, to start before the first is complete. The replacement behavior includes a new script and set of acquaintances, although in practice the script is often the same and only a subset of the acquaintances are changed. When more than one thread is executing in an object, the individual threads access independent versions of the script and set of acquaintances, and are therefore non-interfering.

3.4.1 ACT++

[KML93] describes a C++ class library called ACT++ that implements a variation of the actor model within C++. Computations in ACT++ can include both *active objects* (actors defined using the ACT++ class library) and *passive objects* (ordinary C++ objects, which use the thread of the requestor, rather than having their own internal thread as the actors do). The key features of the model of computation implemented in ACT++, some of which are extensions of the conventional actor model, are:

- An ACT++ program consists of a collection of actors that execute concurrently using independent threads of control (these are "threads" in the operating system sense: lightweight processes).

- Actors communicate asynchronously via *request messages*. Each actor has the ability to buffer messages in a queue called its *mail queue*. Actors can also participate in synchronous communications with objects that are not actors.

- Request messages contain requests for the execution of program segments in an actor called *behavior scripts*. Behavior scripts are implemented as methods of a special type of object called the *behavior object* or *behavior*. Each behavior is responsible for processing exactly one request message.

- The behavior object that will process an actor's next request message is called its *current behavior*. At any point in time only a single behavior object is designated as the current behavior of an actor. The processing of a message by an actor is performed by the execution of a specific method in the behavior object of that actor.

- The processing of a request message by a behavior of an actor can lead to the occurrence of one or all of the following actions:

  1. send asynchronous request messages to itself or other actors.

  2. synchronously invoke methods in itself or in other, non-actor objects.

  3. create more actor or non-actor objects.

  4. specify the next behavior of the actor (called a *replacement behavior*) that will process the next request message.

- Actors can also exchange *reply* messages. As a result of a request message to it, an actor may send a reply message to the requestor with the result of the computation. A special type of object called a *Cbox* is used to buffer reply messages. Cboxes are defined with blocking semantics so that an actor blocks when it tries to extract a reply message from an empty Cbox. (The Cbox concept was introduced in ConcurrentSmalltalk [YT87b]).

- An actor in ACT++ is not restricted to processing request messages in the strict order of their arrival. Instead, a replacement behavior can consult the values of its instance variables to decide which request message to process next.

The ACT++-provided simulation of the components of the actor model in C++ effectively defines a reflective version of the actor model, since ACT++ reifies various aspects of the model, and allows user-defined versions to replace them.

Actor objects in ACT++ are instances of the predefined `Actor` class, whose constructor is:

```
Actor(Behavior* init_beh, int th_num = 1, char* name = 0).
```

This constructor corresponds to the *create* primitive described in the last section. The behavior pointer specifies the initial behavior objects responsible for processing the first message to the actor. Whenever a message is accepted for processing by an actor, a thread (a `Thread` object) is created to execute the method requested in the message. The integer variable `th_num` specifies the number of threads that can be simultaneously active in an actor. The default number of threads in an object is one, but the user can specify more than one thread to provide for concurrent message processing, by allowing two or more behaviors of the actor to process request messages simultaneously. The character pointer `name` associates a name with the actor that can be printed when debugging.

Messages are the means of communication between actors. All messages exchanged between actors are instances of the predefined object class `Message`, which reifies messages within the model. Messages are sent to actors using the `send` method of the `Message` class, which correspond to the *send* primitive described in the last section.

Messages are requests to execute one of the methods in a behavior object of an actor. All behavior objects in ACT++ are instances of user-defined subclasses of the predefined `Behavior` class, which reifies behaviors within the model. The ability to define user-defined subclasses of `Behavior` is one of the means of customization provided by ACT++. Specification of replacement behaviors is the means by which actors retain their ability to process messages. This is done in ACT++ by the `become` operation defined in the `Behavior` class, which corresponds to the *become* primitive described in the last section. The current behavior of an actor is responsible for specifying the replacement behavior that selects the next message to be processed. The replacement behavior object is usually an instance of the same class as that of the current behavior. (Technically, assigning objects of different behavior classes as replacement behaviors is not impossible. But, conceptually, an actor serves a specific purpose usually realized by the operations defined in a single behavior class). The `become` operation has the form:

```
void become(Behavior* b)
```

In ACT++, no low-level mechanism such as locking is available to explicitly define critical sections. Hence, a behavior object must always process a single message. Otherwise, interference between independent threads of control could corrupt the *state variables* -- the instance variables of the current behavior of the actor. Therefore, it is important to always create a new behavior object as the replacement behavior, which prevents the simultaneous execution of independent threads of control in the same object. Also, because the instance variables of the replacement behavior are initialized using the values of those in the current behavior, the `become` operation must be specified only after the current behavior has finished updating the state variables.

Cboxes are an addition to the actor model to provide a separate place to receive reply messages (in conventional actors, the message queue is used for both). There can be many Cboxes declared in a behavior, unlike the requirement for a single message queue. Cbox implements a blocking `receive` operation. If the reply message is available in the Cbox when the operation is invoked, the `receive` call returns immediately. If not, the current thread in the behavior object blocks; it unblocks automatically when the reply message arrives in the Cbox. Different overloadings of the Cbox class can implement different synchronization schemes.

ACT++ also includes a mechanism for synchronization based on a form of explicit interface object, called a *behavior set*. Ordinarily, the same set of operations is available for execution in each behavior, regardless of its state. This does not permit the direct modeling of objects that dynamically change their synchronization behavior depending on their internal state. To model such objects using conventional actors, it is necessary to incorporate synchronizations involving the state variables in the methods of the behavior to realize selective message processing. However, such synchronizations can interfere with the inheritance mechanism of object-oriented languages.

For example, the paper describes a bounded-buffer actor whose class defines two methods: `in` adds one item to the buffer, and `out` extracts one item from the buffer. When the buffer is empty, the `out` method is inapplicable and, in a concurrent application, it may be desired to simply delay processing the `out` method until there is something in the buffer, rather than return an error indication and have the requestor retry later. This could be done by an explicit test in the method itself. However, this approach creates potential problems for subclasses, because the code implementing the conditional test is dependent on the state of the superclass. Extensions made in subclasses may frequently make it impossible to cleanly inherit the superclass methods, requiring a complete redefinition in the subclasses. The solution proposed in ACT++ is to separate the state-dependent synchronization mechanism from the methods of a behavior class, and consider it a separate entity subject to explicit manipulation. The specific notions introduced to implement this selective method invocation mechanism are *behavior sets*, a *next-behavior-set function*, and *object-state functions*.

A behavior set is an object used by a behavior to selectively process messages depending on the behavior's state. A behavior set contains the method identifiers of a subset of the methods of the class of which the behavior object is an instance. Behavior sets are instances of the predefined class `Behavior_Set`. A `Behavior_Set` constructor has the form:

```
Behavior_Set(<list of pointers to functions>)
```

Behavior sets are used by behavior objects to project different interfaces depending on their internal state. As a result, behavior sets are always used as an integral part of behavior objects. The behavior set represents the interface of the behavior object during the selection of a request message for processing. A behavior will execute a message only if the method required by the request message is an element of the behavior set. Otherwise, the behavior will wait for the arrival of an acceptable message.

The behavior set is established by the *next-behavior-set function* at the time a replacement behavior is specified (and also when the initial behavior is created). The next-behavior-set function invokes *object-state functions* to determine the current state of the actor. The object-state functions in turn access the data members of the behavior to determine the actual state of the actor (the object-state functions insulate the next-behavior-set function from changes in the actual representation of state within the behavior object). The next-behavior set function, together with a state variable to hold the current behavior set, are defined as protected members inside a behavior. When defining subclasses of `Behavior`, the user must provide a new next-behavior-set function, and ensure that the value of the current behavior set variable is properly maintained.

In the bounded-buffer example described earlier, the corresponding `Bounded_Buffer` actor would have three possible states: empty, partial, and full. Corresponding to these three states three object-state functions would be defined: `empty()`, `partial()`, and `full()`. These three functions would have access to the data member of the buffer class called `count`, which keeps a count of the number of elements in the buffer. There would be three behavior sets: `empty_set`, which has the identifier for the `in` method; `partial_set`, which has the identifiers for both the `in` and `out` methods; and `full_set`, which has the identifier for the `out` method only. A next-behavior-set function would also be defined that uses the object-state functions to determine which of these behavior sets to establish as the behavior set of the replacement behavior.

The advantage of using behavior sets is that the behavior corresponding to an empty buffer would not have the `out` method in its behavior set, and so would never execute a message requesting an item when the buffer is empty (and so on). As a result, state-dependent synchronizations need not be encoded in the methods of a behavior, thereby allowing reuse of superclass methods through the C++ inheritance mechanism.

A similar synchronization mechanism is described as part of the (non-reflective) Arche concurrent object-oriented programming language [Iss93]. The paper indicates that their mechanism was inspired by the notion of *type states* [SY86]. Specifically, type T defines operations that may be applied on an entity e of type t. A type state of t then defines t operations that can be applied on e according to operations previously performed on the entity. For instance, the only operation that can be applied on a non-initialized integer variable is an assignment.

The concept of type states in [SY86] was primarily intended as an enhanced *type mechanism* to improve software reliability. In Arche, this general idea is used to define *synchronization states*. A synchronization state s of a view (Arche's term for type) v specifies methods that are accessible when an object of type v is in state s. Moreover, *post-states* are introduced to indicate synchronization states that may be reached from any method of a view. State transitions appear within method implementations through the use of the command `become` that names the state in which the transition occurs. A state transition becomes effective only if and when the method terminates. However, if no transition state takes place while a method executes, the previous object state remains valid. The synchronization state is illustrated in [Iss93] using the same bounded buffer example as was used in ACT++.

Syntactically, synchronization states are declared in the clause `state`; post-states of methods are declared in the clause `post`; and post-states of the initialization method appear in the `view` header. The following type `buffer` declares a bounded buffer:

```
buffer =
     view (n: integer) {empty}
```

```
          put: (x: t) ();
          get: () (x: t);
     state
          full: {get}
          partial: {get, put};
          empty: {put};
     post
          put: {full, partial};
          get: {empty, partial};
     end buffer
```

where n is an instance parameter setting the buffer size. Type `buffer` defines three synchronization states, `full`, `partial`, and `empty`. For example, the `full` state only includes the method `get`, and thus only allows the execution of `get`; any caller of `put` is blocked until the object reaches a state including `put`. A possible implementation of `get`, execution of which leads the enclosing object either to state `empty` or `partial`, is:

```
get: () (x: t) =
      begin
            out := (out + 1) mod n;
            if in = out then become empty else become partial end;
return(buf(out));
end get;
```

Effectively, the synchronization states are equivalent to ACT++'s behavior sets, while the state clause's declarations, combined with the state transitions (using *become*) in the methods are equivalent to ACT++'s next-behavior-set function. Both synchronization states and behavior sets are similar to the *enabled sets* of the Rosette language, described below.

[Iss93] later describes how this mechanism can be used in an implementation of the two-phase commit protocol, using objects with `coordinator` and `participant` interfaces. For example, the `participant` interface defines operations `vote`, `commit`, and `recovery`, and two states. In the `phase1` state, the interface allows only the `vote` operation to be performed. In the `phase2` state, only the `recovery` or `commit` operations can be performed. In this case, the mechanism provides a way to explicitly declare the legal state transitions associated with the different objects in the protocol.

3.4.2 Rosette

[TS89b] describes Rosette, which extends the actor model with reflection and other capabilities. In Rosette, an actor is viewed as being composed of four other objects that are responsible for implementing various aspects of actor semantics. These objects are shown in Figure 3.4. The *meta* object manages the other components. The *container* represents the acquaintances and script as a set of slots. The script is decomposed into separate methods, each of which responds to a message of some type. In this way, methods are just other acquaintances of the object (this is similar to the way prototype-based languages represent state and behavior of objects). The *processor* is responsible for locating an appropriate method to handle a message, performing the execution, and coordinating state changes. The *mailbox* is similar to the conventional actor mailbox, but in the extended model may incorporate different policies that determine the next message to be processed. In addition to reflection, Rosette also includes other extensions, such as both active and passive objects.

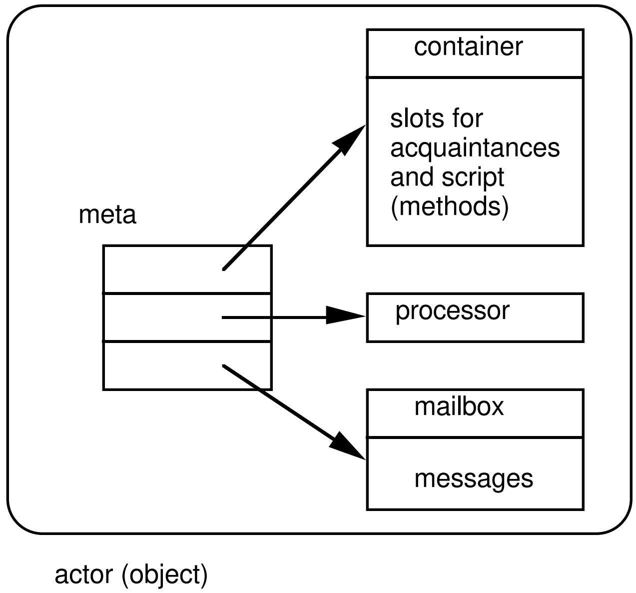

Figure 3.4. Actor Structure in Rosette

One of the uses of reflection in Rosette is to provide a clean way of representing inheritance using message-passing. This involves defining a sharing protocol that establishes internal communication between the actor representing the child and the actor representing the parent. As noted above, when an object begins executing a message, it is locked and cannot begin executing other messages until it is unlocked. The object is unlocked when it establishes its next state. Ordinarily, in order to avoid deadlock due to messages received as a result of the sharing protocol while the object is locked, complex buffering techniques need to be employed. Moreover, code for the sharing protocol and the logic of the application are interleaved in every method. In Rosette, this problem is solved by separating message-passing activities into two levels. The object level deals with messages between objects, while the reflective level deals with messages between containers that are responsible for implementing the sharing protocols. In this approach, a container has one or more acquaintances that are containers from which slots may be inherited. An object may be locked while its container is performing communications in order to obtain a method or other slot value from a parent. Thus, the reflective model of objects permits a consistent "under the hood" treatment of inheritance via message passing.

Rosette also addresses the question of how to factor synchronization constraints into units that can be inherited without having to rewrite them. These units are called *enabled sets*, and are similar in many respects to the behavior sets described earlier in connection with ACT++ (behavior sets and enabled sets are compared in [KL91]).

3.4.3 Reflective Actors

[AFKP+93] describes another reflective version of the actor model. The actors representing the basic reified actor concepts are called *metalevel actors*. To improve performance, a limited reflective model is used in order to avoid the need for interpretation. Each actor consists of three metaactors: a *dispatcher*, a *mail queue*, and an *acquaintance list*. The acquaintance list represents the actor's current state (behavior). ("State" here refers to state as in a prototype-based model, in which the state variables or slots may contain not only data (objects), but also methods; it is because of this that "state" really defines behavior in these models). The dispatcher and mail queue implement the runtime system's communication primitives, so that the interaction between actors can be modified to change the application's basic characteristics.

Specifically, a dispatcher represents the implementation of an actor's transmission behavior. When customized, messages sent by the corresponding base actor are rerouted to the customized dispatcher. An actor's mail queue represents the mail buffer that holds the incoming messages. If a customized mail queue is installed, all messages to the base actor are rerouted through it. A customized mail queue can alter the order of messages to the base actor (to enforce local synchronization constraints, for example).

The paper notes that several protocols that increase system dependability can be expressed in terms of a customized mail queue, dispatcher, and acquaintance list, including two- and three-phase commit, primary back-up, full replication, checkpointing, and encryption. Composition of such protocols can be implemented by transparently manipulating the metaactors of the metaactors. The resulting system allows not only the dynamic installation of generically specified protocols, but also their dynamic removal. The limited form of reflection used supports incremental compilation and increases execution time by only a small amount.

The paper describes an example of using reflection to support mirrored servers for enhanced reliability. In this approach, customized dispatchers transparently duplicate client messages and send them to the mirroring servers. Customized mail queues in the clients eliminate duplicate response messages sent from the various servers (in response to duplicated messages).

The paper also describes mechanisms that can be defined within the actor model for performing various kinds of synchronization. The first is a mechanism to define synchronization constraints on the execution of the operations of a single actor as parts of interfaces in a modular way. This is similar to the behavior sets and enabled sets mechanisms described earlier.

The paper also notes that, in distributed computing, a group of object invocations often must satisfy temporal ordering or atomicity constraints. However, conventional programming languages do not allow for the specification of multiobject constraints in a modular and reusable manner. The paper then describes an approach that allows the direct expression of two types of multiactor constraints: temporal orderings on, and atomicity of, invocations of shared distributed actors. This approach describes multiactor constraints in terms of conditions that must be satisfied for a group of method invocations to be accepted; if the conditions are not met, the invocations are delayed. Multiactor constraints thus coordinate concurrent objects by restricting their activation.

The multiactor constraints are described using *synchronizers*, a special kind of actor that observes and limits the invocations accepted by a group of actors. Synchronizers are implemented via primitive actor communication, whether directly between the constrained actors, indirectly via a central "coordinator", or a combination of these approaches.

3.4.4 Actors in Smalltalk

[Bri89] describes a simulation of the actor model in Smalltalk. The structure of this simulation is similar to the reflective models described above. Each actor has two components, a *mailbox* (message queue), and the *behavior* of the actor, each represented by a Smalltalk object. A process is started for each behavior, which indefinitely removes the next message from the message queue and performs it. The instance variable *aself* denotes the reflective knowledge the actor has about itself; a message sent to *aself* will be recursively sent asynchronously to the actor itself. The actor model distinguishes two types of actors: unserialized actors and serialized actors. An unserialized actor never changes its state (or has no state), and may thus process multiple messages in parallel. A serialized actor may change its state, and thus should process one message at a time. In fully modeling actor semantics, state change must be replaced by behavior change. Accordingly, each behavior object accepts only one message, the specification of replacement behavior.

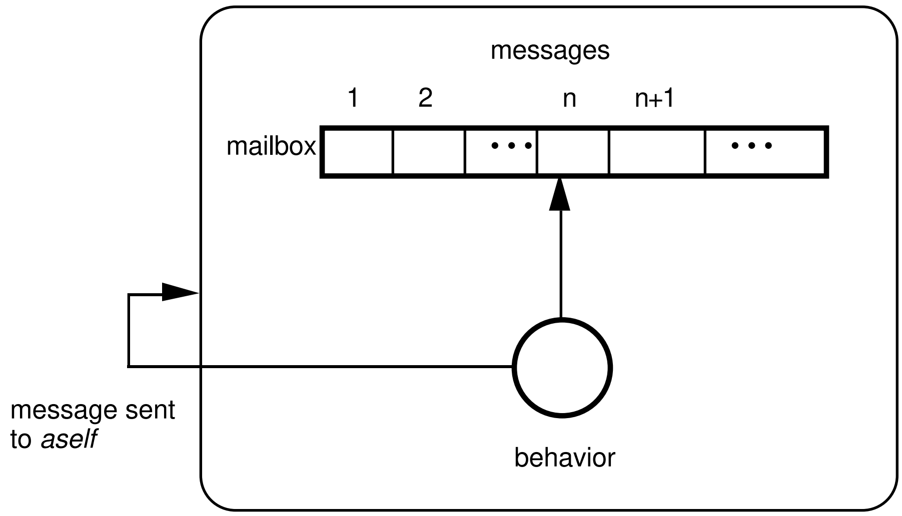

Figure 3.5. An Actor as Two Smalltalk Objects

[Bri89] also describes how these concepts can be used to simulate the different message semantics of the ABCL/1 concurrent programming language (a reflective version of which is described in Section 3.5). ABCL/1 supports three types of messages; *past* (normal asynchronous messages), *now* (synchronous messages), and *future* messages. ABCL/1 *past* messages are modeled using ordinary actor asynchronous messages. ABCL/1 *future* messages are modeled using asynchronous messages and a *future* object. A new *future* object is generated to serve as the reply destination of the asynchronous message. Access to the original expression automatically synchronizes on the assignment of the *future* object. The *now* message is then equivalent to a *future* message and explicit access to (and synchronization with) the *future* object.

3.4.5 Actor Reflection Without MetaObjects

Finally, [Tan91] describes an approach to defining a reflective actor model that is somewhat different from those described earlier. Instead of having two types of objects in the model, base-level objects defined by users, and metaobjects with predefined interfaces corresponding to each base-level object, only base-level objects (actors) exist. However, these base-level objects, in addition to supporting user-defined messages, also support special *reifying* and *reflecting* messages.

In this model, the interpreter of an actor is defined as requiring four data structures:

`main-vars`, the message template for the actor

`main-form`, the script of one operation `state-vars-env`, the environment for representing the internal state of the actor

`queue`, the queue of messages received by the actor but still waiting to be processed.

Reifying messages specify a *continuation actor*, which is defined to accept a message containing these four data structures. When an actor in this model receives a reifying message, and its turn to be processed arrives, it does not go through the usual process of message handling. Instead, it constructs another message containing the four data structures, and sends it to the continuation object as a regular message.

Reflecting messages contain these four data structures. When an actor receives a reflecting message, it installs the four data structures contained in the message into its interpreter.

3.4.6 Commentary

The actor model is a relatively primitive object model (with concurrency) which can be used to describe other object-oriented programming metaphors. One of the reasons for covering these models (and those in the next section) is that, unlike those described in Sections 3.2 and 3.3, they illustrate different *concurrent* object-oriented programming concepts.

These variants of reflection in the actor model follow the basic theme of the other models in representing the basic components of the object model as objects so, that reflection can take place. In the case of these models, there are additional components, such as the mail queue, processor (Rosette) or dispatcher (reflective actors), and acquaintance list (state or behavior object), in addition to messages and whatever represents the object/actor itself. As before, extended behavior can be provided by substituting new versions of these components with enhanced behavior. For example, the sections on Rosette and ACT++ indicate that different mail processing policies can be implemented by replacing the mail queue metaobject. Also, as before, different ways of actually implementing reflection are illustrated. For example, many of the models use an approach involving explicit metaobjects. On the other hand, the model of Section 3.4.5 illustrates an approach using reflective operations defined on base-level objects, along the lines of the facilities for structural reflection described in the prototype-based object model of Section 3.3.

Some aspects of the structure of these models are particularly interesting. For example, in these models, the actors (or rather, the mail queues) have a distinct identity from the associated behavior objects, so that the behaviors can be replaced for the same actor. This is consistent with having separate method and state objects associated with a given base-level object, as in the Cecil [Cha92,93] language and proposed for the RISC object model. This is also consistent with some of the metamodels defined in earlier sections, where the metaobject effectively contains the base object's state and behavior, with the base object largely acting as an assertion of base-level identity. Also of interest is the way the object-state functions of ACT++ encapsulate the state variables, along the lines of the get and set operations proposed for the RISC state representation. This is also consistent with the state representation used in many prototype-based object models. The explicit use of thread objects in ACT++ is also of interest. The subject of explicit threads and processes will be discussed in more detail in Section 3.5.4.

The behavior set, enabled set, and synchronization set concepts described in several of the models are effectively multiple object *interfaces*. The OLE 2 [Bro94] and ODP [ISO93] object models, which are not actor-based, are other examples of models supporting objects with multiple interfaces. Since the models described in Section 3.4 are reflective models, these interfaces are represented as explicit interface objects, similar to the interface concept proposed for the RISC model. The OLE 2 definition of an interface as a set of method pointers may provide the basis for a reasonable definition of such an object, and is essentially the same as that of a behavior set (recall that the behavior set constructor takes as an argument a list of pointers to functions). (However, OLE 2 does not provide for an object to project different interfaces based on its state; rather, the different interfaces must be explicitly accessed by client objects.) The next-behavior-set function for determining which interface to use next, and similar computations for determining the state an object is in, are reflective computations, and some of these operations would constitute operations in a metaobject protocol for these objects. Concepts such as behavior sets are also a form of extended *type* mechanism, with enhancements to support the definition of *protocols* (intended *patterns* of behavior among objects), as opposed to the simple definition of the operations available for a given object. This is illustrated by the connection between the synchronization state and type state concepts identified in Section 3.4.1, and the example in that section of using these concepts to define objects supporting the two-phase commit protocol.

The way in which these models handle asynchronous message passing, and model the message passing of other models, is also of interest. [AFKP+93] notes that, in the actor model, even though the arrival order of messages is nondeterministic, every message sent to an actor is guaranteed to be delivered eventually. This suggests that it might be possible to model this in a synchronous model, with the message being delivered to a "guaranteed mail delivery system". Similarly, it has been noted elsewhere that an asynchronous mail system can be modeled in a model using synchronous communication if the model allows dynamic extensibility of a buffering agent (such as a mail queue) as an atomic action. This approach is used in the OMG Event Service [OMG93]. The ability to model these various forms of message passing in terms of each other is important in the analysis of object model interoperability, and also in the ultimate selection of a base message passing semantics for the RISC model. Similarly, the representation of the various forms of ABCL/1 messages in terms of asynchronous messages plus futures in Section 3.4.4 illustrates an approach that could be used in the RISC model to describe these sorts of variations reflectively. Combining these observations suggests that this could also be done using synchronous messages.

Concepts such as behavior sets, enabled sets, etc. also illustrate the recurring theme of the need to performing synchronization by conditionally blocking execution of specific methods in some way. Similarly, the models illustrate the use of related "communities" of objects (or the "synchronizers" of Section 3.4.3) in performing such synchronization, as well as other functions. As noted earlier, such facilities are related to the integration of concurrency control and transaction facilities in object models. Explicit conflict tables, together with control facilities implemented by separate transaction management metaobjects, would provide another way of specifying related constraints [HG93].

3.5 Examples of Reflection in Non-Actor Concurrent Languages

There are concurrent object-oriented languages in addition to the actor languages. This section describes several of these, in order to illustrate the primitives required in some other approaches to modeling concurrency.

3.5.1 ABCL/R2

[MMWY92] describes ABCL/R2, a reflective concurrent programming language. ABCL/R2 is a descendant of several earlier languages, one of which, ABCL/R [WY88,89] was also reflective. While the computation model of the ABCL languages is a descendant of the Actor computation model described in Section 3.4, it differs from the actor model in a number of respects. For example, an object may accept a message that is not at the head of its message queue. The model also supports both synchronous and asynchronous messages.

An object definition in the ABCL languages has the form:

```
[object object-name
   (state representation-of-local-memory )
   (script
      (=> message-pattern where constraint   ...action...)
                                ...
      (=> message-pattern where constraint   ...action...))]
```

An example of a message pattern (for a buffer object) is `[:put anItem]`. Any symbol starting with a colon in a message pattern is treated as a constant, while other symbols are pattern variables which are matched with incoming messages or their components and assigned values as a result. It is possible to add an expression of the form `@reply-destination` to some types of messages in order to explicitly specify an object to which the reply should be returned.

As noted in Section 3.4.4, the ABCL languages support three types of messages; *past* (normal asynchronous messages), *now* (synchronous messages), and *future* messages. Messages may also be sent in *ordinary* and *express* (higher priority) modes. Thus, the sending of a message M to an object T can be denoted in the following ways:

	ordinary message	express message
past type	`[T <= M]`	`[T <<= M]`
now type	`[T <== M]`	`[T <<==M]`
future type	`[T <= M $ x]`	`[T <<= M $ x]`

(where x denotes a future variable in the future type messages).

In ABCL/R, each object in the system has its own metaobject. Objects and metaobjects can send messages to their metaobjects. The metaobject models both structural and behavioral aspects of an object X and contains the following components (explicitly represented as objects) as the values of its state variables:

1. the message queue in which messages sent to X are placed.
2. the state variables of X.
3. the set of scripts (method definitions) associated with X.
4. the *evaluator* which evaluates methods associated with X.

The values of the instance variables can be manipulated by sending messages to the metaobject. Thus, it is possible to request the metaobject to acquire and replace methods. The methods associated with the metaobject handle the behavioral aspects of X such as message queue management, method search (dispatching), and method execution. (These methods can in turn be modified, because they are modeled and available as values of instance variables of the metaobject's metaobject. Such modifications are made by sending messages to this *metametaobject*). In addition to these structural aspects of an object, the computation aspects, *arrival*, *reception*, and *acceptance* of messages, and the *execution* of scripts, are all explicitly represented.

ABCL/R2 extends this model in a number of ways. For example, in order to reduce the overhead of reflective computation, ABCL/R2 allows the definition of *non-reifying* objects, which lack individual reflective capabilities. A non-reifying object has a special metaobject that does not allow metalevel operations.

An object in ABCL/R2 always belongs to some *group*. Objects in a group share computational resources which are represented as *group kernel objects* at the metalevel. The structure and computational behavior of a group are defined at the meta and higher levels of the group by the group kernel objects. Objects `meta-gen` (metaobject generator), `GMgr` (group manager), and `Eval` (evaluator) are standard group kernel objects. Other group kernel objects can be defined by the user to represent other types of shared resources.

For each object, there is an individual tower of metaobjects which determines the structure of the object, including its scripts (methods). Thus, for example, reflective operations to alter scripts are performed in this tower. In addition, there is a separate *group tower* of metagroups which determines the group behavior, including the computation (evaluation) of the scripts of the group members. Group-wide operations that share group resources, such as scheduling, are defined in this tower.

The group manager represents and manages the group. Messages sent to the group are actually forwarded to the group manager. The paper gives an example of the definition of a "concurrently-controlled-group" by defining a new group with specialized meta-gen and Eval objects.

On object creation, the metaobject generator creates a corresponding metaobject. When a form `[object ...]` is evaluated by the evaluator, a message `[:new statespec lexenv scriptset evaluator gmgr]` is sent to the metaobject generator. A metaobject created by the standard metaobject generator contains: a message queue object, a set of scripts (methods), a set of state variables, references to the evaluator and the group manager, and the execution mode. The reception of a message `M` by a base-level object corresponds to its metaobject receiving a message `[:message M R S]` (where `M` is the message, `R` is the reply destination, and `S` is the sender). The metaobject then places the message `M` into its message queue object. The metaobject also searches for a script that matches a message at the top of the queue and, if it finds one, sends a message to the evaluator to execute the script.

The evaluator is a computation resource shared by the base-level members of a group. Its role is to evaluate the scripts of member objects. Unlike its role in ABCL/R, where it was a stand-alone, private object, the ABCL/R2 evaluator interacts with other group kernel objects and metaobjects for group management, such as scheduling. Its typical behavior is as follows: it accepts a message to evaluate an expression `[:do exp env id gid eval]` whose reply destination is a *continuation object* `C`. On receipt of the message, it evaluates expression `exp` under environment `env`, then sends the result to `C`. The arguments `id` and `gid` are mapped to the pseudo variables `Me` and `Group`, the former denoting the object itself, and the latter the group of the object. A number of object-oriented concurrent programming languages create continuation objects for use in synchronization, or to which results are delegated. The metacircular definition of metaobjects in ABCL/R2 is an example: when a metaobject sends an expression to the evaluator for execution, the metaobject creates a continuation object, which receives a reply from the evaluator, and which in turn indicates the end of evaluation by sending an `:end` message to the metaobject. This allows script execution and message reception to occur simultaneously.

Reflective programming in ABCL/R2 is performed in two ways. In order to affect the behavior of an individual object, the metaobject is sent messages or customized. In order to affect group-wide behavior such as resource sharing, messages are sent to the object's group, (`group-of X`), or the group manager object is customized.

Operations are categorized into reflective and non-reflective operations. In ABCL/R2, the following operations can be reflected upon:

- references to variables (including lexical variables, and pseudo variables `Me` and `Group`).
- message sending (both "past" type (asynchronous) and "now" type (RPC)).
- object and group creation

Other primitive operations such as arithmetic operations are not subject to reflection, and are thus compiled rather than interpreted.

[MMWY92] notes that the current version of the language allows too much freedom in reflective programming, and that a topic of current research is to define an appropriate metaobject protocol that allows only valid customizations, and gives the compiler more apriori information.

An appendix of the paper describes how Time Warp (TW) scheduling can be implemented, as an example of how a scheduling mechanism can be encapsulated at the metalevel of a user program using the reflective capabilities of the language. The Time Warp mechanism was first proposed for controlling the sequence of events in distributed discrete event simulation [Jef85], and was later extended for operating systems [J+87], and process synchronization and concurrency control in database systems [JM86]. A chronological scheduler for multidatabase and distributed object systems using a variant of this approach is described in [GRL94].

As described in [GRL94], the database variation of TW views transactions and data items as objects that encapsulate sequential processes and communicate via timestamped messages. TW uses partial transaction and data object rollback to synchronize concurrent transactions. To implement partial rollback, objects maintain input, output, and state queues. Messages received by an object are inserted in its input queue. Objects process input messages in timestamp order. When a message changes the state of an object, the new state is recorded in the object state queue. In addition to changing the state of an object, the processing of an input message may cause one or more messages to be sent to other objects. Before sending such a message, the object creates an inserts a corresponding *anti-message* in its output queue. Anti-messages are sent only as part of an object rollback. Whenever an object receives the anti-message of a message m, it undoes the effects of m, using the information in its state queue to recover the old state.

A transaction execution begins when a transaction object receives an invocation message from an application. The TW mechanism stamps the message with a timestamp which becomes the timestamp of the transaction. Subsequent messages from the same transaction carry the transaction timestamp. Objects process messages they receive in timestamp order. A synchronization conflict in TW arises when a new message arrives out-of-order, i.e., has a timestamp TS that is less than the timestamp of the last message the object has already processed. TW neither delays read and write operations so they can be processed in timestamp order, nor aborts and restarts transactions with they issue out-of-order messages. Whenever a new message with timestamp TS arrives to a data object out-of-order, the TW mechanism performs a *time warp* as follows: First, the object that received the out-of-order message finds the state associated with the largest timestamp that is less than TS and discards all states with greater timestamps from its state queue. This action rolls back the object to a time just before TS. Next, the object sends all anti-messages that have been recorded in its output queue and that have timestamps greater than TS. These anti-messages cancel the effects of messages incorrectly sent to other objects. Canceling the effects of an incorrect message at some object may also require a partial rollback there. Finally, the object processes the new message and starts re-executing messages with timestamps greater than TS, again in timestamp order. When transactions complete all their operations, they are committed in timestamp order.

In the ABCL/R2 implementation, a *Time Warp group* is defined, whose members are specialized by their individual metaobjects so that they coordinate in running the TW algorithm. By specializing the metaobject generator, the TW algorithm itself is encapsulated at the metalevel, and group membership automatically dictates TW behavior. Messages sent within the group have the form `[target <= message @ reply-destination :vrt virtual-send-time]`. A TW scheduler object is also introduced for controlling computation within the group. The scheduler does not interact directly with the evaluator of the group, but rather interacts with the *metaobject* of the evaluator. This metaobject is customized so that an evaluation request message (corresponding to a base-level method evaluation) is first sent to the scheduler. The metaobject then asks the scheduler for the next method evaluation it should perform, as determined by the algorithm of the scheduler. With this approach, the same TW algorithm can run independently of differences in the scheduling algorithm at the metametalevel (e.g., FIFO versus lowest-local time first; such changes can affect how much rollback takes place as the result of out-of-order messages). Moreover, it would be easy to extend the Time Warp group to add inter-scheduler communication, and/or to adapt to a distributed environment.

This Time Warp mechanism illustrates yet another example of how extensions, particularly those related to concurrency and synchronization, can be implemented using reflective capabilities, together with some form of object "group" or "community" mechanism.

3.5.2 Apertos (Muse)

[Yok92] describes the Apertos reflective object-oriented operating system (formerly called Muse). Apertos attempts to handle problems of object migration and provision of object services in a heterogeneous network, in which objects may have to move between nodes, such as in mobile computing applications.

A base-level Apertos object has a structure similar to that in the ABCL languages described above. Each object encapsulates state and methods which access the state. The semantics and properties of an object may change while an object is running and when it evolves or moves. To deal with this, a metaobject is also defined for each object. In this approach, the object acts as a container of information, while the metaobject defines the semantics of its behavior. The metaobject defines, among other things, how the object communicates with other objects, the memory management mechanism used by the object, and how faulty operations are to be handled. The following pseudocode describes an object's properties at its metalevel, in terms of a `Descriptor` object with eight slots. Each slot denotes a metaobject.

```
metaobject Descriptor {
     myname is       Identifier;
     execMode is     State of execution;
     instance is     Permanent segment;
     text is         Shared segment;
     queue is        Queue of messages;
     userStack is    Execution stack;
     comm is         Method for communication;
     memory is       Method for storage management;
}
```

An object is also supported by a group of metaobjects called a *metaspace*. An object has execution semantics that are cooperatively provided by the metaobjects in its metaspace. Metaobjects in the metaspace provide services such as naming (including generation of object identifiers), segment and page management, file management, and network services. The metaobjects in the metaspace are either directly or indirectly referenced by an object's metaobject when it is in that metaspace. This structure is similar to that used in ABCL/R2 above, in which objects get their semantics not only from individual metaobjects, but also (partially) from metaobjects in a *group*.

For object creation, Apertos has defined several class systems, each of which is tailored for individual programming languages. A class system is a metaspace which provides services for object creation and defining the immutable properties of class instances. The class hierarchy defined by a class system is only available at compile time (the class hierarchy is collapsed when the object is compiled). The class hierarchy and metaobject structure are separate in Apertos, because they have different purposes: the class hierarchy defines the structure of an object, while the metaobjects define the computational semantics. This is similar to the separation of structure and behavioral reflection described in [Fer89] (Section 3.2).

Changes in an object's execution environment are represented by the concept of *object migration*. Objects migrate by changing their metaspaces. For example, the movement of an object from main memory to secondary storage is modeled by the migration of the object to a metaspace which represents a secondary storage. Similarly, an object might migrate to an environment in which debugging facilities are available when it is to be debugged. More significantly from the point of view of mobile computing, disconnected operations for portable computers, performed while these computers are disconnected from the network, are described in terms of object migration. Here, the object migrates to the portable computer metaspace, which limits the services available to the object (e.g., network communications).

Metaspaces are explicitly represented by metaobjects called *reflectors*. A hierarchy of reflectors is defined in Apertos. Basic reflectors include `mZeroObject` (the reflector of reflectors), which provides reflector capabilities to metaobjects, and `mBaseObject` (the reflector of base objects), which provides basic concurrent object capabilities to base objects. `mBaseObject` provides two interfaces, one used by base objects, the other used by other metaobjects. The interface for base objects includes operations:

`Call`—invokes a method defined in the target object

`Reply`—delivers the result of an operation to the sending object
`New, Delete`—creates/removes an object
`Grow, Shrink`—changes the size of an object
`Find, Install`—returns/creates a mapping between an object and its reflector
`Migrate`—moves an object to another metaspace

The interface for other metaobjects includes operations:

`Deliver`—delivers an incoming message send by other reflectors
`Migrate`—handles an incoming request for object migration
`Preempt`—interrupts the running object and begins execution of another (this is usually used by mBaseObject)

Finally, `MetaCore` is a terminal metaobject located in each computer. It has no metaspace, and it provides the common primitives for object execution.

When an object is migrated from one metaspace to another, it is desirable that the new metaspace be such that it can continue to provide the services required by the object (in particular, the object may need to migrate while it is executing). In support of this, reflectors are defined to support a common method called `canSpeak` which is used to check the compatibility of reflectors or metaspaces during object migration. When an object can continue execution when it migrates to a given metaspace, the source and destination metaspaces are said to be *totally compatible*. When the object can continue its execution, but with some restrictions, the metaspaces are *partially compatible*. When an object cannot continue execution in the new metaspace (e.g., a movement to secondary storage), the metaspaces are *incompatible*. The `canSpeak` method checks a separate reflector class hierarchy in determining metaspace compatibility. Metaspaces are designed to take appropriate action when an incoming object's execution is only partially compatible with the metaspace. For example, an object moving from a computer connected to a network to a portable computer disconnected from the network is moving to a metaspace that is only partially compatible with the original one, since it does not support network communication. The portable computer's metaspace must be prepared to take appropriate action if the migrated object sends a remote request .

The paper notes that use of separate metaobjects (and hence reflection) in defining the characteristics of base-level objects frees programmers of base-level objects from certain dependencies on the execution environment. For example, persistent object storage can be implemented as a metaspace, and objects can be made persistent in a reasonably transparent fashion by moving them to that metaspace. A number of other service-related advantages of this reflective approach, useful in general-purpose distributed object environments, are also cited:

- Since metaobjects are responsible for delivering messages, they can choose an optimal network protocol for delivering messages, transparently to the base object.

- Since the internals of an object are described by metaobjects, and the semantics of an object are defined by a metaspace, services can be transparently provided that are optimal for the type of object involved. For example, memory management suitable for the granularity of the object can be chosen.

- Since the execution environments of objects are defined as separate metaspaces, metaspace definitions can be used in migrating objects between old and new system implementations, or to different nodes for purposes of load balancing. .

3.5.3 Explicit Versus Implicit Modeling of Processes

[GW93] describes an object-oriented concurrent programming model that, although not explicitly reflective, illustrates a somewhat different approach to concurrency than those described earlier. Many proposed architectures for concurrent object-oriented systems, including most of those based on the actor model and others described earlier in this report, define a base class of "active" objects as the unit of concurrency, and then require the designer to derive problem domain classes from that class or its subclasses. However, in the approach described by [GW93], problem domain objects are derived from ordinary "passive" object classes (objects that act only in response to messages). The resources for concurrent processing are provided by two special object classes, `Process` and `Community`. While these can be specialized through subclassing, it is not intended that application domain classes be derived from them; they are intended to remain detached from the application domain class hierarchy.

A `Process` provides a single thread of execution and is the fundamental resource for execution. Concurrency is achieved by creating multiple instances of `Process` and assigning them different tasks to perform. Any object can create an instance of `Process` in the usual way, by sending a message to the `Process` class, and can then assign that `Process` work to do by sending it a message of the form:

```
aProcess have:  anObject  perform:  aMessage
with:  aListOfArguments
```

The `Process` object is thereby requested to send `aMessage` to `anObject` using a separate concurrent thread of control. The combination of creating a new `Process` object and sending it the above message has the same effect as "forking" a new process. The `Process` object itself does not know how to do useful work. By sending a specified message to a specified object, it provides motive force by lending its thread of execution to the object that knows how to do application-specific work. A variation of the above message allows the message to be sent repeatedly at a prescribed interval.

Objects within an application can have one or more private `Processes` to support concurrent work. As `anObject` executes the method associated with `aMessage`, it can send messages to other objects. In doing so, the thread of execution of the private `Process` (`aProcess`) is temporarily passed to the receiver, to be returned when the receiver finishes processing the message. In this way, the thread may effectively be passed to any object reachable by the owner. Objects may have individual private processes by creating a `Process` and sending it a message with `self` as the `have` argument, as in:

```
standardProcess have:  self  perform:  standardize
with:  NO_ARGS every: 30 :MINUTES
```

A `Community` is a special type of `Process` that provides a home for other objects. It contains a collection of objects (its members) that work together as a group. A `Community` provides the facilities for its members to communicate with each other as well as with objects in other communities. Since it is a `Process`, each `Community` supplies a thread of execution and message-handling facility for its members. It also provides communication ports through which messages to and from objects in other communities are transparently routed. Communities are designed to facilitate interactions among objects requiring close coordination with each other.

Each `Community` continually monitors its message queue for incoming messages. When a `Community` extracts a message from the queue, it forwards the message to the intended object, transferring the thread of control to that object. In responding to the message, the object can send messages to many other objects. Messages to members of the same `Community` are executed either as direct procedure calls or as efficiently as dynamic binding will allow. Messages to members of other communities are transparently forwarded by the sender's `Community` to the message queue of the receiver's `Community`, with an option for the sender to either wait for a reply or proceed without waiting. The senders and receivers see no differences between remote and local messages. When an object has finished responding to a remote message, it returns the thread of control to its `Community` , which returns the result to the sender by the reply port of the sender's `Community`.

A third kind of object, the `Processor`, represents a physical machine with a real-time operating system kernel that simulates concurrency. Communities and processes are aware that they are associated with a particular `Processor`.

All objects wishing to receive communications from objects in other communities register themselves as *global* objects. This means that the object is accessible from any address space managed by the operating system, including remote computers connected by communications channels. Each globally accessible object is known by a name. A name is a character string and the establishment of the name is accomplished through a registration process, which uses the global name store of the operating system to publish the global object's name for use by other objects in the system. Communication with a global object is possible by any object that "links to" a global object by name, using the returned identity handle as the "address" of the communication.

The approach also supports event-driven execution. Access-oriented programming (triggers) is in some respects a dual of object-oriented programming. In object-oriented programming, procedure invocation (the result of sending a message) causes a hidden modification of state. In access-oriented programming, direct data (state) access causes a hidden invocation of procedures associated with the data. Access-oriented programming allows execution to be data driven, i.e., the system can perform certain functions automatically whenever selected data is updated. This is useful in the construction of real-time applications. It allows the system to implicitly determine the order in which various operations must be performed from its data flow requirements. In distributed systems, data-driven execution is particularly valuable because it provides automatic synchronization of process execution across processor boundaries and avoids unnecessary polling over the network.

To support this, an object class called `Data` is defined that encapsulates arrays of values or scalars. `Data` provides get and set methods on its elements and is subclassed to encapsulate each of the ordinary C data types. The get and set methods hide interactions with the operating system to notify "subscribers" that the data has been modified, examined, or touched. Any `Process` object can subscribe to any `Data` object, then go to sleep, and be awakened by the OS whenever the data is accessed in the specified way.

Another paper describing an alternative approach for modeling concurrency is [Car91], which describes Eiffel //, a parallel extension to the Eiffel language. Briefly, the paper observes that concurrency control can be specified in either an explicit or implicit fashion. In an *explicit* mechanism, concurrency control involves the explicit programming of a thread of control, and typically a specific instruction for explicit message reception. The ABCL languages discussed earlier, as well as Ada, POOL, and Eiffel //, belong to this category. In an *implicit* mechanism, the definition of concurrency control is generally declarative, and explicit programming of a thread of control is not involved. The behavior sets of ACT++ and the enabled sets of Rosette are examples of implicit mechanisms.

The paper argues that each form of control is necessary in some circumstances and that, as a result, what is necessary is an *explicit* control mechanism, together with the flexibility to define the various *implicit* mechanisms as abstractions using the explicit mechanism. This would allow an escape to the explicit mechanism if necessary, and for users to define their own implicit mechanisms as required. In addition to the need for an explicit control mechanism, the paper states (and justifies) a number of other requirements. To sum up, the identified requirements are:

1. An explicit control mechanism is necessary (as noted already).

2. The concept of a process (or thread) must be defined as a class, with the process body being an object method. Defining the process concept as a class allows for the definition of abstractions as subclasses. Since the process body is a method, it can be redefined in the subclasses to provide whatever specific behavior is desired.

3. Methods must be first class objects. In order to build concurrency abstractions, it is necessary to program the synchronization of the operations exported by an object. This in turn requires the ability to dynamically manipulate methods (e.g., to construct enabled sets).

4. Requests (messages) must be first class objects. This allows for the semantics of request selection to be explicit programmed (and modified as required).

The paper notes that this modeling of various aspects of the system as objects corresponds to a defining a metalevel which reifies methods and messages.

3.5.4 Commentary

The models in this section, to the extent that they support reflection, repeat the themes already identified, i.e., explicit representation of object model components as metaobjects, meta-level operations defined for those metaobjects, and explicit representation of key "events" in the processing of messages (e.g., sending and arrival of messages).

The ABCL and Apertos models have a lot in common with the actor models described in Section 3.4, and so many of the basic structural aspects of the models are similar. In particular, in the ABCL languages, the base-level object is effectively only an identifier, since the metaobject has all the state as well as the methods. This is slightly different from the approach used in the models described in Sections 3.2 and 3.3, in which structural components such as methods and state were accessed via the base level.

The models described in Section 3.5.3 are somewhat different from the actor models, although they do exhibit some similarities. For example, the class `Data` of [GW93] is similar to the explicit behavior objects of the actor models (an explicit state representation), and is also similar to the corresponding RISC model concept. The way in which access-oriented programming can be provided using this class illustrates the power of modeling state using get and set operations, and is similar in structure to the OMG Event Service [OMG93].

These models are particularly interesting in that they describe the need for explicit process or thread concept to model concurrent processing in certain object models. This approach might be useful as an aspect of the RISC model, because it explicitly models processes (or threads) and the association of those processes with objects. This permits modeling of passive objects, active objects with only one process, with multiple processes, and other sorts of mixtures, in an explicit way. Some of this process interaction is presumably reflective. In particular, a `Process` object could be considered a metaobject like a method or state object, as illustrated by the ACT++ thread objects described earlier. The requirements identified in the Eiffel // discussion just above, including the need for explicit threads, methods, and messages, are also consistent with the general theme of providing reflective capabilities to allow for the flexible modeling of the different control mechanisms, and are consistent with observations made in Section 2 concerning the RISC model.

These models also repeat aspects of the different applications of reflection, and mechanisms for implementing those applications, described earlier. For example, the "global objects" mechanism of [GW93] is similar to the facilities described earlier for implementing distributed message sending, and is also similar to the Network Objects described in Section 3.6.1 and the general facilities provided in OMG's CORBA specifications [OMG91a].

These models also once again illustrate the use of object *groups* in implementing various facilities. The groups of ABCL/R2, the metaspaces of Apertos, and the communities of [GW93], are similar to the communities in Open C++, and play similar roles in various applications, such as transaction management. This is illustrated, e.g., by the example of time warp scheduling using ABCL/R2 groups. Such groups illustrate a useful structuring for objects in distributed networks. As in the Choices system, there will be objects that constitute significant components of an environment that are shared by many objects, such as the persistence and transaction services provided to all the objects in, e.g., an OODBMS, or in a CORBA-based distributed object system. Such components can be treated as metaobjects which are shared by the objects participating in the group. The mechanism for representing object migration in Apertos provides an approach for describing the needs of objects that must change location in such systems.

3.6 Other Examples of Reflective Concepts

This section contains subsections that illustrate either explicit reflection, or aspects similar to those seen in reflective models. They are included because they contain interesting variations on reflective themes seen already in other models, reinforce the use of reflective concepts in certain kinds of applications, or otherwise contribute to defining the concepts required in the RISC model.

3.6.1 Modula-3 Network Objects

[BNOW93] defines a mechanism for implementing what are called *network objects* in Modula-3. In this approach, a *network object* is an object whose methods can be invoked by other programs, in addition to the program that allocated the object. The program invoking the method is called the *client* and the program containing the network object is called the *owner*. To implement network objects, the reference in the client program actually points to a *surrogate object* (proxy), whose methods perform remote procedure calls to the owner, where the corresponding method of the owner's object is invoked. The client program need not know whether the method invocation is local or remote. The mechanism bears some resemblance to the object adapters used in OMG's CORBA specifications [OMG91a], but also illustrates aspects related to the use of reflective facilities, and in particular some of the mechanisms described above for providing distributed capabilities using reflection (even though the mechanism itself is not strictly reflective).

The surrogate object's type is declared by a stub generator rather than written by hand. An object type as seen by a client actually involves three related types:

- the network object type `T` which is given to the stub generator,

- the surrogate type `TSrg` produced by the stub generator; this is a subtype of `T` with method overrides that effectively perform remote procedure calls, and

- the type `TImpl` of the real object allocated in the owner, which is also a subtype of `T`.

The type `T` is required to be a pure object type; that is, it declares methods only, no data fields. The type `TImpl` generally extends `T` with appropriate data fields.

In addition to providing some form of inter-address space communication, and a method of communicating type information from one address space to another, the underlying system must provide for a certain amount of what amounts to structural reflection. Specifically, the system must provide object types with single inheritance and the ability to test the type of an object at runtime, to find the code for the direct supertype given the code for the type, and to determine at runtime the sizes and types of the fields of an object, given the type of the object.

The approach insulates the main part of a network object runtime from differences in communication protocols via an abstraction called a `Transport.T`. A `Transport.T` is an object that generates and manages connections between address spaces. Different subtypes use different communications methods. Each subtype is required to provide a way of naming address spaces. A transport-specific name for an address space is called an *endpoint*. One of the two connections in a pair of address spaces is termed the *client* side and the other is termed the *server* side. Transports are required to provide a thread that listens to the server side of a connection, and calls into a network object runtime when a message arrives indicating the beginning of a remote call. This latter process is called *dispatching*.

Dispatching takes place as follows. The "wire" representation of a network object is a pair (sp, i) where sp is a `SpaceID` (a number that identifies the owner of the object) and i is an `ObjID` (a number that distinguishes different objects with the same owner). Each address space maintains an object table `objtbl` that contains all its surrogates and all its network objects for which any other space holds a surrogate. If a non-surrogate network object is present in the object table, it is said to be *exported*; otherwise it is *unexported*. The concrete representation of a network object includes a state field that records whether the network object is a surrogate, and, if not, whether the object is exported. If the object is a surrogate, a location field in the object generates connections to the owner's address space, and the dispatcher field is unused. If the object is exported, then the dispatcher field identifies a dispatcher procedure for the object, and the location field is unused. The dispatcher call `obj.disp(c, obj)` unmarshals a method number and arguments from connection `c`, calls the appropriate method of `obj`, and marshals and sends the result over `c`. The dispatcher methods are typically written by the stub generator.

3.6.2 CLOS

The metaobject protocol developed for CLOS [KRB91] has been mentioned several times already[1]. In CLOS, the operations defined for objects are called *generic functions*, and the concept corresponding to message processing is generic function invocation. This form of object model was described in detail in [MH93]. Conceptually, CLOS divides generic function processing into three phases: determining which methods are applicable to a given invocation (as determined the classes of the required arguments and by constraints associated with each method called *method specializers*), sorting the applicable methods into decreasing precedence order (also determined by the method specializers and the classes of the required arguments), and sequencing the execution of the sorted list of applicable methods (since, in CLOS, more than one method may be executed in response to a given invocation). In CLOS, this process is defined in its metaobject protocol as a set of explicit generic functions which operate on defined metaobjects. The sequence of metalevel generic functions invoked in generic function invocation is:

find class (i.e., find the class object using its name)
apply-generic-function
compute class precedence list
compute applicable methods using classes
apply methods

Variations in model behavior could be introduced by altering these generic functions, or the generic functions of metaobjects which they call. (Variations could also be introduced by altering the *sequence* of generic functions invoked, although this is not permitted in the CLOS metaobject protocol.) The metaobject classes defined as part of the CLOS metaobject protocol include classes, slot (instance variable) definitions, generic functions, and methods.

3.6.3 DOM 3.0

[NH93a,b] describe a series of experiments with the GTE Laboratories' DOM 3.0 prototype distributed object manager, which illustrate the wide range of flexibility that can be achieved using aspects of reflective technology.

DOM 3.0 implements a very primitive object model, from which more complex facilities can be constructed. This is based on early RISC-like ideas. From the perspective of an object's client, the most fundamental abstraction in DOM is the *invocation*. Every request for object functionality is an invocation. The body of an invocation is a grouping of objects $(o_1\ o_2\ \ldots\ o_n)$. When DOM receives an invocation, it picks the first argument in the invocation, called the *receiver*, and forwards the rest of the arguments to the receiver. The

---

[1] The discussion of the CLOS metaobject protocol here is rather brief. This is due to its having been discussed in [MH93], and also because it is relatively well-documented [KRB91]. However, in spite of the brevity of the discussion here, ideas from [KRB91] contributed significantly to early thinking about the RISC model, as indicated by the number of citations to it sprinkled throughout RISC model reports.

real responsibility of finding and executing the code implementing the invocation semantics rests with the receiver.

From the perspective of the implementor of an object, the most fundamental abstraction in DOM is the object's *dispatcher*. A dispatcher is a piece of code. When an invocation is forwarded to an object, the object's dispatcher is executed. All objects within DOM, including primitives such as integers and strings, model abstractions such as object identifiers, classes, attached systems, and user-defined objects, provide their functionality to the system by specifying their dispatchers.

To make invocations in DOM, most clients call the procedure `Obj.Invoke`, which provides a uniform method for invoking operations on all DOM objects. The body of `Obj.Invoke` assumes that the first argument of the invocation is the receiver. When `Obj.Invoke` is called, it calls the receiver's dispatcher, including the other arguments in the invocation. `Obj.Invoke` hides the notion of dispatching from clients; all they know is that they are invoking an operation on an object.

The experiments reported on in [NH93a,b] examined a wide range of models, systems, and paradigms, including:

- class-based and generic-function-based object models
- client-server and distributed object concepts
- object-oriented databases
- rule-based systems

with the goal of showing that a simple but customizable notion of dispatching allows for the integration of diverse object models and systems. Using the flexible dispatching provided by the DOM prototype, code written in a wide variety of models and systems, including Modula-3, C, C++, Macintosh Common Lisp, CLIPS (a rule-based system written in C), and object-oriented and relational DBMSs, was integrated more easily than would have been possible using more conventional techniques.

For example, one of the experiments involved the implementation of a "network object" capability similar to that described above for Modula-3. This involved defining objects to serve as local "proxies" for remote objects, and was done by effectively modifying the dispatchers of the proxies to package the contents of messages sent to them into network messages, perform remote procedure calls, and return the responses to the local clients.

3.6.4 Variations in Control Mechanisms[1]

The metalevel control structure defining the execution model in a reflective system can be either expressed in a *declarative* or a *procedural* language. Expressions in a procedural language can be understood only in terms of the behavior of the (metalevel) interpreter of the language (the actions it takes, the order in which it does things, etc.), whereas a declarative language states true facts that can be understood without reference to the behavior of the metalevel interpreter [vHar89]. The models described earlier illustrate the procedural approach to expressing the execution model. An example of the use of a declarative language to express the execution model is found in the *law-governed system* of [MR87]. The basic idea behind this approach is that the discipline governing the exchange of messages between objects should be specified by an explicit component of the system called its *law*. An example of such a discipline is the particular inheritance strategy used in

---

[1] These mechanisms were also described in [MH93], and are included here for completness.

a language. Ordinarily, such a discipline would be fixed, and implicit in the language. In the law-based approach, the discipline is explicit and modifiable. In [MR87], this law is a set of rules represented by Prolog clauses. The law can be tailored to the specific needs of a particular application, e.g., choosing a particular inheritance or delegation strategy, or even using different strategies for different parts of the same system. Because the law is explicit, the law can easily be modified, and adapted to new situations. [MR87] illustrates how simple rule changes can modify a basic object model to support various forms of discipline, including enforcement of various types of encapsulation, simple and multiple inheritance, active values, instance and class methods, and other types of rules. The use of rules to express differences in encapsulation and inheritance is also described in [AT88], as part of a more conventional object model.

Execution models for distributed object systems have also been based on asynchronous messages or events, instead of on synchronous message sending. For example, [JS92] describes such a distributed object system. In this system, a given node in the system is defined as containing:

- a set of objects.
- a *distributor* (similar to an Object Request Broker), which sends and receives messages, and is in charge of event communication.

An active object in this system includes, in addition to state, methods, an identifier, etc.:

- a set of triggers (rules).
- a set of relevant events.
- an event processor, which communicates with the distributor, and determines trigger firing.

Objects are generally idle, but perform actions on receiving an interrupt. An interrupt is caused by either the detection of an internal object event, or the receipt of a message from the distributor. The event processor handles an internal event by having the object enter an idle state, and sending a message to the distributor with the event as its content. Eventually the distributor sends back a message to the object notifying it of the same event. This is handled by the triggers, after which execution resumes from after the interrupt point.

The event processor handles a distributor message (including those sent for internal events) by appending the reported message event to the object's event history (a queue). The predicates of all active triggers are checked against the event history, and the trigger fires if its predicate is satisfied. All fired triggers are executed one at a time in some unspecified order, with the object in an interruptable state. After the trigger executions, control returns to the point prior to the interrupt introducing the message event. (While executing a trigger body, new interrupts may occur, and new triggers may fire).

Method invocation in objects is done through the event mechanism. When an object `O` wants to execute a method at another object `P`, it (conceptually) detects a local event `fstart`. This event is then posted to `P`, and causes the execution of a "trigger body" which is nothing but the desired method. Meanwhile, `O`, on having `fstart` posted to it, fires a trigger that blocks further execution. Once `P` completes execution of the method invoked, it detects an event `fend`. This event is posted back to `O`, and causes a trigger to fire that permits `O` to resume execution. ([JS92] notes that, even if method invocation is not really implemented directly using events, it is important to detect when methods start and end for transaction processing purposes.) When an event message is to be sent to a remote object, the local distributor sends it to the remote distributor. The distributors cooperate to obtain a correct execution. The nature of the cooperation depends on the notion of correctness required.

Constructs similar to the explicit mail queues described above in connection with concurrent object-oriented programming languages have been described in a series of electronic mail interactions relating to the OMG's new work on CORBA 2.0 describing aspects of *asynchronous* message support by Object Request Brokers (ORBs) using *synchronous* messaging as their underlying technology. These interactions described a basic facility for asynchronous messaging in an ORB component based on the following principles:

- Messages to objects should be first class (i.e., explicit) objects. This is independent of whether messages are synchronous or asynchronous, except that requests retained for access by applications should be given a full interface as first class objects.

- The queue of incoming requests should itself be an object, providing the ability to browse the pending request objects and remove them for processing in arbitrary orders. This queue should be a persistent resource with database-like transactional integrity across server outages or machine crashes.

- Activation of object processes to accept requests should not necessarily occur automatically in response to delivery of the request. A wide range of object activation policies, including activation established independently by an external process, should be supported.

It is the representation of the messages and message queues as explicit objects that allows different policies for handling messages to be integrated into the system.

3.6.5 Commentary

The facilities described above illustrate principles seen in the reflective models described in earlier sections. For example, the need for explicit control of the dispatching (message sending) process is illustrated in its use for implementing distributed objects in the Modula-3 Network Objects, and for numerous other purposes in DOM 3.0. In addition to modeling aspects of different object models (including such unusual models as Linda [CG89] and the various rule/logic-based models [Man89]), the ability to explicitly represent dispatching and other control aspects of object models allows for a number of potential generalizations that are important in a distributed object environment. For example, in such systems dispatching could potentially be a highly complex process, requiring selection of methods on the basis of node availability, system load, and other factors, and might also be application-, object-, or model-dependent. It should be possible to define such complex dispatching mechanisms using the RISC metaobject protocol.

These facilities also indicate the range of approaches possible in implementing message processing, and emphasize the essential equivalence of synchronous and asynchronous approaches. In particular, the model of [JS92] illustrates the need to detect key "events" in message processing, whether this is done using an explicit event mechanism, or by the invocation of explicit metalevel operations as in the reflective models described earlier.

The use of explicit queues in the discussion of asynchronous messaging in CORBA effectively involves a form of reflection, similar to that found in the reflective actor models. The Object Request Broker itself could be considered a metaobject in one of the object "groups" or "communities" described earlier. The "distributor" of [JS92] plays a similar role.

The discussion of Network Objects in Modula-3 also (implicitly) illustrates yet another example of an "object group" or "object community" concept, with an address space playing the role of the object community. The Network Objects also resemble the "global objects" described in Section 3.5.3.

Finally, although not described explicitly in Section 3.6.2, [KRB91] illustrates applications of metaobject protocol capabilities relevant to interoperability. For example, [KRB91] illustrates how CLOS objects could be defined to support C++-like encapsulation concepts through the use of the metaobject protocol. The use of reflection in defining Smalltalk-like garbage collection in C++ illustrated in Section 3.3.2 is another example. The ability to simulate the capabilities of one object model in another is one way to address object model interoperability issues, and the ability of reflective facilities to support this illustrates the value of reflection in addressing these issues.

Section 3 has described a number of different object-oriented languages or systems that either explicitly support reflection (meta-level concepts), or that contain aspects related to reflection that address issues from the previous RISC object model report [MH93] that required additional work. In addition to material describing each language or system, commentary was provided on the various features identified. The intent was to look at what designers who are explicitly trying to provide reflection (and thus are explicitly trying to identify the key primitives in their own languages) are actually identifying as key concepts, in order to contribute to a better understanding of the concepts that the RISC model should contain. The next section will discuss conclusions reached as a result of this work, and describe future work planned.

4. Conclusions and Future Work

The primary purpose of this report has been to describe progress made toward identifying the basic concepts of a "RISC" object model. Section 2 has described the preliminary set of basic concepts identified for that model in [MH93]. Section 3 then described a number of different object-oriented languages or systems that support aspects of *reflection* (meta-level concepts), a key aspect of the RISC model, or that address issues from the previous RISC object model report that required additional work. In addition to material describing each language or system, commentary was provided on the various features identified. The intent was to look at what designers who are explicitly trying to provide reflection (and thus are explicitly trying to identify the key primitives in their own languages) are actually identifying as key concepts, in order to contribute to a better understanding of the concepts that the RISC model should contain. This section summarizes the conclusions that can be drawn from the object models described in Section 3, and by comparing, at a high level, the concepts identified there with those identified earlier for the RISC model.

4.1 Conclusions

As noted in the various commentary sections in Section 3, the descriptions of reflective object-oriented languages in Section 3 illustrated a number of basic themes that were repeated over and over, in various forms. In each case there was a desire to be able to extend the language or system (or its behavior) in a modular way, without having to modify the individual application objects. Since the basic operation in any object-oriented language or system is message passing, the basic approach to providing the extensions was by extending what happens during message passing (dispatching). This involved detecting the act of message passing, and inserting additional computation (either in addition to or instead of the invocation of the base-level method requested) to implement the desired extensions. A key to implementing these capabilities was the fact that the basic concepts of each object model were themselves represented by objects, with associated operations, that enabled the extensions to be added in a modular way.

In addition, there was a considerable amount of consistency in the general structure of the various reflective object models examined, even though there were some differences in the specific metaobjects identified, due to detail differences in the characteristics of the individual models examined. In general, all the models required the use of explicit method, state, and interface objects, in addition to metaobjects representing the base-level objects themselves. These are similar to the metaobjects already identified as RISC components, and confirms the overall approach being pursued in the RISC model. Some additional components, in some cases already anticipated in [MH93], will also be required, as noted below.

Some aspects of the structure of these models are particularly interesting. For example, in the actor models, the actors (or rather, the mail queues) have a distinct identity from the associated behavior objects, so that the behaviors can be replaced for the same actor. This is consistent with having separate method and state objects associated with a given base-level object, as in the Cecil [Cha92,93] language, and as proposed for the RISC object model. This is also consistent with some of the metamodels defined in Sections 3.2 and 3.3, where the metaobject effectively contains the base object's state and behavior, with the base object largely acting as an assertion of base-level identity.

The way in which the various models handle asynchronous message passing, and model the message passing of other models, is also of interest. The equivalence of synchronous and asynchronous message passing (given certain additional constructs), and the ability to model different forms of message passing in terms of each other illustrated in some of the models, is important in the analysis of object model interoperability, and also in the ultimate selection of a base message passing semantics for the RISC model (although, as noted at the end of Section 3.1, this decision may be somewhat arbitrary). In particular, the representation of the various forms of ABCL/1 messages in terms of asynchronous messages plus futures in Section 3.4.4 illustrates an approach that could be used in the RISC model to describe these sorts of variations reflectively. Combining these observations suggests that this could also be done using synchronous messages.

Other models are particularly interesting in that they describe the need for an explicit process or thread concept to model concurrent processing in certain object models. This approach would be useful as an aspect of the RISC model, because it explicitly models processes (or threads) and the association of those processes with objects. This permits modeling of passive objects, active objects with only one process, with multiple processes, and other sorts of mixtures, in an explicit way. Some of this process interaction is presumably reflective. In particular, a `Process` object could be considered a metaobject like a method or state object, as illustrated by the ACT++ thread objects described in Section 3.4.1, and the models described in Section 3.5.3. The requirements identified in the Eiffel // model in Section 3.5.3, including the need for explicit threads, methods, and messages, are also consistent with the general theme of providing reflective capabilities to allow for the flexible modeling of the different control mechanisms. Thread or process objects, together with the additional concepts, such a message queues, required for other concurrent object models, would seem to provide a general basis for modeling many concurrent object models.

The descriptions in Section 3 also illustrated a wide range of applications for adding extended capabilities using reflection. Some of these involved extensions on a *per-object* basis, such as changing an object so that it prints when a method is invoked, or automatically deletes itself when it is no longer referenced by another object, as well as extensions for remote or asynchronous messaging. Others involved extensions involving multiple objects, or on a more global basis, such as message broadcasting, object synchronization, transaction management, or access to other shared system services, such as persistent storage (in general, these services are represented as objects as well, as illustrated in Choices).

A number of these applications illustrated capabilities directly related to object model interoperability and applications within heterogeneous distributed object systems. For example, the ability to integrate general system services with individual application objects using the multiple object mechanisms just mentioned is a key element in building distributed, object-oriented application architectures, such as the OMG's Object Management Architecture [OMG92a,b], and the Telops Application Framework. Similarly, the examples of supporting C++ encapsulation in CLOS (cited in Section 3.6.2), and adding garbage collection to C++ (Section 3.3.2), illustrate an ability to simulate the capabilities of one object model in another using reflective capabilities. These examples suggest that, if languages had built-in reflective capabilities (or were based on a core containing them), these capabilities would provide a general means for extending them to ease their integration with other languages having different facilities. The use of metaobject protocols is one of the approaches to object model convergence being considered by the X3H7 object standards committee, and may be the subject of liaison activity between X3H7 and OMG [Tho93].

The use of reflection to add facilities for concurrency and transactions illustrated another important theme. The basis of any of these mechanisms is that an object wait to perform some operation until it is safe to do so, according to some condition or criterion. In some cases, this condition is based on the condition of a number of other objects. Reflection allows the invocation of an operation to be extended so that the appropriate conditions can be checked before proceeding. A number of examples in Section 3 illustrated variations of this approach to concurrency control, i.e., altering the reflective computation to interrupt the flow of method invocation to perform concurrency-related communication, and delaying the operation execution until a condition is satisfied that indicates it is safe to proceed with the operation. In some cases, the checking to be performed is simple consultation of locks or semaphores. In more general or complex cases, the operation invocation can be referred to a transaction processor or other transaction management executive. Such a component could be considered as a metaobject providing a specific service (e.g., a member of an Open C++ Object Community). [HG93] describes ongoing work at GTE Laboratories showing how this approach of interrupting message dispatching to consult generalized transaction management components can used to integrate extended transaction facilities in a distributed object environment. It is important to note that the details of the various concurrency control schemes using metaobjects that were described are not as important as the basic pattern, which is that this can be done at the metalevel, by reflection on the global state of the computation. This makes it possible for different concurrency control policies to be defined in a modular fashion, largely independently of the definitions of the individual application objects.

In general, the RISC concepts identified in Section 2 were largely consistent with those used by the reflective models described in Section 3 to model themselves, although the need for certain extensions was also identified. The models surveyed should provide a good foundation for defining the precise interfaces of the RISC object model primitives (i.e., metaobjects). These reflective models also define a structure that could be easily implemented using the "raw dispatching" capability defined in GTE Laboratories' DOM 3.0 prototype. This was recognized in [NH93b], which noted that DOM 3.0's dispatching resembles a "goto", as compared to the "structured programming" provided by a fully-defined metaobject protocol.

Finally, as noted in Section 3, there is a difference between the RISC object model concept and the instances of object model reflection reviewed in Section 3. In these models, the idea is to take a *single* object model and implementation, and extend it by creating runtime metaobjects that represent pieces of the implementation (and so can be altered by the user/programmer). In the case of the RISC object model, there is no single object model that is to be implemented using metalevel facilities. Instead, what is to be provided is a sufficiently-general set of metaobjects so that a very wide *range* of object models can be defined in terms of them. This is why the selection of the individual primitives is less immediately obvious than it might be if only a single object model were involved, and why a range of object models must be examined. Even though this is so, the overall concept of the RISC model is consistent with the concepts underlying the individual reflective models that were reviewed in Section 3.

4.2. Future Work

Additional research on the RISC object model is being pursued in a number of ways. As noted already, the X3H7 (Object Information Management) standards committee is currently pursuing the approach of describing different object models in terms of a common set of concepts, and of comparing and defining mappings between these object models. In addition, X3H7 and the X3H4 (Information Resources Dictionary System) standards committee are jointly exploring the use of logic in describing object model concepts, and X3H7 is considering other formal specification techniques as well, such as Z [Wor92].

X3H7 is also maintaining close liaison with the X3T3 (Open Distributed Processing) and X3H2 (SQL) standards committees, and the Object Management Group, with the goal of coordinating the development of object model concepts with those groups [Tho93]. Our participation in these activities will contribute to a better understanding of the primitive concepts required to describe the object models used in key standards, and thus to the development of the RISC model, as well as potentially contributing to the interoperability of those standards.

Further direct work on the development of the RISC object model will also be pursued in the coming year, including:

- defining strawman details of RISC metaobjects and an overall metaobject protocol

- validating these definitions by describing sample object models in terms of these definitions (this will be done in conjunction with X3H7 to some extent), and also by verifying the ability to correctly represent aspects of concurrency.

- examining additional classes of object models, in particular concurrent logic-based object models, to ensure they can be represented within the RISC model (papers such as [KTMB86], which identify key relationships between these languages and actor languages, are encouraging in this regard).

- examining additional examples of current work on reflection and metalevel concepts in object models.

In addition, the Distributed Object Management project at GTE Laboratories is independently pursuing a number of related activities, in addition to further formal research on RISC object model concepts. These include:

- experiments with RISC model concepts implemented using the DOM 3.0 prototype, which already supports some RISC-like features.

- experiments with supporting OMG Core object model concepts using the DOM 3.0 prototype and related RISC concepts.

- investigation of support for metaobject protocol concepts in the OMG Core object model, and the use of the metaobject protocol in defining OMG Components.

- further integration of RISC object model concepts with our work on object-oriented transaction facilities [GH92, HG93].

Subsequent reports will describe the results of this research.

Acknowledgements

I would like to thank the members of GTE Laboratories' Distributed Object Computing Department, Michael Brodie, Gail Mitchell, Sandra Heiler, Dimitrios Georgakopoulos, Mark Hornick, Emon Mortazavi, Ben Hurwitz, and especially, Farshad Nayeri, and visiting researchers Peter Lockemann, Ole Anfindsen, and Cristobal Pedregal Martin, for discussions which contributed to some of the ideas in this report.

5. REFERENCES

[AFKP+93] G. Agha, S. Frølund, W. Kim, R. Panwar, A. Patterson, and D. Sturman, "Abstraction and Modularity Mechanisms for Concurrent Computing", *IEEE Parallel and Distributed Technology* 1(2), May 1993, pp. 3-14.

[Agr91] R. Agrawal (ed.), *Data Engineering*, *14*(2), Special Issue on Foundations of Object-Oriented Database Systems, June 1991.

[AH87] G. Agha and C. Hewitt, "Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming", in [SW87].

[Als93] S. Alsop, "I'm Just a Country Editor, But Why Object Model Rivalry?", *InfoWorld*, December 13, 1993, p. 112.

[Ash93] C. Ashford, ed., "Comparison of the OMG and ISO/CCITT Object Models", Report of the Joint Forum/OMG Taskforce on Object Modelling, Draft Final, February 1993.

[AT88] M. Aksit and A.Tripathi, "Data Abstraction Mechanisms in Sina/st", in [Mey88].

[AWY89] Gul Agha, Peter Wegner, and Akinori Yonezawa, eds., "Proc. ACM SIGPLAN Workshop on Object-Based Concurrent Programming", *SIGPLAN NOTICES*, 24,4, April 1989.

[Bee90] C. Beeri, "A Formal Approach to Object Oriented Databases", *Data Knowl. Engrg. 5* (1990) 353-382.

[BG81] P. A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems", *ACM Computing Surveys*, June 1981.

[BNOW93] A. Birrell, G. Nelson, S. Owicki, and E. Wobber, "Network Objects", *Proc. 14th ACM Symp. on Operating Systems Principles,* 1993.

[Bri89] J.-P. Briot, "From Objects to Actors: Study of a Limited Symbiosis in Smalltalk-80", *Proc. ACM-SIGPLAN Workshop on Object-Based Concurrent Programming*, Sept. 1988, in *SIGPLAN Notices 24*(4), April 1989.

[Bro94] K. Brockschmidt, *Inside OLE 2,* Microsoft Press, Redmond, 1994.

[Car91] D. Caromel, "A Solution to the Explicit/Implicit Control Dilemma", *OOPS Messenger*, 2(1), April 1991.

[CG89] N. Carriero and D. Gelernter, "Linda in Context", *Comm. ACM 32*(4), April 1989.

[Cha92] C. Chambers, "Object-Oriented Multi-Methods in Cecil", *Proc. ECOOP '92*.

[Cha93] C. Chambers, "The Cecil Language: Specification and Rationale", Technical Report 93-03-05, University of Washington, March 1993.

[CM93] S. Chiba and T. Masuda, "Designing an Extensible Distributed Language with a Meta-Level Architecture", submitted to ECOOP '93.

[CS92] N. Coskun and R. Sessions, "Class Objects in SOM", *IBM Personal Systems Developer* (Summer 1992): 67-77.

[DMBC+87] U. Dayal, F. Manola, A. Buchmann, U. Chakravarthy, D. Goldhirsch, S. Heiler, J. Orenstein, and A. Rosenthal, "Simplifying Complex Objects: The PROBE Approach to Modelling and Querying Them", in S. Zdonik, and D. Maier (eds.), *Readings in Object-Oriented Database Systems*, Morgan Kaufmann, 1989.

[DMC92] C. Dony, J. Malenfant, and P. Cointe, "Prototype-Based Languages: From a New Taxonomy to Constructive Proposals and Their Validation", in [Pae92].

[DT88] S. Danforth and C. Tomlinson, "Type Theories and Object-Oriented Programming", *ACM Computing Surveys 20*(1), March 1988.

[EG89] Clarence A. Ellis and Simon J. Gibbs, "Active Objects: Realities and Possibilities", in [KL89b].

[Fer89] J. Ferber, "Computational Reflection in Class based Object Oriented Languages", in [Mey89].

[FKMT91] E. Fong, W. Kent, K. Moore, and C. Thompson (eds), "X3 / SPARC / DBSSG / OODBTG Final Report", 17 September 1991.

[GH92] D. Georgakopoulos and M. Hornick, "An Environment for Specification and Management of Extended Transactions and Workflows in DOMS", TR-0218-09-92-165, GTE Laboratories Incorporated, October 30, 1992.

[GMS88] S.J. Greenspan, C.L. McGowan, M. Chandra Shekaran, "Toward an Object-Oriented Framework for Defining Services in Future Intelligent Networks," *Proc. IEEE Intl. Conf. on Communications*, June 1988.

[GRL94] D. Georgakopoulos, M. Rusinkiewicz, and W. Litwin, "Chronological Scheduling of Transactions with Temporal Dependencies", to appear, *VLDB Journal*, Jan. 1994.

[GW93] J. Gilbert, III and R. C. Wilhelm, Jr. "A Concurrent Object Model for an Industrial Process-Control Application", *Journal of Object-Oriented Programming*, Nov.-Dec. 1993, pp. 35-44.

[Hew77] Carl Hewitt, "Viewing Control Structures as Patterns of Passing Messages", *Journal of Artificial Intelligence*, 8(3), June 1977.

[HG93] M. Hornick and D. Georgakopoulos, "Towards Addressing the Limitations of Conventional Transactional Systems in Supporting Advanced Applications", TR-0241-12-93-165, GTE Laboratories Incorporated, December 20, 1993.

[HH87] R.J. Hass and R.W. Humes, "Intelligent Network/2: A Network Architecture Concept for the 1990s," *Proc. IEEE 6th Annual Phoenix Conf. on Computers and Communications*, Scottsdale, Arizona, February 1987.

[HT93] B. Hodges and C. Thompson (eds.), "Workshop on Application Integration Architectures: Workshop Report", National Institutes of Science and Technology, April 1993.

[HZ90] S. Heiler and S. Zdonik, "Object Views: Extending the Vision", *Proc. 6th Intl. Conf. on Data Engrg.*, Los Angeles, Feb. 1990.

[Ibr91] M. H. Ibrahim, "OOPSLA '91 Workshop—Reflection and Metalevel Architectures in Object-Oriented Programming", in J. L. Archibald and M. C. Wilkes (eds.), *OOPSLA '92 Addendum to the Proceedings*, published as *OOPS Messenger*, 4(2), April 1993, 315-320.

[ISO93] ISO/IEC JTC1/SC21. N7054, Recommendation X.902, Basic Reference Model of Open Distributed Processing - Part 2, Descrptive Model, 1993.

[Iss93] V. Issarny, " An Exception-Handling Mechanism for Parallel Object-Oriented Programming: Toward Reusable, Robust Distributed Software", *Journal of Object-Oriented Programming*, Oct. 1993, pp. 29-40.

[Jag92] S. Jagannathan, "Reflective Building Blocks for Modular Systems", in [YS92].

[Jef85] D. Jefferson, "Virtual Time", *ACM Trans. Prog. Systems and Languages*, 7(3), 1985.

[JM86] D. Jeffrson and A. Motro, "The Time Warp Mechanism for Database Concurrency Control", *Proc. IEEE Intl. Conf. on Data Engineering*, Feb., 1986.

[JS92] H. V. Jagadish and O. Shmueli, "Synchronizing Trigger Events in a Distributed Object-Oriented Database", in M. T. Ozsu, U. Dayal, and P. Valduriez (eds.), *Distributed Object Management*, Morgan Kaufmann, 1993.

[J+87] D. Jefferson, et. al., "Distributed Simulation and Time Warp Operating System", *Operating Systems Review*, 21(5), 1987.

[KC86] S. Khoshafian and G. Copeland, "Object Identity", in [Mey86].

[KL89a] M. Kifer and G. Lausen, "F-Logic: A Higher-Order Language for Reasoning about Object, Inheritance, and Scheme", *Proc. 1989 ACM SIGMOD Intl. Conf. on Management of Data*, 1989.

[KL89b] W. Kim and F. H. Lochovsky (eds.), *Object-Oriented Concepts, Databases, and Applications*, ACM Press / Addison-Wesley, 1989.

[KL91] D. Kafura and G. Lavender, "Recent Progress in Combining Actor-Based Concurrency with Object-Oriented Programming", *OOPS Messenger*, 2(1), April 1991

[KML93] D. Kafura, M. Mukherji, and G. Lavender, "ACT++: A Class Library for Concurrent Programming in C++ Using Actors", *Journal of Object-Oriented Programming*, Oct. 1993, pp. 47-62.

[KRB91] G. Kiczales, J. des Rivieres, and D. G. Bobrow, *The Art of the Metaobject Protocol*, MIT Press, Cambridge, MA, 1991.

[KTMB86] K. Kahn, E. D. Tribble, M. S. Miller, and D. G. Bobrow, "Objects in Concurrent Logic Programming Languages", in [Mey86].

[Mae87] Pattie Maes, "Concepts and Experiments in Computational Reflection", in [Mey87].

[Man88] F. Manola, "Distributed Object Management Technology in WINS", TM-0081-12-88-165, GTE Laboratories Incorporated, December 30, 1988.

[Man89] F. Manola, "Object Model Capabilities for Distributed Object Management," TM-0149-06-89-165, GTE Laboratories Incorporated, June 30, 1989.

[MB90] F. Manola and A. P. Buchmann, "A Functional/Relational Object-Oriented Model for Distributed Object Management: Preliminary Description," TM-0331-11-90-165, GTE Laboratories Incorporated, December 31, 1990.

[MD89] F. Manola and U. Dayal, "PDM: An Object-Oriented Data Model", in S. Zdonik and D. Maier (eds.), *Readings in Object-Oriented Database Systems*, Morgan Kaufmann, 1989.

[MDC92] J. Malenfant, C. Dony, and P. Cointe, "Behavioral Reflection in a Prototype-Based Language", in [YS92].

[Mey86] N. Meyrowitz, ed., *OOPSLA '86 Conference Proceedings*, ACM, Sept., 1986, published as *SIGPLAN Notices*, 21(11), Nov., 1986.

[Mey87] N. Meyrowitz, ed., *OOPSLA '87 Conference Proceedings*, ACM, Oct., 1987, published as *SIGPLAN Notices*, 22(12), Dec., 1987.

[Mey88] N. Meyrowitz, ed., *OOPSLA '88 Conference Proceedings*, ACM, Sept., 1988, published as *SIGPLAN Notices*, 23(11), Nov., 1988.

[Mey89] N. Meyrowitz, ed., *OOPSLA '89 Conference Proceedings*, ACM, Oct., 1989, published as *SIGPLAN Notices*, 24(10), Oct., 1989.

[MH92] F. Manola and S. Heiler, "An Approach to Interoperable Object Models", in M. T. Ozsu, U. Dayal, and P. Valduriez (eds.), *Distributed Object Management*, Morgan Kaufmann, 1993.

[MH93] F. Manola and S. Heiler, "Toward a Reduced Object Model for Object System Interoperation", TR-0229-03-93-165, GTE Laboratories Incorporated, May 1993.

[MHGH+92] F. Manola, S. Heiler, D. Georgakopoulos, M. Hornick, and M. Brodie,"Distributed Object Management", *Intl. Journal of Intelligent and Cooperative Information Systems 1*(1), 1992.

[MKIC92] P. Madany, P. Kougiouris, N. Islam, and R. H. Campbell, "Practical Examples of Reification and Reflection in C++", in [YS92].

[MMWY92] H. Masuhara, S. Matsuoka, T. Watanabe, and A. Yonezawa, "Object-Oriented Concurrent Reflective Languages can be Implemented Efficiently", in [Pae92].

[MR87] Naftaly H. Minsky and David Rozenshtein, "A Law-Based Approach to Object-Oriented Programming", in [Mey87].

[Nei87] O. M. Nierstrasz, "Active Objects in Hybrid", *ACM SIGPLAN Notices*, 22(12), December 1987.

[NH93a] F. Nayeri and B. Hurwitz, "Experiments with Dispatching in a Distributed Object System", TR-0236-09-93-165, GTE Laboratories Incorporated, July 1993.

[NH93b] F. Nayeri and B. Hurwitz, "Generalizing Dispatching in a Distributed Object System", submitted for publication, December 1993.

[NN92] F. Nielson and H. R. Nielson, *Two-Level Functional Languages*, Cambridge Tracts in Theoretical Computer Science 34, Cambridge University Press, 1992.

[OH92] R. Orfali and D. Harkey, *Client/Server Programming with OS/2 2.0*, 2nd. ed., Van Nostrand Reinhold, New York, 1992.

[OMG91a] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, OMG Document Number 91.12.1, Rev. 1.1, 1991.

[OMG92a] Object Management Group, *Object Management Architecture Guide*, Revision 2.0, 2nd. Ed., OMG TC Document 92.11.1, Sept. 1, 1992.

[OMG92b] Object Management Group, *Object Services Architecture*, Revision 6.0, OMG Document 92.8.4, 28 August 1992.

[OMG92c] Object Management Group, *The OMG Object Model Components Manual*, OMG Object Model Task Force draft of May 27, 1992.

[OMG93] *Joint Object Services Submission*, OMG TC Document 93.7.1, July, 1993.

[QP91] P. Quinn and G. Preoteasa, "Reconciling Object Models for Systems and Network Management", Uniforum Association, 1991.

[Pae92] A. Paepcke, ed., *OOPSLA '92 Conference Proceedings*, ACM, Oct., 1992, published as *SIGPLAN Notices*, 27(10), Oct., 1992.

[RBPE+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.

[RC92] K. Ramamritham and P. Chrysanthis, "In Search of Acceptability Criteria: Database Consistency Requirements and Transaction Correctness

Properties", in M. T. Ozsu, U. Dayal, and P. Valduriez (eds.), *Distributed Object Management*, Morgan Kaufmann, 1993.

[RS91a] J. Richardson and P. Schwartz, "Aspects: Extending Objects to Support Multiple, Independent Roles", *Proc. 1991 ACM SIGMOD Intl. Conf. on Management of Data*, ACM, New York, 1991.

[RTLB+91] R. K. Raj, E. Tempero, H. M. Levy, A. P. Black, N. C. Hutchinson, and E. Jul, "Emerald: A General-Purpose Programming Language", *Software--Practice and Experience* 21(1), January 1991.

[SC92] R. Sessions and N. Coskun, "Object-Oriented Programming in OS/2 2.0", *IBM Personal Systems Developer* (Winter 1992): 107-120.

[SCBK+86] C. Schaffert, T. Cooper, B. Bullis, M. Kilian, and C. Wilpolt, "An Introduction to Trellis/Owl", in [Mey86].

[Sow93] J. F. Sowa, "Logical Foundations for Representing Object-Oriented Systems", *Journal of Experimental and Theoretical Artificial Intelligence*, v. 5, 1993.

[SW87] B. Shriver and P. Wegner, eds., *Research Directions in Object-Oriented Programming*, MIT Press, 1987 (early versions of some of these papers appeared in *SIGPLAN Notices*, 21(10), October 1986).

[SY86] R. E. Strom and S. Yemini, "Typestate: A Programming Language Concept for Enhancing Software Reliability", *IEEE Trans. Software Engineering*, 12(1), 1986.

[Tan91] T. Tanaka, "Actor Reflection Without Meta-Objects", *OOPS Messenger*, 2(1), April 1991.

[Tan93] A. S. Tanenbaum, "Distributed Operating Systems Anno 1992. What Have We Learned So Far?", *Distributed Systems Engineering 1* (1993), pp.3-10.

[Tho93] C. Thompson, unpublished draft OMG/X3H7 Liaison Statement, December 18, 1993.

[TS89a] C. Tomlinson and M. Scheevel, "Concurrent Object-Oriented Programming Languages", in [KL89b].

[TS89b] C. Tomlinson and V. Singh, "Inheritance and Synchronization with Enabled-Sets", in [Mey89].

[US87] D. Ungar and R. B. Smith, "Self: The Power of Simplicity", in [Mey87].

[vHar89] F. van Harmelen, "A Classification of Meta-level Architectures", in H. Abramson and M. H. Rogers (eds.), *Meta-Programming in Logic Programming*, MIT Press, Cambridge, 1989.

[WA91] G. I. Williamson and M. Azmoodeh, "The Application of Information Modelling in the Telecommunications Management Network (TMN)", *Proc. Telecommunications Information Networking Architecture Workshop* (TINA91), March 1991.

[Way94] P. Wayner, "Objects on the March", *Byte*, January, 1994, 139-150.

[Weg89] P. Wegner, "Granularity of Modules in Object-Based Concurrent Systems", in G. Agha, P. Wegner, and A. Yonezawa, eds., "Proc. ACM SIGPLAN Workshop on Object-Based Concurrent Programming", *SIGPLAN NOTICES*, 24,4, April 1989.

[Wor92] J. B. Wordsworth, *Software Development with Z*, Addison-Wesley, 1992.

[WY88] Takuo Watanabe and Akinori Yonezawa, "Reflection in an Object-Oriented Concurrent Language", in [Mey88].

[WY89] Takuo Watanabe and Akinori Yonezawa, "Reflective Computation in Object-Oriented Concurrent Systems and Its Applications", *Proc. Fifth Intl. Workshop on Software Specification and Design*, *ACM SIGSOFT Engineering Notes*, Vol. 14, No. 3, May 1989.

[X3H793a] Accredited Standards Committee X3, Technical Committee X3H7, "X3H7 Object Model Features Matrix", document X3H7-93-007v4, November 22, 1993.

[Yok92] Y. Yokote, "The Apertos Reflective Operating System: The Concept and Its Implementation", in [Pae92].

[YS92] A. Yonezawa and B. C. Smith (eds.), *Proc. Intl. Workshop on New Models for Software Architecture (IMSA) '92: Reflection and Meta-Level Architecture*, Tokyo, November 1992.

[YT87a] A. Yonezawa and M. Tokoro, eds., *Object-Oriented Concurrent Programming*, MIT Press, 1987.

[YT87b] A. Yonezawa and M. Tokoro, "Concurrent Programming in ConcurrentSmalltalk", in [YT87a].

[YW89] A. Yonezawa and T. Watanabe, "An Introduction to Object-Based Reflective Concurrent Computation", *Proc. ACM-SIGPLAN Workshop on Object-Based Concurrent Programming*, Sept. 1988, in *SIGPLAN Notices 24*(4), April 1989.