A Comparison of Five Object Models

Alan Snyder
Software and Systems Laboratory
HPL-91-80
June, 1991

object-oriented programming; object model; abstract object model; C++; OSI; network management; HP OpenView; distributed systems; Distributed Application Architecture; tool integration architecture; software development environments; Iris; object-oriented database

This document describes the object models of five key Hewlett-Packard object technologies: the C++ programming language, the OSI systems management standard and its extension to the HP OpenView architecture, the Distributed Application Architecture (DAA), the Tickle integration architecture for software development environments, and the Iris object-oriented database. The five object models are described by annotating a description of an abstract object model developed for this purpose. This document is an expansion of a companion document (HPL-91-79) that contains only the description of the abstract object model.

1 Introduction

This document describes a proposed HP object model. The model is based on the abstract object model prepared for the Object Management Group [12], which is itself an outgrowth of earlier work undertaken by an internal Hewlett-Packard task force [13].

The description of the model is annotated with descriptions of the object models of five key Hewlett-Packard object technologies:

- The C++ programming language [4, 14].
- The OSI systems management standard [15] and its extension to the HP OpenView architecture [6].
- The Distributed Application Architecture (DAA), as documented in the HP-Sun submission to the Object Management Group [8, 9, 10]. The description characterizes the properties of the DAA that are common to all possible object managers, and, where relevant, explicitly identifies characteristics specific to the Distributed Object Manager.
- The Tickle integration architecture for software development environments, as documented in [2, 3, 7].
- The Iris object-oriented database [5].

The HP object model provides an organized presentation of object concepts and terminology. It defines a partial model of computation that embodies the key characteristics of objects as realized in HP object technologies. Its purpose is to serve as a common framework for presenting and comparing HP object technologies. It may also serve as a guide to possible evolution of object technologies towards the hypothetical future "grand unified object system".

The HP object model is an *abstract* object model in that it is not directly realized by any technology. Any particular object technology will have its own object model, which we call a *concrete* object model. A concrete object model is likely to differ from the abstract object model in several ways. It may *elaborate* the abstract object model by making it more specific, for example, by defining the form of request parameters or the language used to specify types. It may *populate* the model by introducing specific instances of entities defined by the model, for example, specific objects, specific operations, or specific types. It may also *restrict* the model by eliminating entities or placing additional restrictions on their use.

An object system is a collection of objects that isolates the requestors of services (clients) from the providers of services by a well-defined encapsulating interface. In particular, clients are isolated from the implementations of services as data representations and executable code.

The object model first describes concepts that are meaningful to clients, including such concepts as object creation and identity, requests and operations, types and signatures. It then describes concepts related to object implementation, including such concepts as methods, data structures, implementation templates, and implementation inheritance.

The object model is most specific and prescriptive in defining concepts meaningful to clients. The discussion of object implementation is more suggestive, with the intent of allowing maximal freedom for different object technologies to provide different ways of implementing objects.

There are other characteristics of object systems that are outside the scope of the object model. Some of these concepts are aspects of application architectures, some are associated with specific domains to which object technology is applied. Such concepts are more properly dealt with in an architectural reference model. Examples of excluded concepts are compound objects, attributes and links, copying of objects, change management, and transactions. Also outside the scope of the object model is the model of control and execution.

In the object model definition, *italics* are used to introduce or define new terms. Parenthesized sentences and indented paragraphs are commentary and are not part of the object model.

1.1 Classical vs. Generalized Object Models

As defined below, the HP object model is a generalized object model. As such, it differs profoundly from the object models most familiar in current systems. It is intentionally a significant generalization of the more familiar object models. Before describing the HP object model, it is helpful to compare and contrast the two different kinds of object model. The terms used in this description are defined later, but the intuition should be clear.

A *classical object model*, used in most existing object technologies, is one where a client sends a message *to* an object. Conceptually, the object interprets the message to decide what service to perform. In the classical model, a message identifies an object and zero or more actual parameters. In most classical object models, a distinguished first parameter is required, which identifies the operation to be performed; the interpretation of the message by the object involves selecting a method based on the specified operation. Operationally, of course, method selection could be performed either by the object or by the system.

A *generalized object model*, used for example in the Common Lisp Object System [11] and the Iris database [5], is one where a client issues a request that identifies an operation and zero or more parameters, any of which may identify an object. In the generalized object model, method selection may be based on any of the objects identified in the request, as well as the operation. Because method selection may be based on multiple objects, it is best viewed as being done by the system, rather than by an object. Operationally, of course, method selection could be performed in multiple stages, with the final stage being carried out by an object.

The classical object model is a special case of the generalized object model. The classical sending of a message to an object is equivalent to a generalized request where method selection is based strictly on the operation and the object identified by a distinguished parameter.

For example, sending the message *print a-printer* to the object *a-spreadsheet* in the classical model is equivalent to issuing the request *print a-spreadsheet a-printer* in the generalized model, under the assumption that method selection is based on *a-spreadsheet* and not *a-printer*.

The generalized object model allows an object technology to provide additional functionality. For example, an object technology could support defining a method that is invoked only when the *print* operation is requested on a spreadsheet and a particular kind of printer. This specialized method could take advantage of the unique capabilities of the particular kind of printer.

Distributed systems with classical object models generally assign locations to objects. With a generalized model, it is also possible to assign locations to operations, which can affect even zero-parameter invocations. Thus, remote procedure calls are also a special case of the generalized object model.

2 Acknowledgments

The development of the HP object model was influenced by the participants in the HP object task force, the Object Management Group technical committee, and the ad hoc ANSI unified object model group. Bill Kent in particular made many valuable contributions. This version of the object model includes several innovations made in the context of the ANSI group. Lorenzo Aguilar, Peter Canning, Ian Fuller, Colin Gerety, Paul Golick, Dmitry Lenkov, Peter Lyngbaek, Joe Sventek, Ian Thomas, and Richard Webber provided invaluable assistance in understanding and developing models of specific object systems.

3 Overview

The entities of a computational system include objects, values (including object names and handles), operations, signatures, and types (including interface types).

4 Object Semantics

An object system provides services to clients. A client of a service is any entity capable of requesting the service. (A *client* may be an object, a person, or a computational process.)

This section defines the concepts associated with object semantics, i.e., the concepts relevant to clients. (Figure 1 shows the relationships among several key concepts defined in this section.)

4.1 Objects

An object system includes entities known as *objects*. An object is an identifiable entity that plays a visible role in providing a service that can be requested by a client.

The most common roles played by objects are as parameters specified by clients in requests for service and as results delivered to clients as part of performing a service. Entities playing roles not visible to clients, such as methods that are executed to perform a service, may be objects when viewed at a lower level of abstraction. A computational object system whose computational model is defined in terms of objects is called a reflective object system.

Objects have certain characteristics, which are explained below.

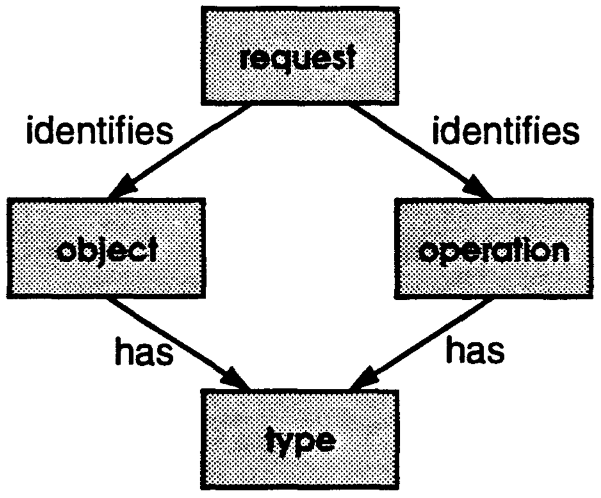

Figure 1. Primary object semantics concepts

C++

A C++ *object of a class type* (i.e., a class instance) that is not an unnamed base class subobject is an object. Note that a C++ *object* is simply a region of storage.

A C++ object of a non-class type is not modeled as an object, because non-instance objects cannot support generic operations. Unnamed base class subobjects are created as components of instances of derived classes. They are not modeled as objects, because they are not visible to clients as separate entities, except where explicit type coercions are used in the case of *repeated inheritance*, where a single derived class object has multiple unnamed subobjects of the same base class. We believe this case is too unusual to warrant a more complex model.

OSI

An OSI *managed object* is an object.

A OSI *class*, *attribute*, *operation*, *notification*, *parameter*, *behavior*, or *package* is not an object. Although these entities have global unique identifiers, which are legitimate parameter values, they have no behavior that is characterized by requests. Note that they do not have ASN.1 distinguished names, and thus cannot be named as the target of an operation (OSI is a classical object model). They are not described by class definitions. They have no associated state.

DAA

A DAA *object* is an object.

Tickle

A Tickle *object* is an object. A Tickle object has a unique associated object type, which implies a maximal potential interface (a set of operations with signatures) to the object, but does not imply any particular methods for those operations (different methods may apply in different contexts). Although Tickle does not define data representations for objects, an object type typically implies a particular data representation for objects of that type. Each Tickle object may have associated infrastructure objects (such as a file), which can be used to maintain state associated with the object. The association between a Tickle object and an infrastructure object is not maintained by Tickle.

Iris

An Iris *object* is an object. Iris objects include *surrogates* and *literals*. *Surrogates* include *functions*, *types*, *users*, *groups*, and various transient objects, as well as objects of user-defined *types*. Literals include atomic datatypes (such as *integers*) and *aggregates* (such as *sets* and *lists*). The *null* value is not an object; it is not a member of any *type*.

4.2 Requests

Clients request services by issuing *requests*. A request is an event, i.e., something that occurs at a particular time. The information associated with a request consists of an *operation* and zero or more (actual) parameters.

A request is to be distinguished from static program text that, when evaluated, causes a request to be issued. The object model does not define the syntax by which programs cause requests to be issued. For example, it does not require that the operation appear first in a request-issuing construct. The object model does not define the evaluation process by which forms in a program are resolved to particular operations or particular values.

The intended meaning of a request is that the operation indicates the service to be performed and the parameters specify both the objects that are to participate in providing the service and any other information needed to specify the result desired by the client.

The object model allows a request to name zero objects or multiple objects, in addition to the usual case of a single object. The role of the identified objects in providing the requested service may be arbitrarily large or small.

Note that the object model does not specify that the operation must be statically identified in client programs. For example, a programming language interface might provide a construct for issuing a request in which the operation name is a variable. Such a construct would be similar to the *funcall* construct in Lisp.

C++

A C++ *function invocation* is a request, except for an invocation of a virtual member function named using explicit qualification, which instead is a method invocation. (Note that only invocations of virtual member functions name generic operations.) Access to a public data member by a client is modeled as an implicit request of a service that returns a reference to the data member.

OSI

An OSI *systems management request* is a request. The parameters of an OSI systems management request include *standard parameters*, *positional parameters*, and *named parameters*. Standard parameters are defined for each base operation (*create*, *get*, *action*, etc.) and correspond to elements of a CMIP protocol data unit. Positional and named parameters are defined for specific actions. Positional parameters correspond to elements of the information syntax datatype defined for a specific action. Named parameters are defined by parameter templates and are identified in requests by unique identifiers. A parameter can be required, optional, or defaulted.

The results of an OSI systems management request have a similar form to request parameters.

Except for the *create* operation, all systems management requests include a standard parameter that identifies a set of target objects within a system. A *simple request* identifies a specific object by specifying an explicit base object, a scope of 0, and no filter. A *complex request* uses a non-zero scope to identify a set of objects, and may use an explicit filter to narrow the set based on the values of the objects' attributes. The requested operation is performed on each identified object. A complex request may also direct a system to perform the operation with atomic synchronization over the entire set of objects. An object may be requested to perform an operation on several attributes with atomic synchronization. The target system is an implicit parameter of all requests. HP OpenView extends the notion of request to include the identification of target objects on multiple systems.

We view a complex request as invoking a higher order operator that issues multiple simple requests, supported by an orthogonal transaction mechanism. See the discussion under operation identity for further details.

(We do not model OSI notifications and confirmations.)

DAA

A DAA *request* is a request. A DAA request includes a distinguished parameter that identifies the target object. A DAA request can be synchronous or asynchronous.

Tickle

A Tickle operation invocation is a request. A Tickle request includes a distinguished parameter that identifies the target object. All other parameters are optional (a default value is provided). A Tickle request can be synchronous or asynchronous. A client is a tool activation operating within a dynamic context. The dynamic context determines which method performs the request. Each tool activation has an associated type schema that determines which operations can be invoked by that client on any given type of object. (We do not model Tickle event notifications.)

Iris

An Iris *function invocation* is a request. All Iris requests are synchronous.

A *request form* is a description or pattern that can be evaluated or performed multiple times to cause the issuing of requests.

Note the distinction between a request (event) and a request form (description). For example, a procedure call in a program text is a request form. Each evaluation of the procedure call results in issuing a distinct request. Similarly, a single user gesture (such as clicking a mouse button "over" an icon) can be viewed as a request form: the gesture can be performed multiple times: each performance results in issuing a distinct request.

Definition: A *static request form* is a request form that issues requests that all identify the same operation. A *dynamic request form* is a request form that issues requests for different operations. (The term *static request form* suggests that the identification of the operation is normally performed prior to execution.)

C++

A C++ *function call* (expression) is a request form, except for a function call that names a virtual member function using explicit qualification, which instead is a method invocation form. (A C++ expression that references a public data member is implicitly a request form.)

OSI

In OSI, a *CMIP protocol data unit* of an appropriate type is a request form. The appropriate types correspond to the base operations, and include *GetInvoke*, *SetInvoke*, *SetConfirmedInvoke*, *ActionInvoke*, *ActionConfirmedInvoke*, etc. A CMIP protocol data unit is a value of a specific ASN.1 datatype. (The OSI model permits other protocols that define other request forms.) The HP OpenView API defines request forms that are C procedure calls. Other APIs could be defined, such as a C++ API.

DAA

DAA request forms are defined by particular language bindings. In the C language binding, a request form is a C *function invocation* naming a function created by the CDL compiler from a CDL *interface definition*.

Tickle

The primitive Tickle request form is an invocation form that names the *operation_invoke* or *operation_invoke_with_wait* primitive. Other request forms may be defined by specific language bindings.

Iris

An OSQL *function invocation* is a request form. All OSQL statements implicitly issue requests, as they are defined in terms of function invocation.

A *value* is anything that may be a legitimate (actual) parameter in a request. A value may identify an object, for the purpose of performing the request. A value that identifies an object is called an *object name*. (Figure 2 illustrates the relationships among several concepts associated with requests.)

There may be values that identify abstract entities other than objects. Such values are called *literals*. (The term literal does not imply that these values are necessarily compile-time constants.) For example, in a particular system, integers might be identifiable in requests, but they might not have all of the characteristics of objects, as defined by that system. In other words, the object model permits but does not require that all entities identifiable by request parameters be objects.

C++

A C++ *value* of a non-class type is a value. A *pointer to a* C++ *object of a class type* is an object name. C++ also provides *reference parameters* and *reference variables*, which allow the pointer to be implicit. A pointer to an unnamed base class subobject is modeled as an object name identifying the enclosing object.

Pointers and references are specified in a request form by expressions that include variable names. Although C++ variables and expressions may denote values (and therefore objects), C++ variables and expressions are not values, and thus are not object names.

C++ values of a class type are never passed directly as function arguments; instead, a *reference* is created, which becomes the argument to a constructor; the *reference* is modeled as the request parameter. Thus, the default call-by-value semantics of function arguments of class type is *implicit* in the model.

OSI

In OSI, a value of an *ASN.1 datatype* is a value.

An *ASN.1 distinguished name* is an object name. It is valid only between the time that an object is created and deleted. OSI requires an object to belong to a specific system. The object is either the system managed object or is directly or indirectly contained within the system managed object. As currently defined, a *distinguished name* is valid only with respect to the system containing the object. HP OpenView supports global *distinguished names*. A *distinguished name* is a restricted form of *ASN.1 attribute value assertion*.

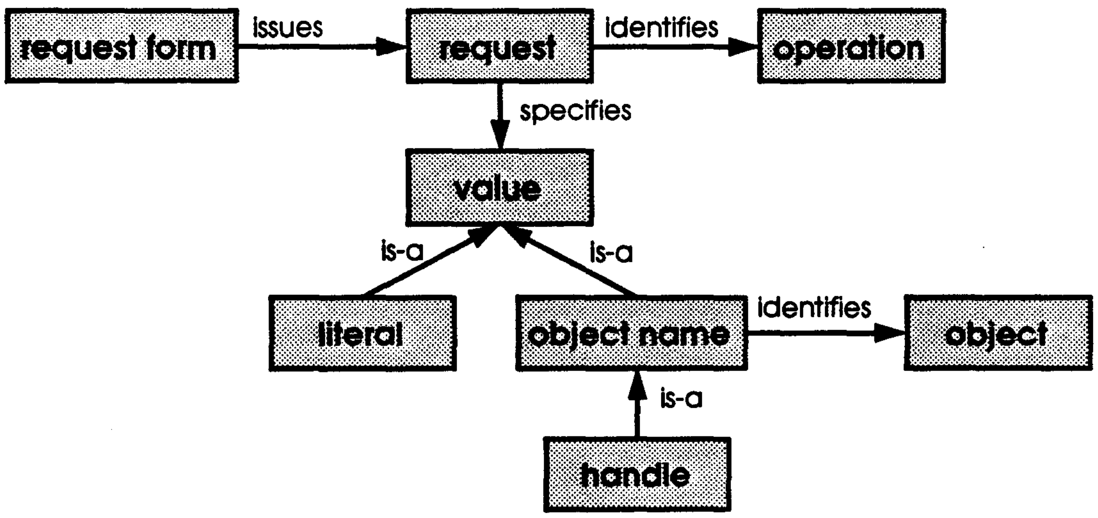

Figure 2. Request concepts

An *ASN.1 attribute value assertion* may be an object name (if it names exactly one object). The referent of an ASN.1 attribute value assertion may change as object attribute values change (and as objects are created and deleted).

DAA

In the DAA, a value is an instance of a CDL datatype. A DAA *object reference* is an object name.

Tickle

In Tickle, a value is an instance of a NIDL datatype. A Tickle *object designator* is an object name.

Iris

An Iris value is an Iris *object* or the distinguished value *null*. All Iris values except *null* are object names. The *null* value is not a member of any Iris *type*.

A *handle* is an object name that reliably identifies a particular object. Specifically, a handle will identify the same object each time the handle is used in a request (subject to certain pragmatic limits of space and time).

An object technology might provide object names that identify different objects at different times or in different "locations", for example, a value that denotes the nearest available printer of a particular kind. These values are object names, but not handles. While both are useful, the existence of reliable object names (handles) is an essential characteristic of an object system.

The distinction between object names and handles can be illustrated by analogy with the Unix[1] file system. Assume Unix files are objects, and consider three possible ways of identifying a Unix file: pathnames, file descriptors, and inode numbers. A pathname is an object name, but not a handle, since the file named by a pathname can change at any instant as the result of renaming files or directories. (A relative pathname has the additional ambiguity of being dependent for its interpretation on the file directory identified as "current" by a given process at a given time.) A file descriptor is a handle, because within a single process (or process tree, if file descriptors are shared), it will always identify the same file object (as long as it remains open). Although an inode number unambiguously identifies a particular file in the context of a single file system, an inode number is not an object name, because it is not a value (it cannot be used as a parameter in normal Unix system calls). Having both pathnames and file descriptors is useful. A major limitation of Unix is the lack of a persistent form of a file descriptor.

The object model allows an object to have multiple handles, in a single context or in different contexts. For example, a handle in the context of a Unix process might be a pointer into the address space of that process. In different processes, the same object would be identified by different pointers. Such handles are valid only within the address space of the process and only during the lifetime of the process. (The Unix file system example above illustrates multiple handles for an object in a single context: A Unix process can have multiple file descriptors that identify the same file. The file descriptors differ in having distinct associated positions into the file.)

C++

In C++, all object names are handles. A C++ object name identifies a unique object within the dynamic extent of the object. After an object is destroyed, any handles for that object are considered invalid. (However, almost all implementations of C++ allow

1. "Unix" is a registered trademark of AT&T.

a handle to be used after the object is destroyed, with unpredictable results that may include accessing some other object.)

OSI

In OSI, an ASN.1 distinguished name is a handle. A distinguished name identifies a unique object within the dynamic extent of the object. After an object is destroyed, any handles for that object are considered invalid.

DAA

The characteristics of a DAA *object reference* are determined by the *object manager* that manages the named object. A Distributed Object Manager *object reference* is a handle.

Tickle

A Tickle *object designator* is a handle.

Iris

All Iris object names are handles.

An object is defined to *participate in a request* if one or more of the actual parameters of the request identifies the object.

The word *participate* is not intended to imply any particular degree of involvement by the object in providing the requested service. At one extreme, the object might actually perform the requested service. At the other extreme, the service might simply manipulate the object name as a value.

A request causes a service to be performed on behalf of the client. One outcome of performing a service may be that results are returned to the client. The results associated with a request may include values as well as status information indicating that exceptional conditions were raised in attempting to perform the requested service.

The object model does not specify how parameters, results, and exceptional conditions are identified by a client. Possibilities for identifying parameters and results include by position (ordered parameters) and by name (keyword parameters). These details would be specified by an elaboration of this model.

C++

C++ requests identify parameters by position. Requests that invoke non-static member functions include a *distinguished parameter* that identifies the target object. The results of a C++ request consist of a single return value that becomes the value of the invocation expression. C++ does not provide a way of returning exceptional conditions, although a proposal to that effect has been accepted by the ANSI C++ committee.

OSI

The identification of parameters in an OSI request form depends upon the communication protocol. In CMIP, standard parameters and positional parameters are identified by position, as elements in an ASN.1 datatype that defines the transmission syntax of a protocol data unit. Named parameters are identified as keyword parameters using the ASN.1 identifier assigned to the parameter definition. Result parameters are specified similarly.

DAA

DAA request parameters are identified by position. A parameter may be an input parameter, an output parameter, or an input-output parameter. A DAA request returns a single *result value*, as well as the output parameters. In the C language binding, a single output parameter encodes the completion status. (Explicit exception indications are under consideration.)

Tickle

Tickle request parameters are identified by position. A parameter may be an input parameter, an output parameter, or an input-output parameter. A Tickle request returns a single *result value*, as well as the output parameters. There is no current provision for returning exceptional conditions.

Iris

Iris request parameters are identified by position as elements of a *tuple* object. An Iris request returns a single *result value*. An Iris operation may generate *warnings* and *errors*; warnings and errors are implicitly collected in a queue associated with the client.

4.3 Operations

An operation is an identifiable entity that denotes a service that can be requested.

The purpose of operations is to characterize sets of requests that have in common some notion of the intended semantics. Such sets are defined by identifying a common operation. It is expected that these sets of requests are sufficiently useful that most request forms will issue requests from a single such set.

The object model does not specify whether an operation is an object or whether a value can name an operation (i.e., whether operations can be parameters).

Several criteria can be applied when identifying operations in a specific object system:

- Operations should be chosen so that common request forms issue single operations.

  Static request forms should be the common case. A request form that issues multiple operations should be distinguished by the use of an *operation variable*, a variable that is evaluated at each execution of the form to determine the operation to be issued. An operation variable should have the expected properties of a variable: its value should be determined dynamically, independent of other aspects of the request form. For example, an overloaded function is not an operation variable, because its resolution is based on the types of the parameter expressions.

  Constructs in procedural programming languages that are analogous to operation variables are *pointers to functions* in C and *funcall* in Lisp: both constructs allow the invocation of a function that is dynamically identified by a value. Naming operations using variables provides an additional dimension of genericity.

- Operations should be chosen to allow useful signatures.

  An operation has a signature that describes the legitimate values of request parameters. Operation signatures should capture the inherent structure in the system. For example, in a statically typed system, the signatures of operations should capture static type information. A poor choice of operations (such as the extreme case of having only one operation) would force the operation signatures to be too permissive (such as accepting all possible values as parameters).

- Operations should be chosen to permit generic operations (page 15).

  On the other hand, there should be generic operations, and there should be generic operations whose binding to specific methods can be determined only during execution. For example, one should not define each method to be a distinct operation.

- Operations should be chosen so that request forms and objects can be characterized by which operations they issue or perform.

  Ignoring such detailed dynamic analysis as dataflow analysis, each request form can be described by the set of requests it can potentially issue. Operations should play an important role in characterizing this set of requests. For example, there are some operations that the requests issued by a request form can identify, and some that it cannot. Similarly, if no single request form can issue requests from two sets, then the two sets of requests should probably identify distinct operations.

  Operations should play an important role in characterizing the requests in which an object can meaningfully appear. An example of such a characterization is: This object is meaningful as the first parameter in requests that identify operations *a*, *b*, and c; this object is meaningful as the second parameter in requests that identify operations *d*, *e*, and *f*; this operation is not meaningful in any request that identifies operations *g*, *h*, and *i*.

- Operations should be chosen to minimize the use of contextual information to select methods.

  If possible, the information needed to select a method to perform a request should be present in the operation and the parameter values.

C++

Every C++ function definition, except a function definition that overrides a virtual member function, defines a distinct operation. (An overriding virtual member function defines a method for an existing operation.) In addition, each declaration of a public data member implicitly defines a distinct operation (whose behavior is to return a reference to the data member).

Operation identification in C++ is more complicated than in most object systems. In C++, an operation is identified by three features: (1) a lexical scope (such as a class definition), (2) the (textual) name of a function (or public data member) declared in that scope, and (3) the parameter types specified in the function signature (omitted for public data members).

The lexical scope is needed to characterize distinct functions with the same name and argument types defined in different scopes. For example, member functions in unrelated classes (neither is a public base class of the other) cannot be invoked from the same request form, and thus are considered different operations.

The parameter types are needed to handle *function overloading*. C++ allows functions with different parameter types to be declared in the same scope with the same name. An invocation of an overloaded function will select a particular function at compile-time based on static type information. Overloaded functions are considered different operations because they are invoked by different request forms. Lexical scoping and overloading allow a single function name to denote many distinct operations.

(Another reason why overloaded functions are modeled as distinct operations is to simplify the semantics of C++ values. Because overloaded function resolution is based on static types, modeling overloaded functions as generic operations would require that values carry both a static type, for overloaded function resolution, and a dynamic type, for virtual member functions.)

The inclusion of the class definition in the identification of member function operations accounts for the semantics of C++ *non-virtual member functions*. Distinct non-virtual member functions with the same function name and parameter types can be defined in a base class and a derived class. Only the base class member function can be invoked on an object of the base class, yet both member functions can be invoked on an object of the derived class; thus, the two member functions are modeled as *distinct* operations. Static member functions are like other non-virtual member functions, except that requests for such operations do not identify a target object.

A member function inherited by a derived class is viewed as being defined in the original base class. Thus, the base class and the derived class support the *same* operation. This choice is made because a single request form can result in the invocation of that member function on objects of either class.

A virtual member function is a single operation, defined by the original introduction of the function in a base class. All overriding definitions in derived classes refer to the same operation. Each definition of the virtual member function that includes a function body is a *method* that implements the operation. Binding selects a method based on the class of the object that is distinguished as the target of the request.

C++ allows the invocation of a specific function definition for a virtual member function, using a qualified name. Because we model these functions as methods rather than operations, such invocations are modeled as *direct method invocations*, rather than as requests. (An alternative is to model them as requests whose binding depends upon context.) Direct invocation of these functions outside a class definition is considered poor style. Modeling each function definition as a separate operation would thus fail to capture the intent of the language design.

Most C++ classes are defined in the external scope shared by all program units; however, nested class definitions are permitted. Thus, C++ class names are meaningful only with respect to a lexical scope. To identify a member function, one must identify a specific class definition, not just the class name.

An interesting special case arises using C++ multiple inheritance. If a derived class defines a virtual member function that overrides functions declared in two public base classes, then the two operations are "linked" in that class (and its derived classes) so that a single function definition provides a method for *both* operations.

A C++ operation is a value of type *pointer to function* or *pointer to class member function*. A C++ operation is not an object.

OSI

The simplest model of OSI systems management operations is to define a fixed set of operations based on CMIP protocol data units (*get*, *create*, *action*, *action-confirmed*, etc.). Note that CMIP distinguishes requests with and without confirmation as distinct protocol data units.

We prefer a more complex model. We model each action as a distinct operation identified by its registration identifier. We model each attribute operation (such as *get*) as a set of operations, one for each attribute to which the operation is applicable. These operations are identified by a pair consisting of the registration identifier of the attribute and the operation name. The implication of this latter decision is that we model an attribute operation applied to multiple attributes as a higher order operator, just as we model complex requests that apply to multiple objects as invoking a higher order operator. We model *get* applied to no attributes as a distinct operation *get-all-attributes*.

The advantage of the more complex model is that it allows much more specific operation signatures. The only risk is the assumption that the transaction behavior of multi-object and multi-attribute requests can be modeled as an orthogonal mechanism.

A systems management *action* is a value. A systems management operation is not an object.

DAA

A DAA operation is identified by an *operation identifier*. A DAA operation is not a value. Operation identifiers are assigned by the CDL compiler based on developer-provided information; the assignment process is implementation-specific. (In the C language binding, operations are named by C functions created by the CDL compiler. However, pointers to these functions cannot be used as request parameters; therefore, they are not values.)

Tickle

A Tickle operation is identified by an *operation name* (a string) and an *operation signature* that specifies the types of the parameters to an invocation of the operation (excluding the target object parameter). An operation is not a value.

This modeling is the best of several unattractive alternatives. There is no entity in Tickle with all of the desired properties of operations. The advantage of the chosen model of operations is that it supports generic operations (the type of the target object influences the method selected) and overloading. Its disadvantage is that an operation may have different semantics for different clients, which would be reflected in disjoint sets of methods for different clients (no request form can invoke a method from multiple method sets). Also, the target parameter type in the operation signature is client-dependent.

One rejected model is to identify an operation by the *operation name* alone. The disadvantage of this model is that operation signatures are virtually meaningless: the same *operation name* can have completely different signatures in different client contexts. It also fails to handle overloading: a single client can use the same *operation name* with multiple signatures.

Another rejected model is to identify an operation by an *operation identifier*. The advantage of this model is that an *operation identifier* has a universal signature, as well as having an intended semantics and a set of associated methods. The disadvantage of this model is that, for any particular client, each *operation identifier* maps to exactly one method. Thus, this model does not support generic operations. (The Tickle architecture is likely to change so that operation identifiers support generic operations.)

Iris

An Iris *function* is an operation. An Iris *function* is either a *generic function* or a *specific function*. Each *generic function* has a set of associated *specific functions*. An Iris *function* is an object.

Operations can be created.

Operation creation produces an operation that is distinct from all previously existing operations. Operation creation can be used to avoid accidental overloading of operations. For two clients to refer to a created operation, the identity of the operation must have been communicated from a common origin to each client. A developer can therefore define a new service (by creating a new operation) with the certainty that the service will be distinguishable from all other services.

C++

C++ does not define a way to create a new operation. Any C++ *class definition* might collide with an existing *class definition*.

OSI

A new OSI operation is defined by registering an action or attribute template with a registration authority.

DAA

The DAA does not define a way to create a new operation. Any CDL *interface definition* might collide with an existing definition. The Hewlett-Packard implementation of the DAA includes a utility that can be used to allocate *operation identifiers*.

Tickle

A new Tickle operation can be created indirectly by creating a new object type, as the identity of an operation includes its signature.

Iris

Iris supports operation creation only for *specific functions* (which are non-generic operations). Each *generic function* and its associated *specific functions* have a function name, which is chosen by the client. A client can create a new *function* with a given function name using the system function *CREATEFUN*. If no *function* with that name exists, a new *generic function* and an associated *specific function* will be created. However, if a *function* with that name already exists, a new *specific function* for the existing *generic function* is created. (A change to Iris is under consideration that would allow the creation of *generic functions*.)

Operations are (potentially) *generic*, meaning that a single operation can be uniformly requested on objects with different implementations, resulting in observably different behavior. In particular, the full range of behavior can be produced using a single request form.

This is a philosophical principle, intended to characterize the power of a computational model. A client can uniformly operate on an unlimited number of different kinds of objects by issuing requests for generic operations. Different code may be executed to perform the operation for different kinds of objects.

One would expect the different behaviors for a given operation to share some basic intent, such as the intent to cause something to be printed. This intent must be satisfied in a correct implementation of a system; however, it may not be enforceable by the system.

Generic operations are to be distinguished from the fixed range of behavior that can be supported using a case statement to select from a predetermined set of possible behaviors. In particular, it is possible to introduce new kinds of objects into a system that provide specialized behavior for existing generic operations.

C++

A non-overriding C++ *virtual member function* is a generic operation.

A non-virtual overloaded member function is not a generic operation because overloaded functions are modeled as distinct operations (see below).

OSI

All OSI systems management operations other than *create* are potentially generic. *Create* is not generic because no objects participate in *create* requests. (An alternative view is that managed object classes are objects, and the *create* operation is generic over its managed object class parameter. I reject this view because the managed object class parameter is not sufficient to allow an object to be created. The target system must possess the independent knowledge of how to implement an instance of the class. Thus, a managed object class is not characterized by the *create* operation.)

DAA

All DAA operations are potentially generic.

Tickle

All Tickle operations can potentially have multiple implementations that can be uniformly requested by clients. However, in general, it is not possible for a given client to produce the full range of behavior for an operation, as the number of methods available to the client is limited by the number of types defining the operation in the client's working schema. In particular, introducing a new type into a Tickle environment will not allow an existing client to invoke the methods of the new type. In this sense, Tickle does not support generic operations. (Extensions to support generic operations are under consideration.)

Iris

An Iris *generic function* is a generic operation.

An operation may be identified in a request form by an *operation name*.

The concept of operation name does not play a significant role in the object model, and consequently is not defined rigorously. The term is defined for use in describing the identification of operations in specific object systems.

This model does not require that an operation have a unique operation name. It is possible that different clients could use different names to refer to the same operation, or use the same name to refer to different operations.

Dynamic identification of operations is possible if an operation variable can be used in a request form.

C++

In C++, there are several ways of naming operations in a request form: (1) by specifying the textual name of the function, (2) by specifying the textual name of a member function qualified by the name of the defining class, and (3) by providing a value of a *pointer to function* type or a *pointer to function member* type. The latter form provides dynamic identification of operations. Note that in the first two cases, the static context is also used to determine the named operation (to resolve overloaded functions and redefined non-virtual member functions). (The operations corresponding to public data members are named by specifying the name of the data member, possibly qualified by the name of the defining class.)

OSI

In the OSI CMIP protocol, an operation name is the ASN.1 datatype of the protocol data unit, possibly augmented by the ASN.1 identifier of an attribute or an action. Dynamic identification may be possible by using a variable to provide the attribute or action identifier, depending upon the language binding.

DAA

Operation names in the DAA are determined by language bindings. In the C language binding, operations are named using the names of C external functions defined by the CDL compiler, and dynamic identification is possible using C pointers to functions.

Tickle

A Tickle operation is named in a request form by an *operation name* (a string) and an *operation signature*. Dynamic identification of operations is possible using a string variable to denote the operation. A C++ binding to Tickle might use the names of specific C++ member functions to name operations.

Iris

An operation name is specified in Iris using an OSQL *function expression.* A function expression can be a function name, a variable of function type, or a function invocation that returns an object name identifying a function object.

4.4 Behavior and Abstraction

The *behavior of a request* is the observable effects resulting from performing the requested service. (The effects may be visible to parties other than the requesting client.) The behavior of a request includes the results returned to the client (including both the values returned and the exceptional conditions reported), as well as indirect effects on the results of future requests (by the same or different client).

In general, the possible behaviors of a request are any arbitrary computation, including computations that issue additional requests. (This statement characterizes the intended power of a computational model.)

C++

The behavior of a C++ request is defined by the *function definition* that is invoked. The *function* may perform an arbitrary computation, including the issuing of additional requests.

OSI

The behavior of an OSI simple request is defined by the *managed object class* of the target object. The *class* specifies the direct and indirect effects of the request, and the relevant constraints on attribute values. Direct effects of an attribute operation are the modification or retrieval of the designated attribute, as defined by the base operation (e.g., *get*). All other effects are indirect, including changes to other attributes, or effects on other objects. The indirect effects of an operation can be an arbitrary computation, including the issuing of additional requests. The behavior of a complex request is the combined effect of the generated simple requests, possibly modified by atomic synchronization.

DAA

The behavior of a DAA request is defined by the target object's *object manager.* The behavior of a request whose target object is managed by a Distributed Object Manager is defined by a *procedure.* The *procedure* may perform an arbitrary computation, including the issuing of additional requests.

Tickle

The behavior of a Tickle request is defined by a *tool* (an executable program). The *tool* may perform an arbitrary computation, including the issuing of additional requests.

Iris

The behavior of an Iris request is defined by the Iris *specific function* that is invoked to perform the request. A specific function is either a *stored function,* a *computed function,* an *external function,* or a *primitive function.* A *stored function* has an explicitly stored extent that is modified using associated *update functions.* A *computed function* is defined by an OSQL *program;* it can perform an arbitrary computation, including the issuing of additional requests. An *external function* is defined by a dynamically-linked code module; it *cannot* issue additional requests. A *primitive function* is one whose definition is either predefined in Iris or created implicitly by Iris, such as the *update functions* associated with *stored functions.* The behavior of an *update function* is to modify the extent of its associated *stored function.*

The behavior of a request generally depends both on the request (the identified operation and the actual parameters) and the state of the computational system. The *state* of the computational system is a representation of the history of past requests; specifically, it represents the effect of past requests on future behavior.[2]

> A behavior can thus be modeled as a function whose parameters include the operation, the (actual) request parameters, and the state of the system. The results of the behavior function include the results of the request and the new state of the system. (This modeling technique is called denotational semantics.)
>
> The state of a computational system is physically represented by data that persists between the execution of requests. This persistent data may include what are effectively references to objects. The object model does not say anything about the stored form of such references. Persistent object references are visible to clients only via the values (object names) that are passed as request parameters and results. For example, a persistent object reference might be revealed to a client as the value that is the result of a service that returns a particular attribute of an object.
>
> The behavior of a request may depend on contextual information (such as the identity of the client), as well as the state of other entities interfaced to the system (such as the state of an attached hardware device) or the effects of direct interactions with users. These additional influences would be modeled as additional *context* arguments to the behavior function.

An object explicitly embodies an abstraction, which is characterized by the behaviors of certain requests. The abstraction embodied by an object is meaningful to its clients. An object that is visible to an end user typically models some real-world object that is familiar to the end user.

> These philosophical principles describe the intent to use objects to embody meaningful abstractions. A client that expects to manipulate spreadsheets should be provided with objects that perform the services expected of spreadsheets. The fact that a spreadsheet object behaves as a spreadsheet is inherent in the object and not dependent upon the client's knowledge. A contrasting example from Unix may help to clarify. A Unix file is a low level abstraction of a mutable byte sequence. No additional semantics are attached to a Unix file; instead, the interpretation of a Unix file is left to applications that manipulate the file. A Unix file might represent a spreadsheet, but the Unix file system does not record this intent. Instead, it is up to the client to know to invoke the spreadsheet application when operating upon this particular Unix file. These statements are not intended to rule out systems that support multiple levels of abstraction.

The behavior of a request may depend upon the identity of the participating objects. In particular, a component of the state of the computational system may be uniquely associated with a particular object. In general, such state cannot be modified without issuing requests that identify specific operations in which the object participates. In other words, distinct objects may have distinct state that is *encapsulated* by operations.

2. In a concurrent system, the history of past requests and results may be insufficient to explain future behavior [1].

In a computational system, any parameter can *explicitly* affect the behavior of a request as the result of programming (such as a case statement on object names), or *indirectly* affect the behavior of a request by participating in requests issued as part of performing the request. In this statement, we are concerned with the *implicit* effect of object identity on behavior that results from the structure of the object system. The most common such effect results from binding an object name to an underlying data structure that realizes the state associated with the named object.

The classic example of encapsulated state is a stack object that provides the services *push* and *pop*: the state implicitly associated with a stack object is the sequence of elements that have been pushed but not yet popped. This state is altered only by *push* and *pop*. It is revealed to clients only by the time-varying behavior of *pop*. The stack abstraction is characterized by the behaviors of *push* and *pop*.

There may be additional components of the state of the computational system that are not uniquely associated with individual objects. For example, consider the operations that define the *created* and *created-by* relations over two sets of objects, program source objects and software developer objects. One could think of the state introduced by these operations as being associated with pairs of objects, or alternatively with the operations themselves. The notion of associated state is not intended to restrict the form of the persistent representation of the state.

C++

In C++, an object name serving as a *distinguished parameter* to a non-static member function can *implicitly* affect the behavior of the request by causing the storage of the object to be made accessible to the method via an implicit parameter. This storage consists of the data members declared or inherited by the class of the object, and it can be viewed as the part of the system state uniquely associated with the object.

If the data members are declared private or protected, then the state is accessible only to methods of the class (or related code). Individual objects are not encapsulated, however: a method can access the state of all instances of its class. Public data members are not encapsulated: they can be read and written directly by clients, once the client obtains a reference to the data member.

C++ contains an encapsulation loophole: If a derived class overrides a virtual member function that is public in a public base class, then there is no way to prevent a client from invoking the base class function on an instance of the derived class.

OSI

The *attributes* of an OSI managed object are explicit components of the system state associated with the managed object. A managed object may have additional implicit associated state that is observable by performing systems management operations. The state of a managed object is encapsulated: it can be accessed or changed by clients only by performing systems management operations. (The state of a managed object can also be viewed via notifications and conceivably can be changed as a consequence of notification confirmations returned by clients.) Attribute values may change implicitly by means not defined by the OSI model (for example, as the result of a state change in an associated hardware device).

DAA

A DAA object may have associated state. The realization of that state is determined by the object's *object manager*. An object managed by a Distributed Object Manager typically realizes its associated state using a file. The only means provided by the DAA for a client to access the state associated with an object is via requests for operations supported by the object.

Tickle

A tool can associate an infrastructure object (such as a file) with a Tickle object (based on the infrastructure identifier in an object designator) to implement state that is uniquely associated with the object. The only means provided by Tickle for a client to access the state associated with an object is via requests for operations supported by the object.

Iris

An Iris object can have associated state defined by *stored functions* that accept the object as a parameter. A *stored function* effectively has an associated data location for each possible *combination* of parameters that satisfy the parameter types declared in its signature; a single argument *stored function* defines state associated with individual objects. The state defined by a *stored function* is encapsulated: it can be accessed or modified only by the *function* and its associated *update functions.*

Objects are distinguishable by clients only to the extent that they affect the behavior of requests.

The object model does not require the use of unique object identifiers in a computational system. Furthermore, even if unique identifiers are used, they need not be revealed to clients.

A system may provide specific services for testing object identity that can be requested by clients. For example, a service could be provided that compares two values to see if they identify the same object. The implementation of this service might be more complex than simply comparing the values as bitstrings. For example, the service might report that two values that previously identified different objects now identify the same object. This behavior could be useful when objects are used to model an evolving understanding of the real world, for example, to model the realization that the entities *Joe's father* and *Bob's brother* are actually the same person. The service might report that two values identify the same object, even though they identify distinct objects at a lower level of abstraction. For example, a system might use multiple objects to represent a single mathematical abstraction (such as an immutable set whose elements are the integers 1 and 2). This distinction would be hidden from most clients.

C++

The identity of a C++ object is embodied in its location in storage. Note that the storage of an object can be reused after the object is deleted. C++ objects are distinguishable by comparing their object names (pointer values that encode their addresses) for equality.

An implementation of C++ may use pointers with different address values to identify the same object in different static contexts. (This situation is most likely to arise in the case of a pointer to an unnamed base class subobject, which is modeled as an object name identifying the same object as is identified by a pointer to the enclosing object.) In normal usage, the use of multiple pointer values is hidden from the programmer by implicit type conversions.

OSI

The identity of an OSI managed object is embodied in its ASN.1 distinguished name. Note that the distinguished name is valid only until the object is destroyed. OSI managed objects are distinguishable by their distinguished names. A client can compare distinguished names for equality.

DAA

The identity of a DAA object is determined by the object's *object manager*, which is responsible for creating *object references* that identify the object. Clients can invoke a DOMF primitive that compares two *object references* to determine if they identify the same object. The semantics of this primitive is ultimately determined by *object managers*.

Tickle

The identity of a Tickle object is embodied in the infrastructure identifier contained in an object designator. The infrastructure identifier is determined by the tool creating the object designator. Object designators need not be unique for each object. The extent of an object designator is undefined. Currently, there is no restriction on the ability of a tool to create object designators and consequently no guarantee that infrastructure identifiers contained in object designators are meaningful. Tickle objects are distinguished using operations. Comparison of object names is not defined.

Iris

The identity of an Iris object is an inherent property of the object. The function *EQUAL* can be used to determine if two object names identify the same object. Note that *EQUAL* returns false when applied to two *null* values.

4.5 Object Creation

Objects can be created. Clients create objects by issuing requests. The result of object creation is revealed to the client in the form of a handle that identifies the new object.

Object creation produces an object that is distinct from all previously existing objects. In other words, objects with similar characteristics (e.g., attribute values) can be distinct; it is not necessary for the client to explicitly assign unique key attribute values to create distinct objects.

C++

In C++, objects can be created using the *new* operator, or implicitly using declarations. Static objects are created when the program begins execution. Automatic objects are created when their declarations are executed. These computations are not operations, and only the *new* operator can usefully be invoked via a request. The creation of an object also involves initialization of the object's storage, which is performed by a *constructor*.

OSI

An instance of a specified OSI managed object class is created using the *create* operation. A superior object may be identified in the request, or supplied as a default by the target system. A name binding must be established for the target system that permits the superior object to contain instances of the specified managed object class. (Objects may also be created by means local to a system that are not specified by OSI. The system managed object is presumably created this way.)

The object is given a distinguished name, constructed by concatenating the distinguished name of its superior object with a relative distinguished name. The relative distinguished name consists of the name and value of the distinguished attribute of the new object defined by the name binding. No other object with the same superior object whose lifetime overlaps with the new object can have the same value for that attribute. A relative distinguished name may be specified in the request, or supplied by the system. The *create* operation normally returns the distinguished name of the new object. (It can be omitted if fully specified in the request.)

OSI does not define how a system knows how to create an instance of a managed object class. HP OpenView defines a registry that associates managed object classes with implementations.

DAA

Object creation in the DAA is a service provided by *object managers*. Object creation is typically made available to clients via operations provided by *factory objects*. The Distributed Object Manager provides an object creation service that allocates an *object reference*, associates a specified *implementation class* with the new object, and optionally creates a file to store the object's persistent state. This service is intended for use only by *factory object* methods.

Tickle

Tickle does not define object creation.

Iris

A new instance of an Iris *surrogate type* can be created by invoking the function *CREATEOBJ*, which returns a handle identifying the new object. Optional parameters can be specified for initialization of the new object.

4.6 Object Destruction

It is possible for an object to be destroyed.

C++

C++ objects can be destroyed, either explicitly or implicitly, depending upon the storage class of the object. The destruction of an object may result in the execution of an operation called a *destructor*.

OSI

OSI managed objects may be destroyed either explicitly (using the *delete* operation) or implicitly (e.g., as a side effect of disconnecting a piece of equipment, or of a crash of the containing system). Explicit object destruction is possible only if defined by the name binding used to create the object. The name binding may declare that destruction is permitted only if the object contains no subordinate objects, or that destruction will cause the destruction of all subordinate objects. Other constraints on object destruction can be specified in the managed object class.

DAA

Object destruction in the DAA is a service provided by *object managers*. A Distributed Object Manager provides a service for destroying an object that is intended for use by the object's methods.

Tickle

Tickle does not define object destruction.

Iris

The Iris function *DELETEOBJ* can be used to destroy a surrogate object.

4.7 Meaningful Requests

Not all possible requests are meaningful: an object cannot participate in providing all possible services, and there may be other constraints on the actual parameters in a request.

Requests could be characterized as meaningful at many levels. This model characterizes meaningful requests at a structural or syntactic level, which corresponds to the notion of type-correctness in statically typed systems like C++ or Ada. In a dynamically typed system like Smalltalk, a meaningful request is one that does not raise the error *message not understood*. While the definition of meaningful request in a system need not be static, it is usually thought of as relatively static. Another possible term is *well-formed* request.

A type system may distinguish an ill-formed request from a well-formed request that cannot be performed. For example, an attempt to divide an integer by a string is considered ill-formed by most type systems. An attempt to divide an integer by zero is also erroneous in a static sense, but is considered well-formed by most type systems. The difference is that a request *form* identifying the division operation is ill-formed if the second parameter expression is a string variable, but is well-formed if the second parameter expression is an integer variable (even if the value of that variable happens to be zero.) The distinction between syntactic and semantic errors is usually based on the declarative power of the type system.

There is no implication that an object system must detect a meaningless request. Issuing a meaningless request may or may not result in an exception condition being reported to the client.

C++

A C++ request is meaningful if it is type-correct.

OSI

An OSI systems management operation request is meaningful if it satisfies a set of conditions either imposed by the structure of the OSI model itself or declared in a managed object class or a name binding. Such conditions include missing parameters, invalid attributes or parameters, invalid operations, and parameter value of wrong type or out of range. (We exclude dynamic conditions, such as nonexistent object, duplicate object, access control violations, constraints among multiple attributes, and general failures to perform an operation.)

DAA

A DAA request is meaningful if the designated target object supports the CDL *interface* that defines the identified operation and the request parameters are consistent with the signature of the operation.

Tickle

A Tickle request is meaningful if: the type of the designated target object is defined in the working schema of the client tool activation to support the indicated operation, the request parameters are consistent with the defined *operation signature*, and the dynamic context maps the operation identifier to an entry point. The meaningfulness of a request is thus client-dependent.

Iris

An Iris request is meaningful if the number and types of the actual parameters satisfy the signature of the *specific function* that is selected to perform that request. The *null* value is a legitimate parameter regardless of the Iris *type* in the signature. (A change is under consideration to allow a signature to disallow the *null* value.)

Each operation has an associated *signature* that may restrict the possible (actual) parameter values that are meaningful in requests identifying that operation. A request is meaningless if the actual parameters do not satisfy the signature associated with the operation identified in the request.

A signature also characterizes the possible result values and exceptional conditions associated with the corresponding requests. This additional information does not affect the notion of meaningful request, but may be used by tools that perform static analysis of programs.

For those readers familiar with typed programming languages, a signature is like a procedure type or a function type. In the case of generic operations, a signature may accept arguments of multiple types. For example, the signature of the *plus* operation might specify that there are two parameters, the actual arguments can be any combination of integers or reals, and the result is an integer if the two arguments are both integers, or a real otherwise.

The object model does not specify the language used to describe signatures. An elaboration of this object model would define a specific formalism for signatures.

The object model does not specify the possible characterizations of results. Typically, signatures will characterize results in terms of types. Signatures may include additional information such as formal or informal descriptions of the behavior of the operation, or information about whether requests are synchronous or asynchronous.

Note that if operation names are values (meaning that operation names can be parameters in requests), then signatures can be considered to be types, as defined below. For example, one might require the value of a parameter to be an operation that accepts two integer parameters and returns one integer result. It is in this sense that operations are shown to have types in Figure 1.

C++

The signature of a C++ operation is specified as part of the original definition (or declaration) of the corresponding *function*. The signature includes C++ *types* for each formal argument to the *function* and the C++ *type* of the result of the *function*. A formal argument of a *class type* is represented in the signature by the corresponding *reference type* (see page 8). A *non-static member function* has an implicit parameter denoting the target object whose type is *reference to class C*, where *C* is the *class* containing the *function definition*.

The types of allowable argument *expressions* is defined not only by the function signature, but also by the set of available *conversions* and (in the case of a formal of class type) *constructors*. In the case of reference or pointer parameters, the possible conversions are limited to trivial conversions from a derived class reference or pointer to a (public) base class reference or pointer, and from any pointer to a *void** pointer. C++ signatures currently do not specify exceptions.

OSI

The signature of an OSI systems management operation is effectively formed by combining multiple specifications of the operation in package definitions and (in the case of *create*) name bindings that are registered with a registration authority. (*Delete* is an exception: it has a fixed signature.)

Multiple specifications allow a subclass to *refine* the definition of an inherited operation. Because the definition of an operation may differ in different classes, the effective signature is a function of a managed object class (the apparent managed object class of the target object, taking allomorphism into account). The signature can be viewed as consisting of multiple *clauses*, one for each managed object class of a possible target object. The signature of an operation can change (by extension) as new subclasses are defined that refine the definition of the operation, or as new classes are defined that include a package defining the operation.

The legitimate parameter values for an attribute operation depend upon the definition of the attribute in each managed object class. The base type of the attribute value is defined in the attribute definition. However, each package that defines the attribute can further restrict the domains (values legal as a parameter to *set*) and ranges (values legal as a result of *get*) of the attribute operations, by refining the sets of required and permitted values.

The signature of *create* contains multiple clauses based on the name binding used to create the object (normally determined by explicit parameters in the request). The signature clauses include the attributes that may be initialized explicitly via request parameters, the types of domains of the initial values, and the conditional packages that may be included.

Action operations are defined only on objects of specific managed object classes, and only on those objects that include certain optional packages. The signatures of these operations would constrain the value of the parameter that names the target object. The signature would include a description of the other parameters that may (or must) be provided and the types of those parameters, as well as the result parameters that may (or must) be returned.

The other standard parameters of systems management operations are constrained by type. Standard exceptions are defined for all signatures; individual signatures may include specific exceptions as well. Each operation signature includes a *behavior*, which informally describes the behavior of the operation.

DAA

The signature of a DAA operation is specified as part of its definition in a CDL *interface definition*. An operation signature includes a CDL description of the parameters of the operation and the result value. An operation signature identifies the operation as synchronous or asynchronous. The type of the target object parameter is implicitly the CDL type *pointer to interface I*, where *I* is the CDL *interface* that defines the operation.

Tickle

The signature of a Tickle operation is specified in two ways. The type of the target object parameter is implicitly determined by the object types to which the operation is defined to be applicable in the client's working schema. The remainder of the signature (describing the explicit invocation parameters) is implicit in the operation (which is identified in part by an *operation signature* that includes a NIDL description of each explicit parameter).

The signature of a Tickle operation is client-dependent; it can vary in the type of the target object parameter. Each tool activation has its own working schema, and an operation may be applicable to different types in different schemas.

Iris

An Iris *specific function* has a signature that defines the number and types of legitimate parameters and the type of the result value. Each *specific function* takes a fixed number of parameters. The parameter and result types are defined in terms of an Iris *type*; the actual parameter or result type is the union of the Iris *type* and the type whose only member is the *null* value. The *specific functions* corresponding to a particular *generic function* must have the same result type, but they can accept different numbers or types of parameters. The parameter part of the signature of a *generic function* is thus a combination of the signatures of the corresponding *specific functions*, and it may change over time as associated *specific functions* are defined or removed. When the last *specific function* for a *generic function* is removed, the *generic function* is deleted.

When an operation is created, its signature is specified.

The object model does not specify whether the signature of an operation can change after it is created.

4.8 Types

A *type* is an identifiable entity with an associated predicate (a single-argument mathematical function with a boolean result) defined over values. A value *satisfies* a type if the predicate is true for that value. A value that satisfies a type is called a *member of the type.*

Types are used in signatures to restrict a possible parameter or to characterize a possible result.

Types are traditionally used both to classify and to constrain. We emphasize the use of types to constrain.

This definition of type is very general. According to this definition, a type such as *integer* would be thought of as having an associated predicate that is true for an integer value and false for any other value. This definition allows unusual types, such as the type of *odd integers*. Note that this definition allows a single value to satisfy multiple types. For example, the value denoting the integer 1 is both an integer and an odd integer.

The object model does not specify what kinds of types may exist in a system. An elaboration of this model would define a specific formalism for types. The choice of formalism may restrict the predicates that can be associated with types in a system. Other types are only conceptual: although not expressible in the system's formal language, conceptual types may be useful when informally describing the system.

Because types are defined as predicates over *values* rather than *objects*, it is possible for a type to discriminate among multiple values that identify the same object (or other abstract entity). For example, a system might provide two forms of object names: a long form that is globally meaningful and a short form that is valid only within one process context. The type *mailbox specified via long object name* would be satisfied by a long name for a given object, but would not be satisfied by a short name for the same object. Such value-dependent types are not required in an object system. If the particular object model does not allow types to distinguish different values that identify the same object (or other abstract entities), then one can view type predicates as defined over objects (and other abstract entities), rather than over values.

Existing object-oriented programming languages provide types that distinguish between different implementations of objects — called classes. A more general notion of interface type is defined below that captures the intuition of generic operations.

The object model does not specify whether types are objects, or whether type predicates are invocable as operations. In some systems, type checking is performed at compile time and types are not represented during execution. The predicate associated with a type need not be effectively computable.

C++

A C++ *type* is a type. A C++ type distinguishes between *value* and *reference* parameters, as well as indicating whether the actual parameter can be modified or used to modify an object. A C++ type is not a value.

OSI

The legitimate values of parameters of OSI systems management operations are defined using ASN.1 datatypes, which includes the type *ObjectInstance* that is satisfied by object names. Although not usable in signatures, a managed object class can be considered to be a type that is satisfied by all of its instances and all instances of its allomorphic subclasses. An OSI type is not a value.

DAA

In the DAA, a CDL *datatype* is a type. A DAA type is not a value.

Tickle

The legitimate values of the explicit parameters of Tickle operations are defined using NIDL datatypes. A Tickle *object type* is used in signatures only to describe the legitimate values of the implicit target object parameter. A Tickle type is not a value.

Iris

An Iris *type* is a type. An Iris type is an object.

A type may have different predicates at different times. The *extension of a type* is the set of entities that satisfy the type at any particular time.

If types do not distinguish multiple values that identify the same object, then one can think of the extension of the type as containing the objects identified by those values, rather than the values themselves.

The extension of a type may change over time, for example, as new objects are created or existing objects are modified by side-effects.

A relation called *subtype* is defined over types. A type *a* is a subtype of a type *b* if any value that satisfies type *a* necessarily satisfies type *b*.

A particular object model defines the subtype relation, which must be consistent with logical implication over the type predicates. (For example, if all integers are reals, then the type integer must be a subtype of the type real.) Subtypes permits a hierarchical classification of objects, which is one of the traditional uses of inheritance. An example of subtype is presented in the commentary following the next paragraph.

C++

In C++, the type *pd* is a subtype of type *pb*, where *pd* is pointer to class *d*, *pb* is pointer to class *b*, and *b* is an unambiguous direct or indirect public base class of *d*. A similar relation is defined for reference types. (This subtype relation is a consequence of not modeling a derived class instance as containing separate objects for each base class.)

OSI

The (conceptual) type implied by an OSI managed object class is a subtype of the type implied by each allomorphic superclass of the managed object class, in the sense that an instance of the managed object class may be operated upon by a client as if it were an instance of the allomorphic superclass. Note that subtyping is not defined for ordinary superclasses, and that allomorphism is *not* transitive.

DAA In the DAA, a CDL *interface* can be derived from one or more CDL *interfaces*. A CDL type *pointer to interface I* is a subtype of any CDL type *pointer to interface J* where *I* is directly or indirectly derived from *J*.

Tickle Subtyping over Tickle object types is defined by type inheritance (the parent relation).

Iris All Iris *types* except the type *OBJECT* are defined to inherit from one or more *types*. An Iris *type* is a subtype of any *type* it directly or indirectly inherits from. All Iris *types* are a subtype of the type *OBJECT*.

An *object type* is a type whose members are objects (literally, values that identify objects). In other words, an object type is satisfied only by (values that identify) objects.

An object may satisfy many types. For example, a Lotus 2.2 spreadsheet object might satisfy the following types (of decreasing specificity): Lotus 2.2 spreadsheet, Lotus spreadsheet, spreadsheet, printable object. In this example, the types are all related by subtyping. Some systems might allow an object to satisfy types unrelated by subtyping. For example, an object representing a particular individual might satisfy both the type person and the type shareholder (there can be people who are not shareholders and shareholders that are not people, e.g., corporations).

C++ A C++ *pointer to class type* or a C++ *reference to class type* is an object type. Note that a plain C++ *class type* has no values (see page 8).

OSI An OSI *managed object class* can be considered to be an object type. However, the only object type that can be used in signatures is the type *ObjectInstance*, which is satisfied by any object name.

DAA In the DAA, a CDL *pointer to interface type* is an object type.

Tickle A Tickle *object type* is an object type. The type *object designator* is an object type.

Iris An Iris *type* is an object type. An Iris *surrogate object* can simultaneously be a member of multiple *surrogate types* that are not subtypes of each other. A *surrogate object* can be dynamically added to or removed from a *surrogate type*.

4.9 Interfaces

An *interface* is a description of a set of possible uses of an object, i.e., its possible appearances as a parameter in a request. Specifically, an interface describes a set of potential requests in which an object can meaningfully participate. An object *satisfies* an interface if it is meaningful in each potential request described by the interface.

In a classical object system, an interface consists of a set of operations. For example, the interface of *printable objects* would consist of the single operation *print*. An object satisfies a classical interface if it is meaningful as the first parameter in any request that identifies one of those operations. For example, a *printable object* is one that supports the *print* operation.

A system supporting the generalized object model might support more complex interfaces that identify specific formal parameters of operations where an object can meaningfully appear. For example, the *printable object* interface would be satisfied by those objects that can appear as the first argument to a *print* request, and the *printer object* interface would be satisfied by those objects that can appear as the second argument to a *print* request.

Interfaces are in some sense duals of signatures. One can imagine the system containing a set of constraints that determine the meaningfulness of requests. Signatures are an operation-centered view of this information, and interfaces are an object-centered view of this information.

The object model does not specify how this information originates. An object might be printable because it defines behavior for the *print* operation. Alternatively, an object might be printable because it has been asserted to satisfy the type *printable object*, which is the declared type of the first argument of the *print* operation.

C++ A C++ *class* whose only members are *pure virtual functions* is an interface.

OSI An OSI *managed object class* is an interface.

DAA In the DAA, a CDL *interface* is an interface.

Tickle A Tickle *object type* is an interface. It defines a set of operations that a client can request of objects.

Iris Iris does not have interfaces. An Iris *type* can be viewed as having an associated interface that is determined by the *functions* defined to take parameters of the *type*. However, an Iris *type* is not itself an interface, because the set of associated *functions* can change dynamically.

An *interface type* is a type that is satisfied by any object (literally, any value that identifies an object) that satisfies a particular interface.

An interface can both describe what a particular object can do, as well as describe how a client intends to use an object. For example, an interface type might be used to declare a formal parameter of a procedure, to indicate how the actual parameter object will be used in the procedure. An interface type captures the client-expected behavior, not the implementation of that behavior (unlike a class type).

Subtyping over interface types means that any object that can be used as described by the smaller interface can also be used as described by the larger interface. Subtyping over interface types is related to set inclusion, at least for simple type systems. The interface consisting of the *print* and *copy* operations is a subtype of to the interface consisting of just the *print* operation, since any object that can be printed and copied can (by implication) simply be printed.

C++ A C++ *pointer to class type* or a C++ *reference to class type* whose class is an interface is an interface type.

OSI An OSI *managed object class* can be viewed as an interface type. However, it cannot be used in signatures.

DAA In the DAA, a CDL *pointer to interface type* is an interface type.

Tickle

A Tickle *object type* can be viewed as an interface type. However, it cannot be used in signatures.

Iris

Iris does not have interface types.

The *principal interface* of an object describes all requests in which the object is meaningful.

The principal interface of an object is the most general interface that describes an object. The principal interface of an object may enlarge or shrink for various reasons, such as the definition of new operations. The object model permits but does not require the existence of a service by which a client can determine the principal interface of an object.

C++

A C++ class defines the maximal set of *member functions* that can be invoked on its instances. C++ does not define the principal interface of an object, in the sense that an object can also be a parameter to an unlimited number of ordinary functions.

OSI

An OSI *managed object class* is the principal interface of its instances.

DAA

The principal interface of a DAA object is determined by the object's *object manager*. The principal interface of an object managed by a Distributed Object Manager is the CDL *interface* used to register methods for the object's *implementation class* with the Distributed Object Manager.

Tickle

The meaningful uses of a Tickle object from a given tool activation is determined by the definition of the type of the object in the tool activation's working schema. Because operations include names that can differ arbitrarily in each schema, there is no *universal* principal interface of an object. This property is a consequence of our modeling choice for operations. Each context can define its own set of operations by choosing arbitrary operation names.

Iris

The principal interface of an Iris object is defined by the *specific functions* that are defined to take parameters of the object's *type* or *types*.

4.10 Templates

A *template* is a type that can be *instantiated* to create a new member of the type. A member of a template created by instantiating the template is called an *instance* of the template.

This definition does not require that a member of a template must have been created by instantiating the template. A template may or may not be an object.

A template may be defined incrementally from other templates. This process is called *template inheritance*.

This definition does not require that a template defined by inheritance be a subtype of the source template(s).

5 Object Implementation

This section defines the concepts associated with object implementation, i.e., the concepts relevant to realizing the behavior of objects in a computational system. Figure 3 illustrates the relationships among several key object implementation concepts.

The implementation of an object system carries out the computational activities needed to effect the behavior of requested services. These activities may include computing the result of the request and updating the system state. In the process, additional requests may be issued.

The most important aspects of object implementation for our purposes are the expressive power of binding (the ability to support generic operations) and the capabilities provided for the sharing and composing of implementation units (the ability to support reuse).

These two aspects are captured by the two parts of the implementation model, respectively: the execution model and the construction model. The execution model describes how services are performed. The construction model describes how services are defined.

There is no single right way to implement an object system. This model is designed to capture common characteristics of current object systems.

5.1 The Execution Model: Performing Services

A requested service is performed in a computational system by executing a program that operates upon some data. The data represents a component of the state of the computational system. The program performs the requested service, which may change the state of the system.

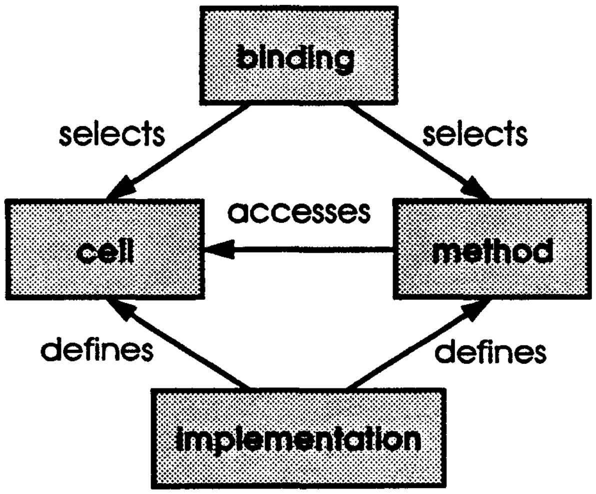

Figure 3. Primary object implementation concepts

A program that is executed to perform a service is called a *method.* A method is an immutable description of a computation that can be interpreted by an *execution engine.* A method has an immutable attribute called a *method format* that defines the set of execution engines that can interpret the method. An *execution engine* is an abstract machine (not a program) that can interpret methods of certain formats, causing the described computations to be performed. An execution engine defines a dynamic context for the execution of a method. The execution of a method is called a *method activation.*

A method might implement multiple operations. The notion of method format is intended to capture heterogeneous systems. The set of method formats understood by an execution engine might change over time. An execution engine might be concurrent, multi-threaded, or sequential.

C++

A C++ *function definition* is a method. A public data member implicitly defines a method that returns a reference to the data member. C++ typically assumes a single sequential execution engine and a single method format.

OSI

OSI does not define the concept of a method. An HP OpenView *management service* (an "M") is a method.

DAA

The implementation of objects in the DAA is the responsibility of *object managers.* An object managed by a Distributed Object Manager is characterized by its *implementation class.* (The *implementation class* is intended to define the principal interface of the object, the behavior of requests to that object, and the representation of the state associated with the object; however, there is no explicit definition of these properties. The DAA does not define a way to create an implementation class. The Hewlett-Packard implementation of the DAA includes a utility that can be used to allocate *implementation class identifiers.*) An *implementation class* may be registered in a Distributed Object Manager with an associated *program* containing a set of *procedures* supporting specific operations; each *procedure* is a method. Typically, an operating system *process* serves as an execution engine. Typically, there are different methods for an *implementation class* for different hardware environments. The notion of method format is not explicit in the DAA.

Tickle

A Tickle *tool* is a method. A *tool activation* is an execution engine. In Tickle, a request is performed by invoking an *entry point* of a *tool activation* (a running process). The *object designator* for the target object is passed to the *tool activation*; the *tool activation* may use the *object designator* to access persistent data associated with the target object. The request parameters may be translated before presenting them to the *tool activation.* Parameters of requests issued by the *tool activation* may also be translated. Parameter translation supports tool integration.

Iris

An Iris *specific function* is a method. The signature of a *specific function* can be defined prior to giving a complete definition; attempting to execute an incomplete *specific function* is an error. Iris assumes a single sequential execution engine and a single method format.

The data operated upon by methods consists of independent units called *cells*. A cell is a (potentially) mutable data structure; changes to one cell do not affect other cells. The contents of a cell might include object names. A cell is defined by its associated *representation*. A representation is a concrete datatype: it defines the format of the data and the primitives used by methods to access or modify the cell. A method can operate only on cells with certain representations.

An object might have zero, one, or multiple associated cells. The cells associated with an object might change dynamically. Operating on a cell is a primitive action; it does not involve issuing a request. The representations understood by a method might change over time.

C++

A C++ *data member definition* defines a cell; the associated C++ *datatype* is a representation. The cells uniquely associated with a C++ object of a class type are the components corresponding to the *ordinary data members* declared by the class and the subobjects of each direct base class. A *static data member definition* defines a unique cell, like a global variable. A *non-static member function* is given access to the cells associated with the target object via lexically scoped variables. Other system state may be accessed via lexical variables or pointer and reference values.

OSI

OSI does not define the concept of a cell. A managed object may have associated data stored in a *management information base*. HP OpenView supports the association of transient or persistent data with an object.

DAA

The implementation of objects in the DAA is the responsibility of *object managers*. The data associated with a Distributed Object Manager object is implicitly determined by the methods registered for the object's *implementation class* (and the methods of associated *factory objects*). In many cases, a *file* is used to represent the state associated with an object, in which case each *file* is a cell. The Distributed Object Manager does not define the representation of a *file*. (It is intended that all registrations of a particular *implementation class* use a mutually compatible representation for the state associated with an object.)

Tickle

Tickle does not define cells or representations. A Tickle *tool activation* may use an object designator to access persistent data associated with an object.

Iris

An Iris *stored function* implicitly defines a cell for each *combination* of possible parameters. The result *type* of the *function signature* implicitly defines the representation of the cell.

An object system includes an infrastructure that serves as the mediator between clients and service providers. (Of course, a service provider can also be a client, by issuing requests of its own.) The infrastructure processes a request by transforming it into a method invocation; this transformation is called *binding*. A *method invocation* identifies a method to be executed, an execution engine to interpret the method, and a set of method invocation parameters, which may identify cells that are to be operated upon by the method. The input to binding includes the request (the operation and parameters), and may also include a *request context*.

Method invocation parameters may include entities that are not values, i.e., are not legitimate request parameters. The execution engine might be created during the binding process. A request context allows the semantics of a request to be client-dependent.

Binding involves the selection of a method to perform a requested service. Binding can be *static*, meaning that the selection of the method can be performed prior to the actual issuing of the request, or *dynamic*, meaning that the selection is performed after the request is issued.

Static binding is performed by compilers based on declarations. A request form often includes variables. The declarations in the program may restrict the possible values denoted by those variables, allowing the proper method to be identified by the compiler.

C++ C++ *virtual member functions* require dynamic binding to select a method when the target object is accessed via a reference or a pointer. Static binding is possible in most other cases.

OSI All binding in OSI systems management is dynamic.

DAA All binding in DAA is dynamic.

Tickle All binding in Tickle is dynamic.

Iris Iris *generic functions* generally require dynamic binding to select a method.

The method selected to perform a requested service can depend upon several factors, including the identity of the objects participating in the request.

The selection of a method based on the participating objects is the basis for the description of operations as generic. In a classical object system, the selection of a method depends only upon the operation and the implementation (type) of the object identified by the distinguished parameter.

The selection of a method may also depend on other factors, such as the "location" of the object or the "location" of the client. (The object model does not define a concept of location.) A specific object technology might impose constraints on the process of binding that provide increased predictability for clients. The object model intentionally does not impose such constraints.

C++ The method selected for a C++ *virtual member function* depends upon the class of the target object.

OSI The method selected to perform an operation on an OSI managed object is determined by the system containing the object. One would expect the selection to be based on the managed object class of the object.

DAA The method selected to perform a DAA request is determined by the *object manager* of the target object. A Distributed Object Manager selects a method based on the object's *implementation class*. The selected method is the one registered in the Distributed Object Manager as implementing the operation for that *implementation class*.

Tickle

The method selected to perform an operation on a Tickle object is determined by the working schema and the dynamic context of the client tool activation. The client-specified operation (name and signature) is first mapped to an *operation identifier*, based on the type of the target object and the definition of that type in the client's working schema. The operation identifier directly determines an *entry point* of a specific tool, as defined by the dynamic context. Pattern matching is then performed to identify a particular activation of that tool (which need not have a unique association with the target object). A new tool activation may be created. Binding is thus client-dependent.

Iris

The Iris *specific function* selected to perform a request that identifies a *generic function* depends upon the *types* of all of the parameters.

Performing a requested service causes a method to execute that may operate upon cells. If the persistent form of the method or a cell is not accessible to the execution engine, it may be necessary to first copy the method or the cell into an executable address space. This process is called *activation*. The reverse process is called *passivation*.

C++

All C++ methods are executable at all times.

OSI

HP OpenView activates a management service as needed to perform requests.

DAA

Activation in DAA is defined by the target object's *object manager*. A Distributed Object Manager activates an object by creating a process to execute the *program* that has been registered with the Distributed Object Manager as containing the methods for the object's *implementation class*. Alternatively, the *program* may be dynamically loaded into an existing process, if the object is a member of the same *object group* as the objects already activated into that process.

Tickle

The Tickle dynamic context manager activates a tool to perform a request if no suitable tool activation already exists.

Iris

All Iris methods are executable at all times.

5.2 The Construction Model

5.2.1 Realizing Behavior

A computational object system must provide mechanisms for realizing the behavior of requests. These mechanisms would include definitions of cells, definitions of methods, and definitions of how the object infrastructure is to select the methods to execute and to select the cells to be made accessible to the methods. Mechanisms must also be provided to describe the concrete actions associated with object creation, such as the allocation of new cells, and the association of the new object with appropriate methods.

An *object implementation*—or *implementation*, for short—is a definition that provides the information needed to create an object and to allow the object to participate in providing an appropriate set of services. An implementation typically includes a description of a cell used to represent the core state associated with an object, as well as definitions of the methods that operate upon that cell. It also typically includes information about the intended type of the object. A cell that is uniquely associated with an object is called the *realization* of the object. (An implementation might also be used to add new behavior to an existing object.)

C++

A C++ *class definition* and its associated *member function definitions* are an implementation. (A derived class implementation includes the base class definitions.)

OSI

OSI does not define object implementations. It requires only that the managed objects implemented by a system behave as specified in their managed object class definitions. An HP OpenView *management service* is an object implementation. A *management service* registers itself with the infrastructure as supporting objects of a specific managed object class. The interface (API) presented to the infrastructure by a management service is called the *managed object interface*; it consists of a set of defined procedure calls.

DAA

Object implementations are defined by DAA object managers. A *program* registered with a Distributed Object Manager can be viewed as an object implementation. The *program* defines methods for a set of operations for objects of a given *implementation class*; a *program* could implement operations for multiple *implementation classes.* A *program* may or may not define how to create an object. The association between a *program* and an object data representation is implicit. (It is intended that all object implementations for a given *implementation class* use a compatible data representation.)

Tickle

A Tickle *tool* is an object implementation. A *tool* defines the implementation of one or more operations in a given context; it could implement operations on multiple object types. A *tool* may or may not define how to create an object. The association between a *tool* and an object data representation is implicit.

Iris

An Iris *type* and the *specific functions* defined to take parameters of that *type* are an implementation.

An object system may allow a method to reference the object(s) identified in the request for the service being performed by the method. This ability, called *self-reference*, allows a method to issue additional requests involving the same object(s). Self-reference is useful when a single method may be executed for different objects. (Self-reference in Smalltalk is indicated by the keyword *self*.)

C++

A C++ *non-static member function* can reference the target object of the request being performed via the implicit parameter *this*.

OSI

This notion is not defined by OSI.

DAA

This notion is not defined by DAA.

Tickle

The target *object designator* is a parameter to a Tickle tool *entry point.*

Iris

An Iris *specific function* can access all request parameters.

5.2.2 Sharing Behavior

An object system typically provides mechanisms that allow objects with the same behavior to share implementations.

However, it is also possible for *different* implementations to support the *same* behavior. For example, a system may contain multiple implementations for the same behavior that operate in different hardware environments. Also, a system may contain a series of implementations for the same behavior that incorporate performance enhancements that were introduced over time.

An *implementation template* is an implementation that can be *instantiated* to create multiple objects that have the same initial behavior. The resulting objects are called *instances* of the template. Self-reference is useful when methods are shared by multiple instances of a template.

In many existing object systems, a class defines both an implementation template and an interface type. Merging these roles in a single entity makes it more difficult for a system to support multiple implementations of the same behavior.

C++

A C++ *class definition* is an implementation template. The associated *member function definitions* are shared by all instances of the class. A C++ *class definition* is not a value.

A C++ *class member* can be declared *private*, which prevents access other than by the class declaration, the associated member function definitions, and a list of explicitly identified *friend* classes and functions.

OSI

An HP OpenView *management service* is an implementation template: it can be instantiated to create multiple objects with the same behavior (instances of the same managed object class). An HP OpenView *management service* is not a value.

DAA

The DAA does not define implementation templates. A *program* registered with a Distributed Object Manager as implementing objects of one or more *implementation classes* is shared by the objects of those *implementation classes* managed by that Distributed Object Manager. However, the *program* cannot be instantiated.

Tickle

Tickle does not have implementation templates. A tool can implement operations on multiple objects. A tool activation performing an operation on an object may have access to other operations not generally available (operations declared *private* in its working schema), by virtue of the *effective type* of the object becoming the type where the *operation identifier* is defined in the client's working schema.

Iris

An Iris *surrogate type* and the *specific functions* defined to take parameters of that *type* are an implementation template. The *specific functions* are shared by all objects of that immediate type. A *stored function* shared by multiple objects implies associated cells that share a representation.

An object system typically provides mechanisms that allow objects with similar behavior to share parts of their implementations. For example, *implementation inheritance* may be provided to allow an implementation to be defined as an incremental refinement of other implementations.

C++

A C++ *class* can be derived from one or more *base classes*, inheriting the declarations of data members and member functions, and the definitions of associated member functions. The *derived class* can declare additional data members and member functions, and can define the additional member functions and redefine inherited virtual member functions.

A C++ *class member* can be declared *protected*, which prevents access other than by classes publicly derived from the base class that declares the member.

OSI

Implementation inheritance is not provided by OSI or HP OpenView.

DAA

The DAA does not provide implementation inheritance.

Tickle

Tickle provides static and dynamic means of sharing its implementations (specifically, methods). Statically, a Tickle tool can be defined as the composition of other tools. Dynamically, a composite tool can inherit methods from its parent dynamic context (the dynamic context of the tool that invoked the composite tool). Tickle does not support sharing of data definitions.

Iris

Iris objects share partial implementations as a result of inheritance. An object of one Iris *type* shares all *specific functions* that accept parameters of a *supertype*, except those overridden by another *specific function*. Additional *specific functions* can be defined that are applicable to the object but not to other members of a supertype.

An alternative technique for providing sharing of implementations is *delegation*. Delegation is the ability for a method to issue a request in such a way that self-reference in the method performing the request returns the same object(s) as self-reference in the method issuing the request.

Delegation can be modeled by making the self-reference context a method invocation parameter (which determines the meaning of self-reference in the invoked method activation) and incorporating the self-reference context of the client method activation in the *request context* of a delegated request (which allows the self-reference context of the client to be used as the method invocation parameter for the new method activation).

C++

C++ does not support delegation.

OSI

Delegation is not provided by OSI or HP OpenView.

DAA

The DAA does not support delegation.

Tickle

Tickle does not support delegation.

Iris

Iris does not support delegation.

6 Glossary

activation. Copying the persistent form of methods and cells into an executable address space to allow execution of the methods on the cells.

behavior (of a request). The observable effects of performing the requested service (including its *results*).

binding. The process that transforms a *request* into a *method invocation*. Binding selects a *method* to perform the requested service, an *execution engine* to interpret the method, and may select *cells* to be operated upon by the *method activation*. See also *dynamic binding* and *static binding*.

cell. An independent data structure that is operated upon by *methods*. A cell is potentially mutable; changes to one cell do not affect other cells. A cell is defined by an associated *representation*.

client. Any entity capable of requesting services.

delegation. The ability for a method to issue a request in such a way that self-reference in the method performing the request returns the same object(s) as self-reference in the method issuing the request. See *self-reference*.

dynamic binding. *Binding* where the selection of the *method* is performed after the request is issued.

encapsulated. An object is encapsulated if it has associated state that can be directly accessed or modified only by particular methods.

execution engine. An abstract machine that can interpret *methods* of certain formats, causing the described computations to be performed.

extension (of a type). The set of values that satisfy the type. See *satisfies*.

generic operation. An operation that can be uniformly requested of objects with different implementations, resulting in observably different behavior.

handle. A *value* that reliably identifies an object. See *object name*.

implementation. A definition that provides the information needed to create an object, allowing the object to participate in providing an appropriate set of services. An implementation typically includes a description of a *cell* used to represent the core state associated with an object, as well as definitions of the *methods* that operate upon that cell. It also typically includes information about the intended type of the object.

implementation inheritance. The construction of an implementation by incremental refinement of other implementations.

implementation template. An implementation that can be instantiated to create multiple objects with the same initial behavior.

inheritance. The construction of a definition by incremental refinement of other definitions. See *implementation inheritance.*

instance (of a template). An object created by instantiating the template.

instantiation. Creating an object from a template.

interface. A description of a set of possible uses of an object. Specifically, an interface describes a set of potential requests in which an object can meaningfully participate. See also *principal interface.*

interface type. A type that is satisfied by any object (literally, any value that identifies an object) that satisfies a particular interface. See *object type.*

literal. A *value* that identifies an entity that is not an object. See *object name.*

meaningful request. A request whose actual parameters satisfy the signature of the identified operation.

member (of a type). A value that *satisfies* the type.

method. A program (an immutable description of a computation) that can be interpreted by an *execution engine* to perform a requested service. A method has an associated *method format* that defines the set of execution engines that can interpret the method.

method activation. A particular execution of a *method*, corresponding to a specific *method invocation.* A method activation has an associated dynamic context that includes *method invocation parameters.*

method format. An immutable description of a *method* that defines the set of *execution engines* that can interpret the method.

method invocation. An identification of a *method* to be executed, an *execution engine* to interpret the method, and a set of *method invocation parameters.* Each *request* is transformed by *binding* into a method invocation.

method invocation parameter. A parameter to a *method invocation.* A method invocation parameter may identify a *cell* that is to be operated upon by the *method activation.*

object. An identifiable entity that plays a visible role in providing a service that can be requested by a client. An object explicitly embodies an abstraction characterized by the behavior of certain requests.

object creation. An event that causes an object to exist that did not previously exist.

object destruction. An event that causes an object to cease to exist and its associated resources to become available for reuse.

object implementation. See *implementation.*

object name. A *value* that identifies an object. See *handle.*

object system. A collection of objects that isolates requestors of services (*clients*) from the providers of services by a well-defined encapsulating interface.

object type. A type whose members are objects (literally, values that identify objects). In other words, an object type is *satisfied* only by (values that identify) objects. See *interface type.*

operation. An identifiable entity that denotes a service that can be requested. An operation has an associated *signature,* which may restrict which actual parameters are possible in a meaningful request.

operation name. A name used in a request form to identify an operation.

participate. An object participates in a request if and only if one or more of the actual parameters of the request identifies the object.

passivation. The reverse of *activation.*

principal interface. The interface that describes all requests in which an object is meaningful.

realization (of an object). A *cell* that is uniquely associated with an object.

representation. An immutable description of a *cell* that defines the format of the data and the primitives used by *methods* to access or modify the cell.

request. An event consisting of an operation and zero or more actual parameters. A client issues a request to cause a service to be performed. Also associated with a request are the results that may be returned to the client.

request context. The information associated with a request other than the operation and the parameters that is needed to determine the result of *binding.* The request context is used to support client-dependent binding.

request form. A description or pattern that can be evaluated or performed multiple times to cause the issuing of requests.

results (of a request). The information returned to the client, which may include values as well as status information indicating that exceptional conditions were raised in attempting to perform the requested service.

satisfies. A value satisfies a type if the predicate associated with the type is true for that value. See *member*.

self-reference. The ability of a method to determine the object(s) identified in the request for the service being performed by the method. (Self-reference in Smalltalk is indicated by the keyword *self*.) See *delegation*.

service. A computation or information processing action that may be performed in response to a request.

signature. A description associated with an operation that may restrict the possible (actual) parameter values that are meaningful in requests that identify that operation. A signature also characterizes the possible result values and exceptional conditions associated with those requests.

state. The information about the history of previous requests needed to determine the behavior of future requests.

static binding. *Binding* where the *method* is selected prior to the actual issuing of the *request*, based on static properties of the client.

subtype. A relation defined over types such that a type a is a subtype of a type b if any value that satisfies type a necessarily satisfies type b.

template. A *type* that can be instantiated to create a new member of the type.

template inheritance. The construction of a template by incremental modification of other templates.

type. An identifiable entity with an associated predicate (a single-argument mathematical function with a boolean result) defined over values. Types are used in signatures to restrict a possible parameter or characterize a possible result.

value. Any entity that can be a legitimate actual parameter in a request. Values that identify objects are called *object names*. Values that identify other entities are called *literals*.

7 References

1. J. D. Brock and W. B. Ackerman. Scenarios: A Model of Non-Determinate Computation. In *Lectures Notes in Computer Science*, Springer, 1981, 252-259.

2. T. Collins, et al. TICKLE: Object-Oriented Description and Composition Services for Software Engineering Environments. To appear in *Proc. 3rd European Software Engineering Conference*, Milan, October 1991.

3. T. Collins, et al. *Tool Composition and Scopes in TICKLE*. Submitted to 1st Conference on the Software Process, Redondo Beach, October 1991.

4. M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

5. D. H. Fishman, et al. Iris: An Object-Oriented Database Management System. *ACM Transactions on Office Information Systems*, 5:1 (Jan. 1987).

6. Hewlett-Packard Company. *HP OpenView: Hewlett-Packard Network Management Architecture*. Internal report, April 1990.

7. Hewlett-Packard Company. *The MIT Documents*. Internal report, Software Engineering Systems Division, June 1990.

8. Hewlett-Packard Company. Sun Microsystems, Inc. *Class Definition Language Specification*. OMG Document 91.1.4.9, March 1991.

9. Hewlett-Packard Company. Sun Microsystems, Inc. *Distributed Object Management Facility Core Specification*. OMG Document 91.1.4.10, March 1991.

10. Hewlett-Packard Company. Sun Microsystems, Inc. *The Object Management Group Object Request Broker RFP Joint Response*. OMG Document 91.1.4.8, March 1991.

11. S. Keene. *Object-Oriented Programming in Common Lisp*. Symbolics Press and Addison-Wesley, 1989.

12. A. Snyder. *An Abstract Object Model for Object-Oriented Systems*. Report HPL-90-22, Hewlett-Packard Laboratories, Palo Alto, California, 1990.

13. A. Snyder. *The Essence of Objects: Common Concepts and Terminology*. Report HPL-91-50, Hewlett-Packard Laboratories, Palo Alto, California, 1991.

14. A. Snyder. *Modeling the C++ Object Model: An Application of an Abstract Object Model*. Report HPL-90-212, Hewlett-Packard Laboratories, Palo Alto, California, 1990. (To be presented at ECOOP-91.)

15. A. Snyder. *An Overview of the OSI Systems Management Computational Model*. Report HPL-91-57, Hewlett-Packard Laboratories, Palo Alto, California, 1991.