An Object Model for Shared Data

Gail E. Kaiser
Columbia University
Department of Computer Science
New York, NY 10027

Brent Hailpern
IBM Research Division
T.J. Watson Research Center
Yorktown Heights, NY 10598

Abstract

The classical object model supports private data within objects and clean interfaces among objects, and by definition does not permit sharing of data among arbitrary objects. This is a problem for certain real-world applications, where the same data logically belongs to multiple objects and may be distributed over multiple nodes on the network. Rather than give up the advantages of encapsulated objects in modeling real-world entities, we propose a new object model that supports shared data in a distributed environment. The key is separating distribution of computation units from information hiding concerns. We introduce our new object model, describe a motivating example from the financial services domain, and then present a new language, PROFIT, based on the model.

Introduction

The classical object model [27] supports private data within objects and clean interfaces among objects. The standard mode of communication between objects is for a client object to send a message to a server object to request some service defined in the server's interface; the client is not aware of the private data hidden within the server and cannot manipulate this data except through side-effects of the server's responses to its messages. This encapsulation makes it impossible for two or more arbitrary objects to transparently and symmetrically *share* data in a tightly-coupled manner. This is reflected in both the compile-time view of objects as information hiding units and their execution-time view as processes. Our goal is to extend the classical object model to support transparent, symmetric, tightly-coupled sharing.

The classical object model permits code and data definitions to be shared via inheritance, but not data values. Data values can be shared symmetrically but non-transparently, by encapsulating the shared data in a third object accessed through message passing [1]. Data can be shared asymmetrically by encapsulating it within one of the objects, which may access it transparently, but the only access available to other objects is through message passing; any sharing is by convention and outside the programming model. It is possible to share data symmetrically and transparently, but only among all instances of a certain class (class variables) or a certain set of classes (pool variables), or all objects (global variables) [11]. We know of one previously proposed object model that does support shared data among arbitrary objects: Self [25] treats all data as potentially shared, transparently and symmetrically, but the data is loosely-coupled. There is no mechanism for compile-time consistency checking of access to any data; instead, all access is through message passing. We drew upon many ideas pioneered in Self during the

development of our object model.

We are concerned with operation in a distributed environment, where the client and server may reside in different processes, and processes may themselves be multi-threaded. (By process, we mean the standard operating system process with its own address space; by thread, we mean a context consisting of registers and a control stack.) Distribution complicates sharing because two objects that share a common subpart may not reside in the same address space, and in these cases apparently direct access must be implemented via message passing. A few distributed object systems (e.g., [26], [22]) support some form of partitioning of subobjects among different nodes, but subobjects cannot be shared.

We propose a new object model that supports data sharing among arbitrary objects in a distributed environment by separating compile-time and execution-time concerns. There are three important components: *facets, objects* and *processes*. Facets are subobjects, the minimal unit of data and control; facets may be shared among multiple objects and may be replicated in multiple processes. Our objects reflect the compile-time aspect of classical objects: each object encapsulates one or more facets and provides an external interface. Our processes reflect the execution-time aspect: each process colocates one or more facets within a single address space and manages a number of *threads*. Objects and processes are orthogonal: objects are not contained in processes nor vice versa.

Our new model is motivated by an important application domain, financial services. Advanced financial services, e.g., "programmed trading", involve: (1) enormous amounts of data; (2) sharing of data among large numbers of users; (3) logical representation of this data as local variables as opposed to entities in an external database; (4) rapidly changing data (e.g., prices); (5) changes to data outside the control of the system (e.g., from the stock exchange wire); and (6) severe economic penalties for making decisions based on obsolete data. These problems have been articulated by other researchers, but not solved (e.g., [20]). In this paper, we are concerned primarily with points 1-3; we have considered points 4-6, but due to space limitations, we mention timing concerns only as needed to explain our decisions regarding other aspects of the design.

Objects are naturally suited for modeling such real-world phenomena, except there is no provision for sharing data among objects. This is the motivation for our extension of the classical object model to support transparent, symmetric, tightly-coupled sharing of data. Transparency is needed since the shared data is logically part of each sharing object. Symmetry is needed to treat multiple users uniformly. Tight coupling is required to guarantee static semantic consistency at compile-time. Tight-coupling is also important at run-time, since our rapidly changing data is similar to the real-time data of manufacturing and telecommunications: the data changes when it changes, and cannot be blocked until

convenient. However, financial applications are more like telecommunications than manufacturing, since operation can degrade gracefully (to a point) as changes are sometimes missed and changes arise on the order of seconds rather than microseconds.

Financial services is also one of the primary motivating applications for database management systems, which support shared data (the database) and separate compile-time (data definition) from execution-time (data access). But conventional databases do not support encapsulation within objects, or even a clear notion of "object", while object-oriented databases [9] do not support sharing among objects. Neither makes execution-time issues such as threads of control and data placement among nodes explicit in the programming model, but buries concurrency and distribution in the underlying database manager. Our object model addresses these problems, although it does not treat persistence or queries. Our object model is not specific to financial services, but is suitable for other applications, such as intelligent network management [18], weather modeling and animation [12], with similar requirements.

We start by discussing an extended financial services example in general terms to motivate our new object model. Then we give an overview of a new programming language, PROFIT (PROgrammed FInancial Trading), based on our model. PROFIT is an extension of C, and most statements and declarations will be written in C. We then present PROFIT's facets, objects and processes. Discussion of timing constraints is outside the scope of this paper. We briefly compare to related work, and summarize our contributions. A subset of PROFIT has been implemented in a pilot study.

Example Portfolio Management System

Consider a financial market, with both stocks and options, collectively called instruments. Our example system manages portfolios made up of combinations of such instruments. For the sake of the example, we assume there are only three companies, *Institutional Books and Materials, Domestic Educational Corporation* and *Supplies, Umbrellas and Novelties*, abbreviated INS, DOM and SUP. Options on their stocks are available with various expiration dates. The system, called *Stock Environment Calculator* (SEC), monitors the current prices of the stocks and options and executes the appropriate purchases and sales (according to certain constraints associated with the particular portfolio by a financial analyst) as market conditions change. We describe three aspects of SEC: consistent access to the current prices of a stock and its options (the 9am stock price along with the 9:10am option price could result in disastrous financial strategies); easy programming of the objects that track the changing prices to determine when action must be taken; and transparent sharing of the prices by all the users of the system.
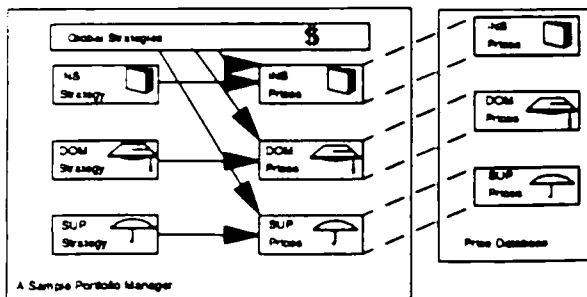
An individual portfolio consists of a set of instruments, their current prices and a set of strategies for when to buy and sell. Each portfolio is organized into subparts, where each part represents the instruments for a particular company, together with any aspect of the strategy specific to that company (see Figure 1). For example, if the price of a stock increased 10% since purchase, buy more. Additional subparts support strategies that cut across companies and maintain any other data needed. A subpart representing the instruments of a particular company contains the current prices of the stock and options. Multiple portfolios will refer to the same company with independent criteria for when price changes are significant to the financial analysts's strategies.

The difficult problem is how to notify the computations reflecting the strategies of these portfolios when the prices of the stocks and options change. There are three ways to structure the solution. The *active value* approach propagates each change to every interested portfolio. *Polling* requires every interested portfolio to poll the current value. The third approach uses *daemons*. We compare these three approaches at an abstract level, and describe PROFIT's implementation of the daemon approach in the remainder of the paper.

In the active value approach (e.g., [23]), any change to any value can be propagated to other parts of the program. The propagation invokes code associated in advance with the data and the kind of change. An exemplary application is changing the speed of a simulated car, resulting in an updated display of the speedometer reading and the consumption rate of fuel. In Figure 2, the active value approach combines the shared data (D) and whatever computation is necessary to monitor (M) the changes to the data in the same object. The monitoring is not a separate thread of control, but instead a side-effect of the procedures that update the data. The monitor code has the responsibility to notify all other interested computations (C) of changes.

The active value approach has a significant flaw with respect to this application: since price changes are frequent and typically small (1/4 point), some changes may be insignificant from the point of view of some portfolios. Thus the system can be flooded by many notification messages to which few portfolios are paying attention. One solution is for the monitor code to know the separate criteria for each interested portfolio regarding what changes are considered important. This would add a significant computational component to the active value and greatly complicate the programming of the monitor code.
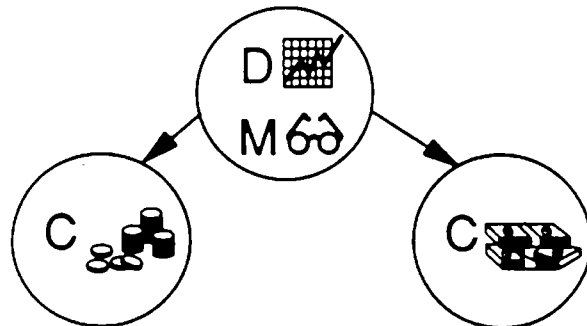

Figure 1: Example Portfolio
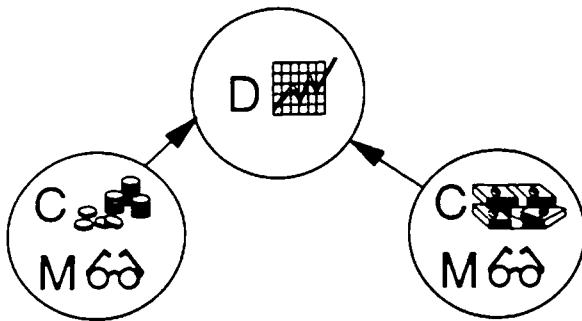

Figure 2: Active Value Approach

Figure 3: Polling Approach

Polling is the traditional means for implementing device drivers, and can be implemented in any imperative programming language. The idea is to check over and over again whether a data item has changed. Figure 3 illustrates this approach: the shared data (D) is passive, and the interested computation objects (C) directly incorporate the monitoring (M) of changes to the data. The computation code must include explicit statements to check whether the shared data has been changed in a manner considered significant by the particular computation.

Polling overcomes the problems with the active value approach, since each portfolio can decide how often to check each price and under what criteria to take action. Unfortunately, a naive implementation — consisting of tedious busy-wait loops — obscures the logic of the main portfolio program. This is not a serious impediment when polling only a single price, but is complicated when the latest prices of multiple instruments must be considered both individually and in combination (Figure 4).
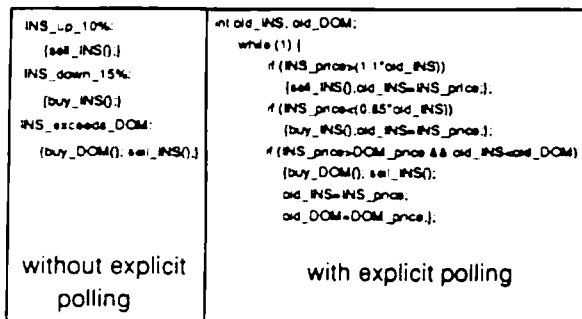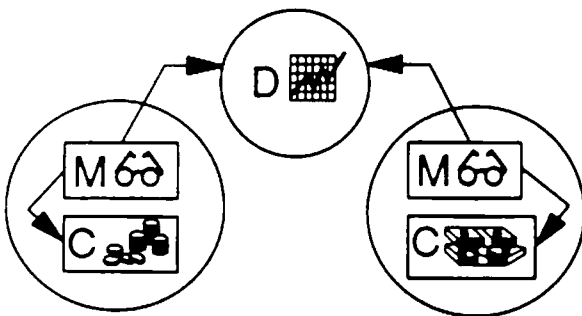

Figure 4: Complexity of Polling

Figure 5: Daemon Approach

We prefer a more sophisticated solution in which the shared prices are monitored by simple daemons [2]. Each daemon contains a trigger that informs the main-line strategy when something "interesting" happens. Active values and polling both employ two objects, one representing the prices and the other the portfolio manager. The daemon approach uses three objects, adding an intermediate object that polls the prices and behaves as an active value with respect to the portfolio manager. The advantages include a simpler programming model and the capability for each portfolio to decide how best to use its computational resources. Figure 5 shows the three objects: passive shared data (D) as in polling, the strategic computation (C) as in the active value solution, and a separate daemon (M).

Overview of PROFIT

We have designed our programming model to provide the appropriate building blocks to easily represent the daemon solution as well as active values and polling. Our contribution is an explicit programming model for *shared data* in a concurrent object system. This is reflected in our design of the PROFIT programming language. There are three main concepts:

- *Facet*, the minimal unit of data and control, in particular, the unit of shared data. A facet consists of a number of named slots, each of which may contain either a data value or procedure code. A facet may execute a single thread of control at any one time.

- *Object*, a statically defined collection of facets representing an information hiding unit. An object defines a context for binding references between facets in the same object and an external interface for passing messages to and from other objects.

- *Process*, a statically defined collection of facets — orthogonal to objects — that must execute at the same physical location. That is, a process represents a single virtual address space. Creation and scheduling of threads, employing single or multiple processors, is handled by processes.
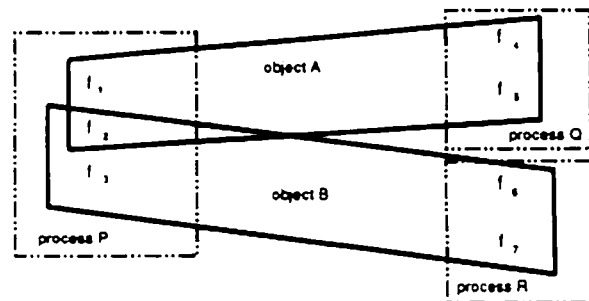

Figure 6: Facets, Objects and Processes

Every facet is a member of one or more objects and one or more processes. In the simplest case, there is exactly one process in the program, and all objects execute on the same machine within the same virtual address space. When there are multiple processes, different facets of the same object may reside in different processes.

Processes are units of separate compilation, while objects may be treated as declarations included during compilation as needed or as source code also compiled separately. Objects and facets can be written independently of processes, and later configured into a system by defining processes; any configuration will provide a logically correct program, although one may be more efficient than another on a given concurrent architecture.

We posit one SEC process containing all the prices, and one additional process for each individual user of the SEC. Each user would define one object corresponding to each of his portfolios, consisting of computation and daemon facets and the relevant previously defined prices facets. The computation and daemon facets would live in the user's process, while the prices facets would be updated only in the SEC process but replicated and thus read in the user processes. The analogy to traditional database servers is not accidental. This relationship between facets, objects and processes is illustrated in Figure 6.

A program specifies the objects and processes that together make up a single application, the physical locations of the processes at execution-time and the initialization code to start the application running. For the purposes of this paper, we assume that facets, objects, processes and programs are all defined statically, so it is not possible to add new components while a program is executing. In the more general case, however, it would be necessary to be able to add user-defined objects (portfolios) and processes to an already-executing system.

PROFIT's facets and objects provide abstractions for programming each of the three approaches to our example above:

- In the active values approach (Figure 2), a prices facet is shared among several objects. This facet must provide the data, change monitoring and notification of interested objects. Each of these objects also contains a non-shared facet that receives the notification and carries out the appropriate financial strategy.

- In the polling approach (Figure 3), the prices facet contains only shared data. Each interested object includes a non-shared facet that carries out both computation and change monitoring. Although this permits the programmer to set the priority of monitoring change, that is, the time interval between polls, it unnecessarily complicates the overall computation by mixing polling activities and the control associated with the main-line strategy.

- In the daemon-based approach (Figure 5), the prices facet contains only shared data. Each interested object includes two non-shared facets, one that carries out the computation and the other that monitors changes. The latter is devoted to polling the shared facet, performing only that computation necessary to determine which changes are interesting to its object, and notifying the strategy facet accordingly.

Processes support the run-time behavior of our example: the execution of a thread within a particular facet makes it possible for the programmer to easily control the rapidity with which change is monitored and acted upon. That is, all relevant timing constraints are expressed directly by the procedures provided within a facet, so each daemon facet can poll/notify at the time intervals appropriate for the portfolio management object(s) containing it. It is the responsibility of the enclosing process to schedule execution of facet threads. Only one replica of a facet can be executing a thread at a time in the general case, requiring significant synchronization overhead, which we ignore in this paper. However, we expect most shared facets will contain only data slots (e.g., shared prices) and no procedure slots, so

replication could be optimized to allow multiple reading threads with no contention and no synchronization. Alternatively, prices facets might not be replicated, in which case the daemon facet would have to poll via interprocess communication, perhaps across a network.

PROFIT is more a language extension than an entirely new language, in that it does not define the details of the base language, in this case the data and procedures that may appear in slots. We intend these to be written in conventional programming language(s); for now we assume C. Thus data slots contain C data values and procedure slots contain C functions. Macros and subroutines will be provided for evaluating slots within the same facet, handling indirection to other facets and objects, referring to any facets and objects returned by these evaluations, interacting with threads, and so on. One issue is whether multiple data slots, of the same or different facets, can point to the same data structure. This is a problem since different facets might reside in different processes, so direct sharing is not always possible. One solution might be distributed virtual memory [15]. We follow a simpler approach: no sharing of data structures, only entire facets (i.e., there is no pointer aliasing). Data structures are copied when transmitted as arguments, whether within the same process or across process boundaries.

Facets

The facet is the minimal unit of data and control. A facet has a unique name and a set of named slots, each of which may contain either a data value or procedure code. Slots are typed, either the type of the data (a C datatype) or the return value of the procedure (a C datatype or void). Procedure slots must be equated to specific C functions at compile-time. Evaluating a data slot returns the value, while evaluating a procedure slot executes the procedure (with the parameters provided) and returns the result of the execution, if any. (Facets correspond closely to Self objects.)

For example, the set of prices for the instruments of the INS company would be represented as a facet, called INS-instruments (Figure 7). The only operations are (implicit) get and put. In this example, there is one possible writer — some agent external to SEC representing the stock exchange wire — and multiple readers from different portfolios. Another example would be the daemon that monitors the changing prices of DOM's instruments (Figure 8). The daemon would keep certain local data such as high and low trigger values, used when deciding whether to notify the corresponding portfolio manager.

INS-instruments	
stock-price	$114.75
1Q-option-price	($115,$.50)
2Q-option-price	($115,$4.75)

Figure 7: INS-instruments Facet

DOM-daemon

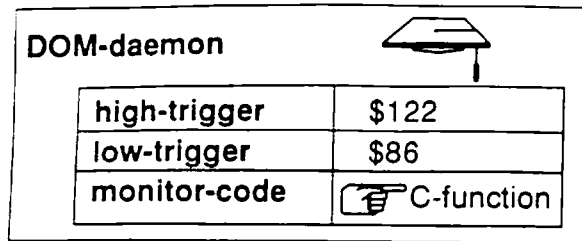| high-trigger	$122
low-trigger	$86
monitor-code	☞C-function

Figure 8: DOM-daemon Monitor

There is a distinguished slot within each facet, called active, that represents the currently executing thread of control. There can be at most one thread executing within a facet. Among other things, the thread indicates its originating object (see next section). If the active slot is null, then the facet is not doing anything. A prices facet, for example INS-instruments, would normally be passive except while get or put is running. The get operation must be able to return multiple prices from the INS-instruments facet to ensure consistent access, while the put operation could be restricted to a single value since price changes are independent. A portfolio management strategy, say involving statistics specific to the INS instruments, would also normally be passive, until the daemon wakes it up after a significant change.

A facet represents a closed scope, meaning every use of an identifier matches an identifier defined within the facet. There are no free variables. Of course, each procedure in a procedure slot of a facet is also a scope, with its own local variables. The facet's other procedure and data slots are global to the procedure, that is, the facet is treated as the procedure's enclosing scope and the procedure can call the other procedure slots and access the data slots. In order to support references between facets, a slot may be declared *indirect*, as in Figure 9. The containing object is then obliged to provide a *binding*, to a slot in some other facet within the same object or to an entry in the interface of another object; see next section. Every object has a *binding table* for this purpose. When code references an indirect slot, then the semantics are to refer to the current object's binding table to resolve the reference. Thus, a procedure in one facet can call procedures or access data in other facets via the corresponding indirect slot in its own facet.

This approach is based on delegation [16], where when one object cannot handle a message, it defers it to another. We modify traditional delegation by binding at the enclosing object level rather than separately for each individual facet.* This means a facet can be written without knowledge of which specific other facets it will delegate to (see Figure 10).

DOM-daemon

high-trigger	$122	
low-trigger	$86	
monitor-code	☞C-function	
strategy-wake	ⓘ	indirect
stock-price	ⓘ	indirect
1Q-option-price	ⓘ	indirect
2Q-option-price	ⓘ	indirect
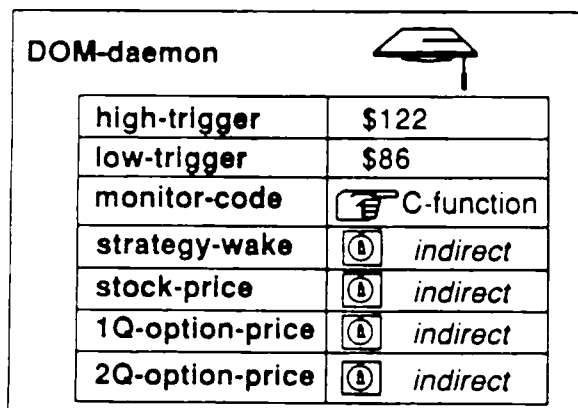
Figure 9: Indirect Slots
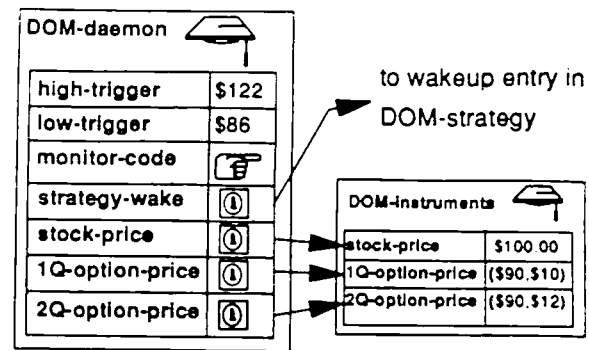
Figure 10: Binding

Objects
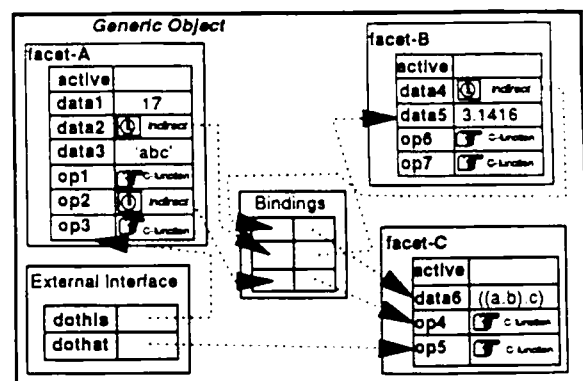
Figure 11: Generic Object

Binding Table	
(facet-A,data2)	(facet-B,data5)
(facet-A,op2)	(facet-C,op4)
(facet-B,data4)	(facet-C,data6)
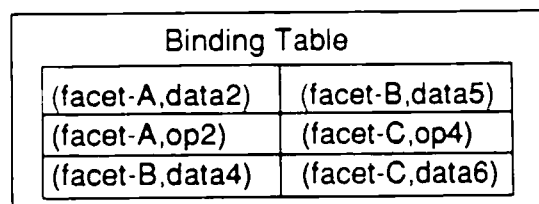
Figure 12: Generic Binding Table

As in the classical object model, a PROFIT object is the compile-time unit of information hiding. It defines an external interface and encapsulates its internal data and procedures. The interface defines the object's unique name and the set of entries (procedures) visible to other objects. PROFIT objects are different in that the internal data and procedures are supplied by a set of facets with bindings between the facets. A generic object is shown in Figure 11. The binding table maps each indirect slot of every facet within the object, either to a slot of a facet in the same object or to an entry in the interface of another object. Figure 12 shows how several facets may be bound together within an object, and Figure 13 the bindings for the SUP portfolio manager object.

---

*Further, we fully delegate to the receiving facet's own context (its own slots) rather than evaluating a slot from the receiving facet as if it were a slot in the delegating facet, i.e., there is no "self pointer". This is relevant if the evaluated slot is a procedure that references other slots.
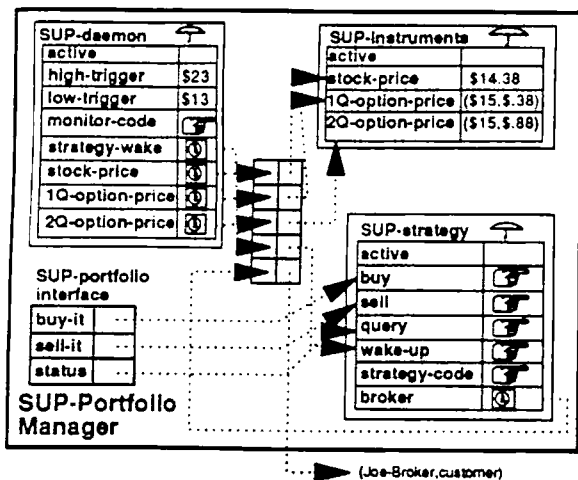
Figure 13: SUP Portfolio Manager Object

When a procedure is executing within a facet, it may directly access only those slots defined in the same facet. Accesses to indirect slots are resolved through the binding table. First we describe calls within a facet, and then calls from one facet to another. A call within a facet is treated like a conventional procedure call. The actual parameters and caller's return point are pushed on the stack maintained by the current thread (i.e., each thread has a separate stack), and control is transferred to the called procedure. Calling a procedure slot means calling a C function. The local variables of the C functions are stored on the stack. All external references made by the C function must appear as slots in the facet. When the procedure returns, any result is left on the stack, and control is transferred back to the calling procedure.

For calls between facets, we consider first the callee and then the caller. If a call arrives while the callee facet is already active, the call is queued. When a call reaches the front of its queue, the facet accepts the call and sets its active slot and binding table according to the calling object. Subsequent indirection is with respect to this binding table. When the call completes, the response is sent to the caller and the facet goes on to the next queued call. The caller initiates a call by pushing the actual parameters and return point on its stack. The caller facet is then released, and can accept a new call. When the call returns, it simply adds itself to the end of the original caller's queue. Note that the caller is not suspended, but may continue by accepting the next call in its queue. When the call returns and reaches the front of the queue, the caller continues execution of that thread at the point where it left off.
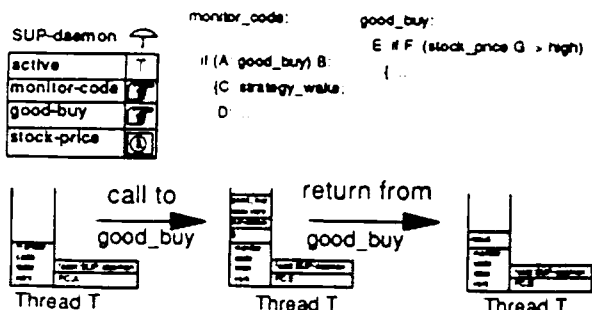
Figure 14: SUP-daemon Call Within Facet

These discussions of both caller and callee viewpoints are equally valid for indirection to another facet in the same object or to an entry in the interface of another object (and ultimately a facet in this other object). However, when a call is made across processes, the calling thread must be suspended at the process level (i.e., the calling facet is not suspended) and a new stand-in thread created in the called process.[**] On return, the suspended thread is resumed by the process. In the next section we discuss the details of implementing these synchronous calls, and also describe mechanisms that permit asynchronous calls.

Consider the following example: The SUP-daemon periodically polls the SUP-instruments prices, to compare to its own trigger values. If the criteria are met, the daemon notifies the SUP-strategy computation. The SUP-daemon's call to evaluate a data slot within its own facet is shown in Figure 14, while the SUP-daemon call to get data from the SUP-instruments facet is depicted in Figure 15. The asynchronous call needed for the SUP-daemon to notify SUP-strategy is discussed later.

When a facet is shared among multiple objects, each of these objects provides a different binding table that must resolve all the shared facet's indirect slots (Figure 16). When a facet is active, only one binding is actually used, the one belonging to the object from which the facet was invoked. Since a shared facet may be invoked from another shared facet, it is necessary for the appropriate binding table (i.e., its pointer or index) to be passed as an implicit parameter.

Communication between objects is a simple extension of the communication between facets. When a call is received at the interface of an object, the object maps the call to a procedure slot of one of its member facets. When the call returns, the object sends the result back to the calling object. Because objects can communicate with many other objects, we associate a queue with each object's external interface. As soon as a call in this queue has been mapped to a particular facet, it is moved from the object's queue to the facet's queue, and the object goes on to resolve the next external call.

(1) call to stock-price

(2) Thread T is queued

(3) T dequeued, value fetched

(4) call returns, T queued

Figure 15: SUP-daemon Call to SUP-instruments

Processes

The classical object model does not specify a particular role for processes. There is a spectrum of relationships between objects and processes reflected in existing concurrent object-based systems ([7, 24, 8, 4]). A PROFIT process represents a single address space and contains one or more threads of control. Each facet may be replicated in multiple processes, and any threads executing in a replica are managed by its containing process. The system support for processes is also responsible for maintaining consistency among all the replicas of the same facet [10]; we do not address how replication is implemented in this paper, but note only that symmetric read/write replication is required. Processes thus represent the execution-time organization of facets. In contrast, objects represent the compile-time organization of facets: Objects do not "live" anywhere, and facets of the same object may reside in different pr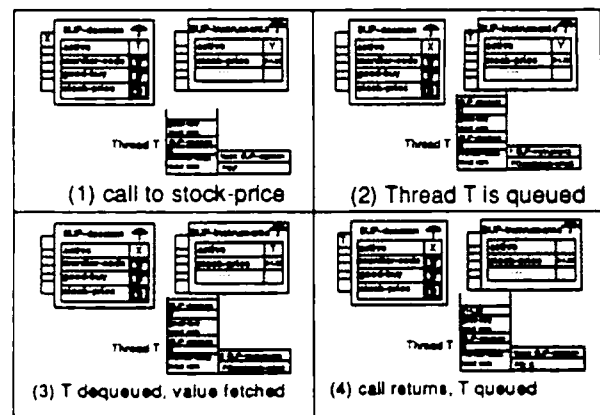ocesses. The external interface and binding table of every object are replicated in every process, with no run-time overhead since this information is determined at compile-time and is not modifiable.

There are also many notions of thread, running a spectrum from light-weight as in Mach's raw threads [21] to medium-weight as in Mach's C-thread package [6] to heavy-weight as in a full Unix process [2]. The common concept is that a thread has a single locus of control. What distinguishes the variety of threads is the ownership of data (address space) and context (registers, control stack). We have in mind medium-weight threads. Along with a simple locus of control, a PROFIT thread maintains context between nested calls (evaluations of procedure slots), thereby preempting recursion. Alternatively, light-weight threads without their own control stacks would preclude recursion or dynamic variables; heavy-weight threads on a per-facet basis, resulting in one facet per process, would likely lead to terrible performance.

In the previous section, we discussed the notion of queuing calls for facets and for objects. Each call in a facet queue is represented by a thread, which provides the context of the call. Since objects have no physical representation, object queues are implemented as separate queues in each process, and processes take over the object's run-time role in mapping external calls. First we discuss the manipulation of threads in facet queues, and then the creation of threads in response to calls (messages) queued for processes. If a call arrives while the facet is already active, the thread associated with the call is queued. When a thread reaches the front, the facet call sets its active slot to reference the dequeued thread; when the call
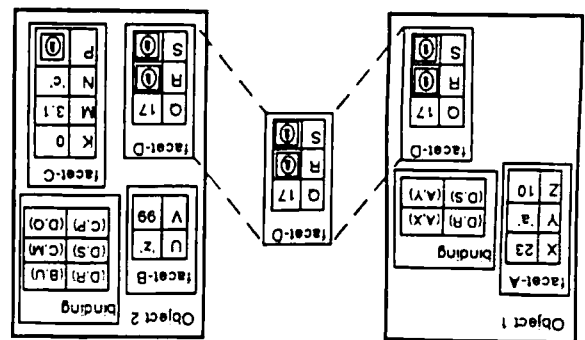

Figure 16: Shared Facet

complete, the thread is queued for the caller. This works only among facets within the same process, where enqueuing and dequeuing of threads is managed by pointer operations. When calls are made across process boundaries, the calling thread must be suspended and a stand-in thread created in the remote process.

There are two situations in which processes create new threads: when a call is made across process boundaries and when an asynchronous call is made. When a facet calls another facet that resides in another process, the calling thread is suspended at the process level. The calling facet accepts the next thread in its queue. Meanwhile, the process sends a message to the other process, indicating (facet, slot), parameters, return point, and some identification of the calling object, needed to select the appropriate binding table. The message is appended to the receiving process's queue. The receiving process removes the first message in its queue. The process creates a corresponding thread and queues the thread for the indicated facet. When the call is completed, the process sends a message back to the sending process's queue. The sending process matches the message with the suspended calling thread, and queues it for the calling facet.

Asynchronous calls work the same as synchronous calls as explained above — with two differences. The first is that the calling facet does not release its thread, but instead continues execution. The second is that there is no return point, when an asynchronous call completes. Note that this does not permit the calling facet to wait for the completion of the asynchronous call (or, more generally, wait for a particular asynchronous call from another facet).

This lack of blocking uncovers an apparent flaw in the PROFIT design: there is no way to make an *atomic* call, except for the trivial cases of gets, puts and procedures whose calls (direct and transitive) happen to be synchronous calls to other procedures within the same facet. When a facet makes a synchronous call to another facet, it relinquishes its thread and serves the next thread in its queue. This context switching may result in arbitrary changes to its data slots. In those cases where it is necessary for certain data items to maintain their current values during a synchronous call to another facet, this data must be stored in the local variables of the calling procedure (C function) — and thus in the thread's stack, where there is no danger of being overwritten by unrelated threads queued for the same facet. This lack of atomicity is intentional, since atomicity directly conflicts with the need for rapid reaction to change: an "atomic" call that can be interrupted (and not rolled back), say to meet timing constraints, is of course not really atomic.

Related Work

Among existing languages, PROFIT is probably closest to Argus [17]. PROFIT facets are similar to Argus objects. If PROFIT objects and processes were not distinct, but instead every object was also a process, the result would closely match Argus guardians. Argus's lightweight processes are analogous to PROFIT's threads. Hermes [3] is another similar language that provides both objects and processes. In Hermes, the process defines the external interface, in terms of typed ports; objects are encapsulated inside processes. Hermes objects are thus like PROFIT facets except they cannot be shared or replicated, although they can be passed by value as arguments across ports.

***An important optimization is to return the thread to a pool for later reuse, rather than deallocating it.

It is also interesting to compare PROFIT to Linda [5], which defines a paradigm for parallel programming quite different from Argus or Hermes. Communication among processes in Linda is via tuples in a global tuple space. Facets can be viewed as the static analog of tuples. On the other hand, facets may be treated as the dynamic analog of Flavors's mixins [19]. Mixins provide data and procedure slots that may be inherited by arbitrary objects. Such inheritance is concerned with structure, however, not contents. In particular, each inheriting object may have different values in the data slots inherited from the same mixin. Facets, in contrast, provide direct sharing of data slot values as well as definitions, in the style of Self. The most significant difference between PROFIT and Self is that PROFIT supports concurrency, both multiple threads and multiple processes, while Self is a purely sequential language. Another important difference is that PROFIT data and procedure slots are provided in C, allowing access to existing application code — which is obligatory for the financial applications envisioned, while Self is a uniform language.

Conclusions

The primary contribution of this paper is our new object model for shared data (facets) based on the separation of compile-time information hiding (objects) and execution-time computation concerns (processes). We have demonstrated by our portfolio manager example a methodology for using our new object model in a practical application domain with specialized requirements.

Acknowledgments

References

[1]   Gul Agha.
      Actors: A Model of Concurrent Computation in Distributed Systems.
      The MIT Press, Cambridge MA, 1986.

[2]   Maurice J. Bach.
      The Design of the UNIX Operating System.
      Prentice-Hall, Englewood Cliffs NJ, 1986.

[3]   David F. Bacon and Robert E. Strom.
      Implementing the Hermes Process Model.
      Technical Report RC 14518, IBM T.J. Watson Research Center,
         March, 1989.

[4]   Andrew Black, Norman Hutchinson, Eril Jul and Henry Levy.
      Object Structure in the Emerald System.
      In *OOPSLA '86*, pages 78-86. September, 1986.

[5]   Nicholas Carriero and David Gelernter.
      Linda in Context.
      Communications of the ACM 32(4):444-458, April, 1989.

[6]   Eric C. Cooper and Richard P. Draves.
      C Threads.
      Technical Report CMU-CS-88-154, CMU Department of Computer
         Science, June, 1988.

[7]   Partha Dasgupta, Richard J. Leblanc Jr. and William F. Appelbe.
      The Clouds Distributed Operating System: Functional Description,
         Implementation Details and Related Work.
      In *8th ICDCS*, pages 2-9. June, 1988.

[8]   David Detlefs, Maurice Herlihy and Jeannette Wing.
      Inheritance of Synchronization and Recovery Properties in
         Avalon/C++.
      Computer 21(12):57-69, December, 1988.

[9]   Klaus Dittrich and Umeshwar Dayal (eds).
      1986 International Workshop on Object-Oriented Database Systems.
      September, 1986

[10] David K. Gifford.
      Weighted Voting for Replicated Data.
      In *7th SOSP*, pages 150-162. December, 1979.

[11] Adele Goldberg and David Robson.
      Smalltalk-80 The Language and its Implementation.
      Addison-Wesley, Reading MA, 1983.

[12] Paul E. Haeberli.
      ConMan: A Visual Programming Language for Interactive
         Graphics.
      In *SIGGRAPH '88*, pages 103-111. August, 1988.

[13] Gail E. Kaiser, Steven S. Popovich, Wenwey Hseush and Shyhtsun
         Felix Wu.
      Melding Multiple Granularities of Parallelism.
      In *3rd ECOOP*, pages 147-166. July, 1989.

[14] Catherine Lassez and Tien Huynh.
      An Expert Decision-Support System for Option-Based Investments
         Strategies.
      Journal of Computers and Mathematics with Applications, 1990.
      In press.

[15] Kai Li and Paul Hudak.
      Memory Coherence in Shared Virtual Memory Systems.
      In *5th PODC*, pages 229-239. August, 1986.

[16] Henry Lieberman.
      Using Prototypical Objects to Implement Shared Behavior in Object
         Oriented Systems.
      In *OOPSLA '86*, pages 214-223. September, 1986.

[17] Barbara Liskov.
      Distributed Programming in Argus.
      Communications of the ACM 31(3):300-312, March, 1988.

[18] Subrata Mazumdar and Aurel A. Lazar.
      Knowledge-Based Monitoring of Integrated Networks.
      In *IFIP TC 6/WG 6.6 Symposium on Integrated Network
         Management*, pages 235-243. May, 1989.

[19] David A. Moon.
      Object-Oriented Programming with Flavors.
      In *OOPSLA '86*, pages 1-8. September, 1986.

[20] Peter Peinl, Andrea Reuter and Harald Sammer.
      High Contention in a Stock Trading Database: A Case Study.
      In *SIGMOD*, pages 260-268. June, 1988.

[21] Richard Rashid, Avadis Tevanian, Michael Young, David Golub,
         Robert Baron, David Black, William Bolosky and Jonathan Chew.
      Machine-Independent Virtual Memory Management for Paged
         Uniprocessor and Multiprocessor Architectures.
      In *ASPLOS II*, pages 31-39. October, 1987.

[22] Marc Shapiro, Philippe Gautron and Laurence Mosseri.
      Persistence and Migration for C++ Objects.
      In *3rd ECOOP*, pages 191-204. July, 1989.

[23]   Mark J. Stefik, Daniel G. Bobrow and Kenneth M. Kahn.
         Integrating Access-Oriented Programming into a Multiparadigm
             Environment.
         IEEE Software 3(1):11-18, January, 1986.

[24]   Robert Strom and Nagui Halim.
         A New Programming Methodology for Long-Lived Software
             Systems.
         IBM Journal of Research and Development 28(1), January, 1984.

[25]   David Ungar and Randall B. Smith.
         Self: The Power of Simplicity.
         In *OOPSLA '87*, pages 227-242. October, 1987.

[26]   Horst F. Wedde, Bogden Korel, Willie G. Brown and Shengdong
         Chen.
         *Transparent Distributed Object Management Under Completely
             Decentralized Control.*
         Technical Report, Wayne State University, 1989.

[27]   Peter Wegner.
         Dimensions of Object-Based Language Design.
         In *OOPSLA '87*, pages 168-182. October, 1987.

1990 International Conference on Computer Languages

March 12-15, 1990   New Orleans, Louisiana

IEEE Computer Society Press
Los Alamitos, California

Washington   ●   Brussels   ●   Tokyo