An Architecture for Dynamic Reconfiguration in a Distributed Object-Based Programming Language

Brent Hailpern
IBM
Old Orchard Road
Armonk, NY 10504
914-765-6481
bth@watson.ibm.com

Gail E. Kaiser
Columbia University
Department of Computer Science
New York, NY 10027
212-854-3856
kaiser@cs.columbia.edu

CUCS-029-91
20 September 1991

Abstract

Distributed applications ideally allow reconfiguration while the application is running, but changes are usually limited to adding new client and server processes and changing the bindings among such processes. In some application domains, such as real-time financial services, it is necessary to support finer grained reconfiguration at the level of entities smaller than processes, but for performance reasons it is desirable to avoid conventional approaches that require dynamic storage allocation. We present a scheme for special cases of fine-grained dynamic reconfiguration sufficient for our application domain and show how it can be used for practical changes. We introduce new language concepts to apply this scheme in the context of an object-based programming language that supports shared data in a distributed environment.

Part of this work was completed at the IBM T.J. Watson Research Center while Dr. Hailpern was a Research Staff Member and Prof. Kaiser was an Academic Visitor. Kaiser is supported by National Science Foundation grants CCR-9000930 and CCR-8858029, by grants from AT&T, BNR, DEC and SRA, by the New York State Center for Advanced Technology in Computers and Information Systems and by the NSF Engineering Research Center for Telecommunications Research.

An extended abstract of this paper titled "Dynamic Reconfiguration in an Object-Based Programming Language with Distributed Shared Data" appeared in the **Eleventh International Conference on Distributed Computing Systems**, Arlington TX, May 1991, pp. 73-80.

keywords: abstract types, distributed system, dynamic object update, portfolio management, soft real-time processing

1. Introduction

This research is motivated by the problem of *rapidly changing data* in a distributed environment, particularly in the financial services domain. For example, on-line stock trading involves: (1) enormous amounts of data (stocks and options); (2) sharing of data among large numbers of simultaneous users (financial analysts); (3) rapidly changing data (as prices of financial instruments fluctuate); (4) changes to data outside the control of the system (from the stock exchange wire); and (5) economic penalties for making decisions based on obsolete data (say, data from before the most recent large transaction in a particular stock). These problems have been articulated by other researchers (e.g., [Peinl 88]), but not solved.

An on-line stock trading program might consist of a shared ‘‘prices database’’ and a number of analyst workstations that execute portfolio management programs. These portfolio managers would monitor the current prices of the stocks and options, and execute the appropriate purchases and sales — as market conditions change — according to certain rules and constraints associated with the particular portfolio by a financial analyst. A key challenge in such a program is that the prices of the various stocks and options change rapidly, perhaps several times a minute, and to a first approximation independently from each other and from the actions taken by any individual portfolio manager. Multiple portfolios will refer to the same instruments, and have separate criteria for when price changes are significant to the financial analysts’ investment strategies. Thus, different portfolios may require information about price changes at different time intervals and/or different granularities of change. What we have in mind is essentially soft real-time processing, where it is not mandatory for every portfolio to be informed of every possibly trivial price change, but where the quality of service is balanced against the computation and communication costs of providing that service.

In a previous paper [Kaiser 90], we introduced a distributed object-based programming model that addresses these problems. This programming model supports an application architecture where price changes are monitored by daemons operating on behalf of individual portfolio managers. The sampling rate of each daemon is specific to the requirements of its portfolio manager, and it notifies its manager of changes at the granularity considered interesting by the manager’s financial strategy. This approach falls in the middle of the spectrum from polling to active values, and combines the advantages of both extremes. Our programming model also supports other architectures on this spectrum, and is not specific to financial services. It is suitable for other applications, such as network management [Mazumdar 89], machine vision [Boult 90] and animation [Haeberli 88], with similar characteristics.

We have developed a language, called PROFIT (PROgrammed FInancial Trading), based on this model. PROFIT is a coordination language [Ciancarini 90] that extends the declarations and statements of some base computation language, such as C, with additional facilities to support distributed computation in the context of rapidly changing shared data. In particular, PROFIT adds *facets* as the minimal unit of data and control, *objects* as collections of facets encapsulated for the purpose of information hiding, and *processes* as collections of facets organizing the run-time structure of the program. Different facets of the same object may reside in different processes, and a facet may be shared among multiple objects although it resides in exactly one process.

An archetypical program includes one facet representing each of the financial instruments available, with these executing in one or more processes as the ‘‘prices database’’. Each of the several portfolios would be represented by an object that includes some subset of the shared price facets, plus additional private facets for monitoring changes to prices and computing financial strategies. Since an object may be distributed among several processes, a portfolio object may include price facets located in a process that resides on the price server and portfolio management facets in a process on an analyst’s workstation. Daemon facets that monitor changes in the market might be located on either machine, reflecting different computation and communication costs tradeoffs.

PROFIT has been designed for early binding of facets into objects and facets into processes, in order to avoid the overhead and error-prone nature of dynamic storage management. Sharing of facets among objects is permitted by indirection tables, since each object provides a different execution context for its facets. Unfortunately, early binding inhibits flexibility in critical ways. For example, in the extreme it does not allow adding instruments, adding portfolios, or changing the composition of portfolios over time. To change anything, it would be necessary to recompile the program and reinitialize, certainly disruptive for an on-line program.

In this paper, we propose to solve this problem by relaxation of the early binding paradigm, to support special cases of late binding. We refer to these cases collectively as *dynamic reconfiguration*. Dynamic reconfiguration gains the flexibility to implement the specific rebindings required for our application domain without abandoning the benefits of early binding. This approach has long been used for operating systems and distributed services, for example, for adding new clients and servers, and changing connections between clients and servers. To our knowledge, dynamic reconfiguration has not previously been supported under program control for changes at a finer granularity than that of an entire process.

To support dynamic reconfiguration, we add three new concepts to PROFIT: breeds, stalls and pens (the ranching metaphor is explained later). Breeds are similar to types in that they define a set of facilities required of those facets that can be substituted for each other in a particular context. Stalls and pens are similar to variables and sets, respectively: stalls ‘‘hold’’ a single facet and pens ‘‘hold’’ a collection of facets, and in both cases which facets are held can be changed during program execution. These new concepts give the PROFIT programmer the ability to change the group of facets operated on by a computation and substitute among a number of facets that all provide a common set of facilities. In this new version of PROFIT, it is possible to change the static organization of an executing program, but without going so far as to support dynamic memory management. Throughout the rest of this paper, we refer to our original design of PROFIT as PROFIT_0 and the extended PROFIT introduced in this paper as simply PROFIT.

We start by summarizing the PROFIT_0 design. Then we elaborate on the dynamic reconfiguration problem. In the subsequent sections, we explain breeds, stalls and pens. We then introduce registries to aid in locating new objects. Finally we describe the ranchhouse as the mechanism for adding new facets, objects, and processes to a running program. We briefly describe the implementation of PROFIT_0 and discuss related work.

2. Background

The PROFIT_0 programming model supports data sharing among objects in a distributed environment. There are three important components:

- *Facet*, the minimal unit of data and control. Although facets may be shared among multiple objects, only one operation at a time may execute within a facet.
- *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 encapsulating the facets.
- *Process*, a statically defined collection of facets that must execute on the same host. That is, a process represents a single virtual address space in which its facets reside, and allocation of computation and communication resources to facets is handled by their process. Multiple facets may execute concurrently within the same process.

Every facet is contained in one or more objects and exactly one process. Objects and processes are orthogonal: objects are not contained in processes nor vice versa. This organization is illustrated in Figure 2-1.

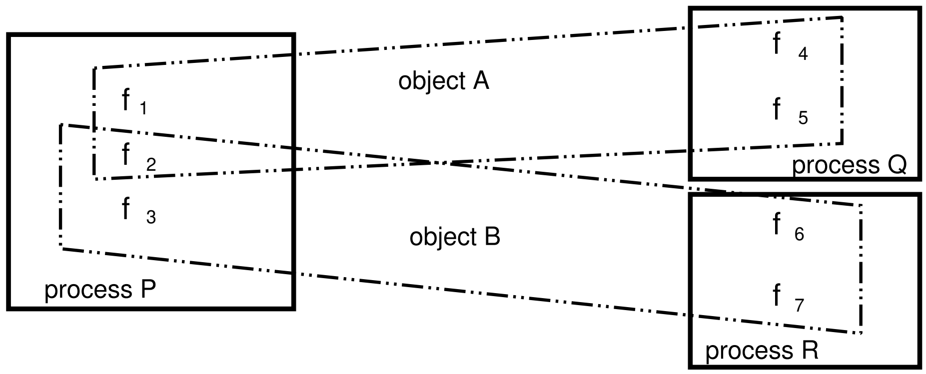

Figure 2-1: Facets, Objects and Processes

This programming model provides a set of building blocks for constructing distributed applications with rapidly changing data. Now the question arises as to what the best way is to present the components of this model. The building blocks could be reflected by collections of subroutines called from an existing language, or integrated into a uniform level of abstraction as part of a new programming language. We chose an intermediate point on this spectrum: we maintain traditional data structures and control flow from a conventional computation language, but add pervasive new constructs — representing our programming model — to coordinate among computations. Relying on an existing computation language allows us to focus on the innovative aspects of our model.

We briefly sketch the primary PROFIT$_0$ language concepts below. A more complete treatment, with several financial examples, may be found in another paper [Kaiser 91].

2.1. Facets

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, with either the type of the data (e.g., a C datatype, if C is the computation language) or the return value of the procedure (a C datatype or `void`). Procedure slots must be equated to specific procedures (e.g., C functions) at compile-time.[1] Data slots may optionally be assigned to a specific value determinable at compile-time or run-time; if not explicitly initialized, data slots are implicitly set to null values. Evaluating a data slot returns the current value, while evaluating a procedure slot executes the procedure (with the parameters provided) and returns the result of the execution, if any. Data slots may be reassigned during execution to new values, but procedure slots cannot be changed. This structure is similar to objects in Self [Ungar 87].

For example, in PROFIT we can declare a facet containing only data slots as follows:

```
FACET Some-instrument
    stock-price: price
    1Q-option-price: struct { strike-price: price,
                              option-price: price }
    2Q-option-price: struct { strike-price: price,
                              option-price: price }
END FACET Some-instrument
```

There is a distinguished slot within each facet, called `active`, that represents the currently executing operation (either evaluation of a procedure slot, or `get` or `put` on a data slot). By definition, there can be at most one operation executing within a facet at any time. Since every facet has an `active` slot, it is not explicitly declared.

Within a facet, every use of an identifier matches an identifier defined within the facet. There are no free variables. In order to support references from one facet to another, one or more slots of a facet may be

[1]Within one of these procedures, the program text can refer to slots in the same facet (both data and procedure) via extended syntax, supported by the PROFIT preprocessor.

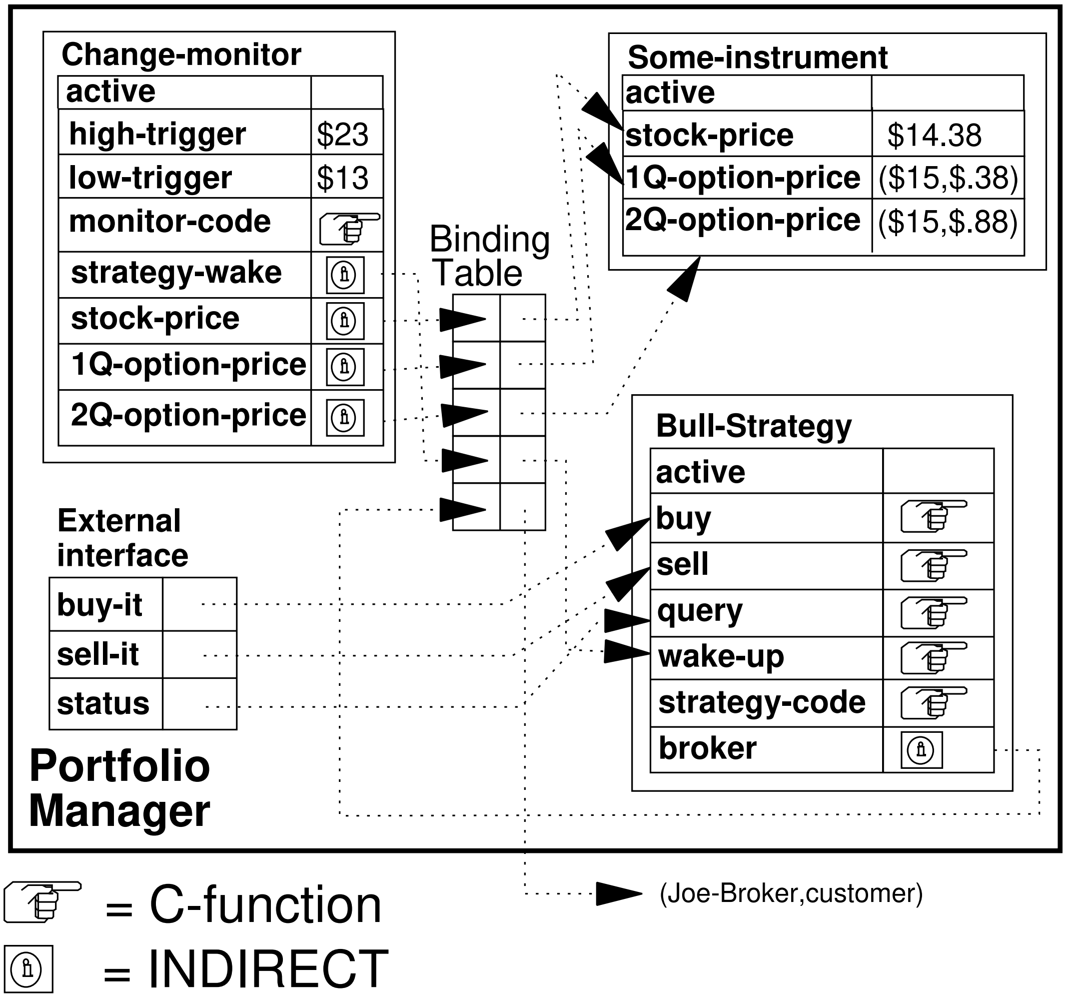

Figure 2-2: Generic Object

declared ***indirect***, as depicted in Figure 2-2. The containing object is then obliged to provide a *binding* to a slot in some other facet. Every object has a *binding table* for this purpose. When a procedure slot is being executed, and the code references an indirect slot (data or procedure) of its facet, then the semantics are to refer to the current object's binding table to resolve the reference.

For example, we would declare the `Change-monitor` facet in Figure 2-2 as:

```
FACET Change-monitor
    high-trigger: price
    low-trigger: price
    monitor-code(parameter declarations):  return type
    indirect strategy-wake(parameter declarations):  return type
    indirect stock-price: price
    indirect 1Q-option-price: struct { strike-price: price,
                                       option-price: price }
    indirect 2Q-option-price: struct { strike-price: price,
                                       option-price: price }
END FACET Change-monitor
```

2.2. Objects

An object defines an external interface and encapsulates the data and procedures in its internal facets. The interface defines the set of entries visible to other objects, representing procedures or `get` and `put` operations on data. An object binds each entry in its interface to a slot in one of its facets. Since facets may contain indirect slots, an object must bind each indirect slot in one of its facets, either to a slot in another one of its facets or to an entry in the interface of another object. In both cases, the result of the mapping is to a pair: either (facet, slot) or (object, entry).

An example portfolio management object is shown in Figure 2-2, showing how several facets may be bound together. This binding would be declared as follows:

```
OBJECT Portfolio Manager
FACETS: Change-monitor Some-instrument Bull-Strategy

ENTRY: buy-it -> Bull-Strategy.buy
       sell-it -> Bull-Strategy.sell
       status -> Bull-Strategy.query

MAP: Change-monitor.strategy-wake -> Bull-Strategy.wake-up
     Change-monitor.stock-price -> Some-instrument.stock-price
     Change-monitor.1Q-option-price -> Some-instrument.1Q-option-price
     Change-monitor.2Q-option-price -> Some-instrument.2Q-option-price
     Bull-Strategy.broker -> Joe-Broker.customer

END OBJECT Portfolio Manager
```

When code (a C function) is executing within a facet, it may access only those slots defined in the same facet. Accesses to indirect slots are resolved through the binding table at run-time. Evaluating a slot results in a call to a C function, for a procedure slot, or to the `get` or `put` operation, for a data slot. A call within a facet is treated like a conventional procedure call. For calls between facets, we consider first the viewpoint of the called facet, and then discuss the calling facet. There is a queue associated with each facet, and an arriving call is inserted in that queue. When a facet is inactive, it accepts a call from its queue. When the called operation completes, the facet places the response in the queue for the calling facet, and it then goes on to accept its next queued call, if any.[2] From the viewpoint of the calling facet, it queues the appropriate operation at the called facet and becomes inactive. The calling facet is not suspended waiting for the response, but may now accept the next call in its queue. After the response to the original call is placed in the caller’s queue and eventually accepted, the facet continues with the operation that made the call from the point where it left off. 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).

[2]This description has been simplified for ease of presentation; PROFIT$_0$ supports asynchronous message passing, and priorities as part of its soft real-time aspect.

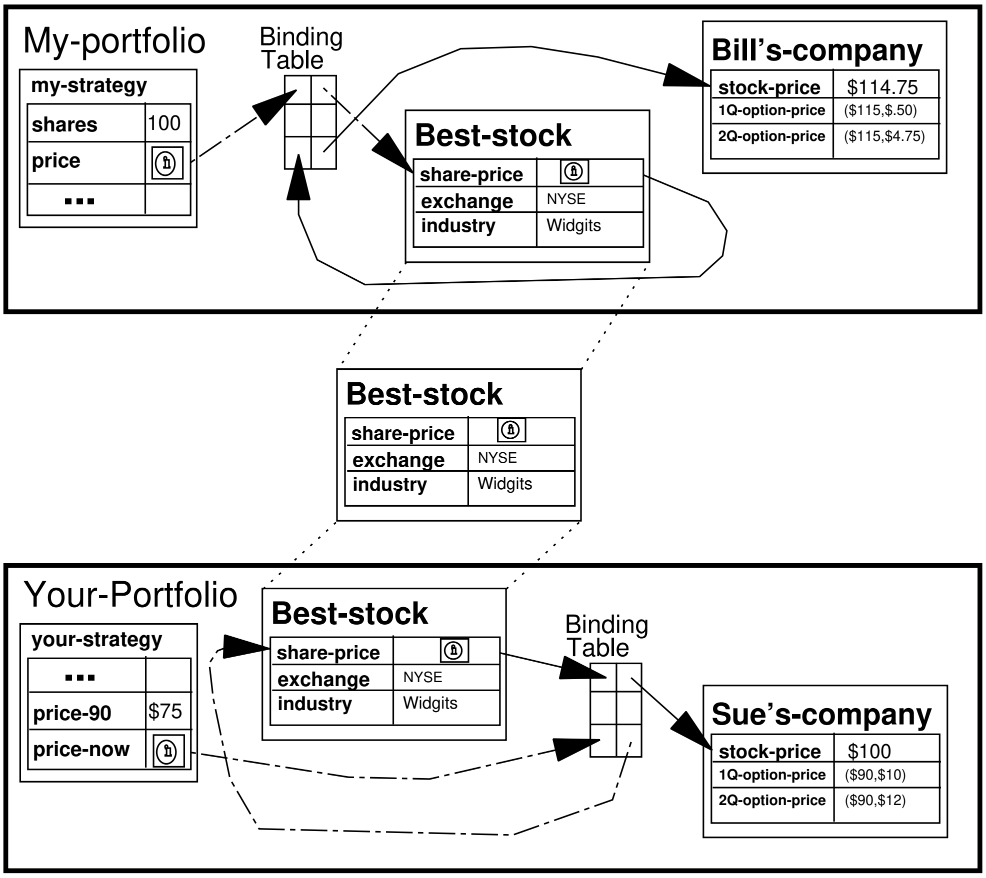

Figure 2-3: Shared Facet

So far, we have considered only the case where a facet is part of exactly one object, and thus there is exactly one binding table that needs to be considered. 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, as illustrated in Figure 2-3. When a facet is active, only one binding is actually used, the one provided by the binding table for the calling object.

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. The call is queued normally at the facet. When the call returns, the object must send the result back to the calling object.

2.3. Processes

PROFIT$_0$ processes are based on the conventional notion of processes in operating systems. Each facet resides in the address space of a particular process, and 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 be distributed among multiple processes on the same or different machines. The only physical representation of objects are their binding and entry tables, which are replicated in every process containing one or more of their facets.

PROFIT$_0$ relies on medium-weight threads similar to Sun’s lwp package [Sun 87]. Along with a simple locus of control, a thread maintains context (that is, a stack) between nested calls, thereby permitting recursion. Each call in a facet queue is represented by a thread, which provides the context of the call.

When a procedure in one facet makes a call to another facet, the first facet’s current thread is suspended and enqueued for the called facet. When a facet removes a thread from its queue, the facet sets its `active` slot to reference the thread and resumes the thread’s execution to evaluate the called slot; when the call completes, the thread is suspended and enqueued for the caller. This works only among facets within the same process, where enqueuing and dequeuing of threads is managed by simple index or pointer manipulation. When calls are made across process boundaries, a stand-in thread must be designated in the remote process. Notice that the execution of a procedure slot is not necessarily atomic. If the procedure accesses an indirect slot, then it relinquishes its control over the facet and the next call in the queue takes over. Any state that must be maintained across an indirect access must be saved in the procedure’s local variables, not in the data slots of the facet.

A process is declared as follows:

```
PROCESS Prices-database
FACETS: Some-instrument Another-instrument A-third-instrument
        Your-instrument My-instrument Database-manager

start := Database-manager.initialize()
<error handling>
END PROCESS Prices-database
```

3. Dynamic Reconfiguration

The PROFIT$_0$ language as described above assumes a fixed, static set of facets, objects and processes determined at compile-time. All entities within the PROFIT$_0$ coordination language are statically allocated, in particular, once a facet is assigned to a process it cannot migrate and no new facets, objects or processes can be added to the program. All connections among entities are also statically determined, specifically, a portfolio cannot substitute one instrument for another or change the number of instruments in its portfolio.

The PROFIT$_0$ language does not support any notion of a facet *type*, or more specifically in this context, any operations that support the creation of a new facet that is an instance of a given facet type. PROFIT$_0$ also does not permit the rebinding of indirect slots in a facet. This restricted organization permitted us to concentrate on the programming model without concern for run-time interface checking, dynamic storage management, naming and locating facets, and so on.

The new result presented in this paper, called *dynamic reconfiguration*, is the relaxation of these restrictions to allow limited changes to the static structure in order to support important special cases motivated by our application domain. Our approach is to support only a small number of well-defined changes to an executing program, which provides high leverage with low overhead. An alternative would have been to add conventional type definition facilities, dynamic storage allocation operations, facilities for directly modifying binding tables, run-time interface checking for newly bound indirect slots, and so on, making the coordination language almost indistinguishable from a full-fledged computation language.

We restrict the extended PROFIT coordination language to support the following special cases: the ability to substitute one facet for another with a compatible interface in a controlled manner, operations over (dynamically-sized) sets of facets where the constituency can be changed, and addition of new facets, objects and processes to an executing program. Considering our domain, this allows us to add new instruments to the prices database, change the composition of existing portfolios, and add new users and their portfolios.

Our approach to providing these capabilities is built on three main concepts:

- *Breed*, which is a partial facet description representing an abstract type.
- *Stall*, which is a collection of facet slots that can be mapped to the corresponding slots in any one facet (called the *occupant*) belonging to an associated breed.
- *Pen*, which contains a *set* of facets (called a *herd*) that all belong to an associated breed. A member of a herd is called a *constituent*.

We first discuss breeds and stalls, which provide a simple facility for changing an executing program: replacing a ‘‘client’’ facet’s binding from one ‘‘server’’ facet to another ‘‘server’’ facet. We next explain pens and herds, which add the ability to operate over changing collections of facets. We chose this terminology because more conventional terms like type, collection, set, interface, variable, group, view, etc. already have multiple meanings in the literature, and we wished to avoid misleading readers who may be familiar with the terms in other contexts. To our knowledge, Computer Science has yet to exploit the ranching metaphor.

4. Breeds and Stalls

In PROFIT_0, each operation is executed by invoking a procedure slot in some specific facet. A procedure slot either contains a C function, or is declared ***indirect***, in which case the enclosing object provides a binding to a procedure slot in another facet or in the interface of another object. Because the program uses static binding, all these indirections can be checked at compile time (e.g., that the referenced slots actually exist and that procedure references refer to procedure slots as opposed to data slots).

We use breeds and stalls to relax the constraint that a binding cannot change over time. Breeds provide the mechanism for declaring which facets can substitute for which other facets. Stalls are the language construct that permits certain slots to actually be rebound to a new occupant. An important concept is that breeds and stalls refer to multiple slots in a single occupant. Hence stalls permit the binding of multiple slots — all in the same facet — as a single unit whereas all bindings in PROFIT_0 are on an individual slot-by-slot basis.

A *breed* is defined as a set of slots representing a service provided by a facet. Each slot is defined by a signature. Every facet that contains this set of slots is a member of the breed, and thus is presumed to provide the defined service. Facets may contain more slots than those included in the breed. Thus a breed corresponds to an abstract type in Emerald [Black 86] or a role in RPDE[3] [Harrison 90]. For example, we can define any facet with procedure slots `1Q-option-price` and `2Q-option-price` to be a member of the `Options` breed, as follows:

```
BREED Option
    1Q-option-price: struct { strike-price: price,
                              option-price: price }
        ON EMPTY { return { 0, 0 }; }
    2Q-option-price: struct { strike-price: price,
                              option-price: price }
        ON EMPTY { return { 0, 0 }; }
END BREED Option
```

The "ON EMPTY" syntax gives error handling code to be used when an uninitialized stall is accessed. This is discussed in Section 5.

Dynamic rebinding introduces two potential problems: interface mismatches between a client and its new server, and the interruption of outstanding calls from the client to its previous server. An interface mismatch arises when the new occupant of a stall does not provide the same facilities as the old one. One way of handling this mismatch would be to adopt the ‘‘message not understood’’ feature of Smalltalk. This approach, however, requires an exception handling mechanism to deal with unexpected responses. Another possibility would be to identify, at run-time, a set of slots to be rebound to some new occupant, and then carry out slot-by-slot interface checking at the time of rebinding. Instead, breeds allow manipulation of a set of slots as one unit and compile-time determination of type conformance. By declaring the breed at compile-time, only one check has to be carried out to ensure that a new binding preserves membership in the breed.

Breeds describe the facilities offered by server facets. A *stall* identifies the particular set of slots within a client facet that can be rebound to corresponding slots in any member of an associated breed. A stall consists of a stall name and a breed name, and the stall as a whole is depicted as ***indirect***. Figure 4-1 illustrates an `Option` stall (`Some-company`) within a generic facet (`Strategy`). The declaration for the `Strategy` facet would look like:

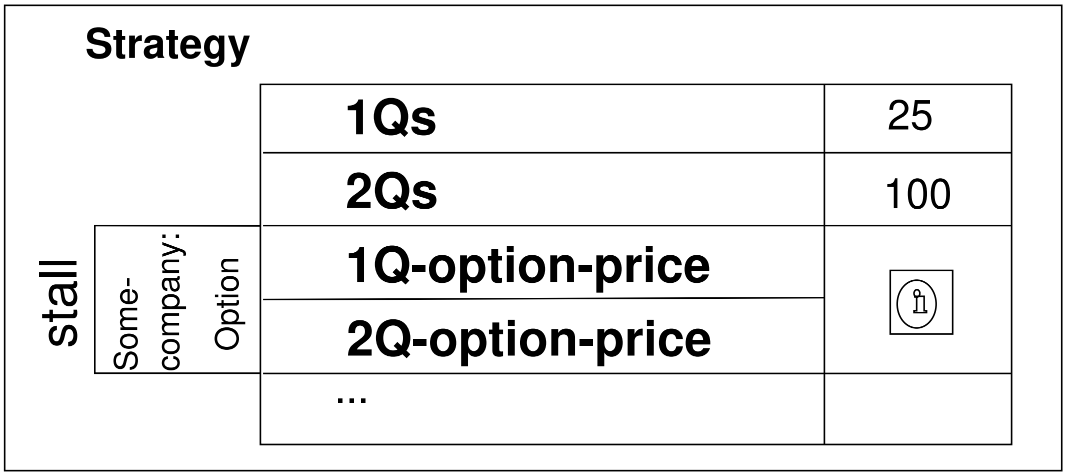

Figure 4-1: Generic Stall

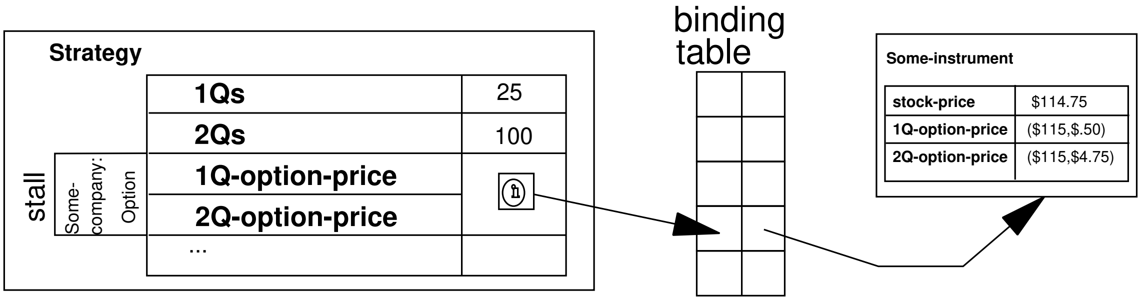

Figure 4-2: Binding an Occupant to a Stall

```
FACET Strategy
    1Qs: price
    2Qs: price
    STALL Some-company: Option
    ...
END FACET Strategy
```

Figure 4-2 shows how the slots of an occupant (the `Some-instrument` facet), those matching the `Option` breed definition, are bound to the `Some-company` stall in the `Strategy` facet. In addition to the original (facet,slot) to (facet,slot) bindings, the binding table is now extended to bind (facet,stall) pairs to occupant facets. Note that a second component (e.g., slot name) is not needed for the range of the (facet,stall) binding, since the breed determines the names of the relevant slots in the occupant. We will discuss how to request rebinding in the next section. The initial binding of an occupant to a stall is done in the map clause of the object declaration:

```
OBJECT ...
FACETS: ... Strategy Some-instrument ...

ENTRY: ...

MAP: ...
     Strategy.Some-company -> Some-instrument
     ...

END OBJECT ...
```

Breeds solve the static interface problem, but there is another problem associated with the dynamics of replacing one occupant with another. Recall that during an inter-facet call, the caller is not suspended but rather accepts the next call in its own queue. This makes it possible for a facet to make a call to an occupant of a stall, and then before that call returns, execute another operation to change the contents of the stall to some other occupant. It is important to define what happens to the ‘‘dangling’’ call. When a dangling call completes its execution and returns its result, the calling facet continues normally from that point. However, subsequent calls to the same stall will be sent to the new occupant rather than the old one. This approach is consistent with our notion of ‘‘reentrant’’ facets: arbitrary changes can be made to a facet state between an indirect call and its return. Stalls simply extend this principle: there is no way to distinguish between changes to the contents of an occupant of a stall and the replacement of one occupant by another.

5. Pens

Using just breeds and stalls, the PROFIT programmer could define portfolios consisting of multiple instruments, and change which particular instruments are included as market conditions change. The programmer would have to declare specific named stalls/slots in his portfolio to be bound to each desired instrument, and although an instrument could be substituted, the stalls/slots in the portfolio code could not be renamed. This would be analogous to having variables declared X1, X2, X3 in a conventional computational programming language (or, alternatively, a fixed size array indexed X[1], X[2], X[3]), with no ability to create more variables (or change the size of the array) on the fly.

The limitations of this ‘‘feature’’ become clear when the programmer wants to permit the user to change the size as well as the composition of a portfolio during program execution. Of course the programmer could declare a large number of stalls, and start out with most of them empty, but then it would be necessary to remember which stalls were empty and program the necessary stall management, reintroducing the problem of dynamic storage management. Instead, we provide a scheme for dynamically determining the number of facets that can support a designated service for the same client facet.

Returning to our ranching analogy, a stall can hold a single animal while a pen can hold multiple animals, collectively called a herd. We therefore extend PROFIT's stall notion to allow the collecting together of a set of facets belonging to the same breed. The collection is called a *herd* and each member is a *constituent*. The *pen* is the language construct that allows herds to be collected (rounded up). The purpose of this extension is to allow a portfolio to contain multiple instruments, where the number of constituents as well as the identity of the constituents can change over time. If only the identity could change but the number was fixed, then a fixed array of N stalls would suffice, as mentioned above. This notion of a herd requires us to be able to select individual constituents and to provide operations over entire herd. The syntax for declaring a pen is similar to that of a stall:

```
 FACET Big-Strategy
    1Qs: price
    2Qs: price
    PEN Many-company: Option
    ...
END FACET Big-Strategy
```

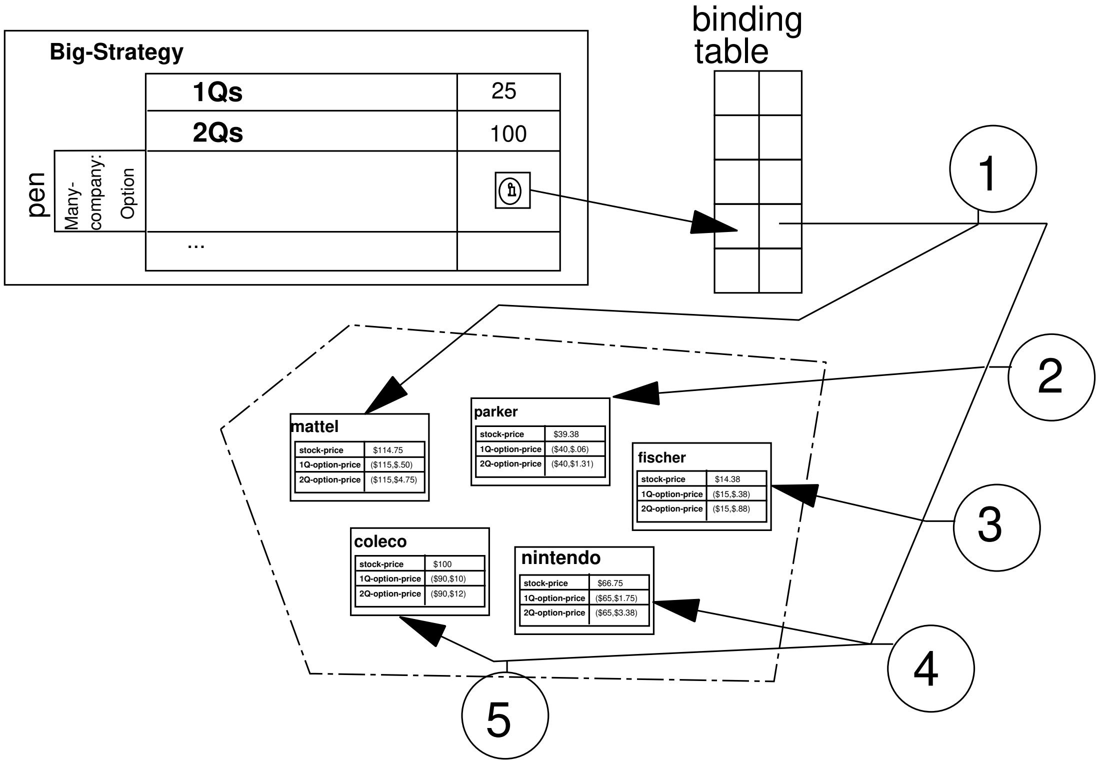

Figure 5-1: Binding a Herd to a Pen

Figure 5-1 shows a toy example of mapping a herd of instruments `mattel`, `coleco`, `parker`, `fischer` and `nintendo` into the `Many-company` pen of the `Big-strategy` facet. Notice that the aggregate structure is effectively represented as part of the binding table, where the pen is linked to an entry in the binding table and this entry links to all of the facets in the herd.

All that is needed for a facet to belong to a breed is to provide the designated set of slots. Since an object interface can also provide a set of slots, objects as well as facets can be members of a breed. Thus an

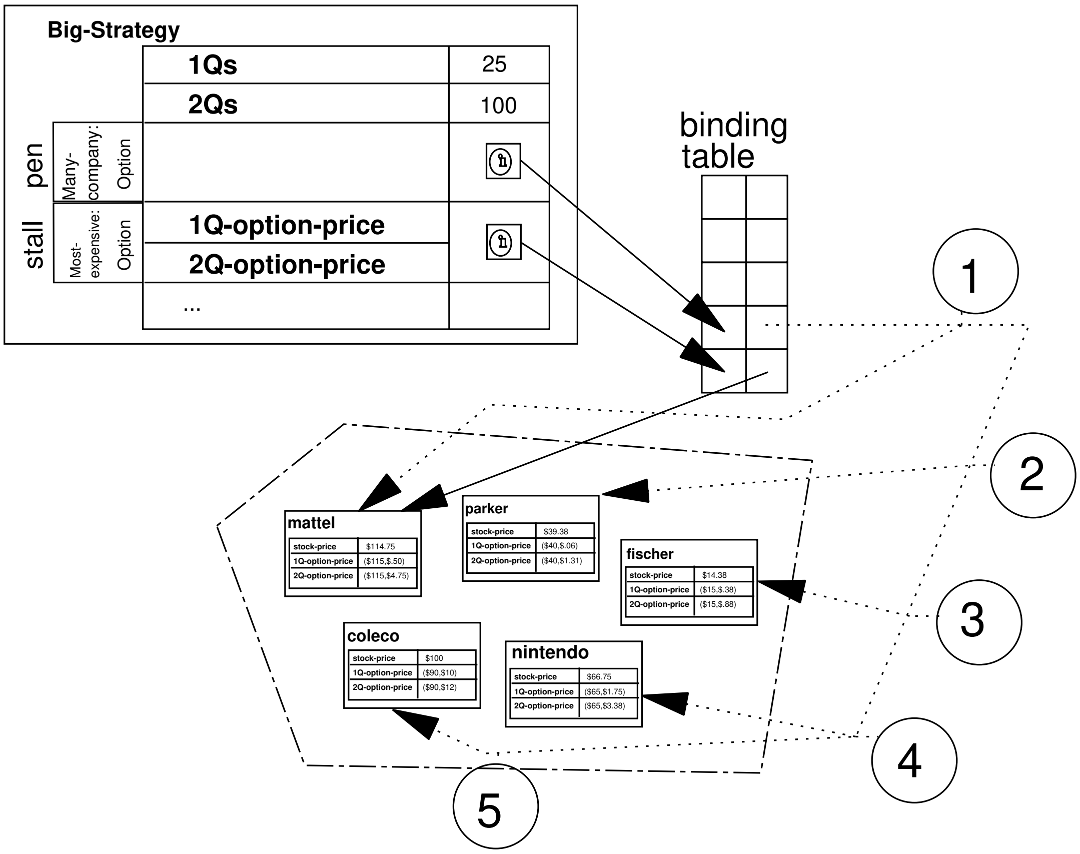

Figure 5-2: Iterating through the Members of a Herd

object can occupy a stall and can be a constituent of a herd. Everything we have said regarding breeds and herds for facets applies similarly to objects. Facets and objects can occupy the same stall (at different times) and be mixed in the same pen. In the rest of the paper, we will often say facet when referring to a member of a breed, but in general, the same statements apply equally well to objects.

In order for a constituent of a pen to be manipulated, it must first be chosen via a ‘‘select’’ operation. The resulting constituent is housed in a stall for reference during the actual manipulation. For example, in Figure 5-2, the `Most-expensive` stall is occupied by the stock in the `Many-company` pen with the highest first quarter strike price. To accomplish this, the portfolio manager executes

```
Query(Many-company, Most-expensive)
  Where MAX(1Q-option-price.strike-price)
```

If this query is successful, as illustrated in the figure, then the slots of the new occupant (`mattel`) can be accessed as "Most-expensive.<slotname>". The case of an unsuccessful query is discussed later.

In addition to querying a pen to produce a single constituent, it is also desirable, sometimes, to iterate over the entire herd. Iterating over a set can be accomplished using a SETL-like ‘‘forall’’ statement, with an arbitrary order of access. The key feature of such a statement is that each element of the set is accessed once and only once. This approach would be difficult to implement for PROFIT, because the constituency of a pen can change while the iteration is in progress, and thus some mechanism would be needed to keep track of which constituents had already been visited. We solve this problem by implementing pens as ordered sets, with the order determined by the relative time of addition to the set (see Figure 5-2). Thus if constituents are added to a pen while an iteration is in progress, it is ensured that they will be accessed appropriately. Thus it is guaranteed that at the end of the iteration, all the current constituents of the herd will have been visited. Furthermore, if no constituent has been removed and added again during the iteration, they will each have been visited exactly once.

The `Query` syntax employed above may be used to replace either the occupant of a stall or the constituency of a pen. To add constituents to a pen without removing those already there, the `Add` syntax below is used. The `Remove` statement is used to remove one or more constituents from a pen or the occupant from a stall. There are seven forms:

```
Query(source-pen, target-pen) Where ...

Query(source-pen, target-stall) Where ...

Add(source-pen, target-pen) Where ...

Add(source-stall, target-pen)

Add(source-stall, target-stall)

Remove(target-pen) Where...

Remove(target-stall)
```

When the target is a stall, a query by definition returns only one element.

Now let's consider the case where a query might fail. For example, the `Many-company` pen above might have been empty, and thus there was no maximum first-quarter strike price. Hence the `Most-expensive` stall would be empty after the query. In general, a query may be unable to find desirable members of a breed to place in a pen or to occupy a stall. The possibility of an empty stall is particularly problematic, because it is necessary to define what happens when a slot of an empty stall is accessed.

One approach would be to include an elaborate exception handling mechanism in PROFIT. However, in keeping with the coordination language/computation language distinction, any exception handling facility should come from the base computation language rather than PROFIT. In case the base computation language has no specific facility, however, PROFIT provides the following simple mechanism: every stall has an associated ‘‘empty’’ bit. This bit can be tested prior to accessing the stall, for example,

```
if ( Most-expensive.empty == 0 ) ...
else ...
```

If an empty stall is accessed, the computation language code associated with the "ON EMPTY" condition in the breed (`Option`) is executed (see section 4). The possibilities for what can be done in this condition are determined by the facilities provided by the base computation language.

It is generally impossible to prevent a stall from becoming empty for three reasons: (1) a query may fail, as described above; (2) during a call to indirect slot, the facet is relinquished to another thread of control, which might make the stall empty; and (3) a stall may indirect to another stall, which in turn might indirect to a facet, and the intermediate stall might be changed at any time. The first two situations can be handled by checking the empty bit and careful programming of calls. But the third problem requires exception handling. Thus PROFIT defines a very primitive form of exception handling ("ON EMPTY") in the breed definitions.

6. Registries

We have discussed the idea that the constituents of a herd can change. The questions arise as to how does a program ‘‘know’’ what facets are available to be added to a herd, and how does it add them. For example, we would like for a portfolio manager to be able to iterate through the available instruments and decide in which to invest, whether or not it has made previous investment in any of these instruments. It should even be possible to consider new instruments that did not exist at the time the portfolio was originally constructed.

A common solution to this naming problem is to provide a ‘‘known place’’ that keeps track of all the names in the program and the locations of the corresponding entities. In PROFIT, this role is fulfilled by *registries*, which list the members of a particular breed. Registries can be queried, and they return one or more registered members. A query might be simple, for example, ‘‘give me a member’’ or ‘‘give me all the members’’, or associative, for example, ‘‘give me the member with the highest current yield’’ (assuming that `yield` is the name of a slot defined for the breed). The resulting facet(s) returned are then bound into a stall or pen.

It is important to note that the query does not return handles, such as pointers or ‘‘facet identifiers’’, that could be stored in variables or passed as parameters. Instead, the result is stored directly in a stall or pen by the run-time system. Throughout our work on PROFIT$_0$, we deliberately avoided introducing pointers to facets, or other kinds of facet identifiers. Such identifiers are ugly in principle because without hardware or operating system support (e.g., capabilities [Tanenbaum 87]), programs can manipulate them in arbitrary ways, forge them, and access the associated facets in violation of integrity constraints, e.g., one could extract the tenth through thirteenth words offset from the beginning of a facet.

In PROFIT$_0$, we were able to avoid explicit pointers or identifiers because all facets were bound statically and all references went through the statically created binding table. This approach has a significant advantage in providing a uniform programming style for slot manipulation, whether the slot is defined directly in the facet or requires indirection to another facet or to an object interface entry. In particular, there is no need for indirect accesses to explicitly dereference a pointer or lookup a facet identifier. We would like to preserve this approach with the extensions discussed in this paper.

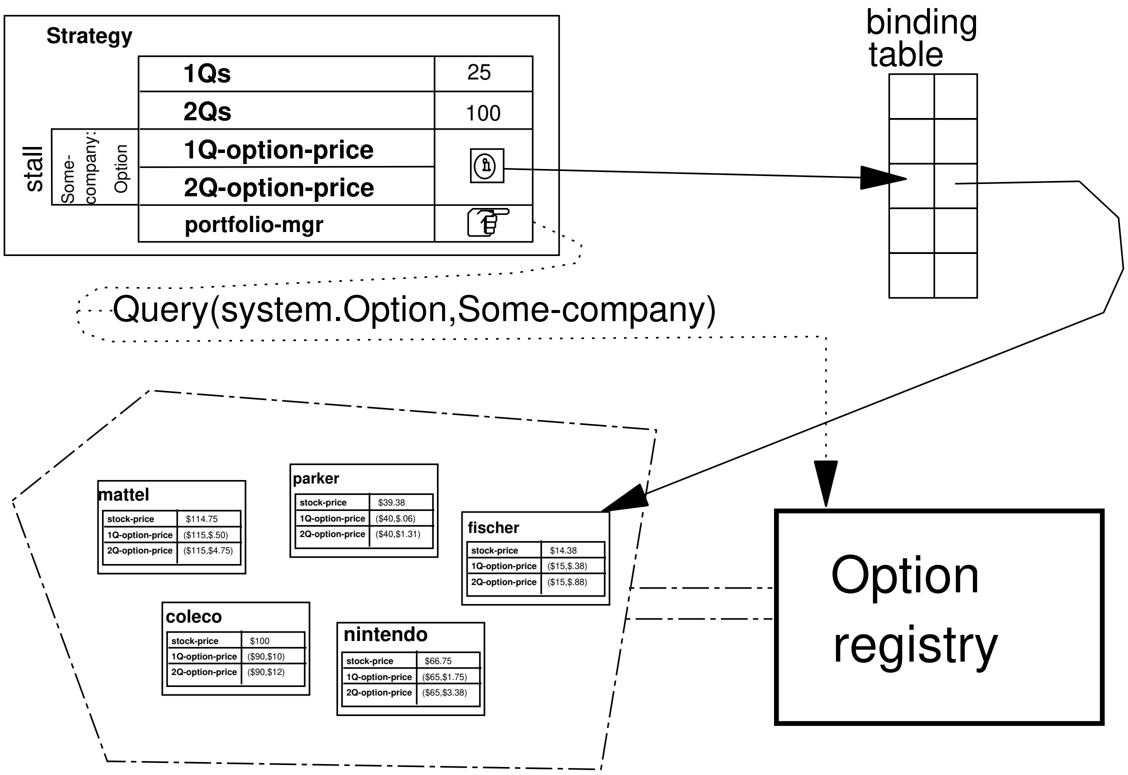

Figure 6-1: Facet Prior to Query

The PROFIT system provides a *registry* for each breed. When a new facet is created (we explain how in Section 7), it is automatically added to the appropriate registries — there may be more than one appropriate registry since a facet can be a member of multiple breeds. Subsequent queries on these registries can place this facet in a stall or add it to a pen. Figure 6-1 shows a facet prior to a query, while Figure 6-2 shows the rebinding of the facet after the query.

This mechanism is rather limiting, since it does not permit registration according to criteria more restrictive than breed. Thus PROFIT extends these system-defined registries to also allow creation of user-defined registries. These registries are constructed using the breed, pen and query facilities, to build up a herd of facets that match criteria more specific than just the breed. Hence PROFIT registries are first-class objects. A potential client can query a user-defined registry in the same manner as a system-defined registry. Figure 6-3 shows a facet making a query to the user-defined `FAO` registry, which the

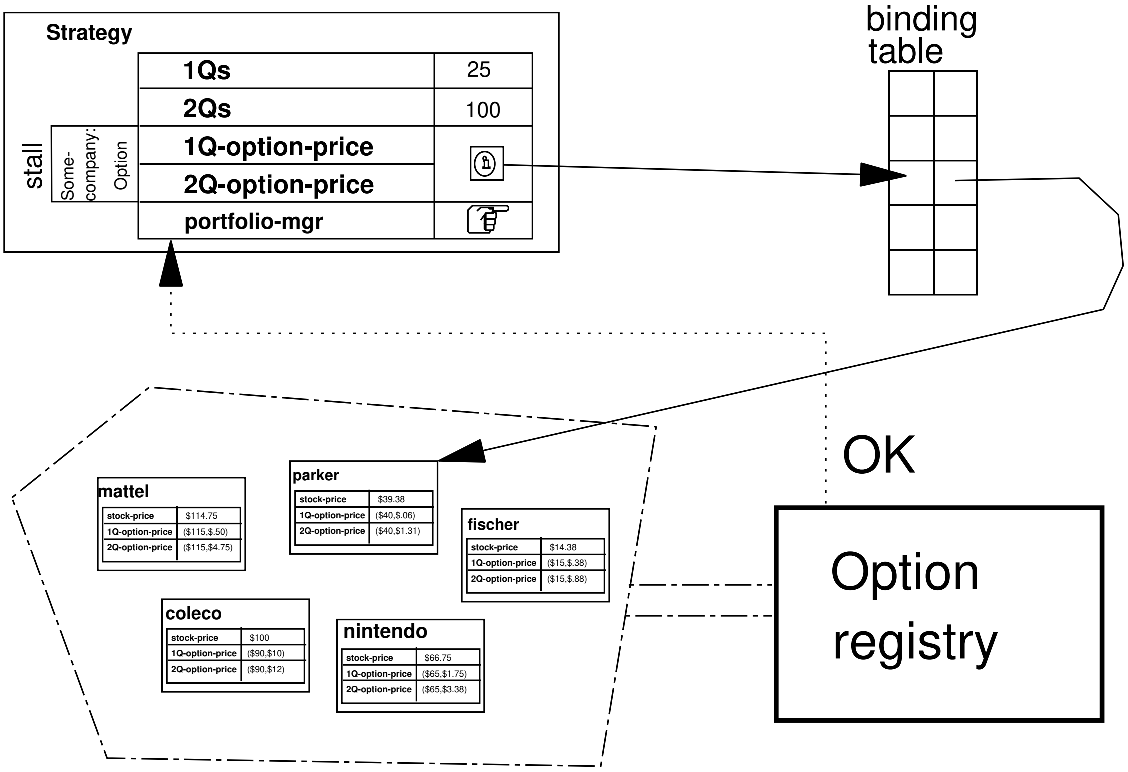

Figure 6-2: Facet After Rebinding

calling facet refers to as `my-registry`. Figure 6-4 shows how a user-defined registry is itself defined with a pen containing those members of a particular breed that are of interest for the purpose of the registry.

As shown in Figure 6-1, queries to the system-defined registry for a particular breed simply name the breed as a qualifier for the `system` registry. But given the existence of user-defined registries, it is necessary to consider how a facet names one of these registries. Any facet that is designed to work only with a particular registry can have this registry hard-wired in its binding table. Similarly, a facet could contain slots representing multiple predetermined registries.

But if more flexibility is desired, then a facet must be able to change the registry it uses, or use a set of registries, where membership in the set can change. This can be done using stalls and pens, as already described, to substitute one registry for another or to select from a herd of registries. A facet determines the set of available registries through the system-defined `Registry` registry, the registry of members of the `registry` breed. In Figure 6-5, we illustrate the `Fancy-strategy` facet using the `Registry` registry to selectively bind the `FAO` registry into its `my-registry` stall, and in turn using the `FAO` registry for subsequent queries to fill its `Play` pen. The syntax for both querying and changing the

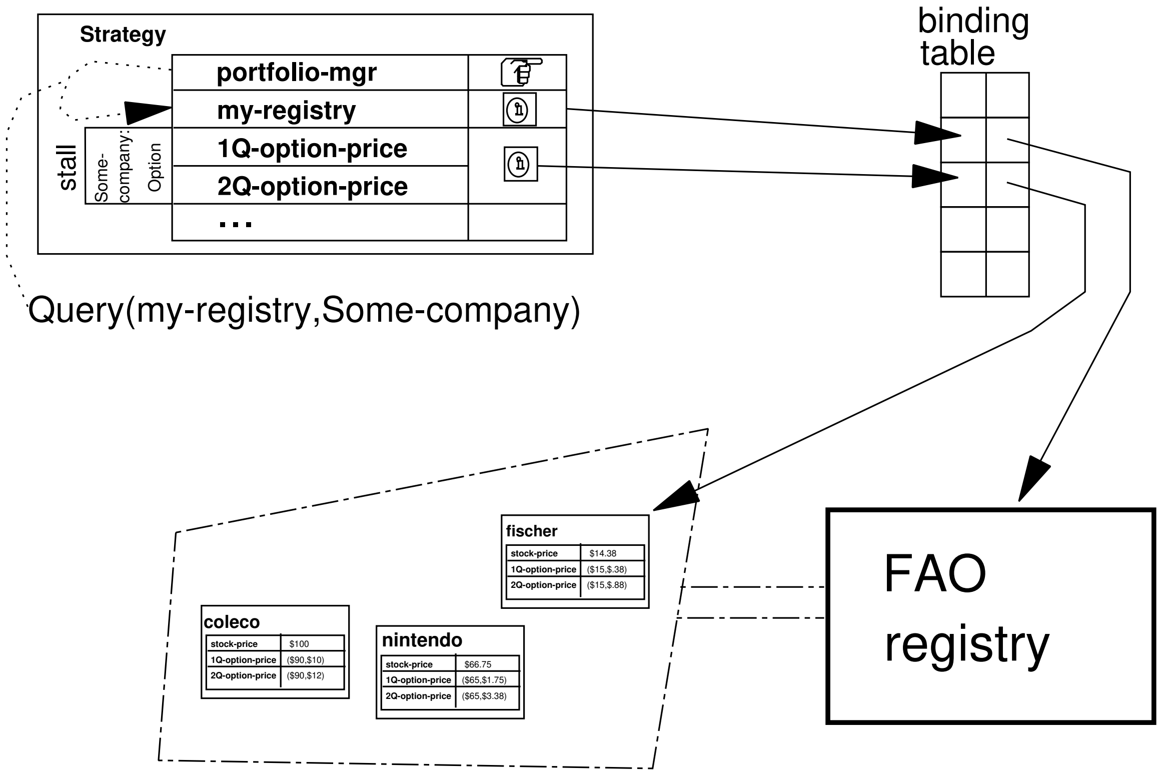

Figure 6-3: Query to User Registry

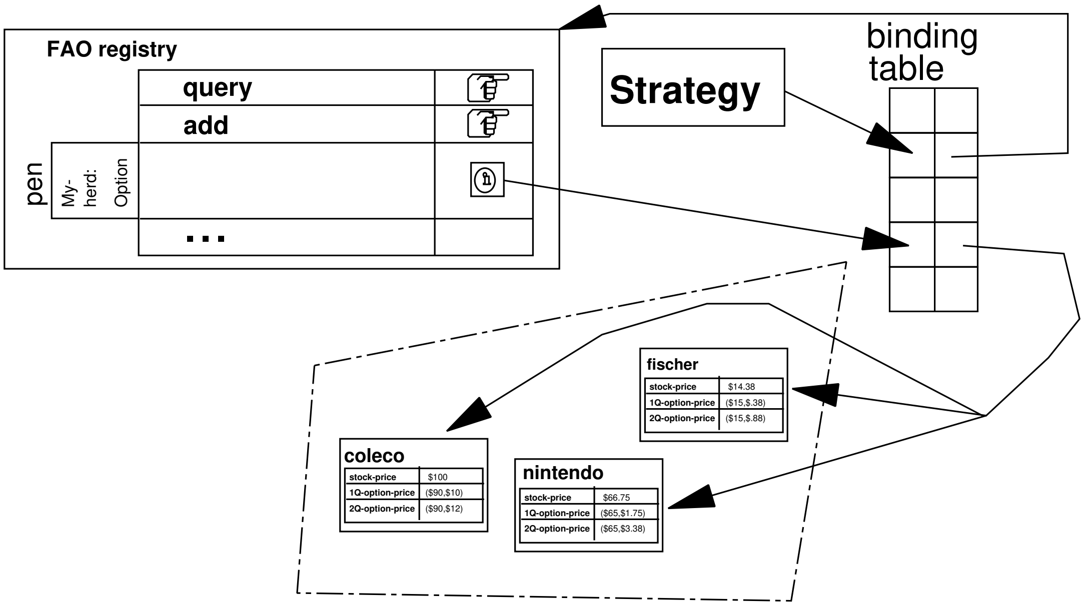

Figure 6-4: Inside a User Registry

constituency of a user-defined registry is thus the same as the query syntax described for pens in the previous section.

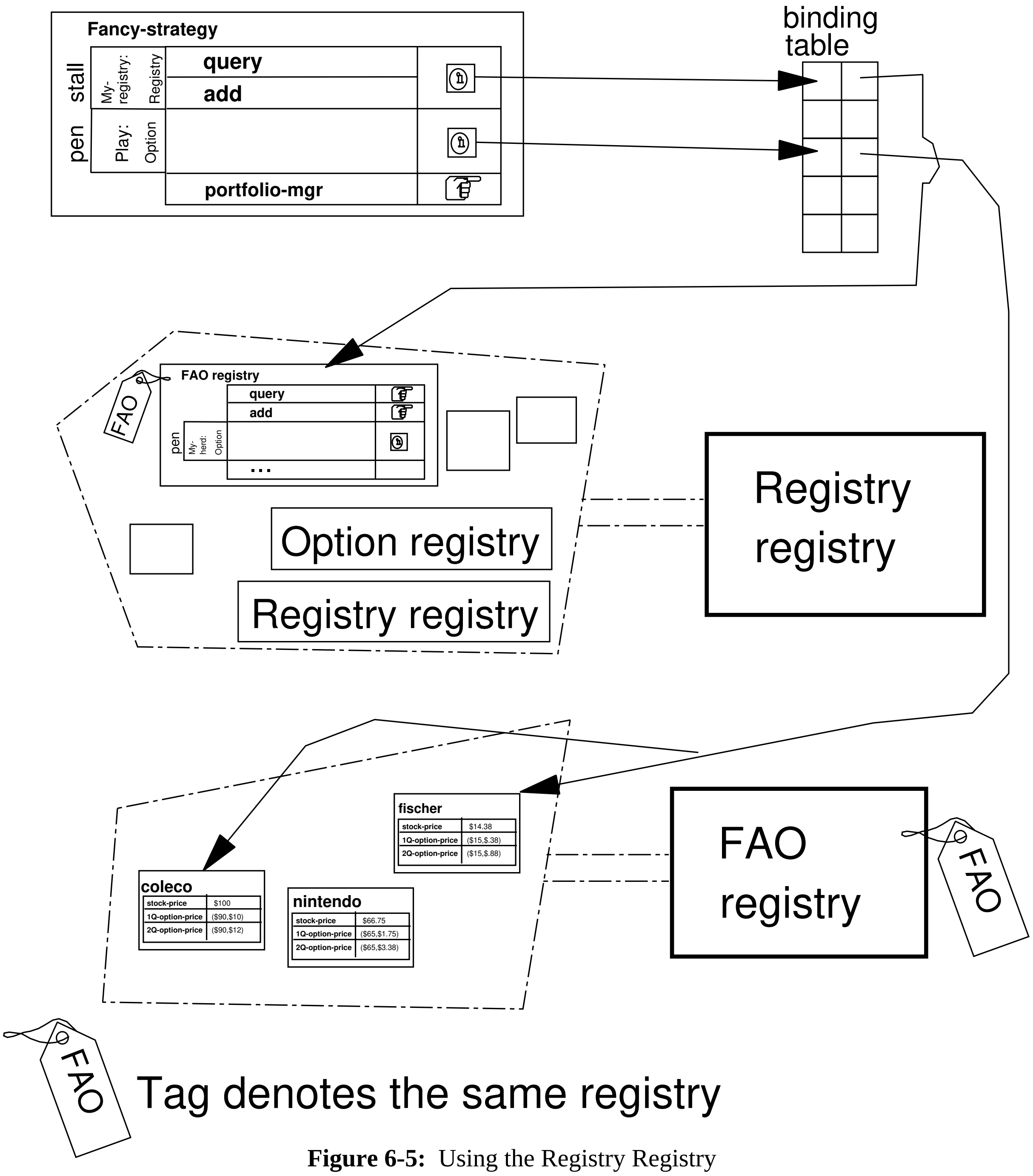

Figure 6-5: Using the Registry Registry

7. Ranchhouses

The discussion so far covers sufficient facilities to reconfigure objects and facets within an existing program. For example, a portfolio manager can concentrate on a different stock by changing a stall and a portfolio can hold a changing variety of stocks using a herd. Previously unconsidered stocks, resident in the prices database, can be evaluated using a registry. However, there is no way as yet to construct entirely new portfolios or to add new stocks to the prices databases.

There are two alternative paradigms for adding facets and objects to a running program, internally and externally. In the internal case, the program itself creates the new entities, e.g., by executing the `new` operation on the type of the entity. This approach is common in object-oriented computation languages like Smalltalk and C++, where the creation/destruction of objects is the primary mechanism for computing. This requires dynamic storage management.

The purpose of PROFIT, however, is to provide a static structure in which computation takes place. When dynamic reconfiguration is necessary, the programmer creates the new entities externally to the program and then combines them with the running program. Thus we need a mechanism to add new processes, objects and facets to a running program. Because most operating systems already provide means for creating new processes at any time, we have chosen to use operating-system processes as our vehicle for adding new objects and facets as well. This approach still requires PROFIT to have the ability to incrementally update existing processes and to connect a new process with already executing processes.

The PROFIT run-time structure consists of a collection of PROFIT processes and a distinguished operating-system process that serves a name server and coordinator, called the *ranchhouse*. Each process contains a collection of facets, a collection of binding tables and external interfaces representing objects, and a coordination component that allows each process to communicate with the ranchhouse.

The ranchhouse acts as a repository for all of the static definitions, configuration information and system registries in a PROFIT program. The static definitions include the definitions of objects (i.e., their names and external interfaces), and the definitions of breeds (i.e., their names, slot names and types). The configuration information indicates the run-time organization of the program, including which facets reside in which processes and the binding tables of objects, as well as the host names and operating-system process identifiers of the PROFIT processes. The system registries map breed names to the list of facets and objects belonging to the breed, and also link breed slots to corresponding slots in each of these members of the breed (e.g., the `Stock-price` slot of breed `Option` could be the fifth slot of facet `mattel` and the fifteenth slot of facet `coleco`).

To make an addition to an executing program, the programmer can define new facets and new objects, and their relationships to old facets and old objects. But the programmer cannot define new breeds, modify existing breeds, modify existing facets, change the external interfaces of existing objects or change the composition of existing processes. The collection of new facets, objects and interconnections are declared in a new process called a *room*. This room is in effect added on to the existing collection of processes in the same sense that a real ranchhouse is extended by adding new rooms onto its end and creating a doorway in what was once the outside wall.

Adding a new room to an executing PROFIT program is accomplished in three phases: compile, link and execute. The compilation phase transforms the textual descriptions of the new components into object code, including the information needed to update the executing program. Compilation takes advantage of symbol table information representing the static definitions known by the ranchhouse. The link phase begins when the object code for the new room is presented to the ranchhouse as an extension. The ranchhouse inspects the room and propagates corresponding changes to existing processes. The execution phase then activates the new process and adds its facets and objects to the relevant system registries. All room additions must be serial, i.e., the linking and execution phases must be executed as an atomic unit, with respect to other additions.

Compilation can proceed separately from linking and execution. Compilation begins by taking a snapshot of the current static definitions from the ranchhouse. The new code is translated and the incremental changes to the ranchhouse information are computed. The incremental changes must be relocatable, in case there have been intermediate links between the time that the snapshot was taken and the time that the resulting room is actually linked. The sequence of compile, link and execute is structured in such a way that out-of-date snapshots cannot be made invalid by subsequent links, although exact offsets need to be computed at link time.

The purpose of the link phase is to connect any new facets to existing objects and vice versa. The ranchhouse performs any necessary relocation in the incremental binding information. The ranchhouse then broadcasts these binding table deltas to all the existing processes, and waits for acknowledgments.[3] Because the system registries have not yet been updated, and existing entries in binding tables have not been changed, there is no way for any currently executing code to refer to a new object or facet. This is desirable, since the new room is not yet running.

Finally, in the execution phase the ranchhouse forks the new process. In its startup code, the new process first sends its facet and object information to the ranchhouse, for it to update the system registries. This action makes the new facets and objects available to the rest of the program, for installation in stalls and pens. The room now executes its initialization thread.

The syntax for a room combines the syntax of facets and objects with that of a process. The following PROFIT code corresponds to figure 7-1. Note that the object definitions can refer to any combination of the new facets and existing facets in the snapshot obtained from the ranchhouse. The process must contain exactly those facets declared in the room.

[3] We assume for this paper a fault-free environment.

```
ROOM With-A-View

<facet definitions>
<object definitions>

PROCESS New-Portfolio
FACETS: <names of facets>

start := foo.bar()
<error handling>
END PROCESS New-Portfolio
END ROOM With-A-View
```

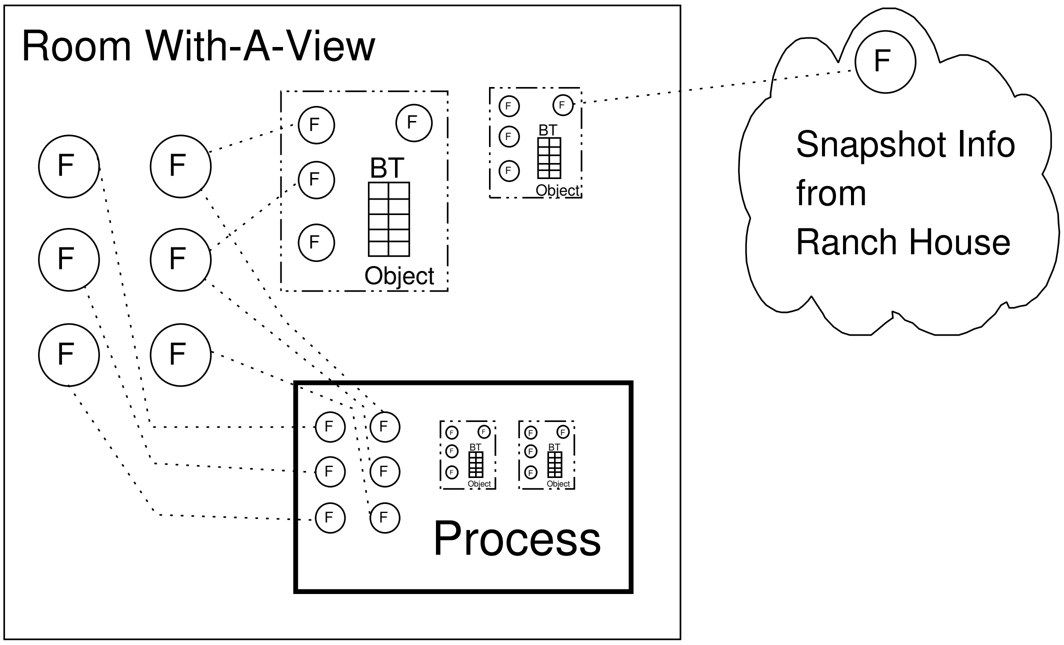

Figure 7-1: Compiling a New Room

8. Implementation

The current PROFIT$_0$ implementation is limited. It supports only a single process, although there may be multiple objects and shared facets. The prices database is updated by a simulated feed. PROFIT$_0$ includes several language facilities related to timing and scheduling, for example, the `everytime` statement repeats a loop within a specified time period after the beginning of the previous execution of the loop and the `pause` statement indicates an opportunity for a higher priority call to take over the facet.

The implementation is in the form of a coordination language for the C computation language. The coordination code is translated into C, and the compiler and run-time support is written in C. Threads are supported using the Sun lightweight processes package (lwp). The parser consists of 4375 lines of C, 247 lines of lex rules and 645 lines of yacc rules. The run-time support consists of an additional 1366 lines of C.

One reasonably large application program, called SPLENDORS [Patel 91a], has recently been completed. SPLENDORS includes 551 lines of PROFIT$_0$ coordination code and 6148 lines of C computation code (1541 lines of which are for the user interface). SPLENDORS provides a specific real-time portfolio management application intended for use by non-programmer financial industry professionals; a library of generic facets for reuse in user portfolios; parameterization and inclusion of library components in programs; and an X windows user interface.

SPLENDORS uses a different facility for dynamic reconfiguration than described here [Patel 91b]. Since the PROFIT$_0$ implementation supports only a single process, it was not possible to use our ranchhouses scheme. Instead, an interpretive approach was followed. The programmer provides a library of generic price and daemon facets, with two versions of each — interpreted and compiled. When the end-user adds new stocks to her portfolio, she also provides parameters for use by the interpretive price and daemon facets. The next time the program is generated using the `make` tool, corresponding facets with these parameters compiled in are automatically included. Thus the performance penalty is only temporary, since the program can be regenerated after every trading day. Further, the non-programmer end user can carry out her own ‘‘programming’’, without the aid of programming staff, provided sufficient generic facets are available in the library. This mechanism is part of the SPLENDORS application system, not part of PROFIT$_0$ per se.

9. Related Work

There is extensive related work on object models, which we describe in our previous paper introducing PROFIT$_0$. Since this paper concentrates on abstract interfaces (breeds) and dynamic reconfiguration, this section addresses work related only to these topics.

9.1. Abstract Interfaces

Emerald [Black 86] uses the notion of abstract types to provide the benefits of static type checking while retaining the flexibility and extensibility of untyped object-oriented languages. Abstract types are analogous to PROFIT breeds. An abstract type defines an object interface: a set of operations, their signatures and, at least in principle, their semantics. Any actual object can implement many abstract types, and any abstract type can be implemented by many different actual objects. Emerald defines a complete type-checking discipline based on conformity of abstract types. Though Emerald permits subtyping, based on type conformity, it does not support code-sharing inheritance. Unlike the explicit sharing in PROFIT, Emerald can share common code only as an implicit optimization.

Object creation in Emerald is effected by an object constructor object, rather than a class prototype or clone. In support of distributed applications, objects are manipulated through location-independent object invocation. It is the responsibility of the run-time system to locate and transfer control to the target object. Emerald uses a small number of explicit location primitives, locate (an object), fix (an object at a node), unfix (an object), and move (an object), whereas in PROFIT location information is never exposed in the programming of a facet. Emerald supports the notion of moving an object to the caller and supports a parameter passing mode termed call-by-move. Such a parameter is passed by reference, but at the time of the call it is relocated to the destination site. In contrast, PROFIT does not support object migration; PROFIT parameters are passed by value and facets cannot themselves be parameters.

RPDE^3 [Harrison 87] is not a programming language, but rather an open-ended, structural framework for integrating tools that manipulate objects. A tool, in general, can perform many functions, and can manipulate objects of a variety of types. To allow for easy extension, both of the functions that can be performed and of the object types that can be manipulated by a tool, each tool is separated into code fragments along two dimensions: function and data. A fragment, called a tool base, is associated with each function, and contains generic code to perform that function. The code is entirely independent of object type. This fragmentation is only loosely similar to PROFIT facets. When type-dependent processing is required, the tool base calls an appropriate ‘‘support’’ method. Separate code fragments associated with specific object types implement these methods: they are called method implementations.

This two-dimensional architecture allows extension in both the functional and data domains to be accomplished by addition of code fragments, rather than by modification of existing code. Each tool base in an RPDE^3 environment thus expects the objects it manipulates to implement a particular collection of ‘‘support’’ methods, and any object that does so can be manipulated successfully. Such collections of methods are called roles, which are analogous to PROFIT breeds. Each object can have multiple roles, allowing for manipulation by multiple tool bases. Both object types and roles form hierarchies: the object type hierarchy supports inheritance of implementations, whereas the role hierarchy supports inheritance of specifications. The two-dimensional organization of RPDE^3 allows it to support both subtyping and code-sharing inheritance. RPDE^3 does not support sharing between objects or concurrent programming.

Hailpern and Ossher [Hailpern 90] have extended the notion of abstract type (breed) to a ‘‘view’’ that includes interface specification, the set of objects that can provide a service (servers), and a set of objects that can consume the service (clients). This serves as a framework for describing different inheritance and delegation mechanisms and for orthogonally incorporating security, priority, and controlled interfaces in an object-oriented system. It also permits the dynamic changing of the set of clients of a view and operations over these changing client sets.

9.2. Dynamic Reconfiguration

Dynamic reconfiguration is common in operating systems and network management systems, as in both cases there are a number of resources to be managed that change relatively infrequently. In the past, reconfigurations such as adding a printer or a new network node were handled manually. As networked systems get larger and faster, it has become necessary to automate these processes, so new management and control approaches are being developed to automatically change configurations, as well as to detect and correct performance bottlenecks and failures [Krishnan 91].

The SOS operating system [Shapiro 89] is similar to PROFIT in that multiple objects (similar to our facets) can be combined into a group (similar to our notion of object), with easy communication among the objects in the group, even though the objects reside in multiple contexts (similar to our processes). A local proxy provides access to the service collectively provided by the group. Proxies may be migrated as needed for service. PROFIT's stalls provide the equivalent of proxies, but without any physical migration except in the sense of installing a local copy of the relevant binding table. SOS effectively supports a coordination language, in the sense of PROFIT, with objects implemented in C++. SOS provides a mechanism for certain cases of dynamic reconfiguration in the form of dynamic classes, where all member functions are called via a dynamic table, but requires dynamic linking capabilities avoided by PROFIT.

The Mercury system [Liskov 88] provides a general interprocess communication mechanism for heterogeneous systems. Servers are written independently of whether their application clients choose communication protocols to provide low latency or high throughput. The performance requirements of the application determines the choice of conventional synchronous RPC, asynchronous sends, or bulk byte streams, all supported by the same call-streams mechanism. Multiple languages are supported through subroutine libraries or language extensions, collectively known as language veneers. Dynamic reconfiguration is supported to a minimal extent through server ports, which are reestablished after network failures and permit binding of new clients to servers during program execution. Thus the facility comparable to stalls/occupants is implicit, and there is nothing comparable to our pens, herds or registries.

The Matchmaker system [Jones 85] is similar to Mercury. It provides an interface specification language for heterogeneous distributed systems, but does not seem to support dynamic reconfiguration. Hermes [Strom 91] is another system that supports distributed programs with well-defined interfaces between processes. New ports can be added to an executing process and existing port connections can be changed, by statements executed from within the existing Hermes code. This is analogous to PROFIT's facilities for filling stalls and pens. New processes can also be added using the `create of` statement, but only from within an existing process. Thus it is not possible to add new facilities that were not included in some sense in the original program.

The Conic [Kramer 89] is also port-based. Task modules, analogous to facets, contain code, data, entry ports and exit ports. These task modules are configuration independent: there is no direct naming of other modules, just sends and receives to ports. Group modules collect together task modules (and other group modules) in a way similar to PROFIT objects. These group modules provide configuration through a special language (e.g., create, link, and unlink). The external interfaces of task modules are the same as that of group modules, so either can be nested in higher-level group modules. Conic logical nodes correspond to our processes. Each is, in effect, a runtime instantiation of a group module. At runtime, the Configuration Manager can accept configuration commands so as to change the existing links between logical nodes and to create new logical nodes.

Conic differs from PROFIT in several ways. Conic does not provide sharing of data between task modules, except for the optimization that messages between tasks in one logical node can be pointer based. Group modules cannot span multiple logical nodes (unlike PROFIT objects, which can span multiple processes). As a result, dynamic reconfiguration in Conic cannot affect the internals of a group module: it is compiled into only one logical node. PROFIT can, however, extend objects with new facets by including them in a new process, via a new room.

Wei and Endler [Wei 91] describe a similar port-based facility with three kinds of modules: definition modules (data type and procedure declarations), program modules (code and port declarations), and configuration modules (which map ports to ports and include ‘‘change script’’ specifications). Definition modules and program modules correspond to PROFIT objects and facets, respectively, while configuration module port mappings correspond to PROFIT binding tables. Reconfiguration is based on commands to create components, link ports, unlink ports, delete components and place components on machines, with change scripts structured as condition/action rules. Users can also execute command sequences externally. In order to refer to dynamically generated components from within a static program, they are described by their structural characteristics through an object selection language rather than by name. This object selection language is navigational, describing which objects are linked through which ports to which other objects, rather than associative as in PROFIT breeds and queries.

The Polylith distributed programming system [Purtilo 91] provides facilities for dynamic reconfiguration of program modules. It provides reliable techniques for programmers to change module implementations, system topology (i.e., the bindings between module interfaces), and system geometry (i.e., the mapping of the structure onto a distributed architecture). A module is an operating-system process whose interface is defined by an abstract data type. The implementation can thus be changed by replacing the entire process with another that maintains the same interface.

The Polylith reconfiguration facilities allow for the suspension of communication between modules during reconfiguration and the transfer of state information of the old implementation of a module to the new one. It defines three groups of reconfiguration primitives, getting a capability for a change, making of series of edits to describe the change, and applying the change atomically. In PROFIT, we describe the changes in the compile phase, and atomically obtain the effective capability for a change and apply it in the link and execute phases. Since PROFIT does not support replacement of executing processes, there is no need to suspend communication between modules or to transfer state information. Polylith permits changes only when the system is in a reconfigurable state; since PROFIT only extends the existing system, a change can be made at any time — except there can be only one change in progress at a time.

Frieder and Segal [Frieder 91] describe a finer grained approach, where individual procedures may be replaced in a running program. Their approach depends on identifying procedures as active or inactive, where an active procedure either has an activation record on the stack, can call a procedure on the stack, or is semantically dependent on a procedure on the stack. Procedures are replaced as they become inactive. When a procedure containing local static data is updated, the updating mechanism invokes a user-supplied ‘‘mapper procedure’’ that converts the data from its old representation to the new one. Old code can call new code, through an ‘‘interprocedure’’ when the interface has changed. All such changes are initiated externally, and there is no way to choose to make changes from within the program. Further, even externally it is only possible to substitute new procedures for old procedures, massaging the interfaces, but it is not possible to make structural changes in the program.

10. Conclusions

The most common form of distributed programs involves client and server processes. These programs can be reconfigured by replacing clients and servers and changing the interconnections among clients and servers. Some applications, however, require a fine granularity of reconfiguration where it is not practical to represent each reconfigurable unit as an operating-system process.

The standard approach to dealing with such fine-grained entities is to enclose multiple such entities within a process. Dynamic storage management and pointers permit components to easily be created, connected and deleted on the fly. But such flexibility invariably comes with a substantial penalty. Our goal throughout this work has been to avoid the overhead and error-prone nature of dynamic storage management but still maintain the benefits of building and evolving a distributed program constructed from fine-grained entities.

In this paper, we present new facilities beyond the original PROFIT$_0$, to permit such evolution. These included extensions to the programming model in terms of breeds, new language constructs for defining and manipulating stalls and pens, and new run-time services embodied in registries. These features allow the program to change itself in response to conditions anticipated by the original programmer — and using only those facilities provided by the original programmer. The ranchhouse scheme ties all these new features together, and goes further by permitting a programmer to dynamically modify a PROFIT program while it is executing in ways that may not have been anticipated by the original programmer — and, more significantly, using code provided by new programmers.

Acknowledgments

Tushar Patel is primarily responsible for the PROFIT_0 implementation, with contributions from Jason Kim, Michael Mayer and Isai Shenker. The SPLENDORS application was developed as part of Tushar's MS thesis research. Discussions with Terry Boult, Jim Donahue, Gary Herman, Catherine Lassez, Aurel Lazar, Harold Ossher, Dan Schutzer, and Mark Segal have contributed substantially to the development of the ideas presented in this paper.

References

[Black 86] Andrew Black, Norman Hutchinson, Eril Jul and Henry Levy.
Object Structure in the Emerald System.
In Norman Meyrowitz (editor), *Object-Oriented Programming Systems, Languages and Applications Conference*, pages 78-86. ACM, Portland OR, September, 1986.
Special issue of *SIGPLAN Notices*, 21(11), November 1986.

[Boult 90] Terry Boult.
Private Communication.
1990

[Ciancarini 90] Paola Ciancarini.
Coordination Languages for Open System Design.
In *International Conference on Computer Languages*, pages 252-260. New Orleans LA, March, 1990.

[Frieder 91] Ophir Frieder and Mark E. Segal.
On Dynamically Updating a Computer Program: From Concept to Prototype.
The Journal of Systems and Software 14(2):111-128, February, 1991.

[Haeberli 88] Paul E. Haeberli.
ConMan: A Visual Programming Language for Interactive Graphics.
In *SIGGRAPH '88*, pages 103-111. Atlanta GA, August, 1988.
Special issue of *Computer Graphics*, 22(4), August 1988.

[Hailpern 90] Brent Hailpern and Harold Ossher.
Extending objects to provide multiple interfaces and access control.
IEEE Transactions on Software Engineering 16(11):1247-1257, November, 1990.

[Harrison 87] William Harrison.
RPDE^3: A framework for integrating tool fragments.
IEEE Software 4(6):46-56, November, 1987.

[Harrison 90] William Harrison and Harold Ossher.
Checking Evolving Interfaces in the Presence of Persistent Objects.
Technical Report RC 15520, IBM Research Division, February, 1990.

[Jones 85] Michael B. Jones, Richard F. Rashid and Mary R. Thompson.
Matchmaker: An Interface Specification Language for Distributed Processing.
In *12th Annual ACM Symposium on Principles of Programming Languages*, pages 225-235. New Orleans LA, January, 1985.

[Kaiser 90] Gail E. Kaiser and Brent Hailpern.
An Object Model for Shared Data.
In *International Conference on Computer Languages*, pages 135-144. New Orleans LA, March, 1990.

[Kaiser 91] Gail E. Kaiser and Brent Hailpern.
An Object-Based Programming Model for Shared Data.
ACM Transactions on Programming Languages and Systems , 1991.
In press. Available as IBM Research Report RC 16442 and Columbia University Department of Computer Science CUCS-046-90, revised December 1990.

[Kramer 89] Jeff Magee, Jeff Kramer, and Morris Sloman.
Constructing Distributed Systems in Conic.
IEEE Transactions on Software Engineering 15(6):663-675, June, 1989.

[Krishnan 91] Iyengar Krishnan and Wolfgang Zimmer (editor).
IFIP TC6/WG6.6 2nd International Symposium on Integrated Network Management.
North-Holland, Washington DC, 1991.

[Liskov 88] Barbara Liskov, Toby Bloom, David Gifford, Robert Scheifler and William Weihl.
Communication in the Mercury System.
In Bruce D. Shriver (editor), *21st Annual Hawaii International Conference on System Sciences*, pages 178-187. IEEE Computer Society, Kona HI, January, 1988.

[Mazumdar 89] Subrata Mazumdar and Aurel A. Lazar.
Knowledge-Based Monitoring of Integrated Networks.
In Branislav Meandzija and Jil Westcott (editors), *IFIP TC 6/WG 6.6 Symposium on Integrated Network Management*, pages 235-243. North-Holland, Boston MA, May, 1989.

[Patel 91a] Tushar M. Patel and Gail E. Kaiser.
The SPLENDORS Real Time Portfolio Management System.
In *1st International Conference on Artificial Intelligence Applications on Wall Street*. New York NY, October, 1991.
In press. Available as Columbia University Department of Computer Science, CUCS-011-91, April 1991.

[Patel 91b] Tushar M. Patel.
Real-time Portfolio Management and Automatic Extensions.
Master's thesis, Columbia University, October, 1991.

[Peinl 88] Peter Peinl, Andrea Reuter and Harald Sammer.
High Contention in a Stock Trading Database: A Case Study.
In *ACM SIGMOD International Conference on the Management of Data*, pages 260-268. Chicago IL, June, 1988.
Special issue of *SIGMOD Record*, 17(3), September 1988.

[Purtilo 91] James M. Purtilo and Christine R. Hofmeister.
Dynamic Reconfiguration of Distributed Programs.
In *11th International Conference on Distributed Computing Systems*, pages 560-571. May, 1991.

[Shapiro 89] Marc Shapiro, Philippe Gautron and Laurence Mosseri.
Persistence and Migration for C++ Objects.
In Stephen Cook (editor), *3rd European Conference on Object-Oriented Programming*, pages 191-204. Cambridge University Press, Nottingham, UK, July, 1989.

[Strom 91] Robert E. Strom, David F. Bacon, Arthur P. Goldberg, Andy Lowry, Daniel M. Yellin and Shaula Alexander Yemini.
Hermes A Language for Distributed Computing.
Prentice-Hall, Englewood Cliffs NJ, 1991.

[Sun 87] *SunOS Reference Manual Section 3L: Lightweight Processes Library*
Sun Microsystems, Inc., 1987.

[Tanenbaum 87] Andrew S. Tanenbaum.
Operating Systems Design and Implementation.
Prentice-Hall, Englewood Cliffs NJ, 1987.

[Ungar 87] David Ungar and Randall B. Smith.
Self: The Power of Simplicity.
In Norman Meyrowitz (editor), *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, pages 227-242. ACM Press, Orlando FL, October, 1987.
Special issue of *SIGPLAN Notices*, 22(12), December 1987.

[Wei 91] Jiawang Wei and Markus Endler.
A Configuration Model for Dynamically Configurable Distributed Systems.
In Bruce Shriver (editor), *24th Hawaii International Conference on System Sciences*, pages 265-274. IEEE Computer Society Press, January, 1991.