Dynamic Reconfiguration in an Object-Based Programming Language with Distributed Shared Data

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

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

12 October 1990

Abstract

On-line distributed applications generally 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 on-line financial services, it is necessary to support finer grained reconfiguration at the level of entities within processes, but for performance reasons it is desirable to avoid conventional approaches such as interpretation and 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 implement this scheme in the context of an object-based programming language that supports shared data in a distributed environment.

Copyright © 1990 Brent Hailpern and Gail E. Kaiser

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. Prof. Kaiser is supported by National Science Foundation grants CCR-9000930, CDA-8920080 and CCR-8858029, by grants from AT&T, BNR, Citicorp, DEC, IBM, Siemens, SRA, Sun and Xerox, by the Center for Advanced Technology and by the Center for Telecommunications Research.

topics: Coordination languages, tools and software engineering; Soft real-time issues and financial applications

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. These problems have been articulated by other researchers, but not solved (e.g., [Peinl 88]).

An on-line stock trading system 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. The key challenge in such a system 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’ 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 90a], we introduced a distributed object-based programming model that addresses these problems. This programming model supports an application architecture where price changes monitored by agents operating on behalf of individual portfolio managers. The sampling rate of each agent 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 or 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, *processes* as collections of facets organizing the run-time structure of the system, and *programs* to enumerate the objects that constitute a system and designate the configuration of communicating processes. Different facets of the same object may reside in different processes, and a facet may be shared among multiple objects.

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. 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 obtain the performance of compiled languages and avoid the overhead 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. Some optimization can be realized by compiling out indirection whenever possible. For example, not all facets are actually shared; the target of indirection for these non-shared facets can be ‘‘in-lined’’.

Unfortunately, such 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 system. 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 system facilities and distributed services for adding new clients and servers and changing connections between clients and servers, but has not previously been embedded in a programming language to support reconfiguration at a finer granularity than that of an entire process.

To support dynamic recovery, 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 for those facets that can be substituted for each other in a particular context. Stalls and pens are similar to variables and arrays, respectively, in that they ‘‘hold’’ facets and which facets are held can be changed during program execution. These new concepts give the PROFIT programmer the ability to change the set 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 ‘‘self-modifying’’ code or some interpretive scheme. 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, and some supporting language extensions for adding new facets, objects and processes to an executing system. We briefly describe the current implementation and discuss related work.

2. Background

The PROFIT_0 object model supports data sharing among arbitrary 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 — orthogonal to objects — that must execute at the same physical location. 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. There is a fourth concept, *Program*, that specifies the objects and processes that together make up a single executable program, the physical locations of the processes at execution-time and the initialization code to start the program running. This organization is illustrated in Figure 2-1.

What we have in mind for the example financial application is a program consisting of one process containing all the prices, and one additional process for each individual user. Each user would define one object corresponding to each of his portfolios, each object consisting of a new financial strategy, the relevant previously defined prices facets, and additional facets for monitoring changes to the prices (called daemons). The strategic facets would live in the user’s process, while the prices facets would be updated only in the price server; the daemons could live in the same process with the prices, allowing lower communication costs than if they lived with the portfolios.

We briefly sketch the primary PROFIT_0 language concepts below. A more complete treatment, with several financial examples, may be found in our previous paper [Kaiser 90a].

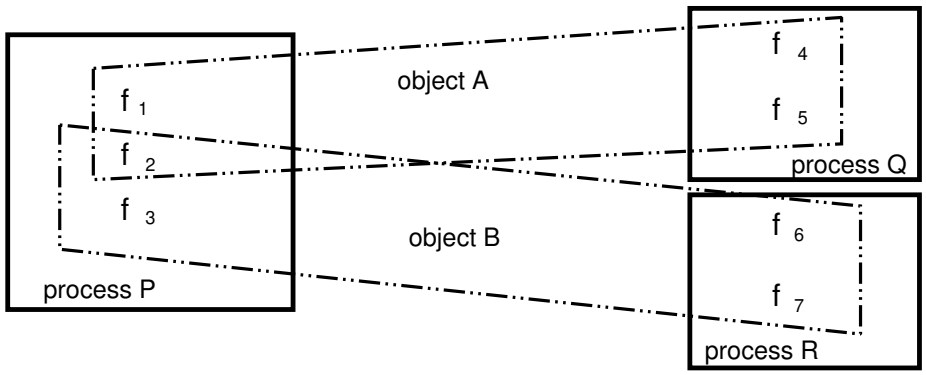

Figure 2-1: Facets, Objects and Processes

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 C functions at compile-time. 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].

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 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 within the same object or to an entry in the interface of another object. 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.

2.2. Objects

An object defines an external interface and encapsulates its internal data and procedures. The interface defines the set of entries visible to other objects, representing procedures or pairs of `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, each object must also 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).

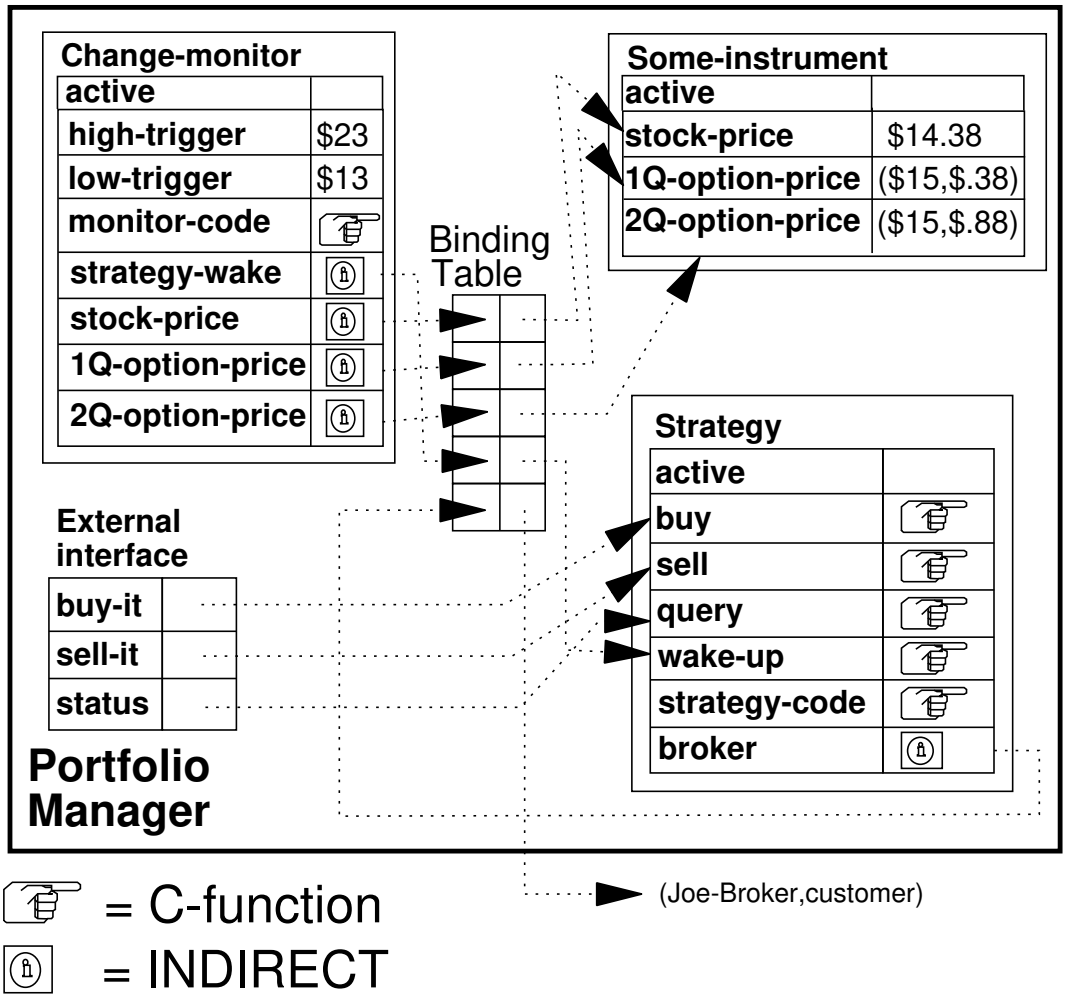

Figure 2-2: Generic Object

When code (a C function) is executing within a facet, it may directly access only those slots defined in the same facet. 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. Subsequent indirection is with respect to the binding table of the object responsible for the call. 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.[1] 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, but may now accept the next call in its queue. When the response to the original call is eventually accepted from its queue, the facet continues with the

[1]This description has been simplified for ease of presentation; in reality, PROFIT$_0$ supports threads, asynchronous message passing, priorities and pre-emptive scheduling as part of its soft real-time aspect. We discuss these components elsewhere [Kaiser 90b].

operation that made the call from the point where it left off.

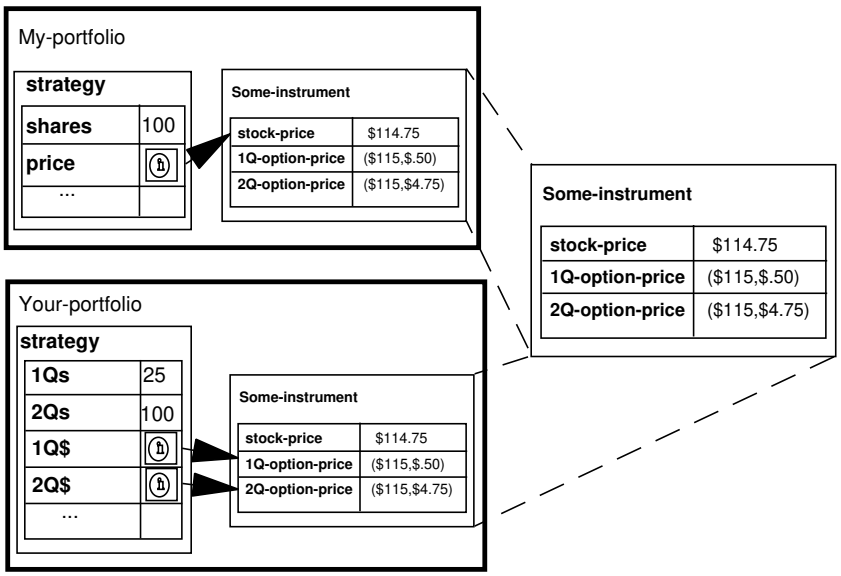

Figure 2-3: Shared Facet

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 belonging to the object from whose facet the active facet was called.

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 close to 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 tables, which are replicated in every process containing one or more of their facets.

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 allocation, naming and locating facets, objects and processes, and so on.

The new result presented in this paper is our relaxation of these restrictions to allow limited changes to the static structure, supporting important special cases motivated by our application domain. We call this *dynamic reconfiguration*. Our approach is to support only a small number of well-defined changes to an executing program, which provides high leverage with minimal performance overhead, except at the moments the changes are actually being made. 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 sets of facets where the constituent elements 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 is a collection of facet slots that can be mapped to the corresponding slots in any member of a *set* of facets (called a *herd*) that all belong to an associated breed.

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 system 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 slot names representing a service provided by a facet. Every facet that contains this set of slots is a member of the breed. Facets may contain more slots than those included in the breed. 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. A breed corresponds to an abstract type in Emerald [Black 86] or a role in RPDE [Harrison 90]. A breed can be extended to include the ‘‘signature’’ of each of the procedure and data slots in the defining set, but in this paper we simplify the discussion by considering only the names of the slots.

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 membership in the breed is preserved upon a new binding.

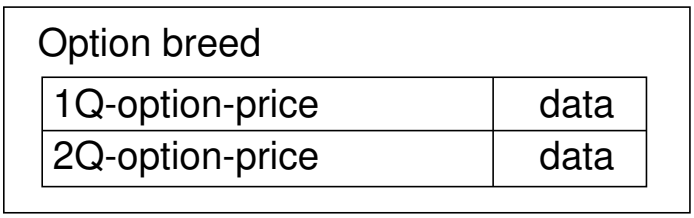

Figure 4-1: Generic Breed

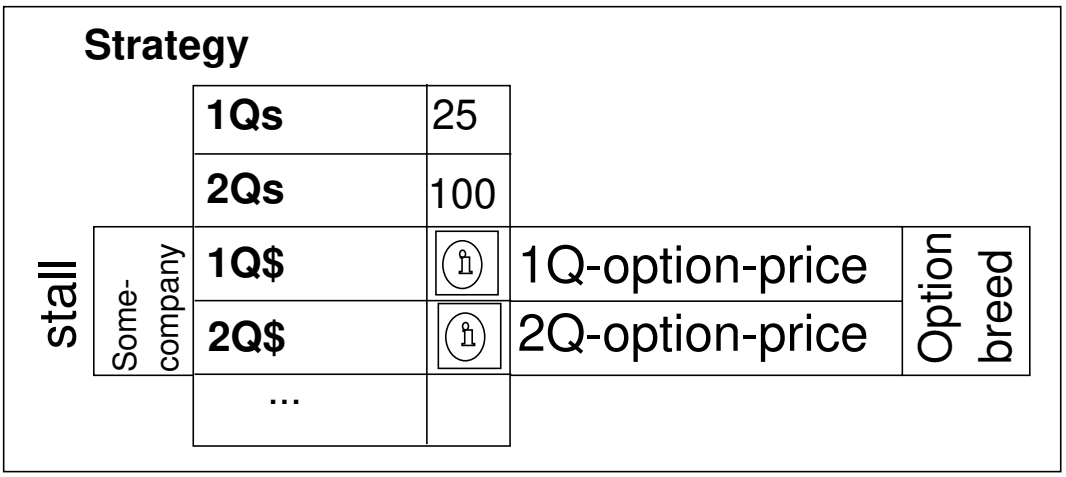

Figure 4-2: Generic Stall

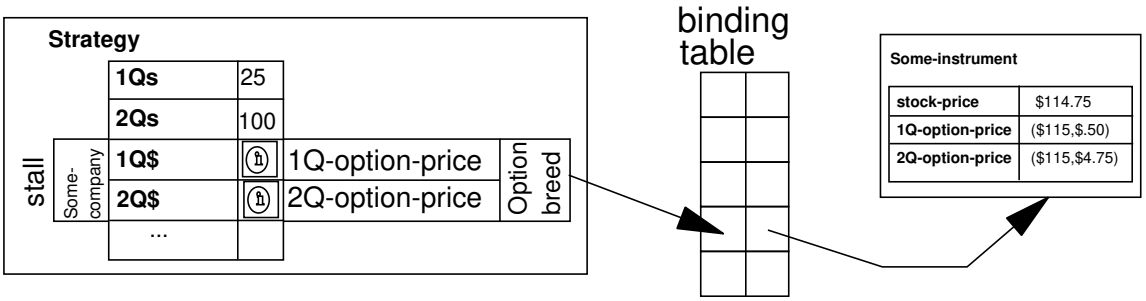

Figure 4-3: Binding an Occupant to a Stall

Breeds describe the facilities offered by server facets. A *stall* identifies the particular set of indirect 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, a set of slot names, a breed name, and a mapping from its own slot names to each of those defined for the breed. Figure 4-1 shows a generic breed definition (for the `Option` breed) and Figure 4-2 illustrates a stall (`Some-company`) within a generic facet (`Strategy`). Figure 4-3 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.

Breeds solve the static interface problem, but there is also a dynamic component of replacing one occupant with another. In particular, a facet may have made a call to the original occupant of its stall, and then before that call returns, execute another operation to change the contents of its stall to some other occupant. It is important to define what happens to the dangling call. The call completes its execution and returns its result, and the calling facet continues 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 shared facets. Outside of ‘‘atomic events’’, arbitrary changes can be made to a facet state. 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 breeds and stalls, the PROFIT programmer can define portfolios consisting of multiple instruments, and change which particular instruments are included as market conditions change. The programmer must declare specific named stalls/slots in his portfolio to be bound to each desired instrument, and although an instrument can be substituted, the stalls/slots in the portfolio code cannot be renamed. This is 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 one wants to change the size as well as the composition of a portfolio during program execution. Of course one could declare a large number of stalls, and start out with most of them empty, but then it is necessary to remember which stalls are empty and program the necessary stall management. For the same reason that most programming languages have some mechanism for dynamically allocating heap variables, we would like to provide a scheme for dynamically determining the number of facets that can provide a designated service to the same client facet.

Returning to our ranching analogy, a stall can hold a single animal while a pen can hold multiple animals, called a herd. We therefore extend PROFIT’s stall notion to allow the binding of a set of slots in a facet to a corresponding set of conforming slots in each member of a collection of facets. The purpose of this extension is to allow a portfolio to contain multiple instruments, where the number of members as well as the membership can change over time. This notion of a collection requires us to be able to address members of the collection and to provide operations over the collection.

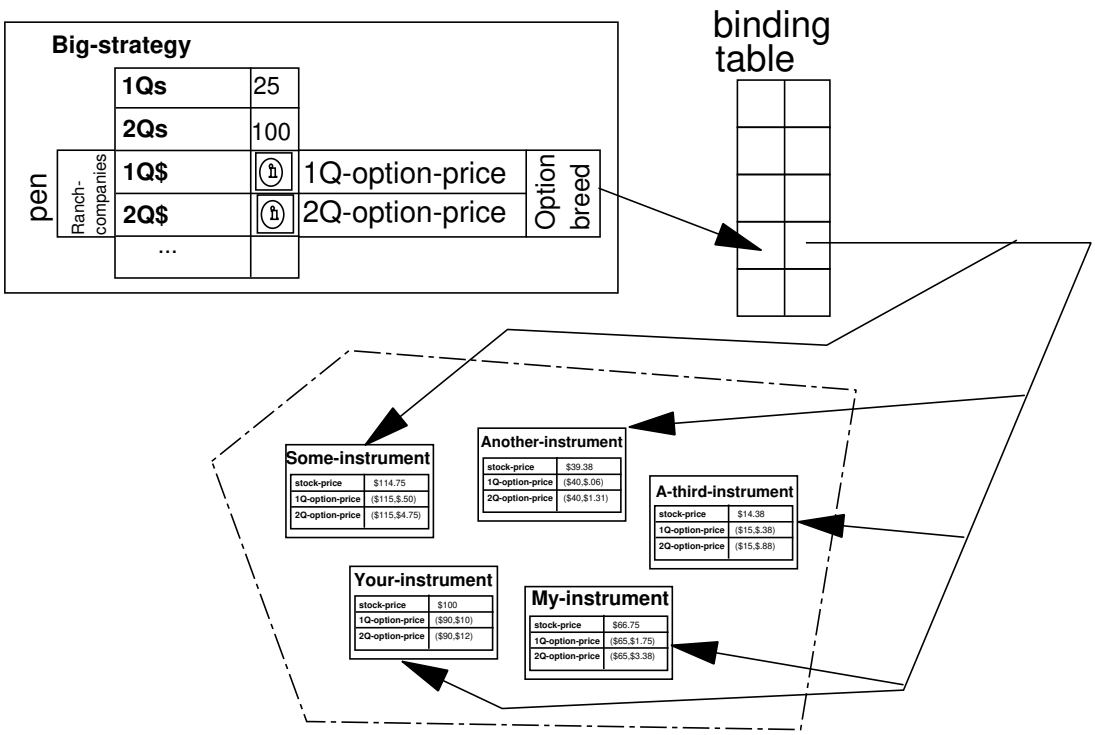

Figure 5-1: Binding a Herd to a Pen

A *herd* consists of a set of facets that all belong to the same breed. The *pen* is the language construct that allows herds to be constructed (rounded up). One way to think of a pen is as a dynamic array of stalls. Figure 5-1 shows an example of mapping a herd of instruments into the `Ranch-companies` 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.

Elements of pens are not addressable by indices, but instead are operated on by various set operations, using SETL-like constructs (e.g., `exists`, `forall`, `from`) [Donzeau-Gouge 87]. Within the scope of one of these set operations, a pen can be treated like a stall to access the slots of an individual member of the herd and treat it as an occupant. For example, a portfolio manager can iterate through the instruments currently held, check their prices and other information, and decide which to sell (or buy).

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. 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.

6. Registries

We have discussed the idea that the membership 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 instruments available and decide which to invest in, 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.

Assume for now that there exist objects, called *registries*, that contain members of a particular breed. These 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 member or members returned are then bound into a stall or pen, respectively.

It is important to note that the query does not return handles, such as pointers or ‘‘facet identifiers’’, that could be stored in a variable or passed as a parameter. 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), programs can manipulate them in arbitrary ways, forge them, and access the associated facets in violation of integrity constraints. Hence 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.

Assume for the sake of discussion that there is exactly one registry for each breed. When a new facet is created (we explain how later), 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 include 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 programming of objects as user-defined registries. These registries are constructed using the breed, pen and query facilities already described, building up a herd of facets or objects that match more specific criteria as

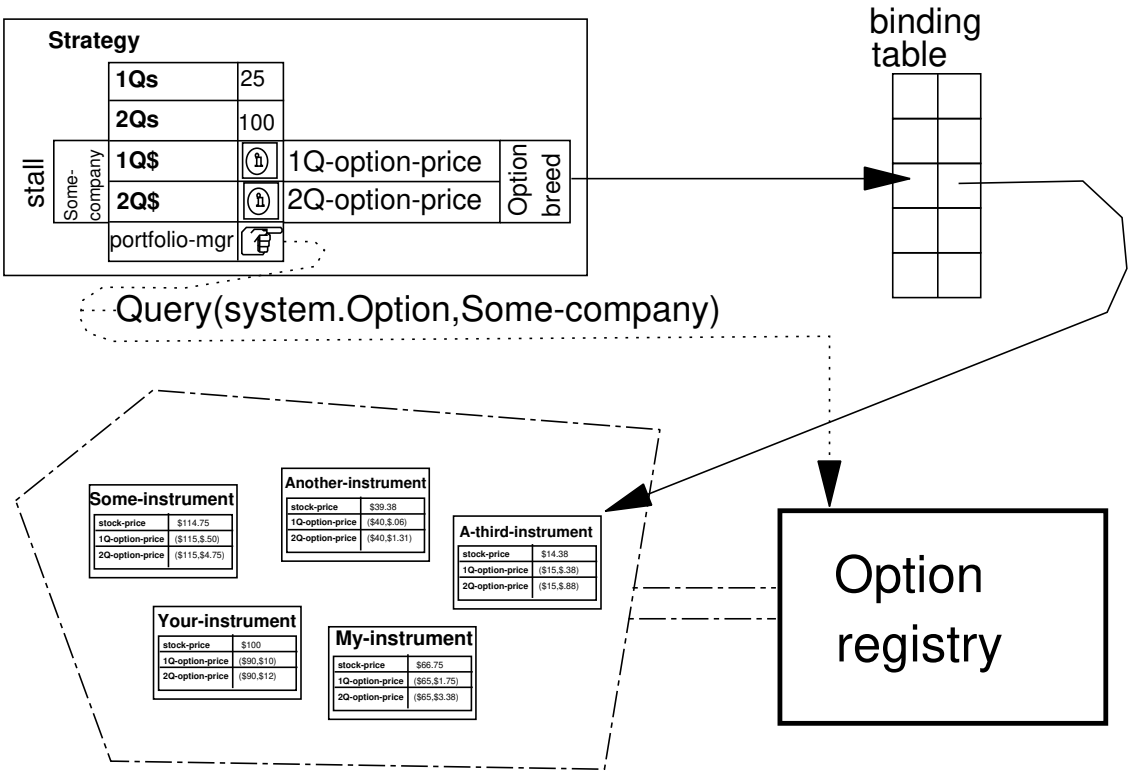

Figure 6-1: Facet Prior to Query

well as 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 a user-defined 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, the question arises as to how does a facet name one of these registries. Any facet that is designed to work with a particular registry can have the appropriate indirection statically built in. 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.

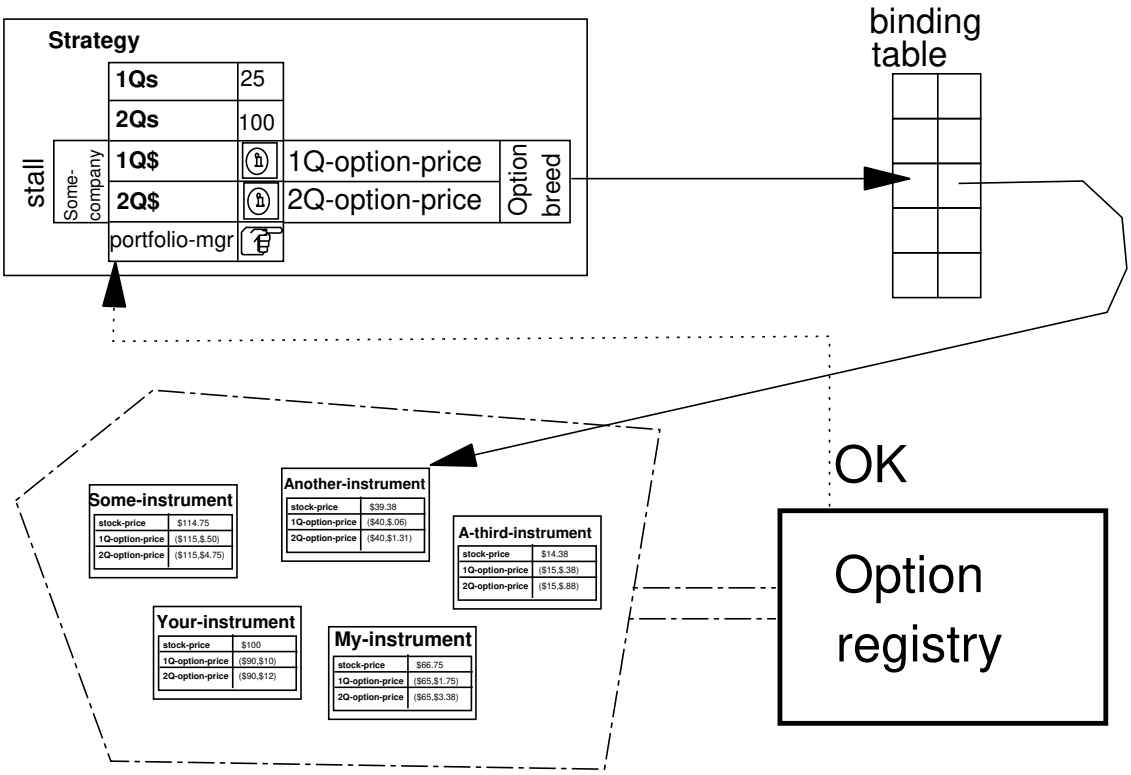

Figure 6-2: Facet After Rebinding

7. Ranchhouses

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.

The purpose of PROFIT, however, is to provide a static structure in which computation takes place. When dynamic reconfiguration is necessary, the programmer should create the new entities externally to the system and then combine them with the running system. This is the approach taken in operating systems, when new processes are added while the operating system is executing.

Thus the programmer adds new facets to a running PROFIT program by defining one or more new processes to contain the facets, compiling and linking it, and initiating execution of the process. The initialization of the new process includes communication with its peer processes to add the new facets to the appropriate registries.

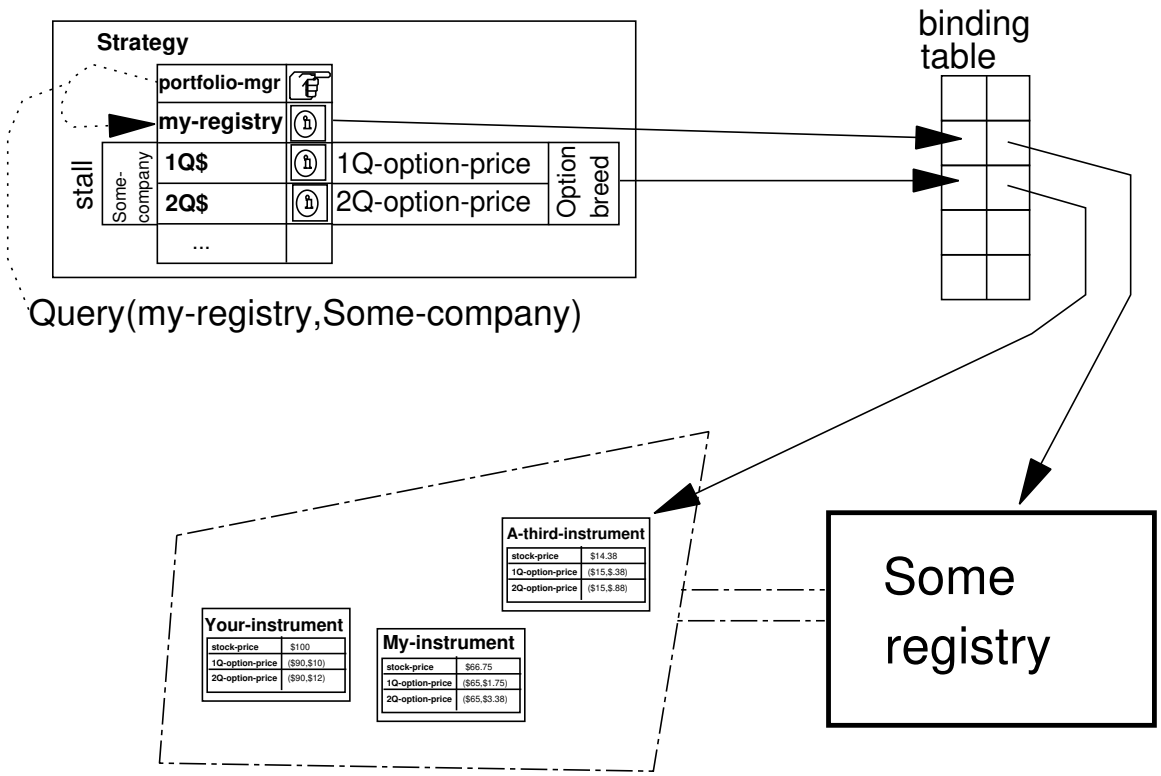

Figure 6-3: Query to User Registry

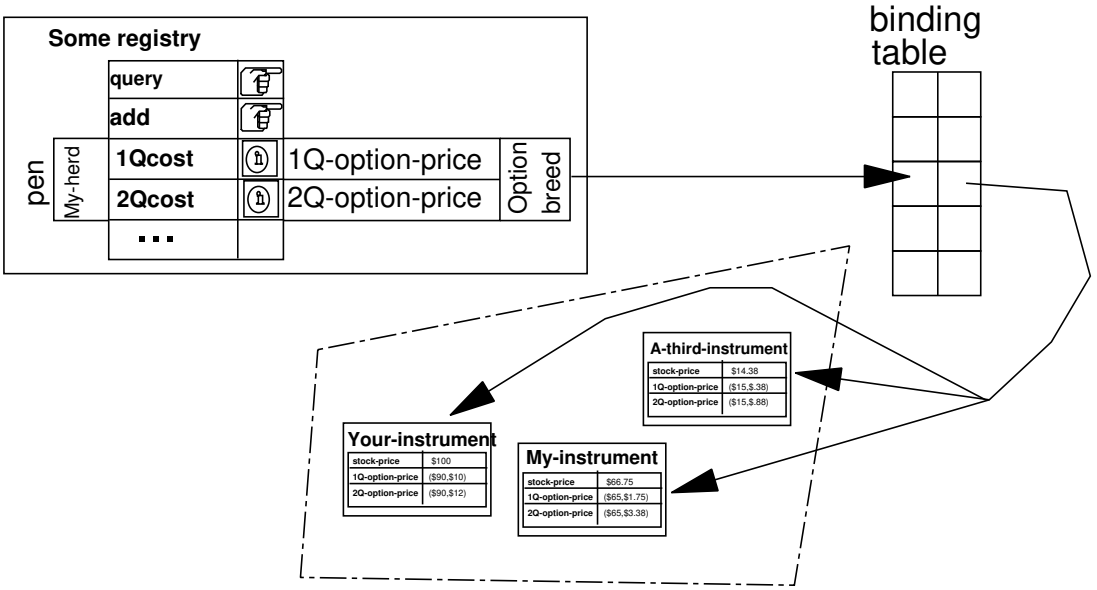

Figure 6-4: Inside a User Registry

New objects can be introduced independently, made up of new and/or existing facets, by generating new binding tables. The programmer then executes a special injection program, whose purpose is to install the

new binding tables in the appropriate running processes where the indicated facets reside.

Both of these techniques add on to an existing program without disturbing its existing structure or execution. This is analogous to how ranchhouses are constructed. An existing house is expanded by adding new rooms onto the end, with the previous outside door (or wall) becoming a door between the old house and the new section.

8. Implementation

There have been three subset implementations of PROFIT, called Asset, Bankrupt and Capital. The latest, Capital, provides facilities slightly different than those described here for adding facets to an executing program, but does not yet support incorporation of new objects or processes. Capital supports only a single process, although there may be multiple objects and shared facets, as well as rebinding of facet indirect slots during execution to previously existing or new facets.

The Capital implementation of PROFIT provides 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. The parser consists of about 3500 lines of C, 200 lines of lex rules and 600 lines of yacc rules. The run-time support consists of an additional 1300 lines of C.

To date, only a few toy stock trading simulations have been implemented in the various versions of PROFIT. A larger scale effort, called SPLENDORS, is now in progress based on Capital to provide a library of generic facets and objects for reuse in user portfolios; parameterization and inclusion of library components in application systems; an X windows interface for applications; and a specific stock trading application intended for use by non-programmer financial industry professionals.

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 dynamic reconfiguration and abstract interfaces (breeds), this section will describe work related only to these topics.

Dynamic reconfiguration is standard in operating systems and network management systems, 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 approaches are being developed to automatically change configurations, as well as to detect and correct performance bottlenecks and failures [Mazumdar 89].

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 carefully avoided by PROFIT.

The Orca programming language [Bal 88] is based on a shared data-object model, which provides reliable and efficient sharing of variables among the processes of a distributed application. There is no dynamic reconfiguration in the sense we describe here, addition of new entities and rebinding of existing entities, in a running system. There is an analogy, however, to migration and replication of data objects. Data objects are represented by a primary copy, for reads and updates, and arbitrarily many secondary copies, with creation of secondary copies and movement of the primary copy dependent on statistical information about usage patterns. The replicas are managed implicitly by the run-time system, and are invisible to the application programmer. In contrast, the placement of facets (shared or unshared) in processes is specified explicitly by the PROFIT programmer, and there is no migration or replication.

The MIT 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 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.

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. Emerald defines a complete type-checking discipline based on conformity of abstract types.

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. 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. 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, analogous to PROFIT breeds. 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, separately and independently. RPDE^3 does not support sharing between objects or concurrent programming.

Hailpern and Ossher [Hailpern 90] have extended the notion of abstract type to a view which 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 special controlled interfaces in an object-oriented system.

Dynamic reconfiguration as discussed in this paper does not address migration [Theimer 86], crash recovery [Strom 85], or general reflection [Watanabe 90].

10. Conclusions

We have presented a new approach to dynamic reconfiguration in on-line distributed applications based on a data sharing model. Our data sharing model consists of facets, objects and processes, with facets as the unit of sharing. Facets reside in a single process but may be shared among multiple objects, and the facets of the same object may reside in different processes. Facets can be written independently of the composition of objects as information hiding units and interface to each other through the binding table(s) of the containing object(s), as well as independently of the composition of processes as collocation units.

The primary contribution of this paper is a new metaphor for abstract types of entities, both facets and objects, that permits the addition of new entities while the distributed application is running. It also supports rebinding of interfaces among the entities. Breeds describe the facilities required by an entity, stalls are collections of slots that are bound to a member of the designated breed, and pens are essentially sets of stalls for representing open-ended collections. Registries provide the means for determining candidates for such bindings. Ranchhouses are proposed as the means for defining the new code that

extends an existing application with new facets, objects and processes.

Acknowledgments

Jason Kim and Isai Shenker contributed at one time or another to the development of Capital. Tushar Patel is responsible for the current implementation effort, and is being aided by Michael Mayer (for the library system) and Vanessa Cole (for the X windows interface). The SPLENDORS application is being developed as part of Tushar's MS thesis research, in consultation with Dan Schutzer of Citicorp and George Beltz of NYSE. Discussions with Terry Boult, Jim Donahue, Gary Herman, Catherine Lassez, Aurel Lazar and Harold Ossher have also contributed substantially to the development of the ideas presented in this paper.

References

[Bal 88] Henri E. Bal and Andrew S. Tanenbaum.
Distributed Programming with Shared Data.
In *International Conference on Computer Languages*, pages 82-91. Miami Beach FL, October, 1988.

[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.

[Donzeau-Gouge 87] V. Donzeau-Gouge, C. Dubois, P. Facon and F. Jean.
Development of a Programming Environment for Setl.
In H.K. Nichols and D. Simpson (editor), *1st European Software Engineering Conference*, pages 21-32. Springer-Verlag, Strasbourg, France, September, 1987.

[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), November, 1990.
Technical Report RC 14016, IBM Research Division, September 1988 (revised March 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.

[Kaiser 90a] 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 90b] Gail E. Kaiser and Brent Hailpern.
An Object-Based Programming Model for Shared Data.
Technical Report CUCS-046-90, Columbia University Department of Computer Science, May, 1990.
Submitted for publication.

[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.

[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.

[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 85] Robert E. Strom and Shaula Yemini. Optimistic Recovery in Distributed Systems. *ACM Transactions on Computer Systems* 3(3):204-226, August, 1985.

[Theimer 86] Marvin M. Theimer. *Preemptable Remote Execution Facilities for Loosely-Coupled Distributed Systems*. PhD thesis, Stanford University, June, 1986. STAN-CS-86-1128.

[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.

[Watanabe 90] Takuo Watanabe and Akinori Yonezawa. Reflection in an Object-Oriented Concurrent Language. *ABCL: An Object-Oriented Concurrent System.* MIT Press, Cambridge MA, 1990, pages 45-70, Chapter 2.