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

Brent Hailpern
IBM
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
914-784-6821/fax: 914-784-6201
bth@watson.ibm.com

Gail E. Kaiser
Columbia University
Department of Computer Science
500 West 120th Street
New York, NY 10027
212-939-7081/fax: 212-666-0140
kaiser@cs.columbia.edu

23 February 1993

Abstract

A distributed application ideally allows reconfiguration while the application is running, but changes are usually limited to adding new client and server processes and changing the bindings among processes. In application domains involving *rapidly changing data*, it is often necessary to support finer grained reconfiguration at the level of entities smaller than processes, even without operating system support for dynamic linking. We present a scheme for special cases of fine-grained dynamic reconfiguration sufficient for a range of application domains 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 while Prof. Kaiser was an Academic Visitor at the IBM Thomas J. Watson Research Center. Kaiser was supported by National Science Foundation grants CCR-9106368, 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 type, distributed system, late binding, network management, portfolio management, soft real-time

1. Introduction

This research is motivated by the problem of *rapidly changing data* in a distributed environment, as arises in many real-world application domains. 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., [28]).

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. Multiple portfolios will refer to the same instruments, but have separate criteria for when price changes are significant to their 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 previous papers [15, 16], 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 each daemon 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 (or notification), 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 [24], machine vision [3] and animation [10], with similar characteristics.

We have developed a language, called PROFIT (PROgrammed FInancial Trading), based on this model. PROFIT is a coordination language [6] (in the sense of Linda [4]) 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 financial instrument available. This collection of facets executes 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 processes that reside on a distinguished database machine 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 cost tradeoffs.

In our previous papers, we described a subset of PROFIT that supported only early binding of facets into objects. In particular, each object was defined as consisting of a static set of facets. This simplification allowed us to focus on issues of sharing, delegation and interfaces by avoiding the complications of dynamic system issues. 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 application.

In this paper, we relax the early binding paradigm and describe how PROFIT supports 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 rapidly changing data applications without abandoning the benefits of early binding.

Three main concepts enable dynamic reconfiguration in PROFIT: breeds, stalls and pens (the ranching metaphor is explained later). Breeds are abstract types [2]: they define the sets of facilities required of those facets that can be substituted for each other in particular contexts. Stalls and pens are similar to variables and sets, respectively: stalls ‘‘hold’’ a single facet and pens ‘‘hold’’ a collection of facets; in both cases the facet or facets being held can be changed during program execution. These 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. It is possible to change the static organization of an executing program, but without going so far as to require generic dynamic linking from the operating system or interpretation from the programming language implementation. Throughout the rest of this paper, we refer to the subset of PROFIT described in our previous papers as PROFIT_0.

One way to aid the application programmer in applying these concepts would be to implement a library of subroutine calls in some conventional programming language, such as C or C++. Another possibility would be to integrate the concepts into a uniform level of abstraction as part of a new programming language. We chose a hybrid approach: we maintain traditional data structures and control flow from a conventional computation language, but add pervasive new constructs representing our programming model as a coordination language. Relying on an existing computation language allows us to focus on the innovative aspects of our model. An additional advantage of this language design technique is that a prototype implementation can be written as a preprocessor plus run-time libraries rather than a full-scale compiler.

We start with an overview of the PROFIT language, including the concepts that support dynamic reconfiguration, and then discuss related work. In the subsequent sections, we present in detail the PROFIT programming constructs that support dynamic reconfiguration, followed by run-time issues. Throughout the paper to that point, we use the stock trading application as a running example, but then briefly consider a network management example. We describe the existing implementation of PROFIT, which includes a simpler facility for more restricted dynamic reconfiguration. Though we describe several fault tolerance issues as they arise, we do not address mechanisms to handle process failures and network partitions in this paper.

2. Overview

The PROFIT programming model supports sharing data among objects in a distributed environment. There are three central 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 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: processes can contain parts of many objects and objects can be distributed over many processes. An object might consist of only one facet, or all the facets in the system might be part of the same object. Analogously, a process might contain a single facet, or all facets might execute within the same process. Our intent is to allow these extremes, as well as a variety of intermediate points on the spectra. One possible organization is illustrated in Figure 2-1.

We briefly sketch these PROFIT language concepts below. A more complete treatment, with detailed financial examples, may be found in [16].

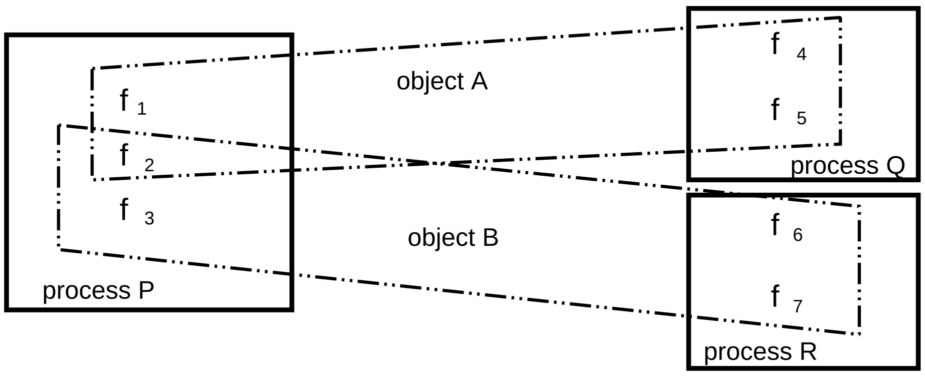

Figure 2-1: Facets, Objects and Processes

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 (e.g., a C datatype or `void`). Procedure slots must be equated to specific procedures (e.g., C functions) at compile-time. Within one of these procedures, the program text can refer to slots in the same facet (both data and procedure) via extended syntax (e.g., PROFIT additions to C). Data slots may optionally be initialized to a specific value determinable at either compile-time or run-time; if not explicitly initialized, data slots are implicitly set to null values. Evaluating a data slot returns its 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 [34].

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 same 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

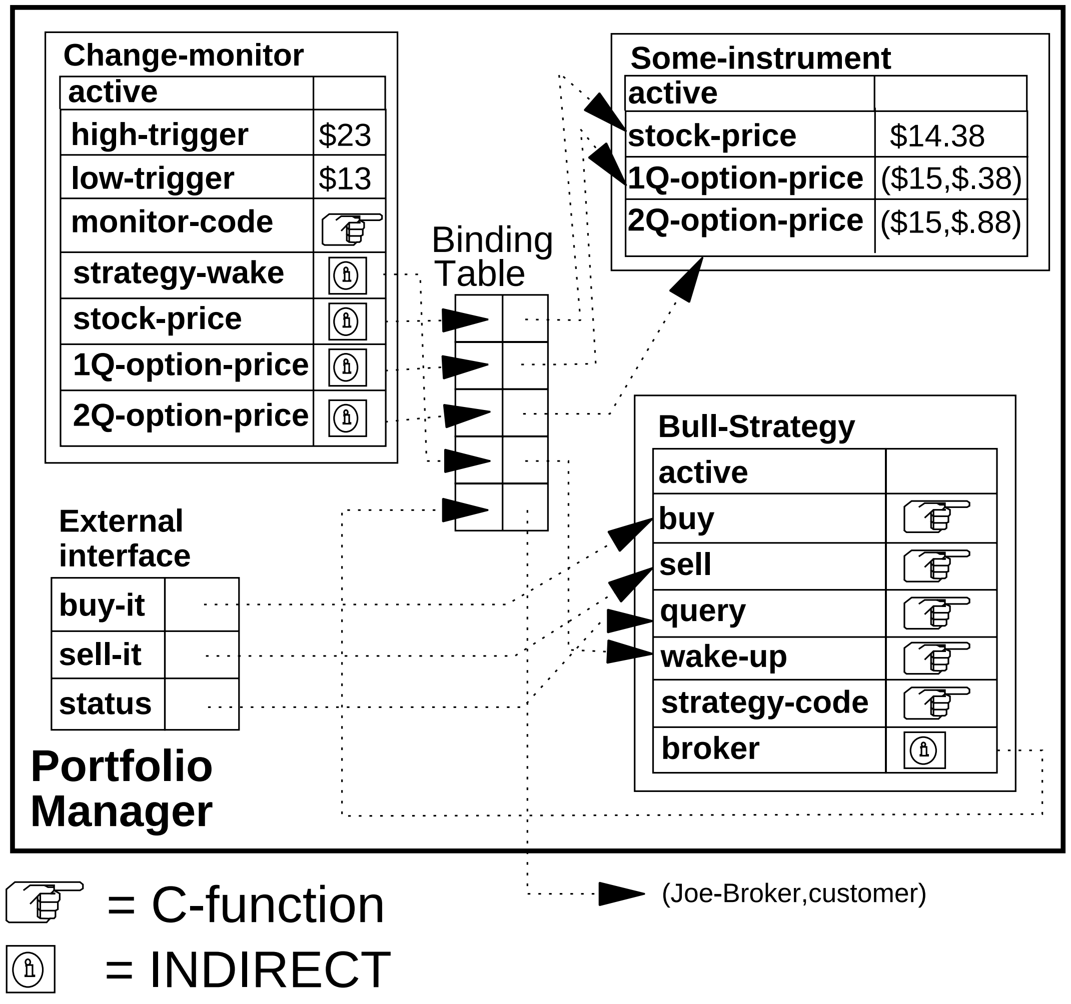

Figure 2-2: Generic Object

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 every 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).

The example portfolio management object in Figure 2-2 shows 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 (e.g., 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 the facet's 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.[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 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).

So far, we have considered only the case where a facet is part of exactly one object, and thus there is

[1]This description has been simplified for ease of presentation; PROFIT also supports asynchronous message passing and priorities.

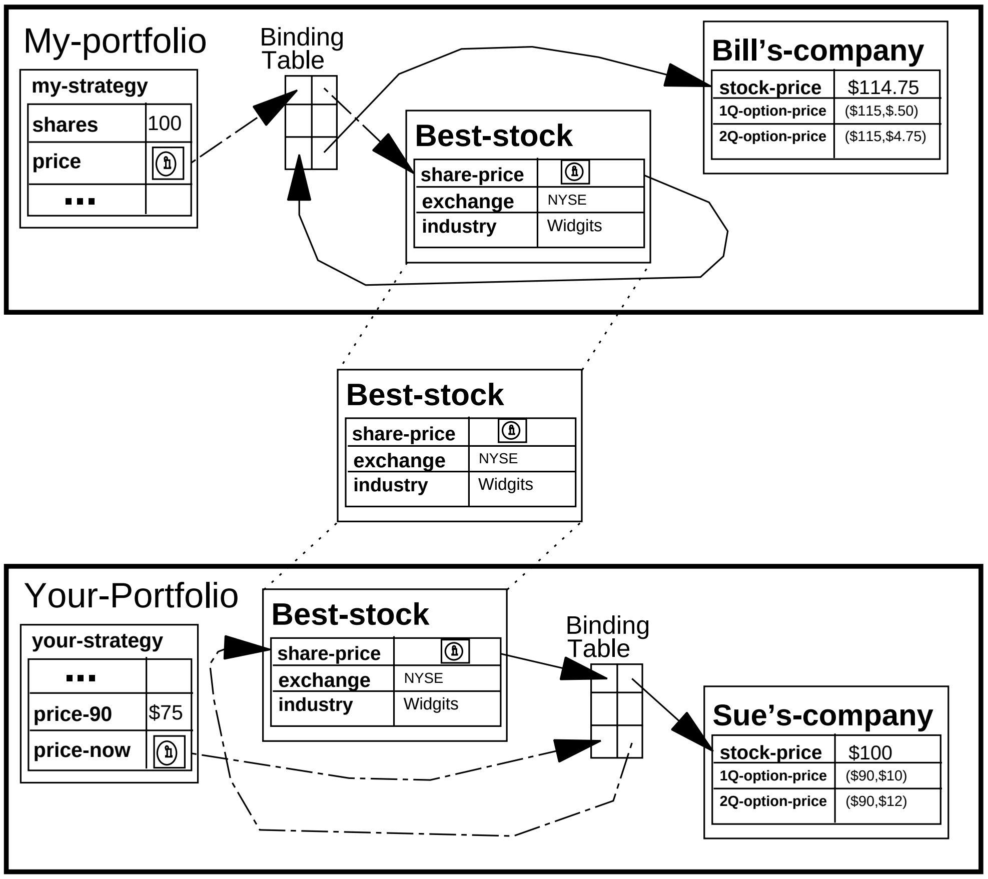

Figure 2-3: Shared Facet

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 independently 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 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 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 relies on medium-weight threads similar to Sun’s lwp package [33], permitting multiple threads to execute within the same process, i.e., the same address space. 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 at 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 at 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 (and thus the thread’s stack), 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
```

2.4. Dynamic Reconfiguration

In our earlier papers, we described the PROFIT$_0$ subset assuming a fixed, static set of facets, objects and processes determined at compile-time. All entities were statically allocated, and no new facets, objects or processes could be added to the program. All connections among entities were also statically determined; specifically, a portfolio manager could not substitute one instrument for another or change the number of instruments in the portfolio. Limiting discussion to this subset permitted us to concentrate on the programming model without concern for run-time interface checking, locating and accessing facets, and so on.

The new result presented in this paper, called *dynamic reconfiguration*, reflects the relaxation of these restrictions to allow limited changes to the static structure in order to support important special cases motivated by the application domains we have studied. Our approach is to support only a small number of well-defined changes to an executing program, providing high leverage with low overhead.

The PROFIT coordination language supports 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 composition can be changed, and addition of new facets, objects and processes to an executing program. For example, this allows us to add new instruments to the prices database, change the composition of existing portfolios, and add new users and their portfolios.

Substituting one facet for another could be handled by making the binding table a first-class entity that can be explicitly modified during execution. Similarly, sets of facets could be handled using the data structuring features of the underlying computation language. These design decisions would imply the manipulation of low-level implementation details, equivalent to accessing stack pointers and contents of registers.

Throughout our work on PROFIT, we have deliberately avoided the direct manipulation of 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. For example, one could extract the tenth through thirteenth words offset from the beginning of a facet.

In the PROFIT$_0$ subset, explicit pointers or identifiers were not an issue, because all facets were bound statically and all references implicitly went through the statically defined 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.

The facilities we present for facet substitution and sets of facets maintain this programming model. We introduce the notion of a binding table whose mappings may change, but only *implicitly* as a side-effect of other operations to be presented in this paper. It is impossible to access the binding table explicitly.

At first glance, it would seem that adding new facets to an executing program could be solved by dynamic storage allocation (although it should be noted that not all computation languages support dynamic allocation, e.g., Cobol and Fortran). However, recall that every facet has its own data and procedure definitions, hence it is not just another instance of a pre-defined class. Thus adding a facet can require that totally new code be added to an executing program. Adding new code to a running process is feasible for many interpretive languages, and is possible even for compiled languages when the operating system supports dynamic linking. But we intend PROFIT to act as a coordination language for computation languages and operating systems without these facilities.

On the other hand, the addition of new processes to an executing program is no more difficult than the process-level dynamic reconfiguration described in our next section on related work. We exploit this fact to enable the addition of facets, by placing all new facets in new processes. These new facets can be incorporated into existing objects, and new objects can acquire existing facets, through the substitution and set mechanisms described below.

Dynamic reconfiguration can be viewed as a form of delegation [22]: when one object receives a message it cannot handle, it passes on or delegates the message to another object. There is nothing inherently static in the delegation concept. PROFIT indirect bindings effectively implement delegation, and hence changing the bindings among facets makes the delegation dynamic.

PROFIT's dynamic reconfiguration capabilities are built on three main concepts:

- *Breed*, a partial interface description or abstract type. A breed represents a service that is provided by every facet or object that conforms to the description. The description is partial in the sense that a facet or object may provide additional services, and hence belong to more than one breed.
- *Stall*, a collection of facet slots that can be mapped to the corresponding slots in any one facet or object (called the stall's *occupant*) belonging to an associated breed, and later remapped under program control to a different facet or object. A stall provides a way of expressing the requirement that a number of slots in a single facet map to corresponding slots in exactly one other facet or object, as opposed to separately binding each of the slots to perhaps a different facet or object.
- *Pen*, a repository for a set of facets or objects (collectively called a *herd*) that all belong to an associated breed. A member of a herd is called a *constituent*. A pen allows grouping together of a variable number of facets or objects selected from a breed, where the contents of the pen can change over time. Operations on a pen include iteration and associative queries.

Breeds and stalls provide a simple facility for changing an executing program: replacing a ‘‘client’’ facet's binding from one ‘‘server’’ facet to another ‘‘server’’ facet within the containing object's binding table. Pens and herds add the ability to operate over changing collections of facets. We chose this terminology because more conventional terms like type, class, abstract type, abstract class, 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.

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 can also be members of a breed. Thus an 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.

PROFIT's run-time system provides a *registry* for each breed. A registry is a pen that automatically contains all the facets and objects that are members of the corresponding breed, and thus supports associative queries across all the members of a breed. New processes are added to an executing application in the style of *ranchhouses*, where a new ‘‘room’’ can be added on to the end of an existing ‘‘building’’.

3. Related Work

There is extensive related work on object models, which we describe in our previous papers. Since this paper concentrates on abstract interfaces and dynamic reconfiguration, this section addresses work related only to these topics.

3.1. Abstract Interfaces

Emerald [2] 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. 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 static type-checking discipline based on conformity of abstract types. Though Emerald permits subtyping, based on type conformity, it does not support code-sharing inheritance. Abstract types in Emerald are equivalent to PROFIT breeds; we chose the term ‘‘breed’’ over ‘‘abstract type’’ because the PROFIT model diverges from that of Emerald in most other respects. Furthermore, we wished to avoid confusion with the similar term ‘‘abstract class’’ (a class that can be subtyped but not instantiated).

Objects are dynamically created in Emerald by an object-constructor object, rather than instantiated from a class or cloned from a prototype. 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). Emerald 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. Location information is never exposed in the programming of a facet.

Beta [21] also provides a concept similar to breeds. It defines ‘‘patterns’’ that are analogous to classes: Beta objects are instances of patterns. Patterns are explicitly organized in a classification hierarchy, by declaring one pattern to be a subpattern of another. *Virtual patterns* are a generalization of virtual functions, making it possible to delay the specification of an attribute, thus allowing attributes to have different bindings in sibling subpatterns. Virtual patterns serve a purpose similar to that of breeds, in that they provide a partial description of patterns. However, breeds support substitution of any PROFIT facet or object that meets its abstract interface, while Beta requires objects to be explicitly declared as instances of a pattern.

RPDE[3] [12] is a 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, 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. 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.

This two-dimensional architecture allows extension in both the functional and data domains to be accomplished by <u>addition</u> 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 is not intended to support concurrent programming, although the underlying object store can be shared.

Hailpern and Ossher [11] have extended RPDE^3’s roles to *views* that include interface specifications, servers and clients. They present a framework for describing different inheritance and delegation mechanisms, and for orthogonally incorporating security, priority, and controlled interfaces in an object-oriented system. Their views also permit the dynamic changing of the set of clients and operations over these changing client sets.

Helm, Holland and Gangopadhyay [13] present ‘‘contracts’’ that define a set of communicating participants and their contractual obligations with respect to the larger grained abstraction represented by the contract. The participants correspond to facets and the instances of contracts to PROFIT objects. In contrast to PROFIT objects, which specify only bindings among indirect slots in facets, contracts give the patterns of communication or protocols among participants. *Contractual obligations* consist of type obligations, which are equivalent to breeds, and causal obligations, which capture the behavioral dependencies between objects. These dependencies are expressed as ordered sequences of sending messages and setting instance variables, an approach useful for implementing simple composition paradigms such as Model-View-Controller [19]. However, the prescription of externally specified protocols (i.e., not embedded in the code) for accessing indirect slots in PROFIT programs would unnecessarily constrain the programmer as well as introduce substantial run-time overhead to validate procedure calls and setting of data slots.

3.2. Dynamic Reconfiguration

Dynamic reconfiguration is common in operating systems and network management systems. In both cases the resources to be managed 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, however, it has become necessary to automate these processes. New management and control approaches are being developed to automatically change configurations as well as to detect and correct performance bottlenecks and failures [25, 20].

The SOS operating system [31] 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 not needed by PROFIT.

The Mercury system [23] 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 [14] is similar to Mercury. It provides an interface specification language for heterogeneous distributed systems. Hermes [32] 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 anticipated in the original program.

Conic [17] 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 special *operations*: 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 important ways. Conic supports linking between typed ports, but provides no syntactic framework (such as stalls) for specifying that sets of ports must map to the same target module. Conic does not provide sharing of data between task modules, except for the optimization that messages between tasks within 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 the facets in a new process.

Wei and Endler [35] 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.

The Polylith distributed programming system [29] 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 a series of edits to describe the change, and applying the change atomically. PROFIT changes work similarly: we first describe the set of changes, and then atomically make the necessary updates to the executing program. 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. In other words, entering a reconfigurable state can be expensive in Polylith but is cheap in PROFIT [30].

Craft [7] describes a resource management system for a processor bank. The system selects a resource according to the attributes desired by the user, which are generally a subset of the catalogued attributes of the resource. There is a central resource repository, where clients can register the services they are willing to provide. Resources are matched up with users dynamically, as needed, and may even be created on the fly when no existing resource matches a user query. The resource repository maintains a set of *prescriptions* for constructing new resources, which involves forking of processes with particular memory images onto specified machines. Dynamic allocation and (re)configuration of resources is at the operating-system process level of granularity.

Frieder and Segal [9] 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 that does have an activation record 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. Even externally, it is only possible to substitute new procedures for old procedures, by massaging the interfaces. It is not possible to make structural changes in the program.

Actors [1] represent the ultimate in a dynamically reconfigurable programming language. Each operation potentially changes the actor that executed the operation into a brand new actor with a completely different behavior. Actors are associated with mailboxes, and messages intended for actors are sent to known mailboxes. Thus an actor *A* can send a message, and by the time a reply appears in the mailbox corresponding to *A*, an entirely different actor *B* can be processing the messages from that mailbox. Actors correspond roughly to PROFIT facets. Actors can be organized into communities, analogous to PROFIT objects, where ‘‘receptionist’’ actors accept all incoming messages and delegate them to the appropriate local actors.

4. Breeds and Stalls

In PROFIT, each operation is executed by invoking a procedure slot or accessing a data slot. When a slot is declared ***indirect***, the enclosing object must provide a binding to a slot in another facet or in the interface of another object. This referenced slot must have the identical type signature as the original slot.

Breeds and stalls permit bindings to change over time while retaining compile-time interface checking. Breeds provide the mechanism for declaring those facets that can substitute for each other. 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; contrast this with Figure 2-2, where `1Q-option-price` and `2Q-option-price` are separately mapped to the corresponding slots in `Some-instrument`.

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. For example, we can define any facet with data slots `1Q-option-price` and `2Q-option-price` to be a member of the `Option` 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 issue is discussed further in Section 5.

Breeds provide a convenient abstraction for large distributed applications that evolve over time. Facets and objects that conform to the breed specification may be interchanged without concern for implementation details, and in fact each member of a breed has its own independent code. Classes, in contrast, presume code sharing — the internal structure and operation code is defined once in the class rather than separately in each of its instances. Types need not imply code sharing, but specify complete interfaces. Breeds allow one to describe only the service required without restricting the object from providing other services. Whereas breeds are static, protocols are dynamic: they prescribe sequences of operations, which could unnecessarily limit the use of the provided services.

By relying on breeds, PROFIT circumvents the problem of interface mismatches between a client and its new server. An interface mismatch could arise when the new occupant of a stall did not provide the same facilities as the old one. 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.

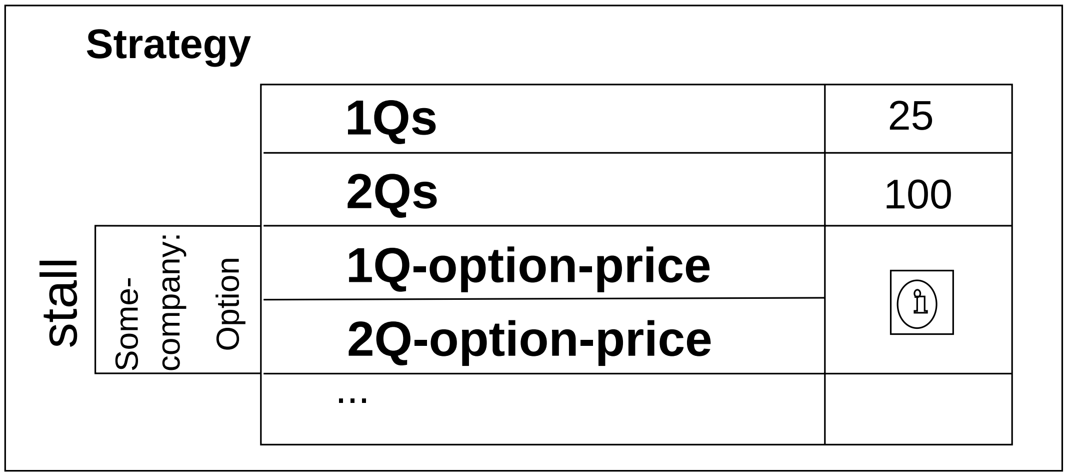

Figure 4-1: Generic Stall

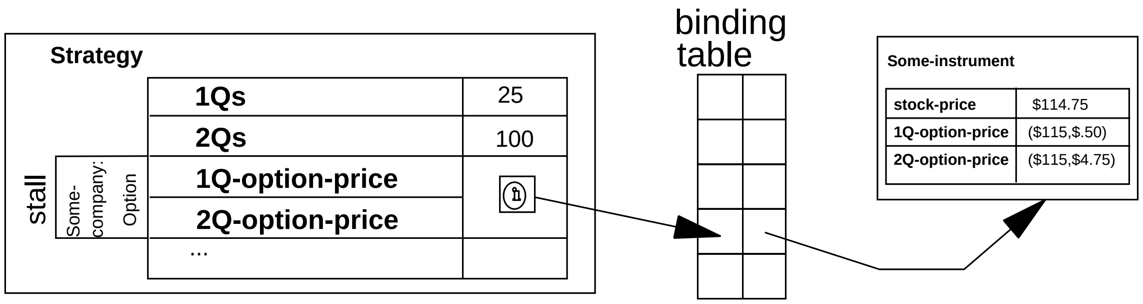

Figure 4-2: Binding an Occupant to a Stall

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:

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

The `Some-company` stall is essentially an `Option`-valued variable.

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 the statement following the call. 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’s 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. Conic uses this scheme in its ‘‘patient monitoring’’ example system [18].

The limitations of this approach 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 this would require complicated storage management by the application programmer. 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 many animals, collectively called a herd. Thus a *herd* is a collection of facets belonging to the same breed; each member is called 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
```

The `Many-company` pen is essentially a set variable containing `Options`.

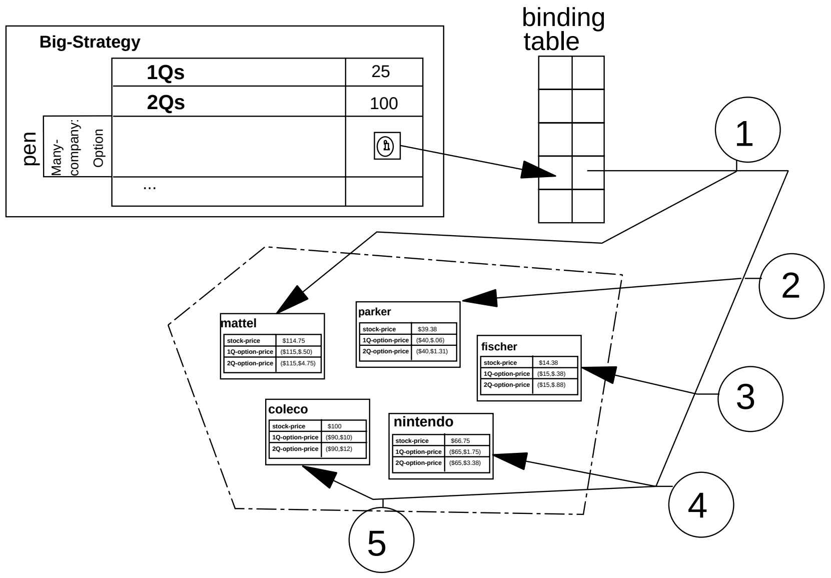

Figure 5-1: Binding a Herd to a Pen

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.

PROFIT provides two kinds of operations on pens: iteration and associative queries. Iterating over a is accomplished using a SETL-like ‘‘forall’’ statement. The key feature of this statement is that each

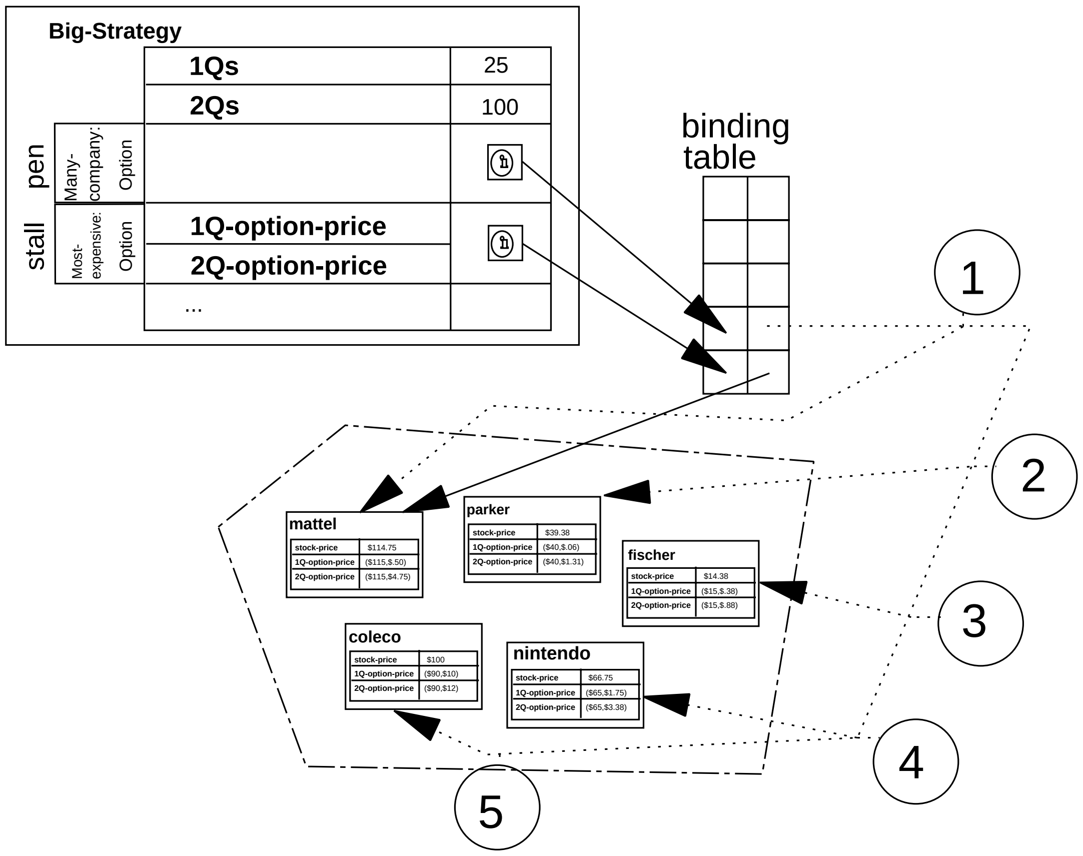

Figure 5-2: Iterating through the Members of a Herd

element of the set is accessed once and only once. This approach is 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). 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.[2] The once-and-only-once property is important since few operations are idempotent (i.e., imagine buying/selling shares), thus an ordered multiset or sequence would be an inappropriate data structure.

In order for a constituent of a pen to be manipulated, it must first be selected. The resulting constituent is housed in a stall for reference during the actual manipulation. For example, in Figure 5-2, the

[2]An alternative approach would be to *brand* each constituent of the pen as it is visited, but this would require the overhead of de-branding after each complete iteration and when any constituent is removed from the pen.

`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 relevant slots of the new occupant (`mattel`) can be accessed as "Most-expensive.<slotname>". The case of an unsuccessful query is discussed later.

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 at most element (a non-deterministic choice if there are multiple possibilities). Additional relational operators could be defined, as long as they refer only to the slots specified in the breed. But efficient implementations might be difficult due to the distributed nature of herds. For example, since an object's binding table is replicated in every process that contains any facet of the object, the manipulation of the stalls and pens represented through the binding table should be executed using a consistency-preserving technique such as two-phase commit. Further discussion of this topic is beyond the scope of this paper.

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 would be 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 from PROFIT. In case the base computation language has no specific facility, however, the PROFIT design includes the following simple mechanism: every stall has an associated ‘‘empty’’ bit, treated as a built-in data slot. This bit can be tested prior to accessing the stall, for example,

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

It is generally impossible to prevent access to a slot in an empty stall 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. 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.

One anomaly with associative queries is that the contents of facets and objects in a pen may change during a query. For example, while querying a prices databases for the instrument with the highest yield, the yields of any and all of the instruments may be changing; in particular, instruments that have already been checked and dismissed may increase their yields. This issue arises because the classical transaction model [8] is not applicable to rapidly changing data, because the updates to this data are generated outside the control of the system. We briefly address this difficulty in a previous paper [16].

6. Registries

We have discussed the idea that the occupant of a stall or the constituents of a herd can change. The questions arise as to how does a program ‘‘know’’ what facets are available. 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 the portfolio’s owner 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 problem is to provide a ‘‘yellow pages’’ directory that keeps track of all the services provided by entities in the program and the means for accessing the corresponding entities. In PROFIT, this role is fulfilled by *registries*, each a system-defined pen that contains all the members of a particular breed. Registries can be queried in the same manner as pens, and they return one or more registered members. Because registries are automatically maintained by PROFIT, applications cannot execute the `Add` or `Remove` operations. A similar approach is described by Craft [7], where computational resources can be associatively accessed in a processor bank.

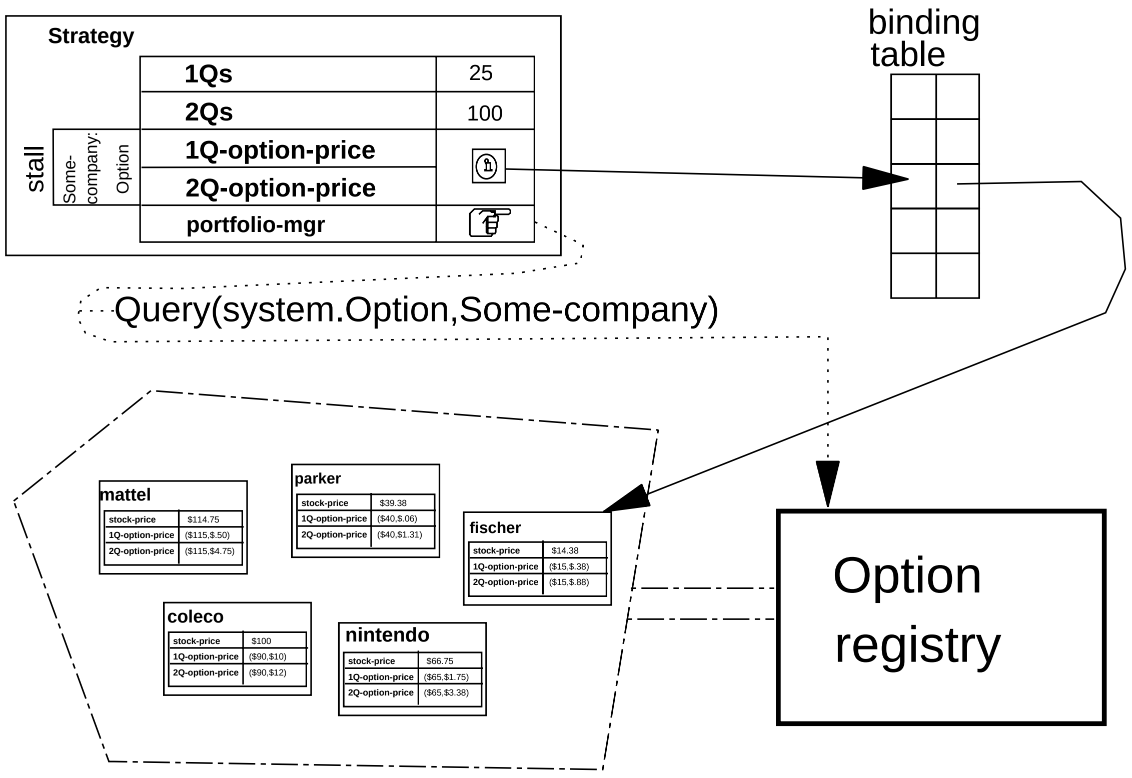

Figure 6-1: Facet Prior to Query

The PROFIT system provides a *registry* for each breed. When a new facet is created (we explain how in the next section), it is automatically added to all the appropriate registries — there may be more than one appropriate registry since a facet can be a member of multiple breeds. This is similar to classes in object-oriented databases, where the class is not simply a type definition but also a repository for all its instances. Subsequent queries on these registries can place the new 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.

7. Run-Time Issues

The discussion so far covers sufficient facilities to reconfigure objects and facets within an executing 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 pen. Previously unconsidered stocks, resident in the prices database, can be evaluated using a registry. However, we have not yet described how one can construct entirely new portfolios or to add new instruments 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, for example, by executing the

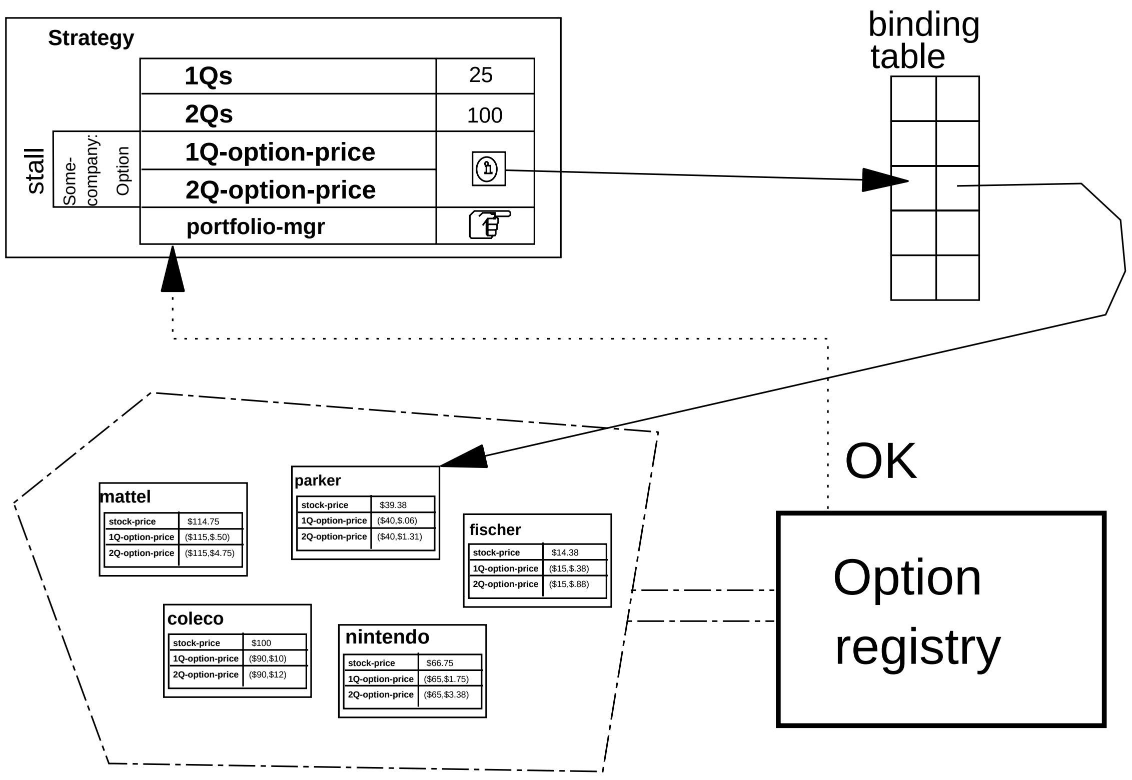

Figure 6-2: Facet After Rebinding

`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 that the new entity be an instance of a known type, with no new code of its own. PROFIT facets and objects, in contrast, stand on their own. Without interpretive or dynamic linking facilities, it is not possible to add new code internally to an executing process.

Rather than rely on dynamic linking from the operating system, the PROFIT 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 given 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 to register new facets and objects, 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 as the coordinator for the program, 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 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 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, including bindings within the new objects to old facets and old objects. But the programmer cannot define new breeds, modify existing breeds, modify existing facets, change the external interfaces or binding tables of existing objects, or change the composition of existing processes. The collection of new facets and objects are declared in a new process called a *room*. A 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 the processes that it coordinates. 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; this is enforced by the ranchhouse.

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 unrelated 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.

Because the 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 appropriate, 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, to update the 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 modified program named in the room definition). The process must contain exactly those facets declared in the room.

```
ROOM With-A-View
MODIFIES PROGRAM BunkHouse;

<facet definitions>

OBJECT Fenster
FACETS: Pane, Glass, Latch, Sash;
...
END OBJECT Fenster

OBJECT Fenetre
FACETS: Glass, Frame, Blind, Shade;
...
END OBJECT Fenetre

PROCESS Skylight
FACETS: Pane, Sill, Latch, Sash, Glass, Frame;

start := Latch.Unlock();
<error handling>

END PROCESS Skylight
END ROOM With-A-View
```

The purpose of the ranchhouse mechanism is to provide language-level syntax for describing the run-time connection of new facets and objects with an existing program. Other languages, such as Self [5], have also taken the approach of providing their own language-specific dynamic linking. If the underlying operating system provides dynamic linking, however, that may provide a more efficient implementation.

8. Network Management Examples

Imagine a small network with three workstations, two printers and a file server, as illustrated in figure 8-1. Two breeds, `Printer` and `Device`, are shown in figure 8-2. We graphically depict two applications, a print manager that allows selection among the available printers (figure 8-3), and a network monitor that shows the status of the various devices on the network (figure 8-4). We represent each node in the network with a facet, and show a subset of this facet's slots next to the node in the

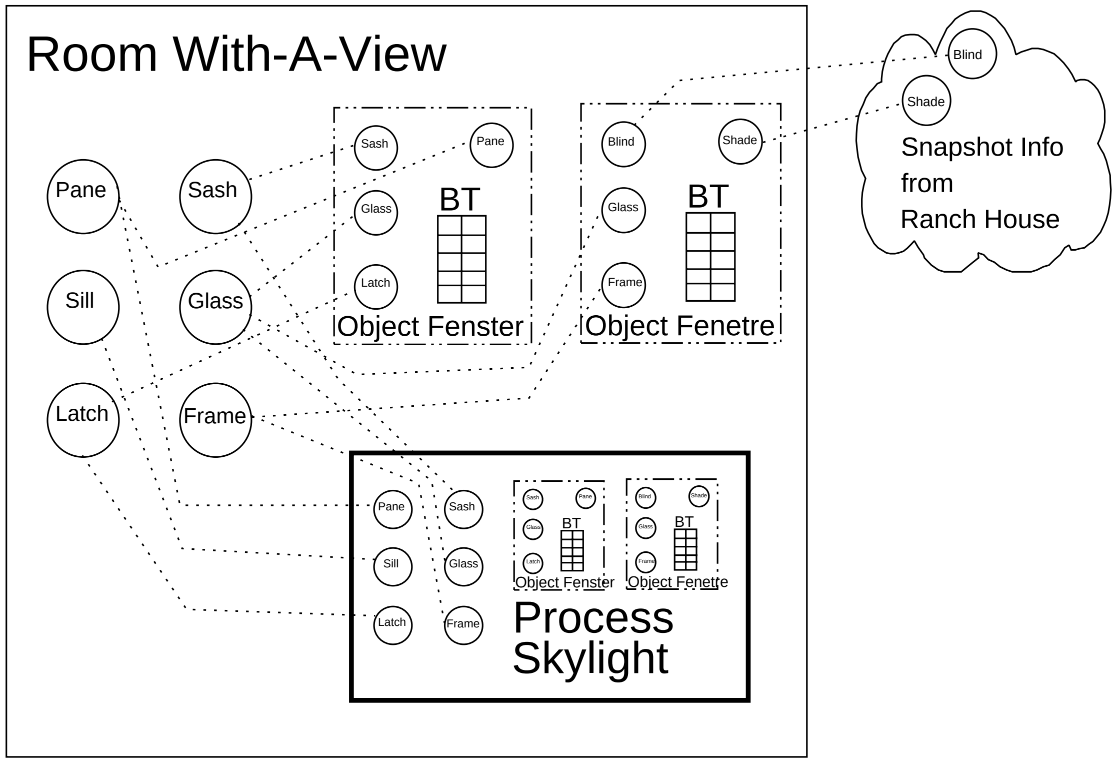

Figure 7-1: Compiling a New Room

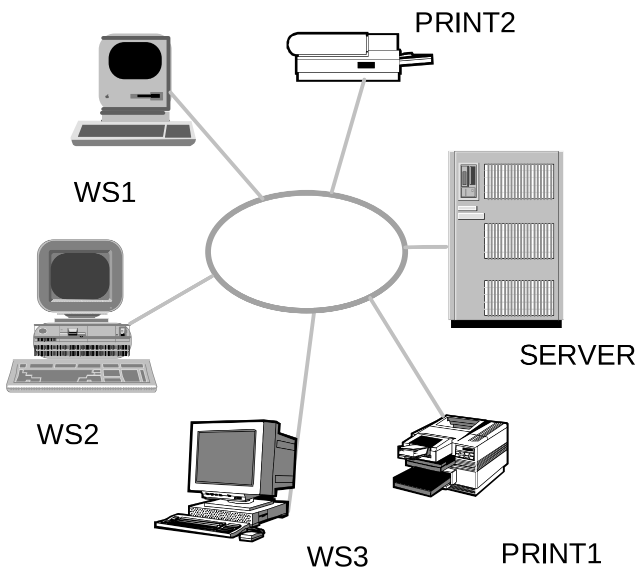

Figure 8-1: Small Network

picture. Those slots that are used in the application are highlighted.

```
BREED Printer
   name: char *;
   location: char *;
   spool: char *; /* directory */
   printqueue: queue;
   errorcode: int; /* 0 ok, 1 jammed, 2 tray empty, etc. */
   enqueue: ... /* function to print */
   dequeue: ... /* function to cancel */
   query: ... /* function to display printqueue */
END BREED Printer

BREED Device
   ip: char *;
   name: char *;
   status: int; /* 0 down, 1 up */
   util: float; /* utilization */
END BREED Device
```

Figure 8-2: Network Breeds

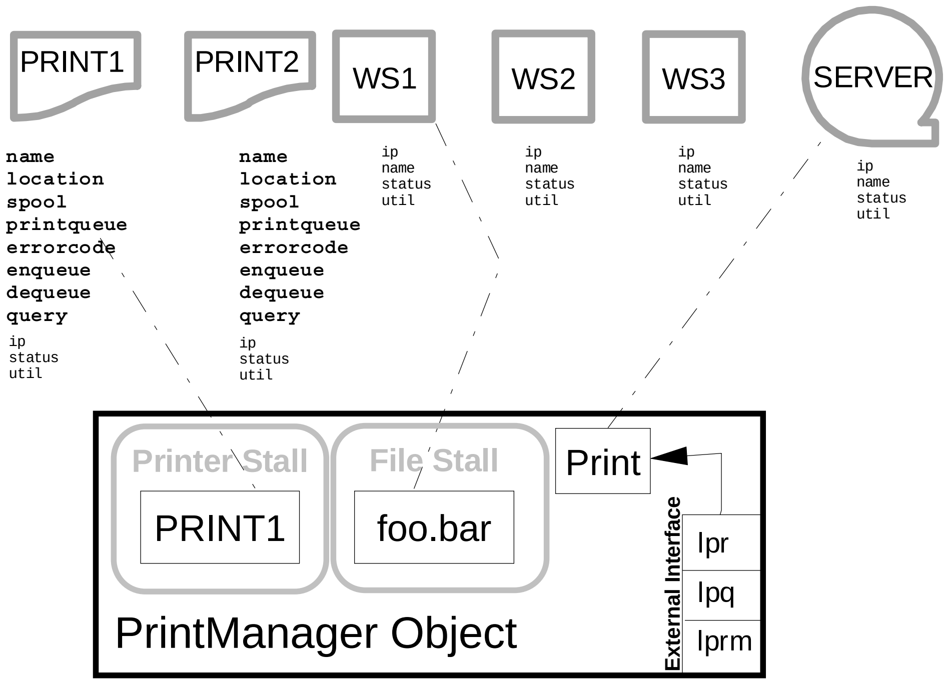

Figure 8-3: Print Manager

The `PrintManager` object in figure 8-3 contains facets with two stalls representing the file to be printed and the printer. The object's interface includes entries to print, display and cancel. To request that a file "foo.bar" be printed, the caller invokes the `lpr` command with the name of the file and, optionally, the name of the printer. The `PrintManager` fills the file stall with the directory entry for "foo.bar". If the `lpr` command specified a printer, then the printer stall is filled with the corresponding node; otherwise, the default printer is used. The `PrintManager` then calls that printer's `enqueue` procedure slot to send the file to the associated print spooler. From the point of view of the PROFIT programmer, the facets representing the file and the printer are local to the `PrintManager` object and can be manipulated directly, even though these facets may be located on different machines as shown in the figure.

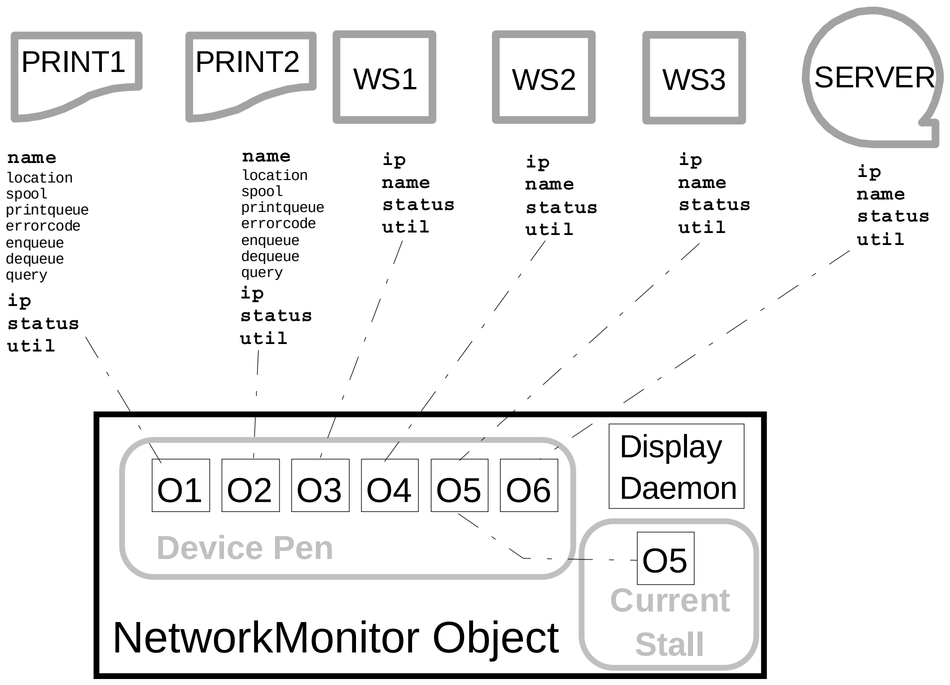

Figure 8-4: Network Monitor

Figure 8-4 shows a `NetworkMonitor` object containing facets with a `Display` daemon, a pen collecting all the devices being monitored, and a stall representing the one device whose displayed information is currently being updated. The display daemon repeatedly iterates through the `Devices` pen, filling the `Current` stall, to refresh its display. A more complex network management system could associate a different daemon with each device. This might be represented as a fixed number of daemons in an array, corresponding to monitor windows, or by a dynamically sized pen of composite objects each representing a daemon together with its device information. Again, all the entities in this complex distributed application can be manipulated by the PROFIT programmer as local variables and procedures.

9. Implementation

The current PROFIT implementation is limited. It supports only a single process, although there may be multiple objects and shared facets. PROFIT 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 pre-empt 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 SunOS 4.1.1 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 [26], has been completed. 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 includes 551 lines of PROFIT coordination code and 6148 lines of C computation code (1541 lines of which are for the user interface).

SPLENDORS uses a different facility for dynamic reconfiguration than described here. Since the current implementation supports only a single process, it was not possible to use our ranchhouses scheme of adding on new rooms. Instead, a partially interpretive approach was followed [27]. The PROFIT 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 uses a simple menu-based interface to provide parameters to the interpretive price and daemon facets, and to tailor generic source code stored in a file. 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 represents the bulk of the SPLENDORS application system.

10. Summary

We have developed a new approach to dynamic reconfiguration in on-line distributed applications based on a data sharing model. Our 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 different 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).

The previously published subset of our language design featured static allocation of facets. The primary contribution of this paper is the presentation of important special cases of dynamic reconfiguration, without resorting to interpretation or dynamic linking. We propose a new metaphor consistent with PROFIT’s data sharing model for expressing dynamic reconfiguration facilities within the programming language. 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 stalls containing multiple members of a breed. Registries are system-defined pens, one for each breed, that automatically contain every member of the breed. Existing programs can be extended by injecting a new process containing new facets and new objects.

Acknowledgments

Tushar Patel is primarily responsible for the PROFIT 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. We would like to thank Harold Ossher, Jim Purtilo and Marvin Theimer for their helpful comments on earlier versions of this paper.

References

[1] Gul Agha and Carl Hewitt.
Actors: A Conceptual Foundation for Concurrent Object-Oriented Programming.
In Bruce Shriver and Peter Wegner (editors), *Research Directions in Object-Oriented Programming*, pages 49-74. The MIT Press, Cambridge MA, 1987.

[2] Andrew Black, Norman Hutchinson, Eril Jul, Henry Levy and Larry Carter.
Distribution and Abstract Types in Emerald.
IEEE Transactions on Software Engineering SE-13(1):65-76, January, 1987.

[3] Terrance Boult.
Using Profit for Machine Vision.
1990
Private Communication.

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

[5] Craig Chambers, David Ungar and Elgin Lee.
An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes.
In Norman Meyrowitz (editor), *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, pages 49-70. ACM Press, New Orleans LA, October, 1989.
Special issue of *SIGPLAN Notices*, 24(10), October 1989.

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

[7] Daniel H. Craft.
Resource Management in a Decentralized System.
In *9th ACM Symposium on Operating Systems Principles*, pages 11-19. Bretton Woods NH, October, 1983.
Special issue of *Operating Systems Review*, 17(5), October 1983.

[8] K. P. Eswaran, J. N. Gray, R. A. Lorie and I. L. Traiger.
The Notions of Consistency and Predicate Locks in a Database System.
Communications of the ACM 19(11):624-632, November, 1976.

[9] 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.
Elsevier Science.

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

[11] 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.

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

[13] Richard Helm, Ian M. Holland and Dipayan Gangopadhyay.
Contracts: Specifying Behavioral Compositions in Object-Oriented Systems.
In Norman Meyrowitz (editor), *OOPSLA/ECOOP '90 Conference on Object-Oriented Programming Systems, Languages and Applications/European Conference on Object-Oriented Programming*, pages 169-180. ACM Press, Ottawa, Canada, October, 1990.
Special issue of *SIGPLAN Notices*, 25(10), October 1990.

[14] 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.

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

[16] Gail E. Kaiser and Brent Hailpern.
An Object-Based Programming Model for Shared Data.
ACM Transactions on Programming Languages and Systems 14(2):201-264, April, 1992.

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

[18] Jeff Kramer, Jeff Magee and Keng Ng.
Graphical Configuration Programming.
Computer 22(10):53-65, October, 1989.
IEEE Computer Society.

[19] Glenn E. Krasner and Stephen T. Pope.
A Cookbook for Using the Model-View Controller User Interface Paradigm in Smalltalk-80.
Journal of Object-Oriented Programming 1(3):26-49, August/September, 1988.
SIGS Publications.

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

[21] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Moller-Pedersen and Kristen Nygaard.
The BETA Programming Language.
In Bruce Shriver and Peter Wegner (editors), *Research Directions in Object-Oriented Programming*, pages 7-49. The MIT Press, Cambridge MA, 1987.

[22] Henry Lieberman.
Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems.
In Norman Meyrowitz (editor), *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, pages 214-223. ACM, Portland OR, September, 1986.
Special issue of *SIGPLAN Notices*, 21(11), November 1986.

[23] 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.

[24] 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.

[25] Branislav Meandzija and Jil Westcott (editors).
IFIP TC6/WG6.6 Symposium on Integrated Network Management.
North-Holland, Boston MA, 1989.

[26] 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*, pages 73-78. IEEE Computer Society Press, New York NY, October, 1991.

[27] Tushar M. Patel.
Real-time Portfolio Management and Automatic Extensions.
Master's thesis, Columbia University, Department of Computer Science, October, 1991.
CUCS-030-91.

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

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

[30] James M. Purtilo.
Comparison of Polylith and Profit.
January, 1992
Private communication.

[31] 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.

[32] 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.

[33] *SunOS Reference Manual Section 3L: Lightweight Processes Library*
Sun Microsystems, Inc., Mountain View CA, 1987.

[34] 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.

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