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

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

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

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.

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.

An on-line 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 stocks and options, and execute the appropriate purchases and sales as market conditions change. The key challenge 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 [8], 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 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 model is suitable for other applications with similar characteristics, such as network and systems management [11], machine vision [3] and animation [5],

We have developed a language based on this model. PROFIT (PROgrammed FInancial Trading) is a coordination language [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 *facet* 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 in 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. In this paper, we propose to solve this problem by relaxing 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, 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 reconfiguration, we add three new concepts to PROFIT: breeds, stalls and pens. Breeds are similar to types in that they define the set of facilities required for facets that can be substituted for each other in a particular context. Stalls and pens are similar to variables and arrays, respectively: stalls "hold" a single facet and pens "hold" a collection of facets, and in both cases the facets that 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. In this new version of PROFIT, it is possible to change the static organization of an executing program, but without going so far as to support dynamic storage allocation 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.

2. Background

The PROFIT_0 object model supports data sharing among arbitrary objects in a distributed environment. There are three important components: facets, objects and processes. A *facets* is 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. An *object* is a statically defined collection of facets representing an information hiding unit. An object defines a context for binding references between its facets and an external interface for encapsulating the facets. A *process* is 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.

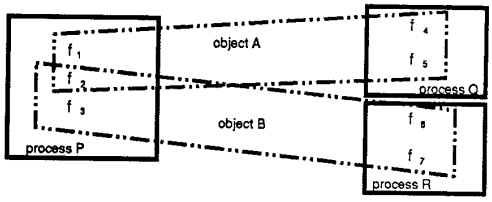

Figure 2-1: Facets, Objects and Processes

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. Within one of these C functions, the program text can refer to slots in the same facet (both data and procedure) via extended syntax, which is supported by the PROFIT preprocessor. 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 [13].

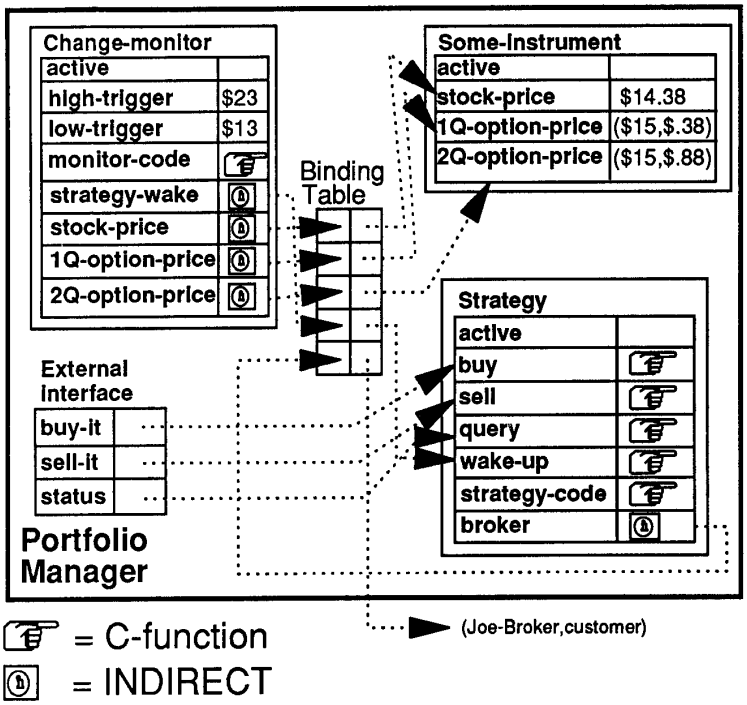

Figure 2-2: Generic Object

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.

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 also binds each indirect slot in one of its facets, either to a slot in another 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, (facet, slot) or (object, entry).

When code (a C function) is executing within a facet, it may directly access only those slots defined in the same 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. When the response to the original call is eventually accepted from its queue, the facet continues with the operation that made the call from the point where it left off.

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. 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.

[1]This description has been simplified for ease of presentation. PROFIT$_0$ also supports threads, asynchronous message passing, priorities and pre-emptive scheduling [9].

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 caller.

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 (and in our previous paper) 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.

In this paper we relax these restrictions to allow limited changes to the static structure. In particular, we support the following special cases: the ability to substitute one facet for another with a compatible interface, operations over sets of facets where the constituent elements can be changed, and addition of new facets, objects and processes to an executing program. This allows us to add new instruments to the prices database, change the composition of existing portfolios, and add new users and their portfolios. We call this *dynamic reconfiguration*. 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 computation language.

Our approach is built on three main concepts: breeds, stalls and pens. A *breed* is a partial facet description representing an abstract type. A *stall* 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. A *pen* 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. 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. Pens and herds 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.

4. Breeds and Stalls

In PROFIT$_0$, each operation invokes 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 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 at least this set of slots is a member of 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 [2] or a role in RPDE [7]. 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.

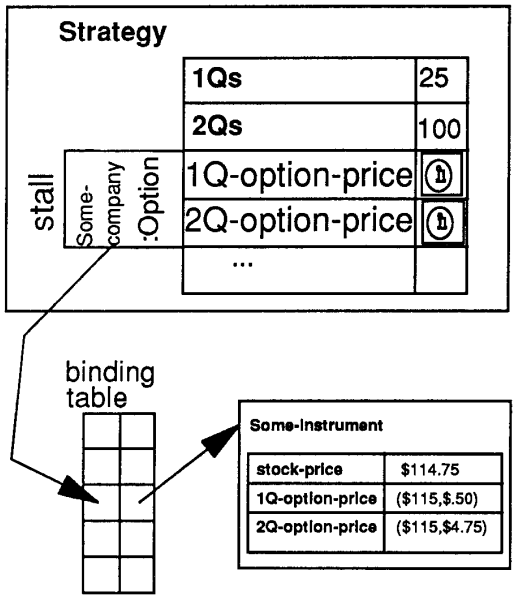

Figure 4-1: Binding an Occupant to a Stall

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, and a breed name. Figure 4-1 illustrates a stall (`Some-company`) within a generic facet (`Strategy`) and 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.

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.

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), 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. 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. We 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, thereby allowing a portfolio to contain multiple instruments, where the number of members can change over time.

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. 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.

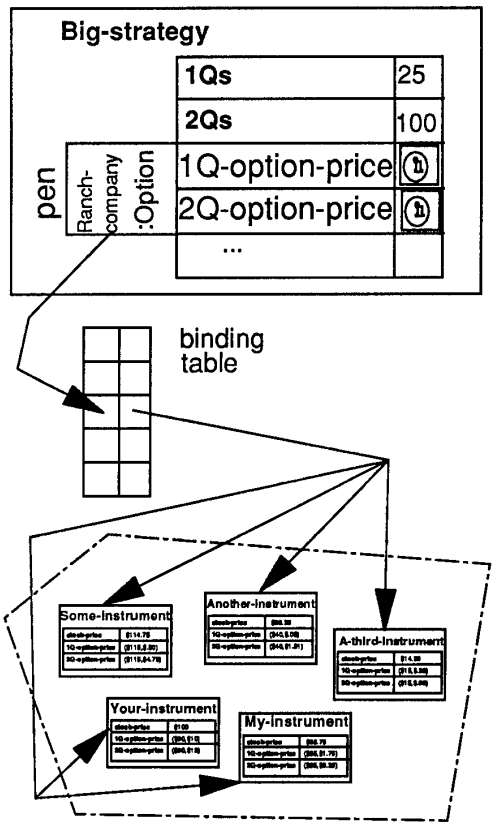

Figure 5-1: Binding a Herd to a Pen

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`). 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 say facet when referring to a member of a breed, but 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. As a result, there is no need for indirect accesses to explicitly dereference a pointer or lookup a facet identifier. Assume for the sake of discussion that there is exactly one registry for each breed. When a new facet is created, it is automatically added to the appropriate registries. Subsequent queries on these registries can include this facet in a stall or add it to a pen.

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 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 6-1 shows a facet making a query to a user-defined registry. The 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.

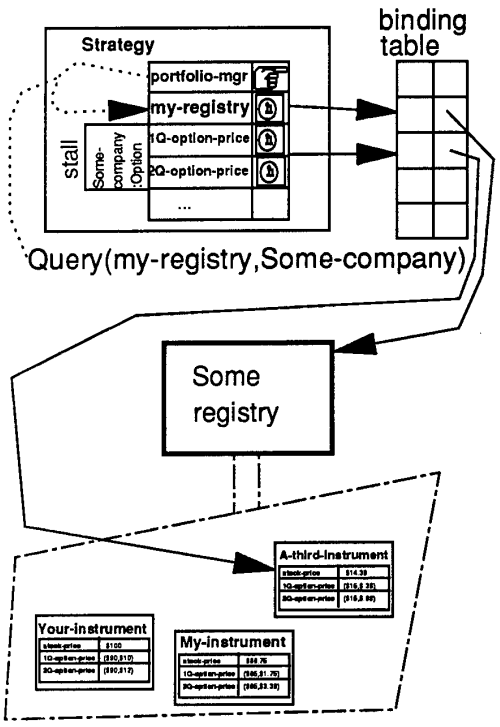

Figure 6-1: Query to User Registry

Queries to the `system` registry for a particular breed simply name the breed as a qualifier. 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.

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 languages, 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 new processes to contain the facets, compiling and linking, them, 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. New objects can be introduced independently, made up of new and/or existing facets, by generating new binding tables. The programmer then executes an injection program that installs 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

The current PROFIT implementation supports only a single process, although there may be multiple objects and shared facets. It includes several language facilities related to timing and scheduling, for example, the `everytime` statement repeats a loop within a specified time period. 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. The parser consists of about 4300 lines of C, 250 lines of lex rules and 650 lines of yacc rules. The run-time support consists of an additional 1300 lines of C.

The SPLENDORS system, now in progress, will 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 financial industry professionals.

9. Related Work

Dynamic reconfiguration is standard in operating systems and network management systems, where there are a number of resources to be managed that change relatively

infrequently. The SOS operating system [12] is similar to PROFIT in that multiple objects can be combined into a group, with easy communication among the objects in the group, even though the objects reside in multiple contexts. A local proxy provides access to the service collectively provided by the group. Proxies may be migrated as needed for service. SOS provides a mechanism for certain cases of dynamic reconfiguration in the form of dynamic classes.

The Orca programming language [1] 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. The Mercury system [10] supports some dynamic reconfiguration through server ports, which are reestablished after network failures and permit binding of new clients to servers during program execution. Hailpern and Ossher [6] have extended the notion of abstract type to a view which includes interface specification, the set of objects that can provide a service, and a set of objects that can consume the service. This serves as a framework for describing different inheritance and delegation mechanisms.

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).

Our original language design featured static allocation of facets. The primary contribution of this paper is the addition of important special cases of dynamic reconfiguration, without resorting to general dynamic allocation. 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 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

Tushar Patel, Jason Kim, Isai Shenker, Vanessa Cole and Michael Mayer contributed to the implementation effort. Discussions with George Beltz, Terry Boult, Jim Donahue, Gary Herman, Catherine Lassez, Aurel Lazar, Harold Ossher and Dan Schutzer influenced the development of our ideas.

Some of this work was done while Prof. Kaiser was an Academic Visitor at IBM Research. Prof. Kaiser was supported by NSF grants CCR-9000930, CDA-8920080 and CCR-8858029, by grant from AT&T, BNR, Citicorp, DEC, IBM, SRA, Sun and Xerox, by the Center for Advanced Technology and the Center for Telecommunications Research.

References

[1] Henri E. Bal and Andrew S. Tanenbaum. Distributed Programming with Shared Data. In *IEEE ICCL*, pages 82-91. October, 1988.

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

[3] Terry Boult. Private Communication. 1990.

[4] Paola Ciancarini. Coordination Languages for Open System Design. In *IEEE ICCL*, pages 252-260. March, 1990.

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

[6] Brent Hailpern and Harold Ossher. Extending Objects to Provide Multiple Interfaces and Access Control. *IEEE TSE* 16(11), November, 1990.

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

[8] Gail E. Kaiser and Brent Hailpern. An Object Model for Shared Data. In *IEEE ICCL*, pages 135-144. March, 1990.

[9] Gail E. Kaiser and Brent Hailpern. An Object-Based Programming Model for Shared Data. *ACM TOPLAS*, 1991. In press.

[10] Barbara Liskov, Toby Bloom, David Gifford, Robert Scheifler and William Weihl. Communication in the Mercury System. In *IEEE HICSS*, pages 178-187. January, 1988.

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

[12] Marc Shapiro, Philippe Gautron and Laurence Mosseri. Persistence and Migration for C++ Objects. In *ECOOP*, pages 191-204. Cambridge University Press, July, 1989.

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