Real-time Portfolio Management and Automatic Extensions: MS Thesis

Tushar M. Patel

IBM
42FA/235
Neighborhood Road
Kingston NY 12401

Columbia University
Department of Computer Science
New York, NY 10027

tushar@cs.columbia.edu

7 October 1991
CUCS-030-91

Thesis Committee: Gail E. Kaiser, Columbia University and Dan Schutzer, Citicorp

Chapter 1
Introduction

The PROFIT language [Kaiser 90, Kaiser 91] is motivated by applications driven by large quantities of rapidly changing data. An example of such an application is the portfolio management application that seeks to take advantage of the market conditions to optimize portfolios. Network management is another such application with large amounts of data that changes rapidly.

Applications such as these require that large amounts of rapidly changing data be monitored. Such monitoring can be accomplished via polling or using the active values (notification) approach. In polling, the various data that are expected to change rapidly are examined periodically. Notification is an approach where the various agents are notified when a change takes place. Polling is CPU intensive whereas the active values approach results in large amounts of network traffic. PROFIT is designed to avoid the polling vs. notification controversy by making it possible to create applications that take an intermediate approach between polling and notification.

This thesis has been motivated by the problem of real-time portfolio management. The primary work is the creation of a real-time portfolio management application — SPLENDORS [Patel 91]. It allows the monitoring of the changing market conditions using appropriate criteria according to the condition being monitored (for example, the prices of a volatile security can be monitored more frequently than that of a stable security) as well as the investment philosophy of the portfolio owner (for example, the same security might be monitored more often for a trader and less often for a long-term stable investor).

SPLENDORS allows the end user to add program components from a pre-defined library of reusable generic components without programming to the *running* system without having to bring it down. This makes it possible to extend the system as appropriate by non-programmers, on the fly, to represent additional securities purchased or to represent portfolios created for new customers by non-programmers. This is done using interpretive constructs. When SPLENDORS is brought down (typically at the end of the trading day), it converts the additional program components to a form that can be compiled. This means that a fully compiled instance of the system can be created for the next trading day. Thus, SPLENDORS has application generator like features that allows non programmers to extend the SPLENDORS system without programming.

However, unlike the usual application generator, the SPLENDORS application generation is extensible. It is possible for the PROFIT and C programmers to create new program components and integrate them with SPLENDORS and making these additions available to non programmers.

At least three implementations of PROFIT have been attempted. The experience gained from the first two have been used in the current implementation known as CAPITAL. It is important to note that CAPITAL is one particular implementation of the PROFIT language and may be replaced by some other implementation in the future.

A real time portfolio management application might typically be used in a brokerage house with several brokers each of whom would have a workstation managing the portfolios of his/her clients. The current implementation of PROFIT and SPLENDORS is designed for a single broker having multiple client portfolios. Different portfolios may share some of the securities. The prices of the securities fluctuates and the strategies are executed dynamically in real time. Each portfolio has its own strategy to manage the securities for that investor.

A problem such as this may be solved in several ways.

- A solution may be created using an existing traditional language such as C. This would require the programmer creating the application to manage the real time requirements of this application. Furthermore he/she would have to worry about the proper way of handling shared data (for example a security that is common among a prices object and a portfolio object) and maintaining consistency among objects sharing such data.

- Another solution would be to use PROFIT to create such an application. Since PROFIT would provide the consistency among shared data as well as provide real time constructs, the PROFIT programmer can concentrate on the financial aspects of the application. While this is superior to an all C approach, this still requires programming skills to accommodate any kind of change.

- The SPLENDORS solution goes a step beyond this. Recognizing that many strategies and monitoring criteria are used repeatedly with slight modification, SPLENDORS provides a way to let the non-programming broker to change the portfolios, associating monitoring characteristics and strategies to manage them. The broker may do this in a dynamic fashion. The changes are activated immediately and code is generated when desired (typically at the end of the trading day for the next session). Furthermore, the programming department of the brokerage firm could continually add additional customized or generic codes to the library for these of brokers. This could be done either in PROFIT or in C.

Chapter 2

Context - PROFIT

The PROFIT language has been motivated by the real time portfolio management application. This involves real time monitoring of market conditions, execution of strategies using up-to-date market conditions information, and updating portfolios on a real time basis to take advantage of the market conditions. This section contains a brief description of PROFIT. For a more detailed description, see [Kaiser 90, Kaiser 91].

There are three main concepts in PROFIT — facets, objects and processes (see figure 2-1)

- A *facet* is a minimal unit of data and control. It is made up of slots. Each such slot contains either data or code. However, in addition to such slots, a facet also maintains a queue of requests for access to its slots and schedules the requests. Only one request may be granted at a time. Each facet contains a special slot known as the active slot, which points to the slot being currently accessed. Slots have priorities associated with them, which are used in scheduling the requests. Each security in SPLENDORS is represented as a PROFIT facet.
- An *object* is a collection of one or more facets. Additionally, it binds relationships between facets in the same object and contains an interface for understanding messages received from other objects. A PROFIT object is illustrated in figure 2-2. Each portfolio in SPLENDORS is represented as a separate PROFIT object.
- A collection of facets executes in an address space, an operating system *process*. Different groups of facets can reside in different processes, possibly on different machines, such as a prices database on a mainframe and the portfolio managers on analysts' workstations. Processes and objects are orthogonal concepts: processes do not contain objects and objects do not contain processes, but both can contain facets.

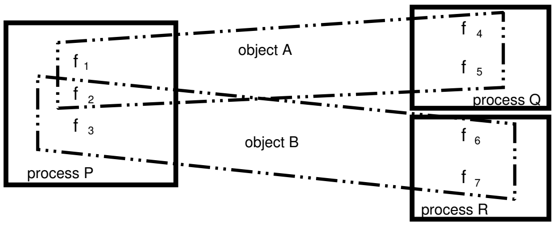

Figure 2-1: Profit facets, objects and processes

Facets may be shared among multiple objects. This is a key characteristic of PROFIT and is exploited throughout SPLENDORS. It makes it possible to have a single copy of rapidly changing data. For example, there is only one facet representing the price of a security even though the security may be contained in multiple portfolios (represented as multiple objects). This in turn makes it easier to maintain consistency among various objects referring to the same piece of shared data.

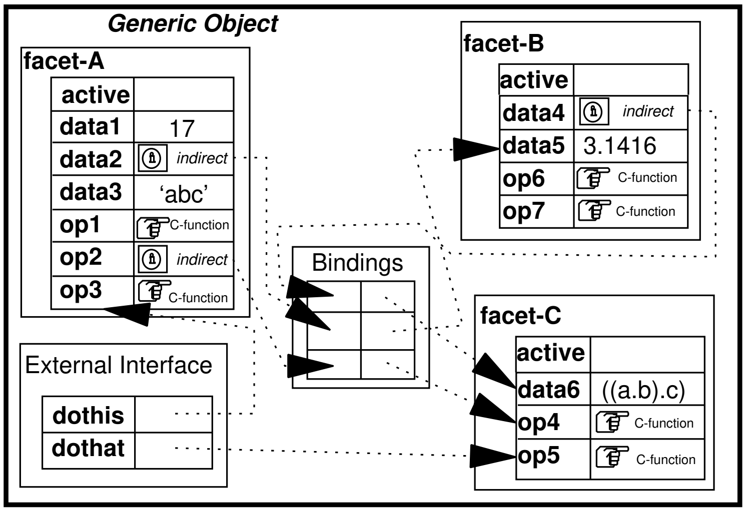

Figure 2-2: PROFIT object

Each facet belongs to one and only one process. There are no copies of facets. Different facets in the same object may reside in different processes, so an object may be divided among multiple processes and there may be parts of several different objects in the same process.

PROFIT programs execute by sending messages between objects. When an object receives a message, it looks up the name of the message in an entry table to see which of its facets knows how to respond to this message, and then queues this message at that facet. All such messages correspond to requests to get or put a data slot or to get (execute) a procedure slot in a facet. A facet may handle a message by doing some processing and then sending a response back to the sender of the message, or it may itself request help from other facets in the same object — or even from other objects. So a facet can also send messages to other facets and to objects, specifically according to the relations specified in its enclosing object(s).

A request is executed in a thread. A thread here refers to a medium weight thread of control which has its own stack. All threads (in a single process) share the same address space. New threads are created when asynchronous requests are executed or when an inter-process request is executed.

In the current CAPITAL implementation, a PROFIT program consists of a single PROFIT file and several C files. Additional PROFIT library files are permitted which may be imported (similar to a macro expansion).

A sample PROFIT file illustrating the basic PROFIT declarations is shown below.

```
FACET f1
   internal num_stocks: int
   internal buy_price: double
   external price: double
   external daemon1(h: double, l:double, t:long, s: char *): int PRIORITY(10)
END FACET f1

FACET f2
   internal num_stocks: int
   internal buy_price: double
   external price: double
   external daemon1(h: double, l:double, t:long, s: char *): int PRIORITY(10)
END FACET f2

FACET f3
   internal num_stocks: int
   internal buy_price: double
   external price: double
   external daemon1(h: double, l:double, t:long, s: char *): int PRIORITY(10)
END FACET f3

FACET daemons
   internal daemon1(high: double, low:double, t:int, s:char *): int PRIORITY(10)
END FACET daemons

FACET I
        internal start_daemons(): int
END FACET I

FACET GE
        internal cur_price: double PRIORITY(5, 10)
END FACET GE

FACET IBM
        internal cur_price: double PRIORITY(5, 10)
END FACET IBM

FACET init
   internal initialize(): int PRIORITY(10)
END FACET init

{* ****************************************************** *}

OBJECT prices
FACETS: DOW GE IBM

ENTRY:

MAP:

END OBJECT prices

OBJECT p1
FACETS:  IBM daemons f1 I

ENTRY:
      start_p1 -> I.start_daemons;

MAP:
      f1.price -> IBM.cur_price,
      f1.daemon1 -> daemons.daemon1;
END OBJECT p1

OBJECT p2
FACETS:  IBM GE daemons f2 f3 I

ENTRY:
      start_p2 -> I.start_daemons;

MAP:
      f2.price -> IBM.cur_price,
      f2.daemon1 -> daemons.daemon1;
      f2.price -> GE.cur_price,
      f2.daemon1 -> daemons.daemon1;
```

```
END OBJECT p1

OBJECT mainobj
FACETS: init

ENTRY:

MAP:

END OBJECT mainobj

{* ************************************************** *}

PROCESS SPLENDORS
FACETS: f1 f2 f3 IBM GE I init
start := mainobj.init.initialize();
DEFAULT_PRIORITY (1,2);
END PROCESS SPLENDORS
```

As can be seen, the PROFIT file contains facet definitions, object definitions and process definition.

Facet definitions consist of slot definitions, which announce the following properties of that slot:

- internal or external (internal if the data/procedure really resides in this facet; external if it resides somewhere else)
- slot name
- number and data types of parameters for procedure slots
- data type (return type for procedure slot)
- priorities for that slot (1st and 2nd numbers represent `GET` and `PUT` priorities. Only `GET` priority is meaningful for a procedure slot). Priorities are optional for any slot. If they are not specified, the default priorities (specified in the `PROCESS` section) are used.

An object declaration specifies:

- the facets that make up the object
- entry points to the object
- the mapping from external slots to slots in some other facet within the same object or to an entry point to some other object. Mapping is specified on a per object basis. If a facet having an external slot is shared among two objects, the external slot in such a facet may be mapped to different slots in different facets.

The process declaration specifies:

- the facets that make up the process
- the entry point to the program - this must be an internal procedure slot
- default `GET` and `PUT` priorities - these are used for slots that don't have any specified priorities

The data or the code actually resides within the containing facet for an internal slot. An external slot is a pointer to some other slot in some other facet. The `MAP` section for each object contains the mapping information that binds external slots to other slots in the same object. An external slot may be bound to another external slot as long as the binding eventually results in an internal slot in some facet. Thus there is a single copy of any piece of data (maintained as an internal slot in some facet) which may be accessed by other facets (which would define external slots and the appropriate mapping). External slots may also be bound to object entry points other objects.

The PROFIT parser parses the PROFIT file and generates a header file (named facetname.h) for every facet for use by the PROFIT programmer. It also generates several C files which contain code and tables including the binding tables used at execution time to resolve external slots. This code is the linked with the non program specific part of the PROFIT kernel (which contains code to schedule requests, handle real time requests and to perform dynamic updates). Additionally, the C code created by the PROFIT programmer (for the procedure slots) is also linked in. The result in the executable PROFIT program.

The PROFIT programmer is responsible for writing code for the procedure slots. Each internal procedure slot must have a C function by the same name. Typically, the PROFIT programmer creates a C file for each facet that contains internal procedure slots. This C file would contain the code for all the internal procedure slots in that facet. The regular C code can be interspersed with PROFIT constructs and statements. The only requirement is that the C file include a header file (named facetname.h) This header file is generated by the PROFIT parser and contains definitions for the various slots as well as expansion for some PROFIT statement constructs.

The syntax for accessing slots (sending a message) in the C file is shown below (d1 and p1 are data and procedure slots, respectively, in that facet):

```
x1 = d1;           /* synchronous data slot GET request   */
put_d1;            /* synchronous data slot PUT request   */
A$_put_d1;         /* asynchronous data slot PUT request  */
x2 = _p1();        /* synchronous proc. slot GET request  */
x3 = A$_p1();      /* asynchronous proc. slot GET request */
```

As can be seen, data slots may have synchronous `GET` and `PUT` requests and asynchronous `PUT` requests. Procedure slots may have synchronous or asynchronous `GET` requests. `PUT` requests for procedure slots and asynchronous `GET` request for data slots are not allowed. The syntax can be summarized as:

- Use the slot name for data slot
- For `PUT` requests, precede the data slot name with put_
- Use _ followed by slot name followed by () for procedure slots. Parameters may be enclosed within ().
- For asynchronous requests, follow these rules and precede with A$.

Such code can be freely intermingled with any other valid C code.

Data slots have `GET` and `PUT` priorities associated with them, while code slots have `GET` priorities. Requests for slots are queued at the containing facet. The priorities are not pre-preemptive. When the current request is finished, the facet selects the next request in its queue according to which has the highest priority (rather than according to which arrived first). Both `GET` and `PUT` requests may run either synchronously or asynchronously. A synchronous request results in the sender's own work being blocked until a response is received. For asynchronous requests, no such blocking is performed; the sender has essentially farmed out part of the processing and continues on with its portion. It should be noted that the blocking applies to a particular request and not the entire facet. The requesting facet is free to process other requests it might have pending.

Different facets may run concurrently within the same or different processes. The PROFIT kernel manages concurrently executing facets and performs round robin scheduling among the active facets within a process. Within a facet, however, requests are not processed concurrently.

PROFIT provides constructs to handle blocking requirements. These include `SENSITIVE_START/END` and `PAUSE`

- `SENSITIVE_START/END` marks a sections as a sensitive section which is a block of code which may be interrupted (i.e. it is a non-critical section). All regions outside such blocks are assumed to be critical regions and may not be interrupted. This choice of constructs as opposed to the traditional critical section forces the programmer to think about what portions of his/her code can really be interrupted. Non-sensitive sections are implemented by assigning a very high priority to them.
- `PAUSE` may be used to allow pending requests to be processed. It is equivalent to `SENSITIVE_START` immediately followed by `SENSITIVE_END`.

Real time constructs (viz. `EVERYTIME`, `SLEEP`, `START_TIMER`) are also provided.

- `EVERYTIME` is used to execute a portion of code repeatedly at a fixed repetition time interval. For example, a block of code enclosed in `EVERYTIME(n)` and `EVERYTIME_END` would result in that of code being executed every n milliseconds.
- `SLEEP` is used to block a thread for a fixed time
- `START_TIMER` can be used to specify that thread must be resumed within a certain time interval.

The PROFIT kernel makes a best attempt to meet the real time requests. A special thread called the timing thread is always running at a very high priority. Any time a new timing request is made, the timing thread wakes up and adds it to its request queue based on when the requester needs to be waken up. The timing thread then goes to sleep until the first thread in its queue needs to be waken up.

In the case of `SLEEP` and `EVERYTIME`, the requesting thread goes to sleep after sending the request to the timing thread.

In the case of `START_TIMER`, the requester simply reduces its priority to a very low level after sending the request. The idea is to let other threads with higher priorities continue until either no more such threads remain or the specified time interval expires. In the former case, the requester will resume execution automatically which may happen before the entire time interval specified expires. In the latter case, the timing thread will discover that the time interval has expired (or is about to expire) and will then change the requesters priority back to its normal level.

All timing requests result in the release of the active slot (of the facet that thread is currently executing) before the requesting thread goes to sleep. It is reacquired when it wakes up. The facet goes on to process some other pending request.

As SPLENDORS was developed, it became evident that a way to dynamically change the portfolio objects was needed. While an elaborate dynamic reconfiguration methodology had been designed [Hailpern 91], it was decided that it would be expedient to use a simpler, limited dynamic reconfiguration method until the more elaborate and complete methodology can be incorporated in PROFIT. The statements available for the limited dynamic reconfiguration include

- ADD_FACET
- BIND_SLOT
- COMPOSITION

ADD_FACET allows the addition of a facet while the program is running. Since there is no notion of an object at execution time, it is not necessary to specify which object contains it. BIND_SLOT binds a slot to some other slot. This allows the binding to change dynamically and to associate binding for slots within dynamically added facets. COMPOSITION may be used to determine the composition of a facet dynamically.

Additionally, a way to read and write to the slots in the newly added facets was needed.

- SGET
- SPUT

SGET and SPUT are used to provide interpretive versions of the GET and PUT. They take the facet and slotnames as arguments along with the value/variable, parameters (for procedure slots) and request type (synchronous or asynchronous). These may be used not just for the newly added facets but also for regular facets. This makes it possible to write generic code because the name of the facet and slot can vary dynamically as the need arises.

To facilitate the use of reusable facets, PROFIT provides an IMPORT statement[1]and has the following syntax:

```
FACET foo: IMPORT generic_foo
```

This results in macro expansion like operation where the generic_foo facet is extracted from a library of such facets resulting in an instance of generic_foo called foo.

The library consists of two files created using the UNIX ar utility. The first of these consists of the facets. The second consists of associated C code. In the above example, the PROFIT code for the facet generic_foo would be created in a file generic_foo.f and added to a library using

```
ar r lib.pl.a generic_foo.f
```

and the C file is added to the corresponding C archive file using

```
ar r lib.cl.a generic_foo.c
```

Once the PROFIT parser extracts the generic PROFIT code, it extracts the corresponding C code (generic_foo.c in the above example) and renames that file to match the name of the instance (foo.c in the above example). It then changes the

```
#include "generic_foo.h"
```

statement in it to the

```
#include "foo.h"
```

[1]The IMPORT statement was implemented by Michael Mayer.

At this point, it is as if the PROFIT programmer had created the `foo` facet and the corresponding C file himself/herself.

Finally, facilities exist to allow the PROFIT programmer to substitute his/her own error handler to handle PROFIT errors via the use of the `inst_profit_error_handler(pfv)` function supplied by the kernel which takes a pointer to a function returning a void as its sole argument. The function doing the customized error handling is invoked when an error is detected with a single argument - an `int` representing the error code).

```
void (*inst_profit_error_handler(pfv))()
void (*pfv)(error_code);
int error_code;
```

Some support for debugging the kernel also exists.

Chapter 3

SPLENDORS Design and Implementation - Overview

SPLENDORS is a soft real-time portfolio management application built using the PROFIT language. The goals of SPLENDORS are to provide a system that manages multiple portfolios consisting of stocks and options on a soft real time basis and to provide a means for non-programmer financial experts to build their own portfolio managers. By soft real time, we mean that the real-time requirements are not as rigid as in a manufacturing situation because in the financial applications domain, the changes may be missed occasionally. This also means that such an application can be created to run on a regular operating system platform and does not require a real time operating system.

The end user of SPLENDORS is the broker in a brokerage firm. Each such broker firm would have a workstation running SPLENDORS which would manage portfolios for all of his/her clients. In addition to the workstations for individual brokers, the system would consist of a mainframe which would receive an incoming real time feed consisting of updated prices (and other information such as volume) of the securities. The mainframe would manage the prices database that the individual workstations may tap into. The current implementation is limited to a single broker and a simulated feed running on another workstation (as opposed to a mainframe).

When the trading day begins, the system begins monitoring the fluctuations in the market. Whenever something happens that might require an action (such as buy or sell) in accordance with some investor's philosophy, the system would send a message containing that recommendation to the broker managing that client's portfolio. Examples of such messages are:

- Sell PEP for customer Simpson. Current Price: 63.125 High Trigger: 63
- Buy something for customer Flintstone. Current cash position: 26%. Trigger: 25%

Such portfolio management requires monitoring of the prices of the securities. The traditional approaches to such monitoring are polling and active value propagation (or notification). In the active values approach, messages describing the changes are automatically sent to all interested portfolio managers. The polling approach requires each interested portfolio manager to explicitly check the current value of the price over and over again.

A real time portfolio management system has to deal with numerous price changes. Using the active values approach would mean flooding the system with a large number of messages (mostly concerning small price changes) even though most of these price changes would not be significant (i.e., most changes would not move the price sufficiently to warrant some action such as buy or sell). Polling would solve this problem, since each interested portfolio could decide how often to poll based on an estimated frequency of "interesting" changes (such as estimate can be made by trial and error or by some mathematical model based on the volatility of the security). However this would mean that the programming of portfolio-wide strategies would have to be interspersed with polling code. This gets complicated and error-prone when the number of securities becomes large and polling of multiple securities is needed.

Although the PROFIT programmer can use either of these approaches, PROFIT is really designed to exploit the tradeoff between polling and active values using an intermediate approach. In the case of SPLENDORS, the active values approach would have resulted in too much traffic, mostly consisting of uninteresting messages (i.e. messages about small price changes that would have no possibility of resulting in some action). The polling approach would have required that the PROFIT programmer interested in programming strategies to represent various investment philosophies also be concerned about polling issues. This is clumsy and error-prone.

One of the tasks that needs to be performed in a real time portfolio management system is the monitoring of prices of various securities on a soft real time basis. This would typically be done frequently (anywhere from several times a second to once every few seconds for each security). Another task is the execution of some computation that would be carried out periodically (once every few minutes to once a day) or when some condition is reached (the price of some security reaches a certain threshold). The purpose of such a computation is to determine whether some action such as buy or sell might be appropriate. These two tasks have have different real time and computation requirements. This insight has been used in the design of PROFIT and is exploited in SPLENDORS. Specifically, SPLENDORS uses a *daemon* to carry out the former task and a *strategy* to execute the latter. A daemon continually monitors the price of a security and notifies the strategy when something potentially interesting (to the portfolio that the daemon belongs to) happens.

3.1. Daemon

A daemon contains knowledge that enables it to use the appropriate polling criteria as in the purely polling based approach. Multiple daemons can be created to monitor the same security and a single daemon may monitor several securities. The polling interval can be different for different daemons monitoring the same security.

Daemons may also contain arbitrary computations (though in practice, such computations would have to limited to those that can execute very fast). The purpose of such a computation is to determine if some portfolio might be interested in the most recent change. The portfolio is notified of the change only if the computation determines that to be the case. This filters out uninteresting changes and ensures that only potentially interesting fluctuations are passed on to the portfolio manager.

Daemons can contain arbitrary knowledge and they may carry out arbitrary computations. In fact, the knowledge in the daemons could be updated dynamically. For example, a simple daemon can be used to monitor the price of a security and inform the strategy when an upper or a lower threshold is crossed. The daemon could contain the requisite knowledge to update these thresholds as the market fluctuates to capture changes in trends [Washington 89].

Daemons are implemented in SPLENDORS using PROFIT facets. Daemons are conceptually associated with securities in portfolios making it possible for different portfolios containing the same security to have different daemons. For example, portfolio manager A, interested in exploiting volatile market conditions and engaged in many short-term trades, could poll the price of security `foo` every second whereas portfolio manager B, who might be interested in longer term investments and is interested simply in being aware of certain thresholds being crossed and investment goals being reached, might poll the price for the same security once every minute or once every hour. Similarly, the same portfolio manager might poll a volatile small growth stock more often than a less volatile large utility.

PROFIT makes it possible to place the daemons with the portfolio objects reflecting the logical association of a daemons with the portfolio for which it is monitoring the security. Such placement would also result in the broker workstation being used for the computation in the daemon, thereby reducing the CPU time consumed on the mainframe managing the prices database. However, it requires that each time the daemon is interested in looking up up a price, a message (and the reply) be sent between the portfolio object on the workstation and the prices database. PROFIT also makes it possible to locate the daemons in the same process containing the prices object (which is a PROFIT object in SPLENDORS that represents the prices database). This minimizes network traffic because uninteresting changes do not generate any messages between an daemon on the mainframe and its portfolio manager on the analyst workstation. Because SPLENDORS is currently limited to a single process, it doesn't have to make this choice. However, we anticipate that when it is extended to handle multiple processes, the latter approach will be used to minimize network traffic.

Since the daemons are basically concerned with monitoring prices of individual securities without having knowledge of the rest of the portfolio, there needs to be another entity to enforce some investment policies for the entire portfolio. This is the *strategy*.

3.2. Strategy

When the *daemon* discovers a change that might be potentially interesting, it sends a message to the *strategy*. Daemons can be constructed to send such messages to multiple strategies associated with a single portfolio. The strategy can execute code and determine if the price fluctuation warrants any action.

A strategy is generally more complicated than daemons. While a daemon is responsible for monitoring price fluctuations and filtering out the uninteresting ones, a strategy concentrates on exploiting interesting changes to optimize the portfolio. It is the strategy which determines what to do with the price changes. It is the strategy which would initiate or recommend the initiation of some action (such as buy or sell) based on the price changes. For example, when the daemon for `foo` recognizes some low threshold being crossed that might result in a buy signal, it informs the strategy of that condition being reached. The strategy then carries out some computation to determine if the purchase of stock in that security is consistent with the investment philosophy of the investor owning the portfolio (such as the maximum holdings in a single company condition hasn't been reached) and if so, carries out some action such as issue a message (recommendation) to the broker managing the portfolio.

One could create portfolios to have strategies associated with a security or a group of securities. This is something that can be accomplished using PROFIT. One could also take the view that the function of the strategy is to recommend actions such as buy and sell when appropriate. Such actions would alter the composition of the portfolio and therefore a strategy is a really portfolio-specific entity unlike a daemon which is typically concerned with monitoring a single security. SPLENDORS is architected around the notion that daemons are security specific concepts whereas strategies are portfolio-wide entities. Just as it is possible to have multiple daemons associated with a security, it is also possible to have multiple strategies associated with a portfolio.

Strategies are implemented in SPLENDORS using PROFIT facets. Each portfolio is required to have at least one strategy. The strategy facets typically execute on the analyst workstation. However, they execute only when they are notified about a potentially interesting change. Since this happens far less often than price fluctuations, the strategy computations have to be carried out less frequently compared to the daemon computations. This in turn implies that a strategy can be considerably more complicated than a daemon.

The explicit distinction between daemons and strategies brings with it several advantages:

First, the daemons are simple to create (because they only monitor the price of a security).

Second, they are typically fast (because they are simple) and a system can therefore have many daemons.

Third, by executing the daemons on the same machine as the prices database and informing portfolio managers on analyst workstations only about interesting changes, one can reduce network traffic substantially. For example, consider the USA trades for IBM stock on July 19 1991 (see figure 3-1).[2]The price changed 341 times. This would result in 341 messages in a purely active values based approach. A polling approach where the IBM price is polled once every 10 seconds would have resulted in 6 times/minute * 60 minutes/hour * 7 hours = 2520 messages. The price changes were all within the range 98 7/8 to 100 7/8 range. If this change was not significant to the portfolio, the hybrid approach supported by PROFIT and used in SPLENDORS would have resulted in no messages at all!

Finally, the strategies are not even told about such uninteresting changes. This means that a strategy is executed less frequently than in a purely polling or active values based approaches. This in turn means that one can use more CPU time each time the strategy is executed, i.e., the strategy can be more complicated (viz. it can use a more realistic model and consider more factors in its computation).

The separation of polling from strategic logic also means that a daemon, executing only the code

[2]This data was supplied by Ken Hardy of Bridge Information Systems, St. Louis.

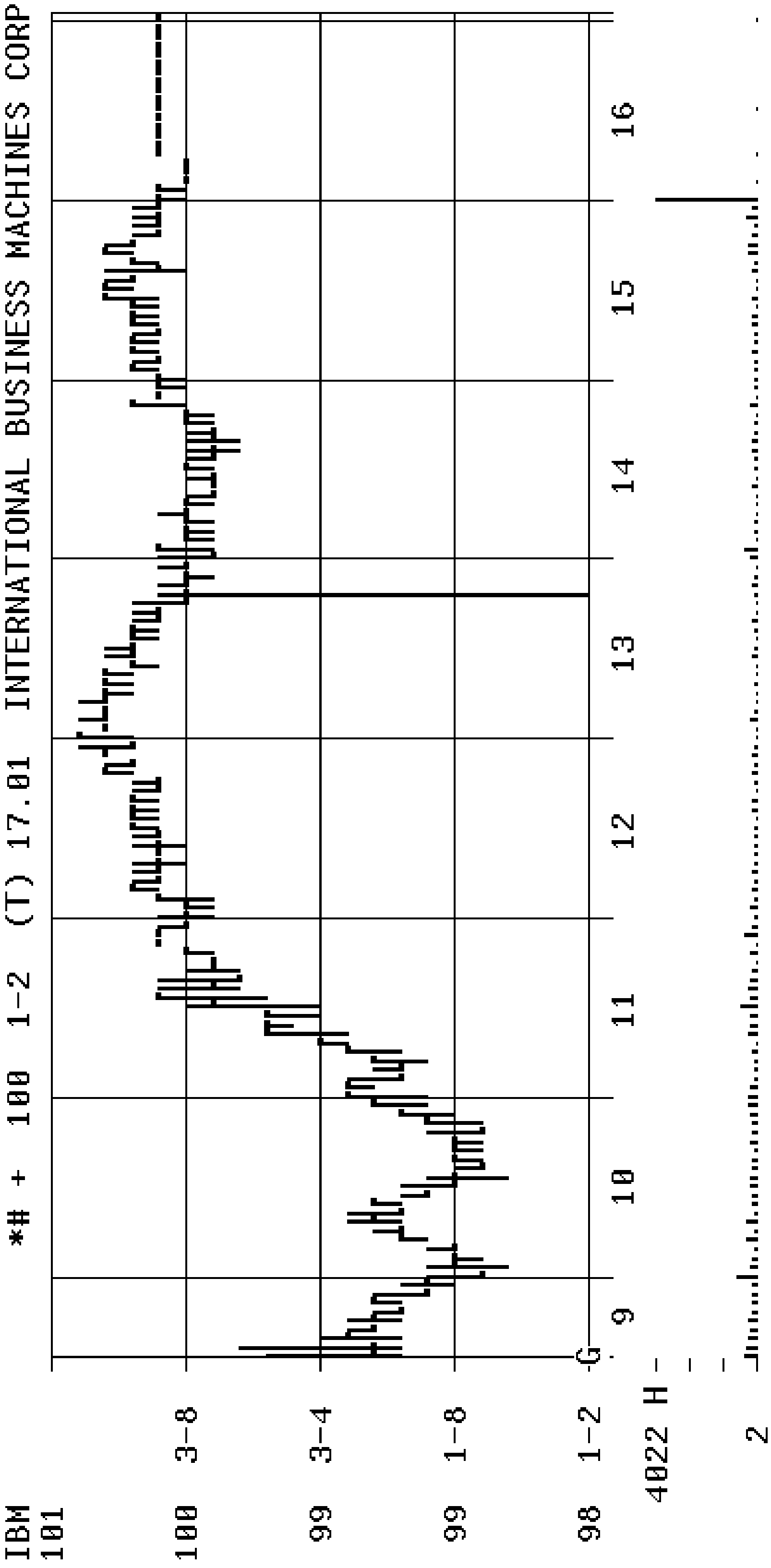

Figure 3-1: IBM stock price on July 19 1991

necessary to poll the price, will execute faster than a purely polling based approach where additional code concerning the strategy also would be executed every time the price is polled. This means that such a separation results in one being able to poll more often than a purely polling based approach. This has two benefits. First, it reduces the number of times "interesting" changes are missed. Second, it makes it easier to guess an estimated frequency to use for polling (because one can be conservative and use a smaller (more frequent) sampling interval).

3.3. Generic daemons

Several organizations of daemons can be created using PROFIT. For example one could create a system which uses one daemon for a particular security for all portfolios. This would result in fewer daemons. However, they would have to contain knowledge about the kind of change that each particular portfolio would be interested in. This could allow the daemon to selectively send messages to a subset of portfolios containing a security when a change occurs that some portfolio might be interested in but others would not. This would complicate the programming of such daemons. Furthermore, addition of such a security to some other portfolio would require that the knowledge about what this portfolio would consider interesting be integrated in the daemon.

In the absence of such knowledge, a message would be sent to all portfolios containing that security when a change occurs that any such portfolio might be interested in. These alternatives would respectively require too much knowledge in the daemon or result in too many messages.

Another possibility is one daemon for all securities in a portfolio. This organization may be useful in specific cases where the strategy of the portfolio requires actions based on prices of multiple securities but in the general case, this organization would result in the least volatile portion of the portfolio being polled (by the daemon) at the same rate as the most volatile. This results in too many messages (if less volatile securities are polled too often) or potentially missed interesting changes (if more volatile securities are polled less often).

For SPLENDORS, we wanted to avoid both of these situations. In SPLENDORS, a separate daemon is associated with each security in each portfolio (with the exception of special situations where additional daemons that monitor multiple securities are possible). This makes the programming of the daemon simple and yet makes it possible to poll at a frequency most appropriate for the single security being monitored. This arrangement also provides great flexibility by allowing for special, customized monitoring characteristics for any security in any portfolio in the entire system without affecting any other security in any other portfolio.

The principal disadvantage of this organization is that the number of daemons required is huge! However, it should be noted that while there may be some cases requiring specialized daemons, most daemons probably are quite similar with very few variations which can be accommodated through parameterization. Thus it would be desirable to avoid having to reprogram similar daemons repeatedly. PROFIT supports and SPLENDORS uses the notion of generic daemons and libraries of such generic entities. Generic daemons and libraries are expanded upon in Chapter 4.

3.4. SPLENDORS architecture

To accomplish the task of portfolio management and optimization, the portfolio manager needs to be aware of significant price changes as they take place. As noted in the previous chapter, this can be accomplished with significantly reduced network traffic by explicitly distinguishing between daemons and strategies.

SPLENDORS uses a single prices database which maintains the updated prices for all the securities. Alternatives that would maintain copies of this database (or subsets of it) on multiple broker workstations are possible but would require significant effort for synchronization. The PROFIT design supports *sharing* of facets without making actual copies, eliminating such concerns. This ability is exploited in SPLENDORS.

A portfolio can be viewed as a collection of:

- securities
- daemons to monitor the price fluctuations for those securities
- strategies to take advantage of such fluctuations

Portfolios tend to be independent of each other (i.e., the portfolio for customer A operates independently of that for customer B) and need limited (or no) interaction among each other. These characteristics make the representation of a portfolio as an object a natural choice. Such an object would encapsulate the data (securities) and procedures (daemons and strategies) to manage and manipulate the data as appropriate.

The overall architecture of SPLENDORS is illustrated in figure 3-2. The major components of SPLENDORS are daemons, strategies, prices and the generics_interpretive_object. The generics_interpretive_object is a collection of generic daemons and strategies similar to those that comprise the generic libraries except that they are interpretive in nature. This is discussed later in this chapter. Additionally, SPLENDORS contains the glue and control routines to carry out operations (such as purchase stock, create a new portfolio, etc.). Once the SPLENDORS system is built, it runs "on its own", monitors price changes of relevant securities, invokes appropriate strategies when the market fluctuates in an "interesting" way and informs the broker when it is time to initiate some action for one of his/her clients.

A single prices object contains the current prices of all the securities. Each such security is represented by a facet. The prices object also contains a feed_manager facet that manages the prices of the securities. Each portfolio is represented by a separate object made up of several facets. Some of these represent the current prices of the securities that make up the portfolio. In fact these facets are shared among the prices object and all portfolio objects containing that security. The PROFIT language is designed to facilitate sharing of facets. Exploiting that ability, SPLENDORS, maintains a single copy of the price facet for any security regardless of the number of objects interested in it. This makes the task of maintaining consistency of data multiple objects may be interested much easier.

Associated with each security facet in a portfolio, is a *daemon* that monitors the price fluctuations for that stock or option. Since daemons may be constructed per security per object, two portfolios containing the same security may have different criteria for monitoring the price fluctuations. Also associated with each

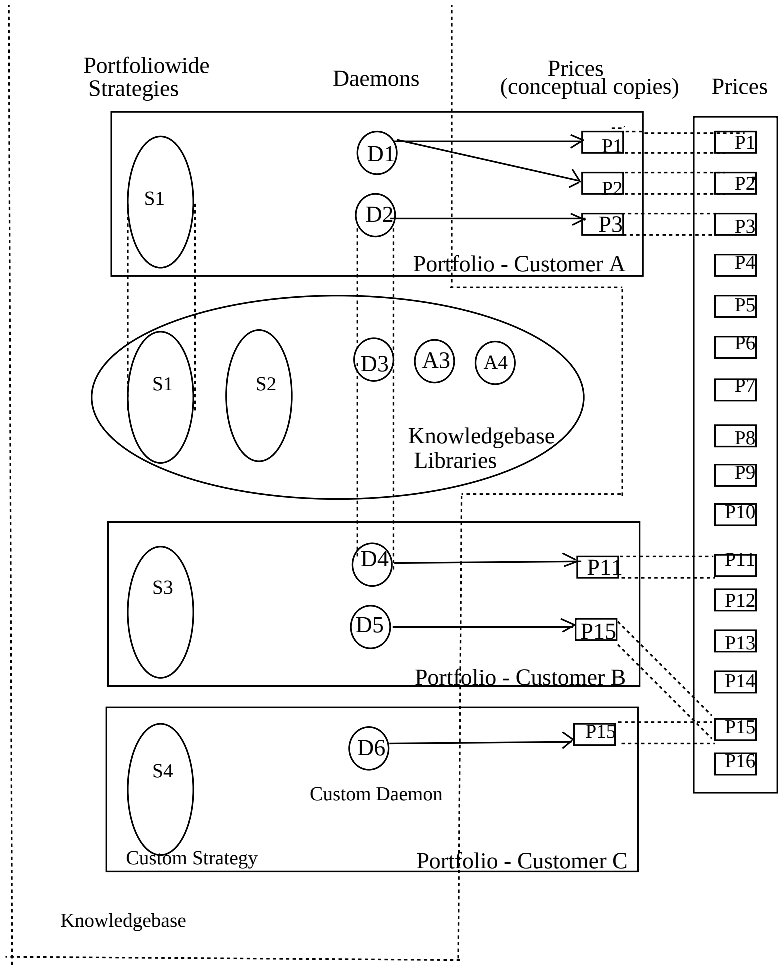

Figure 3-2: SPLENDORS architecture

such security in a portfolio is an accounting facet called security_info_companysymbol_customerid (where companysymbol is the symbol corresponding to the company and customerid is a unique identifier corresponding to a customer) that contains information such as the number of shares, buy price, etc. This data is used in generating reports on that portfolio.

In addition to prices and daemons is the notion of a portfolio-wide strategy. Such a strategy represents the investment philosophy of the investor or fund owning the portfolio. Each portfolio contains at least one such strategy. Also associated with each portfolio is a facet called portfolio_info_customerid to keep track of portfolio-wide accounting information such as cash position, etc. Additionally, each portfolio also contains an initialization facet. The initialization facet is responsible for sending messages to each of its daemon facets to start monitoring the price changes of the appropriate security. These messages are sent asynchronously which results in each such daemon running in its own thread.

Additionally, SPLENDORS contains the generics_interpretive_object. This is used to hold interpretive versions of generic daemons and strategies. These are used to monitor the prices of new securities (purchased since the SPLENDORS system is invoked) and to manage new portfolios (created since the SPLENDORS is invoked).

When the strategy for some portfolio decides it is time to exploit the current market condition by indulging in an action (such as buy or sell), it sends a message to the actions object. This object consists of facets that contain code to execute such actions. The current SPLENDORS actions object contains code that makes recommendations (i.e., construct a recommendation message and inform the SPLENDORS user interface to display it). This allows some human intervention by the broker. Alternatively, once the confidence in such a system is high, code to actually carry out the actions (as opposed to make recommendations to carry them out) could be included in this object.

Finally, SPLENDORS consists of a main_object (consisting of init, portfolio_operations, security_operations, and main_facet facets). The init facet initializes the SPLENDORS system and invokes initialization facets for each portfolio. The portfolio_operations facet is used to add portfolios. The security_operations facet handles orders to buy and sell securities. The main_facet facet contains code to receive, from the user interface, the requests made by the broker and to process them (which is done mainly by sending messages to other facets depending on the request).

Chapter 4

SPLENDORS Design and Implementation - Generic daemons

As discussed earlier, the design point where a separate daemon is associated with each security results in a large number of similar daemons making it highly desirable to have a collection of such daemons which can be customized for specific securities.

For example, an investor may want to monitor several different securities but he/she might be interested in monitoring them using the same criteria (such as sample every *n* seconds, notify the mainline strategy if the price crosses a high threshold *h* or a low threshold *l*). For a scenario like this, it is possible to create and use a generic daemon which can be invoked with parameters *(n, h, l)* to customize it for a specific security/investor. A collection of such generic daemons forms a library which can then be used by the PROFIT programmer in his/her PROFIT programs. PROFIT supports multiple libraries. The parser is informed of the various libraries used via a command line parameter. These are searched in order for an `IMPORT`ed facet.

To allow such generic daemons, SPLENDORS uses the `IMPORT` statement provided by PROFIT. The PROFIT programmer may reuse a previously created generic facet stored in a library by using the `IMPORT` statement in the PROFIT program as illustrated below:

```
FACET foo_daemon: IMPORT generic_daemon
```

A copy of the corresponding C source file (`generic_daemon.c`) is extracted from the library and renamed (as `foo_daemon.c`). The `#include generic_daemon.h` statement in it is changed to `foo_daemon.h`. While the PROFIT programmer may choose to do this manually, in reality, these will usually be automatically created by SPLENDORS as explained later (in Chapter 7).

Here is an example of a generic daemon. This daemon checks to see if the price of a stock goes higher than a certain high threshold or lower than some low threshold and if so, it sends a message to the strategy.

```
FACET gd_absolute
    external cur_price: double
    internal quit: int
    external strategy(p: double, l: double, h: double): int
    internal gd_absolute_daemon(h: double, l:double, t: long): int
END FACET gd_absolute
```

Here's the C code that may be used to implement such a daemon:

```
#include "gd_absolute.h"

int gd_absolute(high_trigger, low_trigger, time_interval)
double high_trigger, low_trigger;
long time_interval;
{
    double temp_price;

    /* initialize slots */
    temp_price = cur_price;
    put_quit(NO);

    /* enter sensitive section */
    SENSITIVE_START;

    EVERYTIME(time_interval)
       if (quit == YES)
          break;

       temp_price = cur_price;

       if (temp_price <= 0)    /* price not yet initialized */
            ;                  /* do nothing */
       else if (    (temp_price > high_trigger)
                 || (temp_price < low_trigger) )
           A$_strategy(temp_price, low_trigger, high_trigger);

    END_EVERYTIME

    SENSITIVE_END;
}
```

The function takes three arguments: `high_trigger`, `low_trigger` and a sampling `time_interval`. If the current price of the security being monitored falls outside the range `[low_trigger, high_trigger]`, the strategy is to be notified.

The function starts with initializing a slot `quit` (see the definition of facet `gd_absolute` above) to the value `NO`. This slot is set to `YES` when termination of the daemon is desired (for example, by a cleanup routine that terminates all daemons by setting the `quit` slot to `YES` for each such daemon). It then enters a sensitive section. The `EVERYTIME` loop is executed every `time_interval` time units. In each such iteration, the value of `quit` is tested to make sure that the daemon is not to be terminated. If it is to be terminated, the `EVERYTIME` loop is exited. If it is not to be terminated, a local copy of `cur_price` is stored into the local automatic variable `temp_price`. The use of `cur_price`, which is an external data slot, results in that value being synchronously retrieved from the facet-slot it is bound to. A local copy is made so that all subsequent uses of it for this iteration refers to the same price (as opposed to if `cur_price` were used in which case it might change within a single iteration). An automatic is used (instead of a static) so that if the same daemon is used to monitor multiple securities, each such invocation has its own instance of `temp_price`. The local copy of this price is then checked to see if it has been set to some initial value (by the simulated feed) and if so, it is then compared to the high and the low thresholds and of the current price. If it crosses either of these, the `strategy` slot is invoked asynchronously.

If the generic daemon is to be used to monitor the price of some stock `foo` represented as a facet in some portfolio object `bar`, the PROFIT programmer would have to create an instance of the generic daemon and perform the requisite bindings for the price and strategy slots.

```
{* the price facet *}
FACET foo
    internal cur_price: double PRIORITY(5, 10)
END FACET foo

{* daemon *}
FACET foo_daemon: IMPORT gd_absolute

FACET strategy_facet
   .
   .
internal strategy(price: double, low: double, high: double): int
   .
END FACET strategy_facet,
   .
   .
   .
OBJECT bar
   .
   .
   .
MAP:
   .
   .
   foo_daemon.cur_price -> foo.cur_price,
   foo_daemon.strategy -> bar.strategy,
   .
END OBJECT bar
```

While this technique would allow reuse of generic daemons, it would require that the SPLENDORS system be recompiled. On a typical trading day, the broker may purchase several securities for various portfolios. Each such security needs to be associated with some daemon to monitor changes in it's prices. Requiring that the system be brought down and recompiled every time a trade needs to be executed would be impractical. What is required is the ability to dynamically add facets to the existing system to represent the newly purchased securities and then associate some daemon with it. This transaction is explained in detail elsewhere. However, this implies that SPLENDORS needs to go beyond providing a library of daemons that can be reused but only by requiring a recompile to providing generic daemons that can be dynamically associated with arbitrary securities in arbitrary portfolios. SPLENDORS accomplishes this by using the limited dynamic reconfiguration features present in CAPITAL currently. It uses an "interpretive" version of such generic daemons as opposed to pre-compiled versions.

This is an example of the `gd_absolute` daemon in its interpretive form. There is no change in the PROFIT file. Only the C files are different.

```
#include "gd_absolute_interpretive.h"

int
gd_absolute_interpretive(high_trigger, low_trigger, time_interval,
            stock_name, customer_name)
    double          high_trigger,
                    low_trigger;
    long            time_interval;
     char          *stock_name;
     char          *customer_name;
{
    double          temp_price;
    char            wrkstr[100];
    char            s[100];

    /* figure out what facet and what slot within that facet   */
    /* contains current price - bind to it                     */
    sprintf(wrkstr, "%s.cur_price", stock_name);

    /* initialize slots                                        */
    put_quit(NO);


    EVERYTIME(time_interval)
       if (quit == YES)
            break;

       /* update binding and read current price                */
       /* a non-sensitive section (don't let others change the */
       /* binding before price is read off                     */
       SENSITIVE_END;
       BIND_SLOT(OBJ_SELF, "gd_absolute_interpretive.cur_price",
                 OBJ_SELF, wrkstr);
       temp_price = cur_price;
       SENSITIVE_START;

       if (temp_price <= 0)    /* price not yet initialized */
            ;                  /* do nothing */
       else if (    (temp_price > high_trigger)
                 || (temp_price < low_trigger) )
           A$_strategy(temp_price, low_trigger, high_trigger);

    END_EVERYTIME

    SENSITIVE_END;
}
```

This daemon is similar to its pre-compiled form shown previously with some exceptions. It takes two additional parameters: `stock_name` and `customer_name`. When the function is entered, the string consisting of the name of the security to be monitored is stored into a local buffer `wrkstr` along with the string ".cur_price" appended to it. The next few lines are similar to the ones in the pre-compiled form. In each iteration of the `EVERYTIME` loop, the `cur_price` slot in this daemon is bound (using `wrkstr`) to the corresponding slot in the facet containing the price of `stock_name`. A message is then sent to it to synchronously get the current price. Once obtained, it is copied into the local variable `temp_price` as before. This binding and request for the current price and making a local copy of it is performed in a non sensitive section to prevent somebody else from updating the binding table while this operation is in progress. The rest of the code is similar to that in the pre-compiled form.

This interpretive version of the daemon is linked into the SPLENDORS system even if no object is using it at that time. This makes all such generic daemons available for use at execution time. The workings of SPLENDORS as it executes a transaction to purchase stock in some company for some customer are explained in the next section.

4.1. Using the generic SPLENDORS daemons

Consider the scenario in which a broker using SPLENDORS decides to execute a transaction such as buying some shares in company bar for customer foo. The broker would begin by choosing the *Buy* action in the *Stock* menu from the initial Splendors menu (illustrated in figure 4-1). SPLENDORS then puts up a panel (figure 4-2) where the broker can provide values for the customer name, the company whose shares are to be purchased, the buy price and the number of shares. Additionally, the broker is allowed to specify via keyboard (or choose from a list via mouse) the name of a generic daemon to monitor the price fluctuations for this security for this portfolio. Based on the daemon specified, SPLENDORS now presents the broker with another panel that prompts him/her for the relevant customization parameters (figure 4-3). Once these are specified, SPLENDORS creates and updates the appropriate facets and objects.

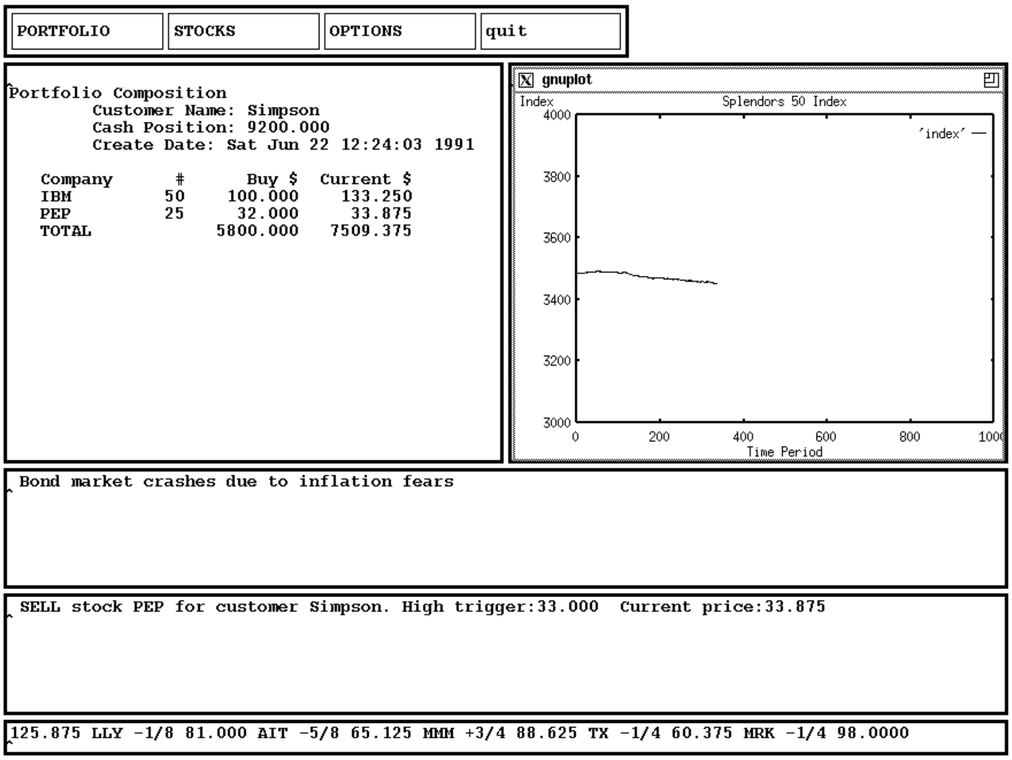

Figure 4-1: SPLENDORS User interface

SPLENDORS creates two new facets to accomplish this.

The first facet called security_info_bar_foo would be used to contain accounting information (represented by data slots) by having appropriate values written to the various slots. This could include things such as number of shares, buy price, time and date of transaction, etc. The second facet created, called daemon_bar_foo, would be used for the daemon associated with this security.

The second facet (daemon_bar_foo) is meant to be the daemon for the newly added security. To do this dynamically without requiring a re-compile, the SPLENDORS system uses a daemon from the generics_interpretive_object (every generic interpretive daemon is loaded in when the system is started). In

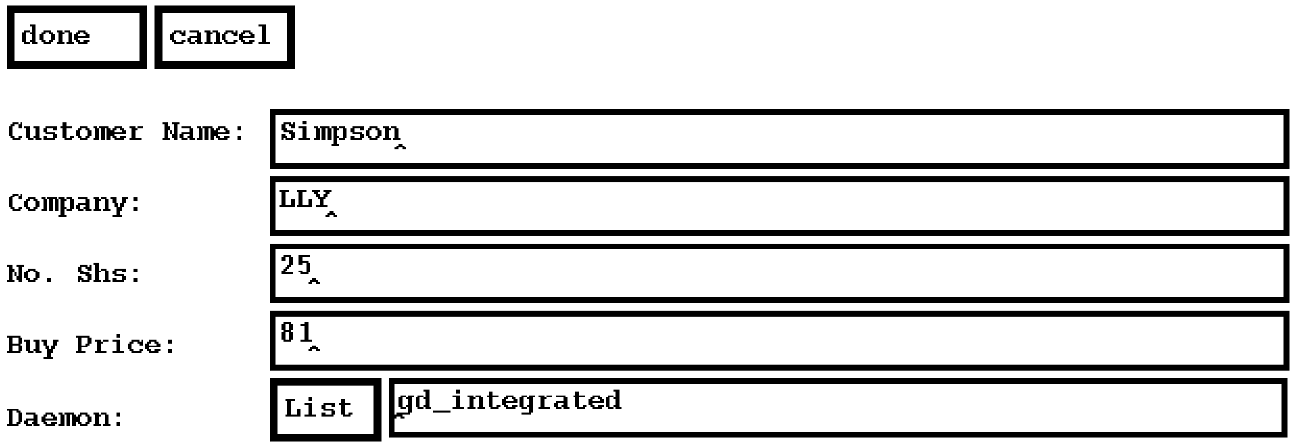

Figure 4-2: splendors User interface - Security/Daemon Specification

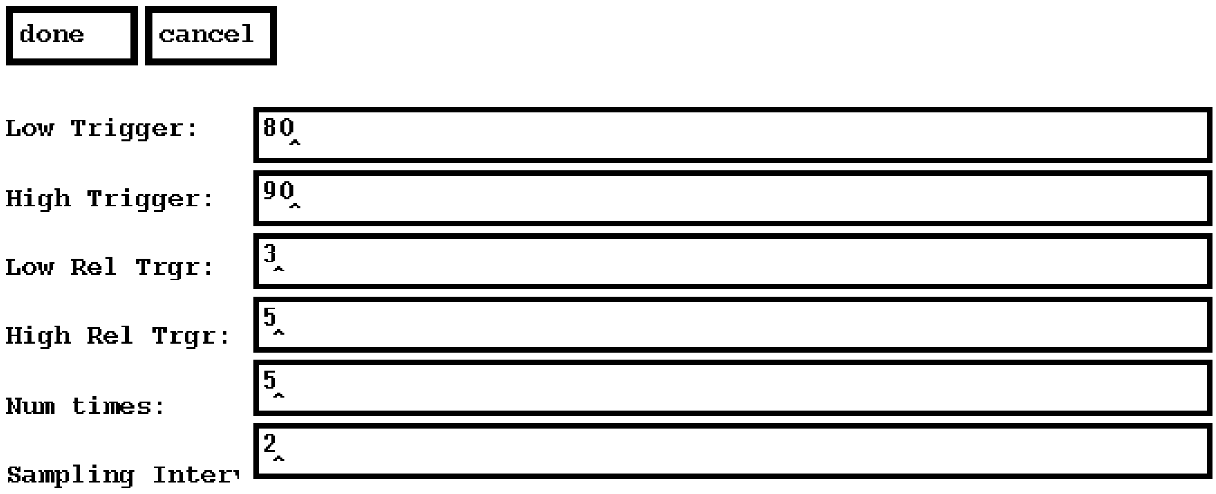

Figure 4-3: splendors User interface - Daemon Parameters Specification

addition to the local data needed by the daemon (provided via invocation parameters and data slots), the code for the daemon needs to be activated. This code is already present in some facet in the generics_interpretive_object. The slot representing the daemon code in daemon_bar_foo is bound to an appropriate entry point to the generics_interpretive_object (based on the choice made by the broker), which in turn binds to some slot in the appropriate facet in the generics_interpretive_object.

Additionally, both the newly created facets have an external slot called `cur_price` representing the current price of the security. To make that accessible, the facet representing the price of `bar` in the prices object is now *shared* with the object `foo`. Then the slot `cur_price` in the facets `security_info_bar_foo` and `daemon_bar_foo` are bound to the price slot in the facet bar. This arrangement continues to ensure that a single copy of the current price for bar is maintained and is shared by all interested facets and objects.

Finally, the portfolio-wide strategy needs to be aware of the newly bought security. To accomplish this, the buy routine simply adds the symbol of the company to a slot known as the composition slot within the portfolio_info_foo facet.

Once all this is done, a message is sent to asynchronously invoke daemon_bar_foo. This results in a new thread being created to execute the daemon. Thus a daemon is now activated for the newly bought security "on the fly" without disturbing the rest of the system and without requiring a recompile.

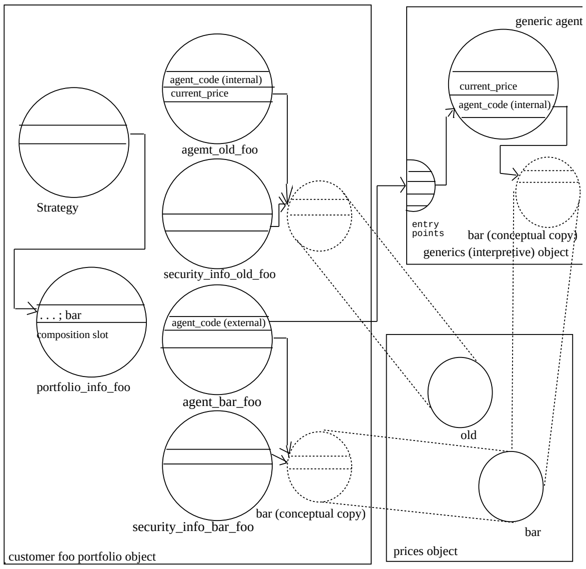

Figure 4-4: splendors dynamic updates

Just as some daemons exhibit a largely generic behavior, so do some strategies. PROFIT constructs can be used to create a library of such generic strategies. If the programmer chooses to use a library of such generic strategies, it is his/her responsibility to ensure that the code can handle additional securities that the portfolio may acquire as the trading day continues. The data structures provided by SPLENDORS can be used to this end. Specifically, SPLENDORS maintains a `COMPOSITION` slot for every portfolio. This slot is accessible to the strategies associated with that portfolio and may be queried by the strategy to discover any changes to the composition of the portfolio.

Chapter 5

Splendors Design and Implementation - Feed and News

Instead of using a real live feed from the stock exchange SPLENDORS uses a simulated feed. The simulated feed runs as a separate UNIX process and communicates with the main SPLENDORS and the SPLENDORS user interface UNIX processes using the standard socket [Sechrest 86, Leffler 86] mechanism.

The feed process generates on a real-time basis,

- updated prices for individual securities
- simulated news messages

and attempts to co-relate the two.

One way to generate the prices would be to use some random number generator to randomly generate a current price for some random stock and then go to sleep for some pre-specified time. When awakened, the same sequence is repeated. Generation of news messages can be accomplished by having a database of such messages and repeatedly execute a loop that randomly chooses one message from this database and then goes to sleep for some time.

The first problem with this approach is that the sleeping for some pre-specified time interval puts a constant load on the system, which is not too realistic. This can be changed to sleeping for some random time (uniformly distributed) between 0 and twice the specified interval in such a way that the mean sleep time is still the pre-specified time interval.

The second problem with this approach is that there may be no constancy between one price for a specific stock and the next price for the same stock. This can be circumvented by using the random number to generate the *delta* which can be applied to the previous price of the same security. This approach would require that prices for all securities be initialized to some appropriate value. The SPLENDORS simulated feed uses a database containing the closing prices on February 22, 1991 to be used as the initial prices.

Third, if an equal number of positive and negative random numbers are generated to generate prices, this would result in a very stable simulated market. It would be desirable to simulate, at will, markets going up and down in addition to the stable markets.

Finally, the news messages being generated should have some correlation with the prices being generated to avoid having situations of the kind where extremely positive economic news is simulated and a market intent on going down are simulated together.

The SPLENDORS simulated feed solves these problems in the following way.

The feed process is parameterized by three parameters - average sleep time between generation of successive new prices for some security, average sleep time between generation of successive news items and the probability of such a message to be positive (i.e., the kind that would result in an upswing rather than a downswing in the market) called `prob_positive`.

The feed process uses a database of 50 stocks, which contains the initial value for the stock price for each company. Once initialized, the feed process executes in an infinite loop generating prices and news and going to sleep for some random time interval (between 0 and 2 * average sleep time).

The feed process also uses another database containing simulated news messages. The database is split into positive messages and negative messages. Associated with each message, is a numerical value called `degree` ranging from -1 to +1. It indicates the impact that particular news item should have on the market.

The feed process maintains a variable called `bias` that ranges in value from 0 to 1 (initially set to 0.5). A value of 0 indicates a completely negative market (every price change is negative) whereas a value of 1 indicates a completely positive market (every price change is positive).

Every time a news message is to be generated, two random numbers `r1` and `r2` are created. `r1` is compared with `prob_positive` to determine if the message should be positive or negative. Once determined, the appropriate database of messages is selected and `r2` is used to select some message from that database at random. To simulate a situation where each successive positive (or negative) message in an already positive (or negative) market have successively smaller effects on the market trends, the `bias` is now adjusted based on the `degree` associated with this message using the following relation:

```
if (bias >= 0.5)
    bias += (1 - bias) * degree;
else
    bias += (bias - 0) * degree;
```

Each time a new price is to be generated, the feed process creates three random numbers `r3`, `r4` and `r5`. `r3` is used to determine the company for which the new price is to be generated. `r4` is used to randomly select the magnitude of the price change (from 0 to 1 in increments of 1/8). `r5` is compared with `bias` to determine the sign of the price change. Once determined, the feed simply adds the price change to the current price of that security (which are independently maintained from the prices database in SPLENDORS) and sends that information to the main SPLENDORS UNIX process and the SPLENDORS user interface UNIX process via sockets.

Additionally, the feed also maintains the SPLENDORS 50 index that is simply the sum of the prices of the securities simulated and writes this out to a file which may be accessed by external programs (other than SPLENDORS). Currently, that file is used to provide a real time graph of the value of the SPLENDORS 50 index.

The feed process uses its own database of prices (as opposed to sharing it with the SPLENDORS process) making it possible to simulate the situation where SPLENDORS may actually miss some price changes (for example, if a large number of prices changes are generated very quickly so that the buffers used to handle such inter-process communication by the socket mechanism fill up resulting in some price change not being communicated and thereby missed). This enables experimentation about the soft real time characteristics of SPLENDORS. Furthermore, the use of a separate UNIX process means that it can simply be replaced by some other process in the future (such as one that might intercept a real feed and simply perform the necessary format conversions to conform to the expectations of the SPLENDORS processes).

One limitation of this simulated feed is that whenever a price is to be generated, every stock in the database has an equal probability of being selected. This in effect results in a market where every company is equally volatile. This limitation occurs because each of the 50 companies is assigned an integral value and whenever necessary, an integer from 1 to 50 is randomly generated to choose the corresponding company. This limitation can be circumvented by assigning a *range* of values to each such company as opposed to a single value. The ranges would be non-overlapping. When a random number is generated, the company whose range contains this random number would be chosen. Stocks with large ranges would be chosen more often simulating volatile stocks compared to those with smaller ranges which would simulate stable stocks. This can be done sometime in the future.

Chapter 6

SPLENDORS Design and Implementation - User interface

The basic SPLENDORS user interface is illustrated in figure 4-1[3]. The user friendly interface is designed for financial analysts who may be familiar with computers as users but would have no knowledge of programming.

The user interface runs as an independent UNIX process and communicates with the SPLENDORS UNIX process and the feed UNIX process via sockets. This has two advantages.

The communication between the user interface process and the SPLENDORS process consists of commands (from interface to SPLENDORS) and recommendations (from SPLENDORS to interface). The communication between the user interface process and the feed process consists of new messages and the prices of the securities (from feed to interface).

The user interface contains enough knowledge about the SPLENDORS commands that it can put up appropriate panels based on the command selected by the user. All responses for the various prompts are collapsed in a single string (in a predetermined format) and passed to the SPLENDORS process for processing.

The separation of the user interface from the rest of SPLENDORS has two advantages. First, the X interface may be replaced with some interface (as long as it follows the communication protocol which consists of passing strings and preceding them with some word to identify them) if necessary. Specifically, the expert user may choose to use a command line interface as opposed to the menu based X interface. Such a user interface can be easily accommodated. A limited command line interface has been created for SPLENDORS to illustrate this.

Second, this separation means that it would be possible to support a setup where the individual broker needs only a relatively inexpensive, less powerful workstation (or X terminal). A more powerful workstation would be shared among a few such brokers (along the lines of departmental servers), to execute their processes and communicate with the mainframe for the prices database.

---

[3]Vanessa Cole developed the X interface for SPLENDORS.

Chapter 7

Automatic Conversion from Interpretive to Compiled Form

An essential task of a portfolio management system is to allow the addition of new securities (i.e. handle the purchase of new securities as the trading day progresses). For a real time portfolio management system such as SPLENDORS, this task is complicated by the fact that not only do such additions to the portfolios have to be supported for things such as accounting, but in fact, these securities have to be monitored appropriately, in line with the philosophy of the investor and in the context of his/her entire portfolio once the purchase is completed.

As mentioned earlier, securities are represented by facets and portfolios by objects. This means that addition of securities to a portfolio has to result in the creation of new facets and addition of customers have to result in the creation of new objects. This requires the alteration of the SPLENDORS program itself. Such alteration done in a traditional fashion would require that the system be brought down and regenerated through a recompile after the new artifacts are added, and then be started again. This is unacceptable in an environment such as the financial markets where many such transactions happen frequently (every few seconds).

To integrate such dynamic actions, it is essential that PROFIT provide a way to add facets and objects dynamically. A general, elaborate scheme to do this has been partially designed [Hailpern 91]. However, due to time constraints, it has not yet been implemented in CAPITAL. To accommodate the needs of SPLENDORS, a simplified version of dynamic reconfiguration is provided by CAPITAL. This simplified version allows the addition of facets and objects, modification of binding rules and provides a way to make `GET` and `PUT` requests in an interpretive fashion. The interpretive version of these requests specify the names of the appropriate objects, facets and slots, which are interpreted by the CAPITAL kernel at execution time and the request is queued to the appropriate facet.

While such interpretive requests make it possible to accommodate dynamic updates to the SPLENDORS system, it does so at the usual cost of flexibility viz. performance because each time such an interpretive request is made, it has to interpreted and resolved at execution time. Pervasive use of these constructs in a large performance-critical application would probably cripple the system. More importantly, the interpretive versions of `GET` and `PUT` requests go against the conceptual philosophy of the underlying PROFIT language because they require the requester to have the knowledge of the target facet and slot - knowledge that really belongs only in the binding rules in the PROFIT code and not in the C code itself. As a result, the use of these should be minimized and for as short as possible (basically confined to the artifacts created due to the activities performed during the trading day).

To provide the flexibility offered by the interpretive version along with the performance offered by the compiled version, SPLENDORS provides a way to convert these interpretive constructs into those that may be pre-compiled, the idea being that such conversion would be done at the end of the trading day allowing the regeneration of a fully compiled version for the subsequent day.

To accomplish this, the following statements must be converted:

- `SGET`
- `SPUT`
- `BIND_SLOT`

Because the `SGET` and `SPUT` constructs were created to provide the function of the `GET` and the `PUT` requests, the conversion needed for these is easy. `BIND_SLOT` requires the update of the binding tables. Additionally, the definitions of the `object` and the `process` PROFIT constructs also have to be updated to reflect the dynamically created artifacts.

To achieve this, SPLENDORS requires conventions that the PROFIT programmer creating artifacts for SPLENDORS must adhere to. It also maintains a list of the dynamically added artifacts. When it is time to perform the conversion from interpretive to compiled form, SPLENDORS uses the original PROFIT file as input, duplicates portions of it, scans the list of the dynamically added artifacts and generates appropriate code for these (using the knowledge about the conventions). The result of this is a new PROFIT file that can be parsed by the PROFIT parser without any further manual actions.

Additionally, C files have to be generated. The PROFIT programmer creating generic entities (such as daemons) that are intended for dynamic use must create two files containing C files - one containing interpretive versions so that it can be used for dynamically added securities. A second file, called the *base file* contains code that will be used in doing the conversion from interpretive to compiled form. There are two main differences between these two files. The first is that the interpretive requests are replaced by ones that would be pre-compiled. The second difference is that all occurrences to the security name and the portfolio names are replaced with some pre-determined character strings (such as `SECNAME` and `PFNAME` respectively) For example, a generic daemon intended for dynamic additions (interpretive) might have something like

```
#include "gdi_absolute.h"
```

and

```
    sprintf(wrkstr, "%s.cur_price", stockname);
    BIND_SLOT(OBJ_SELF, "generics.cur_price", OBJ_SELF, wrkstr);
    temp_price = cur_price;
```

The generic daemon code in the base file intended for use in converting the interpretive version to a pre-compiled version would be coded as

```
#include "daemon_SECNAME_PFNAME.h"
```

and

```
temp_price = cur_price;
```

When SPLENDORS does the conversion, it uses this file and simply substitutes all occurrences of `SECNAME` and `PFNAME` with appropriate names. For the above example, if customer `bar` purchases a security `foo`, this would result in a C file for the daemon containing

```
#include "daemon_foo_bar.h"
```

and

```
temp_price = cur_price;
```

This substitution is done by using the standard UNIX `sed` program in the following fashion:

```
/* form a temporary string containing the UNIX command to execute */
sprintf(temp,
   "sed -e 1,$\"s/SECNAME/%s/g\" -e 1,$\"s/PFNAME/%s/g\" %s>%s",
   new_security,       /* company for which the daemon is being created */
   portfolio_name,     /* customer who owns this security */
   basecfilename,      /* file name of base file for some daemon */
   cfilename);         /* file name for file to hold the pre-compiled version */
system(temp);          /* execute the command */
```

This results in the base file being operated upon by `sed` and a global substitution being performed, replacing all occurrences of `SECNAME` and `PFNAME` by the security name and portfolio name respectively. The changed file is written to the name specified by `cfilename`, which would be previously set to reflect the name of the security, portfolio and the daemon.

This means that if a generic daemon (say gd_absolute) is used for two new securities (say alpha and beta) for customer bar, when SPLENDORS does the conversion, two C files named alpha_bar_daemon.c and beta_bar_daemon.c are created. They have identical code (non-interpretive) except that they refer to slots in different facets (alpha vs. beta) because they use different include files (alpha_bar_daemon.h vs. beta_bar_daemon.h). The include files are generated by the PROFIT parser when the updated PROFIT program is compiled.

It should be noted that no modifications need to be performed to either the updated PROFIT file or the newly created C files. The only thing that needs to be done is that the PROFIT parser needs to be invoked and `make` needs to be invoked (currently, the makefile needs to be manually updated to reflect the additional C files - this restriction can be removed if an additional convention is established viz. that the subdirectory containing SPLENDORS contains no extraneous C files, i.e., all C files that exist in that directory have to be compiled and linked in). The update currently needed is to simply add the name of the file associated with the newly added security to a list of such files. This list is maintained in the variable `USROBJ` in the current makefile.

The limitation of this approach is that strict adherence to conventions is needed by the PROFIT programmer creating generic daemons (or other such artifacts) intended for dynamic use including the names of facets and slots associated with a security. However, the PROFIT programmer creating customized daemons that do not have to be reused in a dynamic fashion need not conform to any of these conventions.

If such a generic daemon for dynamic use is created by the PROFIT programmer, SPLENDORS has to know about it. This knowledge is contained in two places. One is a header file that simply contains constants associated with each such daemon and requires the addition of a definition of a constant associated with the new generic daemon.

```
#define DC_NEW_DYNAMIC_GENERIC_DAEMON 11
```

This definition simply associates a unique numeric value with each generic daemon. It is used in the rest of the code to distinguish between them as appropriate. It is also used to record the daemon associated with a newly purchased security.

The C file in SPLENDORS containing the code to perform the conversion has to have code added so that SPLENDORS knows what base file to use when generating C code and what parameters to specify for the daemon. This is done by adding a `case` to a `switch` statement as follows:

```
case DC_ABSOLUTE:
        fprintf(fpout, "high_trigger: double, low_trigger: double,
                time_interval:long): int\n");
        daemons_list[i] = "gd_absolute_daemon";
        sprintf(cfilename, "%s_pflio%s_daemon1.c", new_stocks[i],
                portfolio_name);
        basecfilename = "base_absolute.c";
        break;
```

This approach to automatic conversion from interpretive to compiled forms is designed specifically for the SPLENDORS system. However, it is accomplished strictly through the use of conventions in SPLENDORS. The CAPITAL kernel knows nothing about this. Such conventions could be established in other applications thereby providing the capability to perform such conversions in other applications in the financial applications domain as well as other domains.

Chapter 8
Future Work

Major areas for future work in CAPITAL/SPLENDORS are in the following areas:

- Replacing the limited dynamic reconfiguration in CAPITAL with a more complete design already partially completed [Hailpern 91].
- Extending CAPITAL to incorporate the multi-processing features of PROFIT.
- Extending PROFIT and CAPITAL to handle distributed systems.
- Extending SPLENDORS (or creating some other sample application) to be a multi-process, distributed system capable of handling many brokers (possibly in the hundreds) and clients (possibly in the thousands).
- Possibly replacing the simulated feed in SPLENDORS with a real feed.

8.1. Multiple Processes

The addition of multiple processes in PROFIT requires extensions to both the CAPITAL parser as well as the run-time kernel. The parser needs to be extended to handle the specification of the various processes. This can be handled by the parser in a way similar to the way it handles multiple objects. Additionally, the parser would have to generate additional code to handle some of the inter-process communication (most of this would be done by the kernel). Finally, the parser might extend the binding table it currently generates to include the information about the process that the target resides on (in addition to the object, facet and slot information it currently generates).

The kernel needs to be extended to be able to handle multiple processes. It might start out by creating one thread per process at initialization to execute the procedure slot specified by the `start` statement for that process. Any of these could result in additional threads due to asynchronous requests. Additional threads may also be created when a facet makes a request to a slot in a facet in some other process. The current PROFIT definition requires one `start` statement per process. This may not be strictly necessary if there are situations where some processes need to be started (from some other process), possibly conditionally. If this is a real possibility, the PROFIT definition may need to be extended.

The PROFIT design requires that each facet belong to one and only one process. This makes synchronization between processes easy (because there are no multiple copies of data to synchronize). Synchronization concerns are handled by the PROFIT programmer through judicious use of `sensitive` sections, `PAUSE` statements and `priorities`.

A facet in one process could make a request to a facet in some other process. The extended binding table (with the process information included in it) would allow the parser to determine the target process at compile time for such inter-process requests. Such inter-process communication might be handled by requiring that the parser generate a `send_ip_request` and `receive_ip_request` functions for each process. When a facet makes a request to some facet in another process, the kernel (in the sender's process) would recognize that and simply call `send_ip_request`. This routine would package the parameters along with some kind of format string to interpret them in a message, which it would then send to the process containing the requested facet. The `receive_ip_request` in the receiving process would receive the request, use the format string to parse it, retrieve the parameters and queue the request at the appropriate facet. The actual communication between the processes could be done through the use of sockets.

One possible problem with this design point is that some kind of processing in `send_ip_request` and `receive_ip_request` is necessary to explicitly determine who the sender and the target are so that the request and the response can be sent to the appropriate processes. In the worst case this would be names (or some identifiers) requiring a lookup in some table (analogous to processing required in processing "interpretive" requests). Such a lookup might be too expensive for a time critical process generated numerous messages frequently (such as the prices process in SPLENDORS). This needs to be investigated to determine if this is a real problem.

Another possibility would be to do away with `receive_ip_request` and `send_ip_request` altogether by using remote procedure calls using the standard RPC mechanism. The CAPITAL parser currently generates `get_slot, put_slot and get_proc` functions. These are designed to be called from the C code. This is done using the C `#define` directives generated by the CAPITAL parser in the header files it creates (i.e., the parser generates these C `#define` directives, which result in these functions being called when the programmer uses a PROFIT construct to access a slot in some facet). The CAPITAL parser could be extended so that it generates these routines (`get_slot`, `put_slot`, `get_proc`) for each process. When the slot being accessed belongs to some other process, these functions would be called on the target process through the RPC mechanism. In the case of asynchronous requests, the function called (`get_slot, put_slot or get_proc`) will return as soon as the request is received and a thread to execute it in is created whereas for synchronous requests, they do not return until the request is complete. This would result in the sender being blocked for synchronous requests as required by PROFIT whereas asynchronous requests would be blocked only long enough to be received by some interprocess server.

The collection of all PROFIT processes would form the PROFIT program. The requisite knowledge about the target processes (for example UNIX process IDs) could be contained in some table that is initialized when the program starts. This table could be maintained in its own process, which would then serve as a name server.

One potential problem with the parser may arise when a very large application consisting of a large number of processes is created. If an application such as SPLENDORS is extended to even a modest environment consisting of 100 brokers (each having his/her own workstation), this could generate a system consisting of at least 101 processes (one for each brokers workstation, 1 for the mainframe running the prices database). What is needed is the ability for the PROFIT parser to generate a separate binary executable file for each process consisting of the code corresponding to the facets in that process created by the PROFIT programmer, code corresponding to the facets in that process created by the parser and the runtime kernel.

8.2. Distributed Systems

Once CAPITAL is extended to handle multiple processes, very few extensions would probably be needed to allow it to be distributed (i.e., allow separate processes to execute on separate workstations on a network). This is especially true if a mechanism such as sockets or RPC, which works in the same way across machines as it works across processes on the same machine, is used as the underlying method of communication between PROFIT processes.

The kernel (or more strictly, the "glue" generated by the parser) would have to be extended so that it contains information on where a particular PROFIT process resides. This information would probably consist of, at least, the machine it resides on and its process id. A thread in the name server process might create such a table (called `location_table`) initially, and then go to sleep (as opposed to terminating) and wake up every time a new process is added. With each such addition, the `location_table` could be updated to reflect the newly added process. The other thread(s) in the name server process can immediately begin using the updated `location_table` to resolved any requests.

Some provision also has to be made for the situation where the facet to whom a request is made does not exist (for example, due to a machine failure). Additionally, some way to handle the situation where the remote machine breaks down in the middle of processing a request is needed. One possible way to handle this would be through a timeout mechanism where the requester gets control back with some error code after some timeout value expires. In the case of asynchronous requests, the situation is more complicated because PROFIT intentionally does not provide a way to indicate the completion of the asynchronous request, thus making it difficult to detect the situation where the processing of an asynchronous request begins but never ends (due to a machine failure in the middle of such processing).

An additional possibility is the dynamic integration of separately compiled PROFIT processes into a running PROFIT program. This would result in the process maintaining the `location_table` being notified, which would then propagate that knowledge to all other processes in the system.

8.3. Distributed Splendors

Once a multi-process, distributed version of CAPITAL is completed, SPLENDORS could be extended to take advantage of these capabilities to provide a system where each broker uses his/her own workstation to run a PROFIT process that would execute all his/her portfolio objects. The prices database would run on a mainframe and be shared among all the analyst workstations.

The main problems with such a system are likely to be performance and size related.

If our modest system of 100 brokers consists of, on average, 100 clients each, we end up with a system with 10000 portfolio objects. If each such portfolio consists of five securities on average, the total system consists of 50000 securities. Since each such security results in two facets under the current SPLENDORS architecture, our system would be 100000+ facets big.

The runtime kernel will also have to be designed to handle these problems. The current CAPITAL design consisting of a single binding table for the entire system is clearly inappropriate when the system could consist of 100000+ facets. This can be easily alleviated by having a distinct binding table for each process, which would also better conform to the PROFIT model.

The biggest challenge in such a system is likely to be performance related. Consider the communication requirements in such a system. If on average, each portfolio requests a single price from the prices database every second, we still end up with 10000 requests per second. If the prices database runs in its own process independent of processes that execute the portfolio objects, these are all inter-process requests. Moreover, this is only an *average* demand on a relatively modest system. The *peak* demand for a system that provides for growth is probably at least an order of magnitude larger. If this communication demand is not carefully considered, it could easily cripple the system. The ability in PROFIT to split up an object among multiple processes may be used to split up the prices database among more than one mainframe should this become necessary. It would also be alleviated considerably if careful distinctions are made between volatile and non-volatile securities and daemons constructed and invoked appropriately. Carefully distinguishing between trading and investment strategies could also help.

Chapter 9
Related work

The work in this thesis relates to the following topics:

- Real time portfolio management
- Application-specific application generators [Bernstein 90]
- Generic software entities (such as strategies, daemons, etc.)

There is a plethora of software that claims to be portfolio management software. Most of this, however, addresses a specific area of portfolio management - security selection. While security selection is an important aspect of portfolio management, the two are by no means the same [Brennan 90]. Many of these portfolio managers do not have any real time characteristics. They allow the user to input the current price along with other parameters to do an analysis of the merit of the particular security.

PORT-MAN [Chan 88] in an investment consultation system created with the banking industry in mind. While not limited to securities, it describes the use of frames to represent individual investor goals, specific instruments not unlike the way facets are used to represent strategies and securities. However, *PORT-MAN* is primarily a goal decomposition system and not a portfolio management system. It has no real-time content at all. Instead, it is designed as a consultation system.

Leinweber [Leinweber 88] describes a software system that incorporates real-time characteristics and an "alarm window" that displays securities whose alarms have been triggered due to the violation of some threshold. However, unlike SPLENDORS, this system is geared to traders. It does not describe how such thresholds can be associated with specific customer portfolios depending on the investors philosophy.

The real-time nature of SPLENDORS, combined with the ability to have arbitrary computations specified as strategies means that such analyses can be carried out on a real-time basis, automatically taking into account the latest market conditions. Furthermore, by having the philosophy of the individual investor incorporated in such strategies (or having separate strategies to represent the investor’s philosophy) means that the analysis carried out in real-time and is specific to the individual investor. A certain change in the market may violate the threshold for a certain investor while it may not be interesting to some other investor. SPLENDORS will raise the alarm only for the appropriate investors.

The underlying motivation behind PROFIT (and SPLENDORS) is to take an intermediate approach between polling and active value propagation. Chakravarthy [Chakravarthy 90] does some performance analysis between the two approaches and concludes that while the active value approach might be better in general, this is not always the case. The flexible approach offered by PROFIT can be used to incorporate advantages of both polling and the active value propagation approach.

SPLENDORS allows the user (broker) to create new portfolio managers for additional customers or to modify existing portfolios due to the purchase/sale of existing securities. The resulting portfolio manager is integrated in the running system. When the system is brought down, SPLENDORS generates code to reflect all such changes so that a pre-compiled version can be started the following trading day. In effect, SPLENDORS generates code much like an application generator. However, it is very specific to a particular application. Thus it is essentially an application specific application generator.

Igor [Bernstein 90] is another example of an application specific application generator created to translate VIF specifications into declarations which are then subsequently used by the VHDL analyzer. It does not, however, generate code that can be integrated into a running system without having to bring it down as in SPLENDORS.

Chapter 10
Evaluation and Conclusion

The primary purpose of SPLENDORS is to show the feasibility of the use of PROFIT to create such an application. This has been achieved. SPLENDORS succeeds in illustrating how the market may be monitored in a real-time fashion to optimize individual portfolios, each of which could follow a different investment philosophy.

SPLENDORS uses the intermediate approach between polling and active value propagation supported by PROFIT. Although it is not currently supported, it is possible to include code in SPLENDORS to count the number of times the prices are updated and the number of times messages are actually sent. Such a capability can be used to run experiments and obtain numerical results on the benefit provided by such an intermediate approach in this environment.

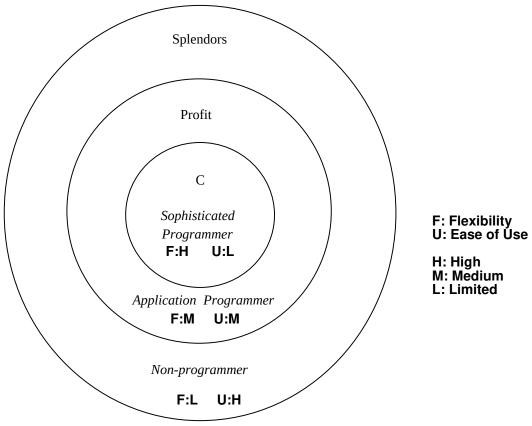

Figure 10-1: SPLENDORS/PROFIT/C system

SPLENDORS uses PROFIT as its underlying basis which in turn uses C as its underlying basis. This arrangement results in various levels of flexibility and ease of use as depicted in figure 10-1.

The ability to integrate C code easily means that the programming department in the brokerage firm can create customized specific strategies. Additionally, they can also create specific generic strategies and make them a part of the knowledgebase. The non-programming broker can then create (and customize via parameter specification) many portfolio managers from such a knowledgebase without doing any programming at all.

The benefit provided by PROFIT can be seen from the amount of code written by the PROFIT programmer versus the C code generated to create an executable system. The C code written to create the SPLENDORS system consists of approximately 2500 lines of C code. The feed manager is an additional 500 lines of C code and the X user interface is an additional 1,000 lines of C code. The PROFIT parser expands this C code and then in addition to the expanded source, generates approximately 3,000 lines of additional C code. This does not include the PROFIT kernel (about 1,000 lines of C code). This illustrates the power of PROFIT and the level of relief provided by it to the PROFIT programmer. PROFIT succeeds in handling the various scheduling and co-ordination concerns and completely freeing the PROFIT programmer from them.

SPLENDORS makes it possible for the non-programming broker to create portfolio managers that can monitor, in real-time, the market and suggest actions tailored to the investment philosophy of the investor. It allows the broker to integrate new securities, daemons and portfolios in an executing SPLENDORS system instance without the need to bring it down.

SPLENDORS also illustrates how the knowledge of the specific domain can be used to define an architecture (primarily, the organization of various components and a set of naming and other conventions) that can be exploited to make such dynamic integration possible at a minimal cost. It illustrates the use of interpretive code to achieve flexibility, which is then converted to a compiled form as soon as possible to achieve performance. The interpretive method of adding components dynamically is clearly limited and inconsistent with the PROFIT design of not allowing the programmer to get control of arbitrary handles. It should be replaced by the more elaborate method of handling dynamic updates already partly designed [Hailpern 91].

Finally, SPLENDORS illustrates the use of libraries of pre-defined generic components created to conform to the SPLENDORS architecture. Since such conformance is the only requirement, this shows how one can create such generic reusable components for other PROFIT applications. The principal limitation of this work is that it is restricted to a single process. This is intentional as this work is the first phase in this project.

Another concern is the performance requirements in the future systems. Further work on this project should clearly define the performance requirements early on.

Another limitation is the rather crude way of handling news items. In effect, news items are simply displayed. It is up to the individual broker to interpret them and update any data in daemons if appropriate. If this system were to be re-designed, some thought might be given to how one could go about integrating some expert system with SPLENDORS so that news items can be used to trigger appropriate strategies.

Chapter 11
Acknowledgments

Mike Mayer implemented the basic basic library facility in PROFIT and Vanessa Cole is responsible for the SPLENDORS user interface. George Beltz of The New York Stock Exchange provided information on the prices feed. Ken Hardy of Bridge Information Systems, St. Louis supplied the data on IBM trades on July 19 1991. We would like to thank Dan Schutzer of Citicorp and Catherine Lassez of IBM for useful discussions and Shyhtsun Felix Wu for bringing the dynamic thresholds example to our attention.

Appendix A

CAPITAL Users Guide

A PROFIT program consists of the following components:

- PROFIT source
- C source (created by the PROFIT programmer)
- Header files for C source (generated by PROFIT parser)
- C source (created by the PROFIT parser)
- Kernel object files

The PROFIT syntax and various PROFIT statements are explained and illustrated in chapter 2.

Each facet containing internal procedure slots has a C source file of the same name as the facet name. It contains one function for each internal procedure slot. The C source file must have an `#include "facetname.h"` statement where `facetname` is the name of the facet. This header file is created by the PROFIT parser when it is invoked against the PROFIT file.

In addition to creating the various header files, the parser also generates the following C source files:

- _bind.c
- _priority.c
- _typesizes.c
- _comm.c
- <processname>.c

To create an executable PROFIT program, the following steps should be followed:

- create the PROFIT source file (for example, prog.p)
- create the appropriate C source files with the appropriate `#include` statement
- invoke the parser against the PROFIT source file `profit prog.p`
- invoke the C compiler against the C source files (user created and parser generated) with the pathname of the directory containing permanent include files specified via the -I flag cc -c -I/proj/profit/src/incrun abc.c
- if the kernel object files are not available, compile the kernel source with the pathname of the directory containing the kernel header files specified via the -I flag cc -c -I/proj/profit/src/comm sched.c
- link the object codes for the C files created by the PROFIT programmer, the C files generated by the parser and the kernel files

These steps have been incorporated in a `makefile` (/proj/profit/demo/splendors/makefile) which can be used instead of executing these steps one at a time.

References

[Bernstein 90] David B. Bernstein and Rodney Farrow.
Automatic Maintenance of Routine Programming Tasks Based on a Declarative Description.
In *12th International Conference on Software Engineering*, pages 310-315. Nice, France, March, 1990.

[Brennan 90] Peter J. Brennan.
Portfolio Managers' Software Wish List.
Wall Street Computer Review , February, 1990.

[Chakravarthy 90] Sharma Chakravarthy and Susan Nesson.
Making an Ojbect-Oriented DBMS Active: Design, Implementation, and Evaluation of a Prototype.
In *International Conference on Extending Database Technology*, pages 393-406. Venice, Italy, March, 1990.

[Chan 88] Y. Y. Chan, E. G. Saw and T. S. Dillon.
An Expert System For Portfolio Management Using Both Frames and Production Rules.
In *8th International Workshop on Expert Systems and their Applications*, pages 463-481. Avignon, France, May-June, 1988.

[Hailpern 91] Brent Hailpern and Gail E. Kaiser.
Dynamic Reconfiguration in an Object-Based Programming Language with Distributed Shared Data.
In *11th International Conference on Distributed Computing Systems*, pages 73-80. Arlington TX, May, 1991.

[Kaiser 90] Gail E. Kaiser and Brent Hailpern.
An Object-Based Programming Model for Shared Data.
In *1990 International Conference on Computer Languages*, pages 136-144. New Orleans LA, March, 1990.

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

[Leffler 86] Samuel J. Leffler, Robert S. Fabry, William N. Joy, Phil Lapsley, Steve Miller and Chris Torek.
An Advanced 4.3BSD Interprocess Communication Tutorial.
UNIX Programmer's Manual Supplementary Documents.
University of California, Berkeley, 1986, pages PS1:8-1 - PS1:8-41.

[Leinweber 88] David Leinweber.
Knowledge-Based Systems for Financial Applications.
IEEE Expert :18-31, Fall, 1988.

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

[Sechrest 86] Stuart Sechrest.
An Introductory 4.3BSD Interprocess Communication Tutorial.
UNIX Programmer's Manual Supplementary Documents.
University of California, Berkeley, 1986, pages PS1:7-1 - PS1:7-25.

[Washington 89] Richard Washington and Barbara Hayes-Roth.
Input Data Management in Real-Time AI Systems.
In *11th International Joint Conference on Artificial Intelligence*, pages 250-255.
August, 1989.

Table of Contents

List of Figures