Choices, *Frameworks and Refinement**

Roy H. Campbell, Nayeem Islam,
and Peter Madany
University of Illinois at Urbana-Champaign

ABSTRACT: We present a method for designing operating systems as an object-oriented framework of generalized, abstract components. The framework is specialized into further subframeworks to implement subsystems of the operating system. Each subframework introduces constraints and relationships between the abstract classes of the components. The constraints are inherited by the instantiations of the framework. *Choices* is an object-oriented operating system designed and implemented using frameworks. In this paper, we explain the application of our design approach to *Choices*. We describe the following subsystems and their subframeworks: virtual memory, persistent storage, process management, message passing and device management subframeworks. We discuss the advantages and disadvantages of using frameworks to design and implement object-oriented systems.

* This work was supported in part by NSF grant CISE-1-5-30035 and by NASA grants NSG 1471 and NAG 1-163.

1. *Frameworks in an Object-Oriented Operating System*

Frameworks [6, 5] characterize the architectural design of an object-oriented system. The Model-View-Controller of Smalltalk-80 systems [14] and the Unidraw graphical editor [24] are two documented examples of frameworks for graphical user interfaces. In this paper, we present a framework for *Choices,* an object-oriented operating system.

The design of *Choices* [3] comprises a hierarchy of frameworks. In that design, the concept of a framework subsumes the conventional organization of an operating system into layers [22]. Frameworks not only allow the design of layers, but they also permit the construction of more complex structures. The use of frameworks permits design and code reuse and the consistent imposition of design constraints on all software, independent of the level at which it may be used.

The object-oriented operating system approach builds system software that models system resources and resource management as an organized collection of objects that encapsulate mechanisms, policies, algorithms, and data representations. A class defines a collection of objects that have identical behavior. Class hierarchies define relationships between classes that share common behavioral properties. Inheritance and inclusion polymorphism permit the methods of a concrete subclass to implement operations on an abstract class. A framework of classes defines an architecture that expresses the organization of an object-oriented design of a system. The framework can be refined into subframeworks, corresponding to the composition of a large complex system out of smaller interacting subsystems. A particular operating system implementation is just one of many possible ways that a framework for an operating system can be "instantiated."

Choices was designed from the beginning as an object-oriented operating system implemented in C++. The system runs stand-alone on the Sun SPARCstation II, Encore Multimax, Apple Macintosh IIx, IBM PS/2, and AT&T WGS-386. It supports distributed and shared memory multiprocessor applications, virtual memory, and has both conventional file systems and a persistent object store. The system has over 300 classes and 150,000 lines of source code.

In this paper we will describe how we have used the object-oriented notion of a framework in our work. *Choices* has the following frameworks: process management and exception handling, scheduling, synchronization, memory management, persistent storage, device management, message passing, communication protocols, application interface, and instrumentation. We will discuss how particular frameworks in *Choices* have contributed to the organization of the system, techniques we have found helpful for building frameworks, and why frameworks are useful.

In Section 2 we review the concept of a framework, and in Section 3 we discuss the techniques for building frameworks. Section 4 introduces the frameworks in the *Choices* object-oriented operating system. Each of the major frameworks of *Choices* are then described in turn: the virtual memory subframework in Section 5, the process subframework in Section 6, the persistent storage subframework in Section 7, the message passing subframework in Section 8, and the device management subframework in Section 9. The advantages of explicitly using a framework for design are discussed in Section 10, and Section 11 describes how, in practice, subframeworks evolve over time and what changes we have made to the *Choices* frameworks. In Section 12, we conclude by reviewing the lessons we have learned from using frameworks to design an object-oriented operating system.

2. *What is a Framework?*

A framework is an architectural design for object-oriented systems. It describes the components of the system and the way they interact. In frameworks, classes define the components of the system. The interactions in the system are defined by constraints, inheritance, inclusion polymorphism, and informal rules of composition (see Section 3 for details on these techniques). *Choices* frameworks use single inheritance to define class hierarchies and C++ subtyping to express inclusion polymorphism. In practice, we have found that the design of a complex system such as an operating system is best defined as a framework that guides the design of subframeworks for subsystems. The subframeworks refine the general operating system framework, as it applies to a specific subsystem.

The framework for the system provides generalized components and constraints to which the specialized subframeworks must conform. The subframeworks introduce additional components and constraints and subclass some of the components of the framework. Recursively, these subframeworks may be refined further. Frameworks simplify the construction of a family of related systems by providing an architectural design that has common components and interactions. An instance of a framework is a particular member of the family of systems.

Frameworks both support and augment the traditional layered approach that has been used to design operating systems. In both approaches the problem domain is divided into smaller domains. A layer represents an abstract machine that hides machine dependencies and provides new services. The abstract machine is presented as a set of subroutines. A framework introduces classes of components that encapsulate machine dependencies and define new services. A layer introduces an interface between implementations that is constrained by the set of calls that are defined. A framework defines interfaces in the form of the public methods of abstract classes. It imposes restrictions on the implementation of an interface by the constraints it imposes. In the layered approach, the design of each layer is independent. Algorithms or data structures in one level may be similar to those in other levels, but the level approach to design has no way to express that similarity. Instead, a framework may have several different instantiations and implementations within a system; it may be reused. The constraints of a framework allow more complex interactions than between levels. The framework approach subsumes the layered approach because the basic properties of the layered approach can be modeled by frameworks. However, the framework approach also allows the constraints within a particular layer to be expressed. Finally, a framework can be defined in terms of abstract classes that are bound to specific concrete classes at run-time using inheritance and inclusion polymorphism. This provides the compile time independence that is exhibited by, for example, the application interface layer but also allows dynamic binding as, for example, is necessary to allow device drivers to be added or changed in a running system.

3. Techniques for Building Frameworks

In this section, we identify and describe some useful techniques for implementing frameworks. We provide example uses from *Choices*.

- *Abstract* classes provide generalized interfaces for *concrete* classes. *Concrete* classes are implementations of *abstract* classes. A framework of abstract classes introduces constraints between objects in the system that are specialized and augmented by corresponding concrete classes. In *Choices,* the persistent storage framework involving *PersistentStores, PersistentStoreContainers,* and *PersistentStoreDictionaries* introduces constraints on the partitioning of various kinds of disks, the provision of various formats of logical files, and the implementation of various methods of file naming.
- *Inclusion Polymorphism* refers to a subclass being a subtype of a superclass. This allows a subclass to be used wherever a superclass is expected. In *Choices,* all devices and device controllers are derived from a device-driver framework. Any device written to use an abstract controller interface may use any instantiation of a controller such as a SCSI bus controller. Further, given a request for a particular implementation of an I/O interface, the system is free to bind that request to any convenient implementation of the interface provided that the class of the object requested and class of the service offered satisfies the subtyping requirement.
- *Constraints* are descriptions of relationships between abstract classes of frameworks or the relationship between concrete and abstract classes within a framework. The use of constraints is most evident in how instances of concrete classes are combined. For example, in the message passing subframework certain primitives intended for distributed memory computing cannot be mixed with those that assume a shared memory architecture.
- *Dynamic code loading* allows one to specify an abstract class when the system is designed and add a concrete descendant class of the abstract class at run-time. For example, a *Choices* device driver consists of a *DevicesController* class and a number of *Device* classes. A new device driver can be added to the system by loading the concrete subclasses of the

DevicesController and the *Device* classes that form the device driver.

- *Delayed Binding* is the ability to determine dynamically the methods to which an object responds (often referred to as the signature of the object). In object-oriented systems this binding is not known until run-time. In C++, delayed binding is a result of using virtual functions. All abstract classes in *Choices* use virtual functions.
- *Conversion* allows objects to be changed at run-time into other objects. Conversion does not modify the original object; instead, a new one is created using the data of the old object. Subclasses of *ProxiableObject* implement the conversion process by responding to the `asA` message [16]. The method takes an argument that may be the name of either a concrete or an abstract class and returns a reference either to an instance of the argument or an instance of a concrete subclass of the argument, respectively. The `asA` method uses the `supports` method to ensure that the underlying data is compatible with the given class. For example, in the *Choices* device management subsystem a serial line can be converted to an input stream and an output stream. In the persistent storage system, a persistent store can be converted into a persistent object.

Another type of constraint we describe involves abstract class relationships. We will use a modified version of Pressman's [20] entity-relationship and instance connection notation to describe abstract class relationships. We also annotate the links between the abstract classes with labels to make the type of relationship between classes more explicit. By presenting information about abstract classes we are able to provide a concise, yet high level description of the system. These relationships are maintained by the instances of the concrete class implementations of the abstract classes. Figure 1 shows the notation we will use to represent one-to-one and one-to-many relationships. These relationships may be mandatory (denoted by an additional line) or optional (denoted by a small circle).

Figure 1: Description of Links between Abstract Classes

4. Choices Frameworks

The framework for *Choices* defines abstract classes that represent the fundamental components of an operating system. Subframeworks specialize these components for use in the context of a subsystem of the operating system. The *Choices* framework imposes design constraints upon the subframeworks, which ensure that they may be integrated into a coherent system. The *Choices* framework consists of three abstract classes: `MemoryObject`, `Process`, and `Domain`. Figure 2 shows an abstract class relationship diagram that defines how these components interact. The classes represent the three general components from which operating systems are built: storage for data, threads of control which execute a sequential algorithm, and an environment that binds the names processed by the threads of control to storage locations. The figure shows that a `Process` must have a `Domain` and that several `Processes` may have the same `Domain`. The `Domain` has several `MemoryObjects` that store program code and data. A `MemoryObject` may be associated with one or more `Domains`. Specializations of the components are required in order to implement the different subsystems of an operating system. For example, the `MemoryObject` is specialized in the virtual memory subsystem to represent both physical memory and virtual memory. In the file system, the `MemoryObject` is specialized to represent disks and files. The constraints imposed between the abstract classes of the *Choices* framework are inherited by the subframeworks. Thus, in either case one or more `Domains` may be associated with a file, a virtual memory, or with physical memory. Similarly, `Processes` are associated with a `Domain`.

In more detail, the system has one kernel `Domain` with which are associated various system `Processes`, see Figure 3. System `Processes` execute operating system programs. Physical memory is mapped into virtual memory one-to-one and is represented by a `MemoryObject` associated with the `Domain`. Storage associated with a

Figure 2: An abstract class relationship diagram for the three fundamental components of Choices

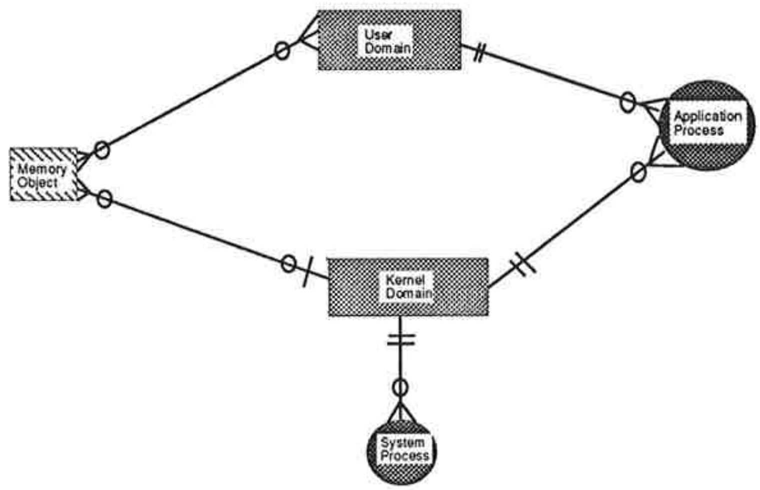

Figure 3: An abstract class relationship diagram for the three fundamental components of Choices

running `Process` is allocated by the kernel `Domain` from this `MemoryObject`. Figure 3 shows an abstract class relationship diagram that defines how these components interact.

Application processes are associated with user `Domains`. The virtual memory used by a user process is divided into regions represented by `MemoryObjects` or data stores associated with the `Domain` of the application. `ApplicationProcesses` execute application programs in "user mode." They may also execute operating system procedures in the kernel in "supervisor mode." When a user `Process` executes in user mode, it is associated with a user `Domain`. When a user `Process` executes in supervisor mode, it is associated with the kernel `Domain`.

A particular region of virtual memory can be shared between two or more concurrent applications. In this case, the `MemoryObject` representing the region is associated with two or more user `Domains`. A region of virtual memory can also be shared between applications and the operating system code. In this case, the `MemoryObject` is associated with both the kernel and user `Domains`.

Before describing the subframeworks built from the *Choices* framework, we must clarify what it means for *Choices* to be an object-oriented system. In Figure 4, we describe the constraints that we impose on the fundamental components of *Choices* to make the system object-oriented. Each object in the system is related to a class and this

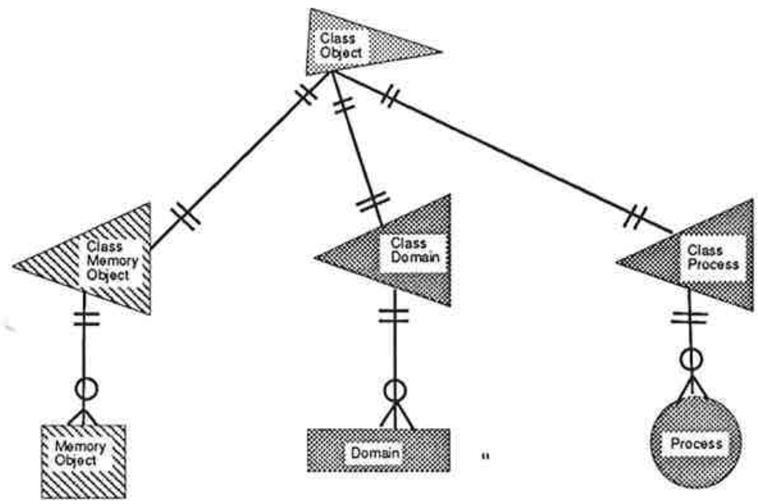

Figure 4: Runtime representation and access of Classes in Choices

relationship is represented explicitly at run-time. *Choices* objects represent classes and the class `Class`. Each constraint is implemented at run-time by a link between objects. For example, the figure shows instances of `MemoryObject`, a `MemoryObject` class object and an object representing the class `Class` and includes the subclassing and instantiation relations.

The components of the subsystems of *Choices* are defined in various subframeworks. For example, the virtual memory subframework [12] inherits the constraints imposed upon `MemoryObject` and `Domain`, defines specializations of these classes, and introduces new abstract classes to define the necessary additional components that are required to implement a virtual memory system.

In the next five sections, we discuss the following subframeworks of *Choices:* virtual memory, process management, persistent storage, message passing, and device management.

There are four parts to the description of a *Choices* subsystem and its subframework. First, a generalized set of orthogonal **components** is defined. Second, an **architectural overview** of the subframework is given. The architectural overview consists of the abstract classes corresponding to the components. A set of concrete classes that are implementations of the abstractions are also described. Instances of these concrete classes constitute the *Choices* operating system at runtime.

The abstract class relationship diagram for the subframework specifies the *constraints* between the various classes. Third, a **design overview** provides detailed accounts of the methods of the classes. Finally, the interaction of this subframework with the rest of *Choices* is provided.

5. Virtual Memory

The *Choices* virtual memory system supports multiple 32 bit virtual memory address spaces, one and two level paging and shared memory. The system is implemented by representing the components of the system as objects. Each virtual memory is supported by a `Domain` which provides the mapping between the virtual memory addresses used by processes and storage. A virtual memory can provide access to multiple different data stores. Each data store is mapped into a region of virtual memory and is represented by a `MemoryObject`. The data store represented by each `MemoryObject` is paged and may be larger than physical memory. Multiple applications may share the data in a `MemoryObject` by mapping it into each of their `Domains`. A `MemoryObject` may be shared across a network by multiple applications using distributed shared virtual memory techniques. Memory mapped files are supported by allocating a `MemoryObject` that represents the file. In this section, we examine the virtual memory system framework of *Choices*.

COMPONENTS The virtual memory system of *Choices* has the following components:

1. The MemoryObject
   represents a data store. The store might contain a process stack, code, heap, or data area of a program. Any one of several subclasses of `MemoryObject` may be used, including subclasses of `PersistentStore` that represent various kinds of disk and files. When a `MemoryObject` is cached in memory, virtual memory addresses may be used to reference the contents of the store. The `MemoryObjectCache` that caches the `MemoryObject` pages the contents of the store to and from the data store into physical memory.
2. The Domain (Address Space)
   maintains the mapping between virtual addresses and data stores. When a `MemoryObject` is added to a `Domain`, the

`Domain` assigns a virtual address range to the contents of the data store and builds a `MemoryObjectCache` to cache the contents of the data store in physical memory. Processes accessing the data store use virtual memory addresses. If the data in the data store is not resident in physical memory, a page fault will occur. The `Domain` maps a page fault virtual address into an offset within the data store and sends a message to the appropriate `MemoryObjectCache` to fetch the appropriate page of data from the data store.

3. The PageFrameAllocator
   allocates and deallocates physical memory. It is used by the virtual memory system to reserve pages for paging.
4. The AddressTranslation
   encapsulates the address translation hardware of the computer. The virtual memory system makes requests to `AddressTranslation` to add and remove virtual memory to physical memory page mappings.
5. The MemoryObjectCache
   stores the mapping between virtual memory pages of a `MemoryObject` and the physical memory pages in which the data is actually stored. The mappings in the `MemoryObjectCache` are maintained in a machine independent form.

ARCHITECTURAL OVERVIEW This section describes the virtual memory system class hierarchy and the relationships between its abstract classes. The class hierarchies for the virtual memory system are shown in Figure 5. These hierarchies show the abstract classes and their concrete subclasses. The abstract class interfaces are preserved in the subclasses and superclass code is reused in the subclasses.

Figure 6 is an entity diagram showing the relationship between the abstract classes of the virtual memory subframework. The diagram

```
AddressTranslation —— TwoLevelPageTable

MemoryObject —— PersistentStore ─┬─ File
                                 └─ Disk

                  ┌─ PagedMemoryObjectCache
MemoryObjectCache ┤
                  └─ SpecialPurposeCache ——— PrePagedMemoryObjectCache
```

Figure 5: Virtual Memory System Class Hierarchies

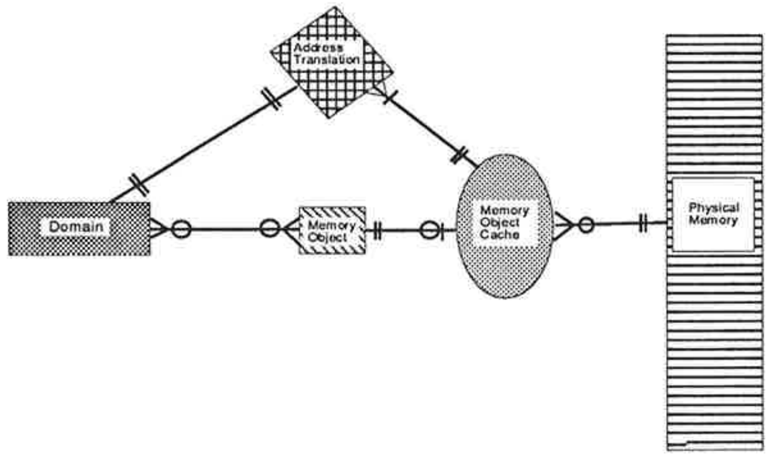

Figure 6: An Entity diagram showing the relationship between objects of the virtual memory subframework

shows one or more Domains sharing one or more MemoryObjects. The MemoryObject reads and writes data from and to a data store represented by a FileObject. Each MemoryObject that is mapped into virtual memory has a corresponding MemoryObjectCache that records in a machine independent way the mapping of pages of the data store that have been copied into physical memory. The Domain handles page faults by requesting the MemoryObject to read the page into physical memory. The MemoryObject returns the page frame that contains the data to the Domain which then adds a virtual memory mapping for that page frame to its AddressTranslation object. Each Domain has its own AddressTranslation object but in a single processor system, only one of the AddressTranslation objects will be active at any one time. Page replacement algorithms may free physical memory pages for reuse. For each physical page, the MemoryObjectCache records all the AddressTranslations that map virtual addresses to that page. The page replacement algorithm selects pages of information to return to the data store and removes the hardware virtual memory mapping by making requests to the appropriate AddressTranslation.

DESIGN OVERVIEW This section discusses the methods of particular classes in more detail. A MemoryObject supports access to an array of equal-sized logical units, where each unit is a block of bytes. A unit corresponds to a disk block, physical memory page frame, or number of bytes. The main access methods for a MemoryObject are read and write. The buildCache method of a MemoryObject returns a MemoryObjectCache. The MemoryObjectCache uses page frame units to read and write physical page frames that cache the contents of the MemoryObject. The MemoryObject converts page frame unit requests from its cache into units that are appropriate for its permanent storage. This allows, for example, the virtual memory system to page from a disk in blocks or a file in bytes.

A Domain maps a set of MemoryObjects or data stores into a virtual address space so that the contents of the data in the stores can be accessed by a virtual address. For example, the MemoryObjects may contain the local variables, the shared variables, the stack, and each file that a Process references, respectively. A Domain associates protection with each MemoryObject and it ensures that the virtual addresses that it uses for each memory object do not overlap. Domains have operations to add and remove MemoryObjects, to lookup or find the MemoryObject at a particular virtual address and to handle a page fault repairfault. Each Domain has an AddressTranslation object which, when activated, controls the hardware memory management unit of the processor. In order to fix a page fault, the Domain sends a cache message to the appropriate MemoryObject to obtain a physical page mapping for the missing virtual memory address. It then sends an addMapping message to the AddressTranslation.

The Disk specifies the behavior of permanent storage in *Choices*. Its subclasses are device drivers. It exports the methods read, write, sizeOfUnits and numberOfUnits. The design of device drivers is described in Section 9 on device management.

The MemoryObjectCache caches all, part, or none of the data of a MemoryObject in physical memory. It keeps track of the physical address of each unit that has been cached. Its main methods are cache, release and protect. The cache method ensures that a particular unit is in the cache and returns the corresponding physical memory address. The release removes a unit from the cache. Each unit is given a protection level when it is cached; protect sets the maximum protection level of a unit and can change the protection of an already cached unit.

The PagedMemoryObjectCache is a concrete class of MemoryObjectCache. It implements cached data using page frame sized physical memory storage units.

The machine dependent code associated with the page mapping hardware is encapsulated in `AddressTranslation`. There is one `AddressTranslation` per `Domain`. On a shared memory multiprocessor, several `AddressTranslations` may be active, one for each processor. `AddressTranslation` has methods `addMapping`, `removeMapping` and `changePermission`. `addMapping` is invoked by the `Domain` after querying `MemoryObjectCache` for a physical address using the `cache` method.

Every `Processor` has an `AddressTranslation` which is responsible for mapping whatever memory management unit (MMU) or translation lookaside buffer (TLB) is provided on the processor to the page table data maintained in an `AddressTranslation`.

The `PhysicallyAddressableUnit` (PAU) is a machine-independent page descriptor associated with a `MemoryObjectCache`. Dirty and referenced bits are maintained in the PAUs by the `MemoryObjectCache` and are used in machine independent paging algorithms. The `PageFrameAllocator` manages physical memory page allocation and has `allocate` and `free` methods. PAUs corresponding to free pages that are not in use by any `MemoryObjectCache` are kept in the `PageFrameAllocator`.

THE CHOICES *FRAMEWORK AND VIRTUAL MEMORY* The virtual memory framework interfaces to many of the other subsystems in *Choices* through the protocols it inherits from the *Choices* framework. The `Domain` provides an interface through which system processes anipulate virtual memory. The `MemoryObject` allows the memory mapping and caching of many different data stores, supports sharing, and allows policies and mechanisms involved in paging to be customized for a specific region of virtual memory. In the next section, we examine process management.

6. *Process Management*

Choices is a multitasking operating system that supports multiple threads of control or processes. Processes provide the *active* computational part of an operating system. We model a process as an object that has methods that may be invoked to change its state. *Choices* supports grouping a number of Processes together into a gang. Gang scheduling permits the processes in a gang to be dispatched on a multiprocessor simultaneously. A variety of other process scheduling policies are supported.

COMPONENTS The process management framework of *Choices* has the following components:

1. The Process
   is a control path through a group of C++ objects. A `SystemProcess` runs in the kernel and is non-preemptable. An `ApplicationProcess` runs in user and kernel space. An `InterruptProcess` is used to handle the occurrence of an interrupt. Each process is associated with exactly one `Domain`, and it executes in that `Domain`. `Processes` may share a `Domain` with other `Processes`. Context switching is light-weight between processes in the same domain and is heavy-weight between processes in different domains.
2. The ProcessorContext
   saves and restores the machine dependent state of a `Process`. Every `Process` has exactly one `ProcessorContext`, and each `ProcessorContext` belongs to exactly one `Process`.
3. The Processor
   encapsulates the processor dependent details of the hardware central processing unit (CPU) including the hardware CPU identification numbers and the state of the hardware interrupt mechanism. It also contains a pointer to its ready queue, a queue of `Processes` that are ready to run.
4. The Gang
   is a group of `Processes` that should be gang scheduled, or run simultaneously, on the processors of a multiprocessor. The `Gang` allows the collection of `Processes` to be manipulated as a single unit.
5. The ProcessContainer
   implements scheduling in *Choices*. For example, processes are run by inserting them into a `ProcessContainer` ready queue. The `Processor` removes a `Process` from its ready queue `ProcessContainer` before dispatching the `Process`. For multilevel feedback queues and other scheduling disciplines, the `ProcessContainer` insertion and removal methods are specialized to provide a given scheduling policy.

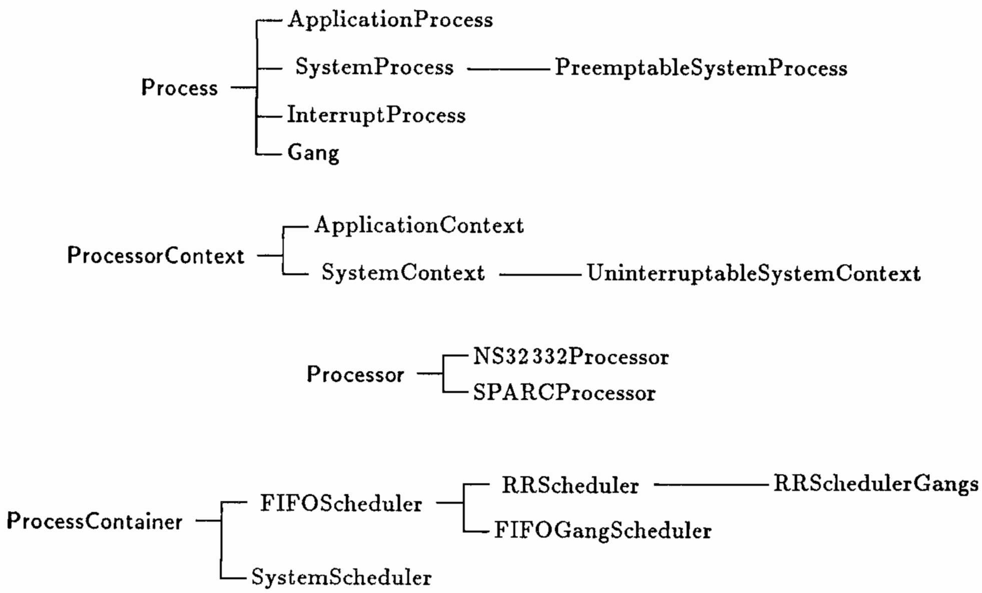

Figure 7: Process System Class Hierarchies

ARCHITECTURAL OVERVIEW The process management system has the following class hierarchy and relationships between its abstract classes. The class hierarchies for the process system are shown in Figure 7. These hierarchies show the abstract classes in bold font. The abstract class interfaces are preserved in the subclasses and superclass code is reused in the subclasses. A `Process` has four concrete subclasses. The `SystemProcess`, `PreemptableSystemProcess` and `InterruptProcess` are kernel processes associated with the kernel `Domain`. `ApplicationProcess` is a user process associated with a user `Domain`. The abstract class `Gang` is also a subclass of `Process`. The `ProcessorContext` class hierarchy mirrors the `Process` class hierarchy and manages the processor-dependencies associated with implementing a thread of control. The `Processor` hierarchy is processor-dependent and has subclasses for each type of central processing unit to which *Choices* has been ported (the class hierarchy is incomplete). The `ProcessContainer` has concrete subclasses for FIFO scheduling, round robin scheduling and two corresponding schedulers that handle `Gangs` as well as regular `Processes`.

Figure 8 is an abstract class relationship diagram for the process subframework. Each `Processor` has exactly one running `Process`

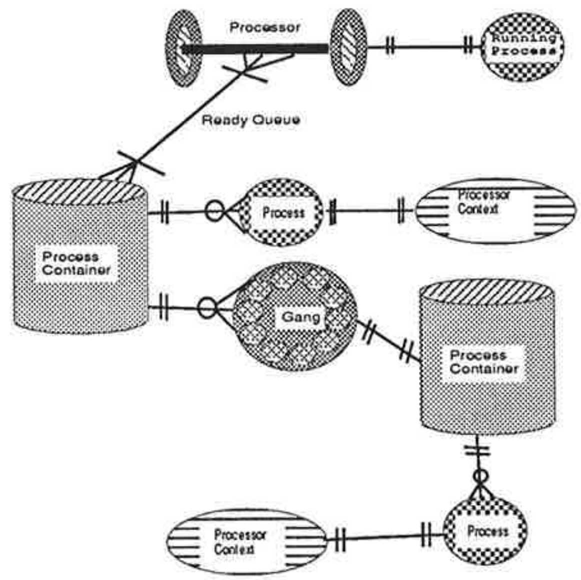

Figure 8: Abstract class relationship diagram for Process Management

and one SystemScheduler ready queue from which it dispatches Processes. The SystemScheduler gives priority to the System-Processes in the ready queue over ApplicationProcesses. When the currently running Process blocks or voluntarily relinquishes its Processor, the Processor retrieves a new Process to run from the ready queue. A particular ProcessContainer may be a ready queue of more than one, but not necessarily all, the Processors in a multi-processor system allowing such systems to have scheduling partitions. A ProcessContainer may contain more than one Process or Gang. Each Process has a ProcessorContext and a ProcessContainer which is the ready queue to which it should be added when it is ready to run. A Gang has a FIFOGangScheduler and a special Process-Container for holding its Process members. When a Gang is first dispatched from the ready queue, the SystemScheduler notes the number of gang members and assigns the processor to dispatch an actual gang Process member from the FIFOGangScheduler. The FIFOGangScheduler is empty and the Processor busy waits for the scheduler to fill. When the required number of Processors have been collected, the SystemScheduler adds the Gang members in the gang

ProcessContainer to the FIFOGangScheduler. The processors then dispatch gang members from the FIFOGangScheduler one by one.

DESIGN OVERVIEW Some of the more important methods of the process system classes are as follows. A Process has methods block to block the process, giveProcessorTo to give the processor to a specified process, relinquish to return the process to the ready queue in order to allow the processor to run other processes, and ready to put the process on its ready queue. The methods block and relinquish are implemented using giveProcessorTo. Its methods save and basicRestore are used for saving and restoring processor independent information during a context switch. These methods are called by the giveProcessorTo, which is the primary method that implements context switching in *Choices*. The methods becomeUninterruptable and becomeInterruptable are not handled directly by a Process but instead are translated to methods on the appropriate ProcessorContext since they are machine dependent operations.

ProcessorContext implements processor dependent methods including checkpoint for saving and restore for restoring processor dependent state, registers, frame pointer, stack pointers and program counter. The checkpoint and restore are called by the giveProcessorTo of Process.

Processor has methods to query and initialize the central processing unit. The chipInitialize method initializes the central processing unit. The installExceptions method installs exception handlers. The flushAddressTranslationCache method flushes virtual addresses from the MMU Translation Lookaside Buffer. The idleContainer method returns the ready queue ProcessContainer.

A ProcessContainer has methods add and remove and isEmpty. The Gang has methods addMember to add processes to it, scheduleMembers to schedule the gang members by adding them to the FIFOGangScheduler, ready to add the gang to the ready queue and returnProcessor to relinquish the gang processes' processors.

THE CHOICES *FRAMEWORK AND PROCESS MANAGEMENT* The implementation of a process is encapsulated completely within the process management framework and provides abstractions for many of the other subframeworks like the message passing system and device driver system. In the next section, we examine the persistent storage framework.

7. *Persistent Storage*

Tapes and disks provide persistent storage of data for a lifetime that is independent of power being supplied to the computer system. The *Choices* persistent storage framework [16] introduces a hierarchy of classes that can be combined to build both standard and customized storage systems. It is flexible enough to support both persistent storage systems and traditional file systems efficiently [23].

COMPONENTS The framework contains the following major components:

1. The PersistentStore
   stores and retrieves blocks of persistent data and has random access methods. A `PersistentStore` is a subclass of `Memory-Object`.
2. The PersistentObject
   encapsulates and provides an operational interface to the data managed by a persistent data store.
3. The PersistentStoreContainer
   divides the contents of a `PersistentStore` into an indexed collection of nested, smaller `PersistentStores` (i.e. a collection of `Files`). The `PersistentStoreContainer` shares storage devices by dividing a `PersistentStore` into smaller ones. Its methods create, make accessible, and delete these nested `PersistentStores`. `PersistentStoreContainers` supports the multiple levels of storage management in the framework and can be nested to an arbitrary depth. The `PersistentStoreContainer` in the lowest layer divides a disk into several partitions. The `PersistentStoreContainer` in the next layer subdivides partitions into logical storage for various types of files.
4. The BlockAllocator
   manages the allocation of the `PersistentStores` within a `PersistentStoreContainer`. In particular, it keeps track of

which data blocks are currently allocated to a `PersistentStore` and which blocks are free. Subclasses encapsulate various mechanisms to manage block allocation, including free-lists or bit-maps.

5. The PersistentStoreDictionary
   maps symbolic names to the indices used by `Persistent-StoreContainers`. The indices may be used to refer unambiguously to the contents of a `PersistentStore`. While `Files` must be contained in exactly one container, they can be named by several dictionaries. Within any dictionary, the keys must be unique, but several keys may map to the same logical name. An example of a `PersistentStoreDictionary` is a System V UNIX directory, which maps fixed-length symbolic keys to indices or logical names called *inumbers*.
6. The PersistentArray, RecordFile, and AutoloadPersistentObject are three different models for structuring the data within files: as arrays of bytes or words (defined by subclasses of `PersistentArray`), as collections of records (defined by subclasses of `PersistentRecordFile`), and as data structures encapsulated by persistent objects (defined by subclasses of `AutoloadPersistentObject`). The first model is suited to the C programming language and the UNIX and MS-DOS operating systems. The file system presents a random-access interface to sequences of bytes and imposes no additional structure. The second model fits programming languages like Cobol, PL/1, and Pascal and operating systems like VMS. The file system presents data as records that can correspond to the types of data structures of the language. The third model fits programming languages like C++ and object-oriented operating systems like *Choices*. The object storage system presents data as objects that are instances of user-defined subclasses of `AutoloadPersistentObject`.

ARCHITECTURE OVERVIEW The persistent storage system has the following class hierarchies and relationships between its abstract classes. The persistent storage framework categorizes most persistent data into two fundamental classes: `PersistentStores` and `Persis-tentObjects` shown in Figure 9 and Figure 10, respectively. A `Per-sistentStore` provides random access to an uninterpreted sequence of blocks of data while a `PersistentObject` interprets the data as

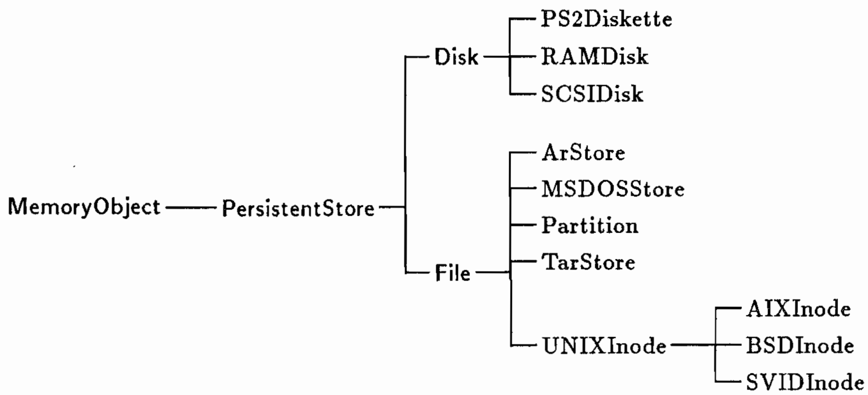

Figure 9: Persistent Store Class Hierarchy

having a format. For example, a UNIX inode is a `PersistentStore`, while a UNIX directory is a `PersistentObject`. A disk is a `PersistentStore`, but a table of descriptors for the files stored on a disk is a `PersistentObject`. All `PersistentStores` have the same interface, much of which is inherited from `MemoryObject`, but the interfaces of different subclasses of `PersistentObject` differ greatly.

The concept of a `PersistentStore` is used both for physical and logical storage devices, allowing reuse of code. The concrete subclasses of `PersistentStore`, shown in Figure 9, belong to one of two categories represented by the following subclasses:

- `Disks` that encapsulate physical storage devices like hard disk drives, floppy disk drives, and RAM disks. `Disks` communicate with objects in the I/O subsystem.
- `Files` that encapsulate logical storage devices like UNIX inodes and disk partitions. Each file has a source `PersistentStore` that supplies it with data from a lower level of the file system. `Files` provide a *window* into their source `PersistentStore`.

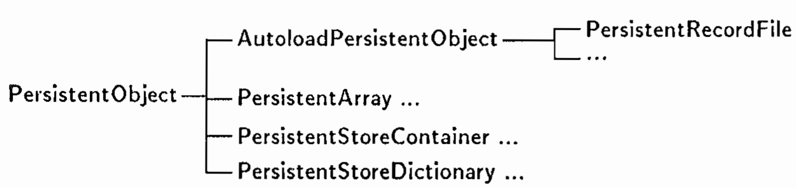

Figure 10: Persistent Object Class Hierarchy

The size of this window can be fixed or variable and can range from zero up to the size of the source `PersistentStore`. The window can be contiguous or divided into discontiguous regions of blocks. Ultimately, the data read from and written to a `File` is also read from and written to a `Disk`.

The `PersistentObject` class defines objects that encapsulate and provide operations on the data managed by a persistent store. Subclasses of `PersistentObject`, shown in Figure 10, abstract the organization, sharing, naming, and data structuring properties of the persistent storage framework.

The persistent storage framework divides a persistent storage system into three layers and is, therefore, an example of a framework that subsumes a traditional layering structure. Figure 11 is an abstract class relationship diagram for the persistent storage system. The top layer contains objects that present application interfaces, the middle layer contains objects that name files and structure the data within

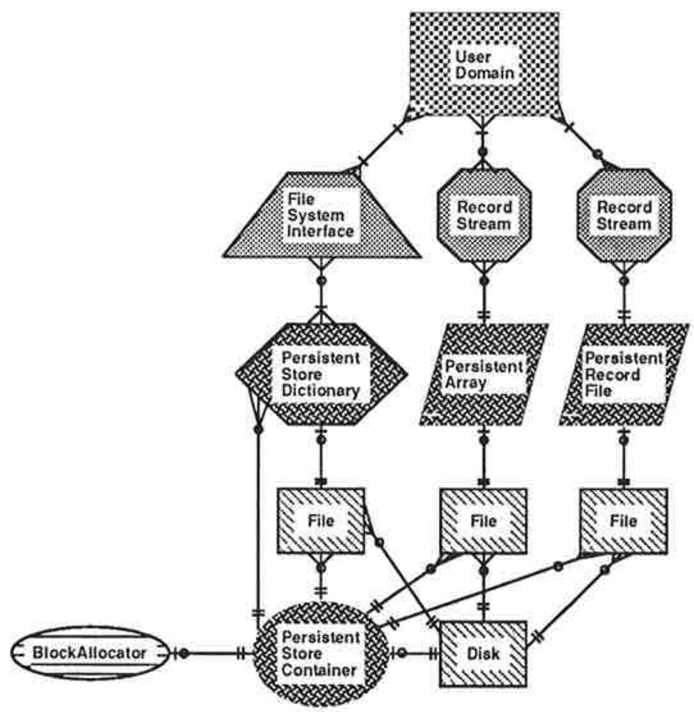

Figure 11: Abstract class relationships for the Persistent Storage System

files, and the bottom layer contains objects that store and organize persistent data. The bottom layer can be further divided into several levels.

A user `Domain` must have one or more `FileSystemInterfaces` to access persistent storage. The `FileSystemInterface` allows the process to open zero or more `PersistentObjects` which include `RecordStreams` and to examine the contents of one or more `PersistentStoreDictionaries`. Multiple processes in the same `Domain` may share the same `FileSystemInterface`.

The middle layer contains `PersistentStoreDictionaries`, `PersistentArrays`, and `PersistentRecordFiles` that structure the data that can be accessed through `Files` and `PersistentStoreContainers`. For example, a `PersistentStoreDictionary` structures the data in a BSD container subclass of `PersistentStoreContainer` to have the BSD UNIX format.

`PersistentArrays` give read/write access to bytes of data and `PersistentRecordFiles` give read/write access to variable or fixed length records. Each `RecordStream` user interface has either a corresponding `PersistentArray` or `PersistentRecordFile`. A `PersistentArray` or `PersistentRecordFile` may be opened and shared by many different `RecordStreams`, some of which may have been opened by processes in different `Domains`.

The lowest layer contains `Files`, `PersistentStoreContainers`, `BlockAllocators`, and `Disks`. Each `File` has a `PersistentStoreContainer` and a `Disk`. `PersistentStoreDictionaries` access a `PersistentStoreContainer` as part of the implementation of opening a storage object and access dictionary information that is stored in a `File`. The classes in this layer have concrete subclasses which, for example, format physical data on disk as a System V UNIX file system. In this case, the `File` would behave like a System V UNIX inode, the `PersistentStoreContainer` would behave like a System V UNIX inode system, and the `Disk` would behave like a disk partition.

`Files` and `Disks` inherit a common interface from `MemoryObject` which allows the lowest layer to be recursively divided into many layers. To subdivide the lowest layer, one emulates a `Disk` using a `File`. The `File` can now be managed like a disk by a `PersistentStoreContainer` and can contain the data for other "higher" level `Files`. One application of this recursion is to divide physical disks into partitions and build file systems in each partition. Another possible application is to build a System V UNIX file system out of a BSD UNIX file.

DESIGN OVERVIEW Some of the more important methods of the file system classes are as follows. `PersistentStores` provide access to raw data, but other interfaces are also needed to satisfy the requirements of the clients of a file system. Some examples include: a container, which treats the data as a collection of files; a dictionary, which treats the data as a collection of file names; and a record file, which treats the data as a collection of records. Subclasses of `PersistentObject` define these and other customized interfaces to a `PersistentStore's` data. The `PersistentObject` class and its abstract subclasses provide methods that control the activation and deactivation of persistent objects, how these objects are mapped into memory, and how they are garbage collected. Each `PersistentStore` has an associated `PersistentObject` class that provides a data abstraction and encapsulation of the persistent data in the store. At run-time, there is a one-to-one correspondence between an instance of a `PersistentStore` and its associated `PersistentObject`.

The `PersistentStore` `asA` method returns a reference to the store's `PersistentObject`. If the `PersistentObject` has not yet been instantiated, the method instantiates the object by mapping the store's data into memory. The `PersistentObject` encapsulates this data as its state data. The `PersistentStore` thus provides the underlying data for its associated `PersistentObject`. `PersistentObjects` and their underlying `PersistentStores` provide the foundation for the *Choices* persistent storage framework. They implement *object-oriented* access to persistent data. The `asA` method takes an argument that may be either a concrete or an abstract `Class`[1] and returns a reference either to an instance of the argument or an instance of a concrete subclass of the argument, respectively. The `asA` method relies on the `supports` method to perform the following steps:

1. determine if the stored data structure is compatible with the requested `Class` or any of its subclasses, and

1 See [17] for a description of first-class classes in *Choices*.

2. if the requested `Class` is compatible with the stored data structure:
   (a) return the requested `Class` if it is concrete,
   (b) otherwise, return the appropriate concrete subclass.
3. if the requested `Class` is incompatible with the stored data structure, return zero.

Several file system clients may access the same persistent data. To provide data consistency for concurrent updates to persistent data through the methods of a persistent object, the `PersistentStore` ensures that there is, at maximum, only one instance of its associated `PersistentObject`. When a `PersistentObject` is no longer needed in primary memory, its finalization code calls the `close` method on its underlying `PersistentStore` to inform the `PersistentStore` that it is also no longer needed in primary memory. A further `asA` request will instantiate a new `PersistentObject` that uses the existing persistent data.

`PersistentStores` also provide methods to report the size of their blocks and records and to report and set their length in both blocks and records. Block and record sizes are given as numbers of bytes. In general, records may span blocks.

The major methods supported by `PersistentStoreContainers` are `create`, `open`, and `close`. The `create` method returns a newly created `File`. The `open` method takes an index as an argument and returns the corresponding `File`. The `close` method informs a `PersistentStoreContainer` that a currently open `File` is no longer being used by any other object in the system.

`Files` whose size can change, e.g. those that represent variable-length files, use the `allocate` and `free` methods of `BlockAllocators` to request and release the blocks of storage. `Allocate` reserves a block of storage and returns its index, and `free` releases a block of storage that is no longer needed.

Naming is orthogonal to storage organization. Using symbolic names, `PersistentStores` can be opened from, created in, added to, and removed from `PersistentStoreDictionaries`. The `open` method takes a key as an argument and returns the named `PersistentStore`. It obtains the `PersistentStore` by invoking the `open` method on its `PersistentStoreContainer` using the id-number that corresponds to the key. Two methods, create and add, allow PersistentStores to be added to dictionaries. The create method performs the same function as open for existing keys; if the key does not exist, however, the operation creates and returns a new PersistentStore. The add method takes a symbolic key and a PersistentStore as arguments. It inserts the key and the id-number of the PersistentStore into the dictionary. The remove method deletes a mapping from a key to an id-number.

APPLICATION INTERFACES The interfaces provided by the naming and data structuring classes are abstract enough to be used directly by application programs; but conventional file systems commonly define an additional layer of abstraction between files and application programs.

A FileSystemInterface object unifies the name-spaces provided by PersistentStoreDictionaries by parsing sequences of symbolic keys, called *pathnames,* and resolving them to PersistentObjects. Each symbolic name is interpreted sequentially by the instance of PersistentStoreDictionary specified by the pathname prefix composed of the previous symbolic names. An example of a FileSystemInterface is the UNIX file system interface, which uses a root directory, a current directory, and a mount table to provide a unified name space for all files within a computer system. A FileSystemInterface that implements the BSD version of UNIX file naming would also use symbolic links.

The public methods of the FileSystemInterface are similar to several UNIX system calls including: open, stat, link, unlink, mkdir, and chdir. These methods manipulate or return references to RecordStreams, PersistentObjects, or PersistentStores.

Subclasses of the RecordStream class provide stream-oriented application interfaces for both PersistentArrays and PersistentRecordFiles. RecordStreams provide the concept of a *current file position,* i.e. the location within the file where the next read or write will occur.

Because RecordStreams introduce the concept of a current file position, they support the setRecordNumber method, which allows programs to reset the current position, and the recordNumber method, which returns the current position. Application programs can read from and write to RecordStreams sequentially. The read and write methods also update the file position. Each instance of `RecordStream` gets data from or sends data to an underlying `PersistentObject`.

THE CHOICES *FRAMEWORK AND THE PERSISTENT STORAGE* FRAMEWORK The persistent storage framework interfaces to all the other *Choices* subsystems through the protocols it inherits from the `MemoryObject`. The `MemoryObject` allows physical and logical storage like disks and files to be used interchangeably, allowing redirection and recursive layering.

8. *Message Passing System*

Many modern operating systems support distributed computing on local area networks of workstations using message passing systems [4, 21] or distributed shared memory [15]. Some operating systems also provide message passing on shared memory machines [1, 21] for parallel programming. This section describes a subframework for message passing designed to support parallel message-based applications. It describes facilities for creating structured messages and sending and receiving messages on a variety of architectures. In *Choices*, messages are sent to `MessageContainers` that are similar to Mach ports [21]. Communication may take place between entities in the same address space, between different address spaces (or different protection domains) on the same machine or between different machines. If the underlying hardware is unreliable, message delivery may be unreliable or some special recovery protocol such as exactly-once may be implemented. The message passing system supports applications on the Encore Multimax shared memory multiprocessor and a network of SPARCstations. The software architectures of the system has been geared towards high performance [11].

COMPONENTS The message passing system has the following components:

1. The MessageContainer
   is a named communication entity for buffering messages. A `MessageContainer` can have multiple senders and multiple

receivers. Once a `MessageContainer` has been created, it is registered with a `NameServer` using an appropriate name. A process intending to send a message to a `MessageContainer` must look the name up in the `NameServer`. On lookup, a sender is given a handle called a `ContainerRepresentative` that forwards messages to the `MessageContainer`.

2. TheKernelMessageSystemInterface and UserMessageSystem-Interface
   support two alternate implementations of the message passing system, the former in the kernel and the latter in user space. In the `UserMessageSystemInterface`, a send or a receive passes message data through user shared memory, not through the kernel. In the `KernelMessageSystemInterface`, a send or a receive passes message data through the kernel and the kernel checks any message parameters. The semantics of the message passing system interface allows messages to be sent and received synchronously or asynchronously.
3. The Transport
   class specifies the mechanism that is used to move a message from a sender to a receiver. A local message may be transported by a separate process or copied by the sender and receiver processes. A remote message is transmitted across the network.
4. The Synchronization
   between processes may be through busy-waiting or blocking.
5. The DataTransfer
   class concerns the buffering strategies used in sending a message. On a shared memory multiprocessor, message sends may be double buffered, single buffered, or passed by reference. In a distributed system, messages are buffered for the message transport mechanism.
6. The Reliability
   class allows messages to be sent unreliably, with at-most-once semantics, and exactly-once [8] semantics.
7. The FlowControl
   class uses rate based flow control to ensure that the sender and the receiver are not overrunning one another's data buffers.

ARCHITECTURAL OVERVIEW The message passing system class hierarchies and the relationships between its abstract classes are as fol-

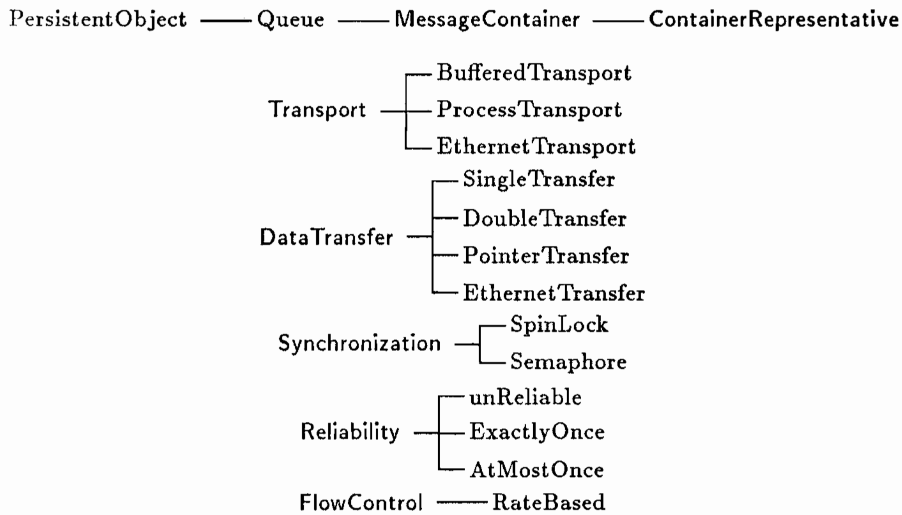

Figure 12: Message Passing System Class Hierarchies

lows. Figure 12 shows the class hierarchies for the message passing system. These hierarchies include the abstract classes and the concrete classes that are subclassed from them. The abstract class interface is inherited by the subclasses and the superclass code is reused in the subclasses.

The abstract classes in the system are `MessageContainer`, `KernelMessageSystemInterface`, `UserMessageSystemInterface`, `Transport`, `Synchronization`, `Data Transfer`, `Reliability`, and `FlowControl`. Concrete subclasses implement the various design options.

There are three concrete subclasses of `Transport`. `ProcessTransport` uses a separate process to transport the message. `BufferedTransport` requires the receiver to fetch the message. `EthernetTransport` delivers the message across an ethernet.

`Synchronization` has two concrete classes. A `Semaphore` blocks the process. A `Spinlock` busy waits on a shared variable.

The class `DataTransfer` has four concrete subclasses. In the concrete class `DoubleCopy` the message is copied into a temporary buffer by the sender and then copied from the temporary buffer into a receiver buffer by the receiver. In the concrete class `SingleCopy` the message is copied once from the sender buffer to a receiver buffer in a shared memory region. The receiver incurs the cost of the copy. In the concrete class `PointerTransfer` buffer pointers are exchanged between the sender and the receiver but data is not physically copied. The concrete class `EthernetTransfer` manages ethernet driver buffer regions, and copies the data from network to user buffers and vice versa. The three concrete subclasses of `Reliability` implement the functionality implied by their names. The concrete subclass of `FlowControl` implements the functionality rate based flow control. It has the concrete class `RateBased`.

Given the above framework it is possible to create a specific message passing system. Instances of these concrete classes appear in an implementation of a framework. For example, a collection of instances of the classes, `KernelMessageSystemInterface`, `BufferedTransport`, `SpinLock`, and `DoubleCopy`, defines a message passing system that is kernel based, lets the receiver process incur most of the cost of message transfer, provides synchronization through spin-locks and copies the message into the kernel domain and then into the user address space (double copy semantics).

Figure 13 is an abstract class relationship diagram for the message passing system. Several `ApplicationProcesses` send messages through the `MessageSystemInterface` but there is only one `MessageSystemInterface` for all the `ApplicationProcesses`. The `MessageSystemInterface` knows about multiple `ContainerRepresentative` and multiple `MessageContainers`. A send uses the `ContainerRepresentative` to deliver the message. On a receive the `MessageContainer` is used to buffer messages. For shared memory an optimization transfers control between the `ContainerRepresentative` to the `MessageContainer` with very little overhead. When the `ContainerRepresentative` and `MessageContainer` are on different machines the overhead of ethernet packet setup and transmission is incurred. There may be several `Transport` options for the `ContainerRepresentative` and `MessageContainer`. The `Transport` class has several `Reliability` mechanisms to choose from, but this functionality is optional. There may be a variety of optional `FlowControl` mechanisms to choose from and a variety of `DataTransfer` mechanisms to use.

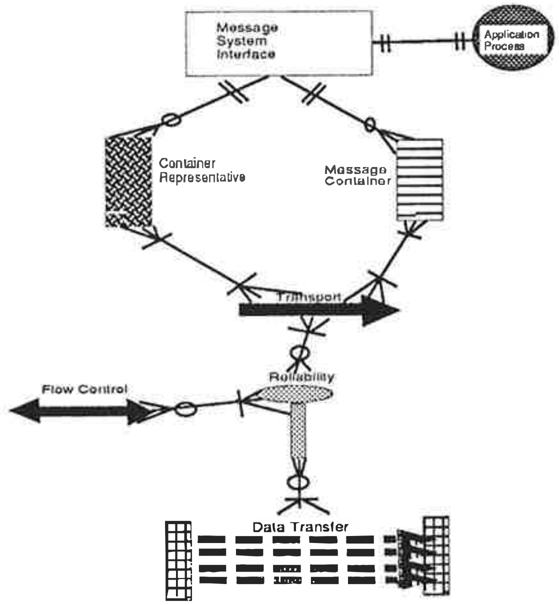

Figure 13: Abstract class relationship diagram for the Message Passing System

DESIGN OVERVIEW The important methods of the message system classes are described in this section. A `MessageContainer` has a queue associated with it. Concurrent access is permitted on this queue as message senders and message receivers deposit and remove messages from the `MessageContainer`. The class `MessageContainer` exports three operations: `get`, `put` and `isEmpty`. `put` adds a message to the container, `get` retrieves a message from the container and `isEmpty` checks to see whether the `MessageContainer` is empty. There is one receiver thread per `MessageContainer`. Multiple senders may send messages to the `MessageContainer`. Once a `MessageContainer` has been created it can be registered with the `NameServer`.

A process intending to send a message to a `MessageContainer` must look up its name in the NameServer using a appropriate name. On lookup, a sender is given a handle called a `ContainerRepresentative`. A `ContainerRepresentative` has the information to locate and deliver messages to the `MessageContainer` for which it is a representative. A representative needs to differentiate between three cases:

- when the sender and receiver share an address space.
- when the sender and receiver are on the same machine but do not share an address space.
- when the sender and receiver are on different machines.

The `ContainerRepresentive` has a `send` method that invokes the appropriate method on a `Transport` object (see below).

Two styles of communication are supported in the `KernelMessageSystemInterface` and `UserMessageInterface`: asynchronous and synchronous. In the *Asynchronous* style, when a message is sent, the process does not wait for the message to be delivered to the buffer of the receiving process. A copy of the message is made and the kernel returns from the call immediately. When a process attempts to receive a message of a particular type, if the message is not in its container, the receive returns immediately with an *identifier* that the receiver process can later use to get the message. If the message is in the container the receiver receives the message immediately. In the *Synchronous* style when a message is sent, the process blocks until the system can send the message to the receiving process. When the receiving process receives a copy of the message the sender is unblocked. When a process does a blocking receive, it waits for the sender to send the message. When the message system is implemented in user space a send or a receive does not involve passing data through the kernel but is implemented using user shared memory. When the message system is implemented in the kernel, message parameters are checked by the kernel.

The `Synchronization` class defines the two operations `acquireResource` and `releaseResource`. `acquireResource` is used to gain access to the shared or critical region and the operation `releaseResource` is used to exit from or give up the shared resource.

It is possible to send messages unreliably, with and without notification of failure using the class `Reliability`. It is possible to set the timeout period for retransmissions. The abstract class defines the operations `deliver`, `deliveryWithNotification`, and `setTimeout`. In the concrete class `unReliable`, all delivery is unreliable. The class `FlowControl` receives information about packet loss from `Reliability` and changes the interpacket gap if necessary. It exports the method `regulate` to regulate the flow control.

THE CHOICES *FRAMEWORK AND MESSAGE PASSING SYSTEM* Various parts of the *Choices* framework use message passing. The `KernelMessageSystemInterface` and `UserMessageSystemInterface` provide interfaces for parallel and distributed computing for `ApplicationProcesses`. The `DistributedNameServer` uses it for maintaining a consistent view of the name space.

9. Device Management

The I/O architecture of *Choices* allows a `Process` to communicate with peripheral devices. Several different I/O devices were examined before a set of abstract classes were designed to provide a uniform interface for device management. Although the *Choices* device drivers were influenced by those of UNIX, there are some notable differences. First, *Choices* device drivers have an object-oriented design and this decomposition leads to less complex device drivers for asynchronous I/O. Second, the device management subframework does not use the file system for naming nor does it use the file system interface.

COMPONENTS The device management subframework of *Choices* has the following major components:

1. The Device
   acts as a server for components of other *Choices* subframeworks. For instance, the concrete class `DiskDevice` acts as a server of the classes of the file system framework. In turn, most of the `Devices` act as clients of `Devices-Controller` objects. For instance, two `DiskDevices` representing disks attached to the same hardware controller act as clients of the same `DiskController`.
2. The DevicesControllers
   represent hardware I/O controllers. A `DevicesController` acts as a server for possibly several `Devices`. A `DevicesController` is not visible to the user of a device. I/O operations should only be requested from a `Device`.

3. The DevicesManager
   supports the addition and removal of devices and controllers. Each system has only one object of this class. When a `DevicesController` is loaded into the system it registers itself with the `DevicesManager` object. Hardware controllers and devices that are added to the system are also registered with the `DevicesManager`. The `DevicesManager` is informed of the addition and removal of hardware components by the system administrator or by the cooperation of hardware and machine-dependent software. The `DevicesManager` matches physical controllers with registered `DevicesControllers`. For each physical controller a "matching" `DevicesController` is instantiated. In addition, for each physical device a `Device` is constructed and returned when the method `attachDevice` is invoked on the `DevicesController`. The new `Device` is then bound to the `NameServer`.

ARCHITECTURAL OVERVIEW The device system class hierarchy and the relationships between its abstract classes are described in this section. The class hierarchies of the device management system are shown in Figure 14. These hierarchies show the abstract and concrete classes of the device management subframework.The abstract class interfaces are preserved in the subclasses, and superclass code is reused in the subclasses. The class hierarchies show subclasses of `Devices` and `DevicesController` for handling disks and character devices such as keyboards and serial lines.

Figure 15 is an abstract class relationship diagram for the device management framework. `DeviceControllers` register themselves on creation with a `DevicesManager` and are associated with one `DevicesManager`. There is a `DevicesController` for each type of device. A `DevicesController` controls a group of `Devices` of the

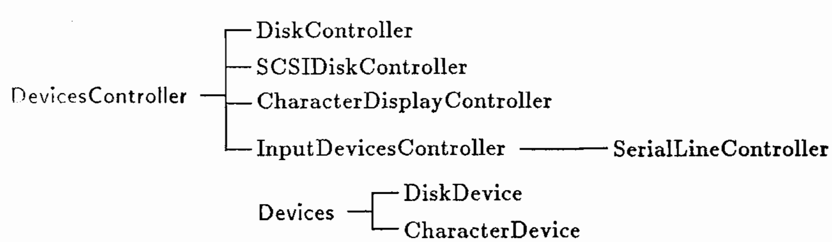

Figure 14: Device Management Class Hierarchies

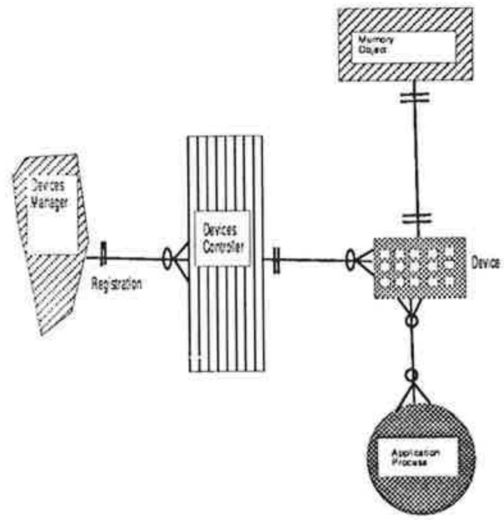

Figure 15: Abstract Class Relationship diagram for the Device Management framework.

same type. A class of `Devices` may be used with more than one class of `DevicesControllers`, and are thus highly reusable. User `Processes` interact with `Devices` which translate the user commands to controller specific commands. As a result, `Devices` may need to perform buffering between the user `Process` and a particular `DevicesController`.

DESIGN OVERVIEW The more important methods of the device system classes are described in this section. The protocol of a `Device` depends on the physical device it represents. For instance, `DiskDevices` have read and write methods that transfer a number of disk blocks. On the other hand, a `CharacterDevice` has additional methods to transfer strings of characters, such as `stringArrived`, and control methods such as `setBaudRate` to set control parameters. When a newly created concrete subclass of `Device` is created it is bound to a `NameServer`. `Processes` may access this device after looking it up in the `NameServer`. An appropriate interface to `Device` may be chosen by invoking the `asA` on `Device`. This is often necessary since the default interface to `Devices` may be too low level. Because `Devices` act as intermediaries between higher level I/O objects and the `DevicesControllers`, they need to buffer input and output requests to peripheral devices.

The interface between a Device and a DevicesController is based Command objects. User requests for reads and writes on Devices cause the construction of one or more Commands which are then sent to a DevicesController object using the sendCommand method. There is an extensive class hierarchy of Commands. Examples of Commands include, FlushOutputCommand, to flush output to a character device, and setParityCommand to set the parity on a serial line. The interface between the DevicesController and the Device uses pointers to pass Commands for two reasons. The first is that a Device can be reused with different DevicesControllers. For example, a DiskDevice can be used as the Device of a machine-dependent DiskController and a machine-independent SCSIDiskController. The second advantage of this interface is that it does not force a DevicesController to have a specific interface that depends on its devices. The protocol of a DevicesController subclass can change without requiring a change to existing Devices. On the other hand, this interface cannot be checked at compile-time to ensure consistency. As this interface is internal to the device management framework, this does not appear to be a problem. A rich set of command classes makes this interface extremely flexible [13].

THE CHOICES *FRAMEWORK AND* DEVICE *MANAGEMENT*

Other frameworks use the device management framework with the help of classes that provide communication between the two frameworks. For instance, a Disk is a PersistentStore that does Input/Output using a DiskDevice. Devices are converted to objects in other hierarchies using the *Choices* conversion mechanism. Conversion is a term introduced in Smalltalk [7] and used for the collection classes. We generalize the conversion mechanism to apply to any class. We also combined the conversion mechanism with double dispatching [9] so that new inter-subframework classes can be added to the system without changing existing classes inside the frameworks.

10. Advantages of Frameworks

In this section, we describe some of the major advantages of using frameworks for designing an operating system. We demonstrate the advantages with examples from the *Choices* operating system.

- *Code Reuse* is normally achieved through the reuse of existing components and through polymorphism. With frameworks, code can also be reused through inheritance. The use of virtual functions in C++, for example, allows large bodies of code to be reused. In the persistent storage subframework, several abstract classes, including `PersistentStoreContainer` and `PersistentStoreDictionary`, implement all public operations. These operations are defined using several simple operations that subclasses must implement.

- *Design Reuse* is achieved in frameworks by reusing abstract concepts from one subframework in another framework. For example, `MemoryObjects` may be used in the persistent store subframework as well as in the virtual memory subframework. Frameworks allow this commonality to be described and reused.

- *Portability* is achieved in frameworks by separating machine-dependent parts of design from the machine-independent parts. For example, an abstract class may have implementations of the machine-independent parts of a component, but machine-dependent parts will be specified by pure virtual functions that must be supplied by a subclass. For example, there is a CPU class that is machine-independent but it has concrete subclasses that are tailored to the SPARC, i386, NS32332, and MC68030 processors.

- *Rapid Prototyping* of different concepts is possible in frameworks because it supports code and design reuse. Code reuse and design reuse reduce coding time and design time, respectively. Once an abstract class has been built, it is only necessary to supply implementations of its pure virtual functions in a concrete class. For example, we were able to compare and contrast several message delivery mechanisms in the *Choices* message passing subframework.

- It is possible to customize for *performance*. For example, in the message passing subframework we allow synchronization through semaphores and spin-locks. For hypercube applications, the spin-lock version is a faster synchronization mechanism.

11. Evolution of the Subframeworks

The subframeworks described above have been reached though iterative improvement of their designs. A concept was tried and when it did not work or it proved inflexible it was modified. Often subframeworks need to be modified to encorporate a new concept. This may require substantial changes to the original framework. Often the changes lead to better insights into the original concepts as well. We have encountered instances of both types of changes to our frameworks and in this section we discuss evolutionary changes to the message passing system and persistent storage frameworks.

One of the most important aspects of the design is *reuse*. For example, it is possible to combine a `SpinLock` class with either a `SingleTransfer` or `DoubleTransfer`. An earlier design merged the transfer and the synchronization hierarchies. This was clearly a mistake as the synchronization and transfer mechanisms are separate concepts. The old design forced the transfer mechanism to be replicated for both the semaphore and spin-lock modes of synchronization. Keeping these separate allows one instance of the transfer mechanism to be used with several synchronization mechanisms.

During the evolution of the persistent storage subframework, there have been three versions [19, 18, 16]. The first version supported the design and construction of UNIX-like file systems, and its major abstraction was the `MemoryObject`. Inheritance was used to both model *is-a* and *has-a* relationships. For example, subclasses of `PersistentObjects` inherited from subclasses of `PersistentStores`. This overuse of inheritance made the framework inflexible.

The second version restricted the use of inheritance to model *is-a* relationships. Besides the `MemoryObject`, the version introduced the `MemoryObjectContainer` and `MemoryObjectDictionary` abstractions. We found that the framework could then model many types of stream-oriented file systems, including both UNIX and MS-DOS file systems. Despite the improvements in the second version, it was still incapable of modeling object-oriented or record-oriented file systems, since it lacked a well-defined concept of a `PersistentObject`.

The current version of the framework supports the design and construction of stream-oriented, object-oriented, and record-oriented file systems. It was motivated by an effort to model a persistent object store and to be a dynamically extensible system. The features of the `MemoryObject` class that related to persistent data were split into a subclass of `MemoryObject`, class `PersistentStore`. This enabled the *Choices* file system framework to be further refined without requiring changes to the *Choices* virtual memory framework, which also relied on the `MemoryObject` class as a key abstraction. The addition of the `PersistentStore` class to the framework allowed the constraint that the data managed by each `PersistentStore` must also be encapsulated by a `PersistentObject`. This version refined the abstractions used in the second version and added abstractions for `PersistentArrays`, `PersistentRecordFiles`, and `AutoloadPersistentObjects`.

12. Conclusions

Our experience has shown that an object-oriented framework is an effective technique for designing a complex software system such as an operating system. In this paper, we have shown how complicated components of the operating system can be designed and the interfaces between the different components defined using frameworks. We also show how a framework for a system can be used to help design the subframeworks required for subsystems of the system. Parts of the framework are refined and specialized for the subframework. There are, however, critical parts of a framework that have only informal definition. In particular, we found that a suitable notation for expressing many of the informal constraints between components of a system is lacking. The relationships that can be expressed by the classes in C++ were insufficient to express all the constraints that accompanied the design of the *Choices* frameworks. There has been little work in formally specifying constraints. A notable exception is the work on contracts [10]. In a more recent paper we have discussed techniques for more formally and concisely describing frameworks [2]. We intend to use these concise techniques for further describing all the subframeworks described in this paper.

References

[1] Forest Baskett, J. H. Howard, and John T. Montague. Task Communication in DEMOS. *ACM Operating Systems Review,* pages 23–31, November 1977.

[2] Roy H. Campbell and Nayeem Islam. A Technique for Documenting the Framework of an Object-Oriented System. In *Second International Workshop on Object-Orientation in Operating Systems,* Paris, France, October 1992. IEEE Computer Society Press.

[3] Roy H. Campbell, Vincent Russo, and Gary Johnston. Choices: The Design of a Multiprocessor Operating System. In *Proceedings of the USENIX C++ Workshop,* pages 109–123, Santa Fe, New Mexico, November 1987.

[4] David Cheriton. The V Distributed System. *Communications of the ACM,* pages 314–334, 1988.

[5] Peter Deutsch. *Levels of Reuse in the Smalltalk-80 Programming System.* IEEE Computer Society Press, Cambridge, Massachusetts, 1987.

[6] Peter Deutsch. *Design Reuse and Frameworks in the Smalltalk-80 Programming System.* ACM Press, Cambridge, Massachusetts, 1989.

[7] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, Massachusetts, 1983.

[8] A. Goscinski. *Distributed Operating Systems: The Logical Design.* Addison-Wesley, Sydney, Australia, 1991.

[9] Kurt J. Hebel and Ralph E. Johnson. Arithmetic and Double Dispatching in Smalltalk-80. *Journal of Object Oriented Programming,* March/April 1990.

[10] Richard Helm, Ian Holland, and Dipayan Gangopadhay. Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. In *ECOOP/OOPSLA '90,* pages 169–180. ACM, 1990.

[11] Nayeem Islam and Roy H. Campbell. "Design Considerations for Shared Memory Multiprocessor Message Systems". In *IEEE Transactions on Parallel and Distributed Systems,* October 1992.

[12] Ralph E. Johnson and Vincent F. Russo. Reusing Object-Oriented Design. Technical Report UIUCDCS-R-91-1696, University of Illinois, May 1991.

[13] Panagiotis Kougiouris. A Device Management Framework for an Object-Oriented Operating System. Master's thesis, University of Illinois at Urbana-Champaign, August 1991.

[14] Glen E. Krasner and Stephen T. Pope. A Cookbook for Using the Model-View-Controller Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming,* pages 26–49, 1988.

[15] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. In *Proceedings of the ACM Symposium on Principles of Distributed Computing,* pages 229–239, 1986.

[16] Peter W. Madany. *An Object-Oriented Framework for File Systems.* PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, June 1992.

[17] Peter W. Madany, Roy H. Campbell and Panagiotios Kougiouris. Experiences Building an Object-Oriented System in C++. In *Technology of Object-Oriented Languages and Systems Conference,* Paris, France, March 1991.

[18] Peter W. Madany, Roy H. Campbell, Vincent F. Russo, and Douglas E. Leyens. A Class Hierarchy for Building Stream-Oriented File Systems. In Stephen Cook, editor, *Proceedings of the 1989 European Conference on Object-Oriented Programming,* pages 311–328, Nottingham, UK, July 1989. Cambridge University Press.

[19] Peter W. Madany, Douglas E. Leyens, Vincent F. Russo, and Roy H. Campbell. A C++ Class Hierarchy for Building UNIX-Like File Systems. In *Proceedings of the USENIX C++ Conference,* pages 65–79, Denver, Colorado, October 1988.

[20] Roger Pressman. *Software Engineering.* McGraw-Hill, New York, New York, 1987.

[21] Richard Rashid. Threads of a New System. *Unix Review,* 1986.

[22] Vincent F. Russo. *An Object-Oriented Operating System.* PhD thesis, University of Illinois at Urbana-Champaign, October 1990.

[23] Vincent F. Russo, Peter W. Madany, and Roy H. Campbell. C++ and Operating Systems Performance: A Case Study. In *Proceedings of the USENIX C++ Conference,* pages 103–114, San Francisco, California, April 1990.

[24] John M. Vlissides and Mark Linton. Unidraw: A Framework for Building Domain-Specific Graphical Editors. In *User Interface Software Technologies,* pages 81–94. ACM SIGGRAPH/SIGCHI, 1989.

[received June 23, 1992; accepted July 2, 1992]