Technical Report

UCAM-CL-TR-86
ISSN 1476-2986

Number 86

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

The Entity System: an object based filing system

Stephen Christopher Crawley

April 1986

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted December 1985 by the author for the degree of Doctor of Philosophy to the University of Cambridge, St John's College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Table Of Contents.

1 Introduction.

1.1 Background. 1

1.2 How the Entity System Happened. 2

1.3 Requirements for an Entity System. 3

1.4 The Structure of the Thesis. 4

2 Persistent Storage and Storage Interfaces.

2.1 Abstraction, Data Storage and Data Models. 6

2.2 Concrete Data Models. 7

2.2.1 The Unit of Data. 7

2.2.2 Addresses and Keys. 8

2.2.3 Files and Explicit File I/O. 9

2.2.4 Memory Mapped Storage Interfaces. 10

2.2.5 General Points. 11

2.3 Storage Interfaces in Programming Languages. 12

2.3.1 Conventional Interfaces. 12

2.3.2 Checkpointing as a Storage Interface. 13

2.3.3 Data Flattening. 14

2.3.4 Persistent Storage. 16

2.4 Databases and Database Management Systems. 17

2.4.1 Database Data Models. 18

2.4.2 Data Definition in DBMS's. 19

2.4.3 Data Manipulation Facilities in DBMS's 21

2.4.4 Data Evolution in DBMS's 22

3 The Basic Entity System

3.1 What is an Entity? 24

3.2 Entity Operations. 30

3.3 Some Examples of Entities. 34

3.4 Comparison with Other Languages and Systems. 42

4 Projections and Perspectives.

4.1 Projections and Perspectives. 46

4.2 Mechanisms for Projection. 48

4.3 Projection Ambiguity. 56

5 An Experimental Entity System.

5.1 The Scope of the Experimental System. 63

5.2 A Standard Storage Interface. 65

5.3 The Basic Entity System. 70

5.4 The Projection Mechanism. 75

5.5 A Model Filing System. 78

5.6 Lessons from the Experimental Entity System. 80

6 Implementation Issues.

6.1 Language Independent Class Definition. 84

6.2 Class Management. 92

6.3 Implementation and Resolution Function Management. 93

6.4 Host Machine and Operating System Problems. 94

6.5 Improving Entity System Efficiency. 97

7 Conclusions. 101

References. 105

Appendix A. Dynamic Modules in Modula-2 112

Appendix B. Glossary. 115

Chapter 1 - Introduction.

1.1 Background.

Developments in programming languages have provided increasingly powerful facilities for algorithmic and data abstraction. Data abstraction in the form of record declarations emerged with COBOL, and algorithmic abstraction started with FORTRAN. Algol 60 marked the start of formal data types and type checking. Programming languages such as Pascal and Algol 68 developed these concepts, but still used types merely as a way of defining data representations in an abstract way. Languages like Simula 67 and later CLU and Smalltalk had object based type systems, allowing the user to define typed objects in terms of the operations that they support, combining both representation and algorithm into a single object abstraction.

Until recently, little work has been done on extending data typing concepts beyond the bounds of a single program. I/O in strongly typed languages is typically performed by reading and writing data as an untyped stream of characters; LISP, Prolog and similar languages dump and restore the program's entire workspace for long term storage of complex data structures.

By contrast, database systems address the problems of many programs using the same data, and of coping with changing data descriptions and access requirements. Databases have traditionally taken a table and file based approach to storing complex data.

Attention has recently focused on extending data typing beyond the bounds of an executing program. The DTL language [Hughes 83] which draws from the Michael Jackson design methodology, models a program as a data transformer which converts one typed data stream into another. PS-Algol [Atkinson 84] which arose out of interest in storing graphics data structures, extends a representational type system by allowing data in the heap to persist from one run of a program to the next. The Eden system [Lazowska 81] [Almes 83] addresses the issue of object based programming with persistent objects in a distributed environment. None of these examples really addressed the issues of evolving programs and data requirements.

1.2 Origins of the Entity System.

In 1979/80 Jim Mitchell, who was at the time a researcher at Xerox PARC, spent 6 months on sabbatical leave at the Cambridge University Computer Laboratory. In late '79 he organized a series of informal seminars on topics related to programming languages. One outcome was a general feeling of dissatisfaction with the programming support environment in use at the time. As a consequence the Programming Environment Research Group (PERG) was formed to try to improve things. At the time, most programming in the Laboratory was in BCPL for the RSX and TRIPOS operating systems. Support for program development was elementary.

It was soon realized that in order to build the desired sort of programming environment, the file system needed to provide considerably more functionality than was available from either RSX or TRIPOS. The file system needed to tie together the components of a modular program such as the source and object code, compilation commands, time stamps. Both small and large components needed to be stored efficiently. Support was needed for incremental improvements to the functions of the file system with a minimum of disruption. Finally, a mechanism was needed for ensuring that files were treated consistently, so that for example the user couldn't accidentally send object code to a line printer. The HADES database system [Wilkes 82], was a starting point but something considerably more general was needed.

From the start, PERG used the term *entity* to describe an object held in the file system, since it was "more than a file". The most popular design was that of an attributed file system [Mitchell 81] in which entities were modeled as a collection of strongly typed attributes with abstract interfaces. An alternative system [Singer 81a] [Singer 81b] modeled each object as a <class, implementation, representation> triple. Various hybrid schemes were proposed using both classes and attributes, and there was a lot of inconclusive argument over potential methods for representing attributes. Finally, I came to implement a trial entity file system under TRIPOS. From this experience I concluded that attributes were not appropriate and that what was needed was a lower level standard storage interface with support for data structuring [Crawley 82].

By this time I was working on my own; the other members of the PERG group either having left the Laboratory, or being too involved with their own theses to contribute. My next activity was the implementation of the "experimental" entity system described in Chapter 5. This was written in Modula-2 for UNIX, and incorporated various levels of support for projection[1] and two versions of a standard storage interface. Development continued on the experimental entity system until early 1984. I was dissatisfied with the storage interface, but was unable to resolve the problems until it was suggested by a colleague [Pardoe 84] that storage objects could be treated as entities in their own rights.

Over a period of 3 or 4 months in 1985, a new entity system kernel was written in Mesa to run under the Xerox Development Environment on a Dandelion workstation. The treatment of storage objects as entities has made a considerable impact on the new system. It has also benefited from being reimplemented in a more suitable language (Mesa) and programming support environment (Pilot / XDE). Unlike previous versions, the XDE entity system is intended to develop into a "production" system that could be used for programming environment research.

1.3 Requirements for an Entity System.

The aim of an entity system is to provide abstract data typing at the filing system level, in such a way as to support the development and evolution of large scale software systems. In summary, the requirements are as follows.

1) The entity system should provide a strictly type checked interface to the data it stores.

2) The entity system's type system should be extensible. The programmer should be able to design and implement new abstract data types at any time.

3) The mechanisms for representing objects should allow object representations and associated algorithms to evolve without undue inconvenience.

1 Projection is a form of dynamic coercion of operations on entities. See Chapter 4.

4) The data stored in the entity system should be relatively secure against corruption due to bugs in client programs and in service routines. When something does go wrong with an entity, the spread of corruption to other entities should be contained.

5) The entity system should be capable of supporting client programs in a number of programming languages.

6) The entity system should be a suitable basis for building such things as
   - user level filing systems,
   - integrated software development environments, and
   - large scale applications systems.

7) It should be possible to implement the entity system efficiently on computers with conventional architectures, and in decentralized and distributed computer systems.

1.4 The Structure of the Thesis.

The rest of the thesis is organised along the following lines.

Chapter 2 is a survey of the spectrum of persistent storage interfaces and techniques as used by application programs. The first section discusses abstraction in data storage and data models. The remaining sections survey storage interfaces provided by conventional file systems, programming languages and database systems. Special attention is paid to support for abstraction and for incremental change.

Chapter 3 describes the basic entity system. The first two sections define an entity and associated concepts, and outline the generic operations. Section 3 presents some example entities, and illustrates the need for entity polymorphism. Section 4 compares entities with object abstraction in other systems.

Chapter 4 describes the part of the entity system which provides polymorphism. The first section defines the concept of projection of operation. Section 2 describes and illustrates a number of techniques for resolving projections. The last

section discusses the problem of projection ambiguity, and describes some methods for avoiding it.

Chapter 5 is an outline of the "experimental entity system" implemented in 1983/84. The first section discusses the constraints within which the system was implemented. Sections 2 through 5 respectively describe the low level storage interface, the basic entity system kernel, support for projection, and a simple environment built to test the system. The final section outlines a number of lessons that were learnt from implementing the experimental entity system.

Chapter 6 discusses a number of implementation issues which were not explored in the experimental entity system. Section 1 discusses language independent class definition and the problems of passing operation arguments from one language to another. Sections 2 and 3 discuss naming and management of entity classes and implementations. Section 4 deals with problems in implementing an entity system in a range of host architectures. Section 5 presents some methods for improving entity system performance.

Chapter 7 offers some conclusions based on the rest of the thesis. Then it provides a sketch of various ideas for programming environments which the author and colleagues are currently exploring using the XDE entity system.

Appendix A is a description of the dynamic modules package used in the experimental entity system.

Appendix B is a glossary of entity system terminology which may prove useful in case of confusion in the body of the thesis.

Chapter 2 - Persistent Storage and Storage Interfaces.

The entity filing system draws ideas from a large variety of persistent storage systems including conventional file systems, database systems and programming languages with support for persistent data. This chapter surveys the spectrum of storage interfaces from the viewpoint of the interface used by the client. In this context the client is typically an application program, though it could equally be a higher level storage management layer.

2.1 Abstraction, Data Storage and Data Models.

Data storage can be viewed by the client in either of two ways which are reflected in the design of a storage interface.

The *concrete view* is that the storage system is a mechanism for storing bits, bytes or words of state. It may also provide structuring primitives (such as records), and support a variety of methods of accessing and updating the state. The point is that the interface treats the data in a totally general way with no regard for meaning of the bit patterns. It is the client's responsibility to interpret the bit patterns as representing higher level information. Thus in this view, the storage medium is exposed to the user.

The *abstract view* is that the storage system is a mechanism for storing abstract information pertaining to a model of reality used by the client. The client does not need to know about the way that this information is represented, or how the representation is held by the storage system.

As in programming languages, data abstraction in the storage interface is a matter of degree. Some abstract interfaces limit themselves to information that can be held in flat structures such as records and arrays, while others support trees and networks. Although most abstract interfaces only support representational abstraction, it is also possible to support the level of data abstraction found in object based programming languages.

A storage interface allows the client to define a *data model* for the stored data within the constraints of the interface. With a concrete interface the data model deals with storing representations. With an abstract storage interface, the stored data model corresponds to a model of reality [Kent 78]. In database systems in particular, the data modeling facilities are founded on one of a number of theoretical database models [Date 83] [Kerschberg 76].

It is important that the data model supported by a storage interface is compatible with the nature and structure of the client's data and the pattern of use. It is also desirable for the storage interface data model to be compatible with the client's programming language, though this is not normally the case. Support for data abstraction in storage interfaces lags well behind the facilities available in most programming languages.

The need for abstraction must be balanced against the cost of implementing it. While this is important in programming languages, it is much more so in storage interface design. Persistent storage systems are generally based on storage devices with access times 4 or 5 orders of magnitude slower than main memory. The concrete approach to data storage recognizes this in providing an interface that is motivated by the need to access permanent storage efficiently. The key to an effective abstract data storage is masking the differences between primary and secondary storage from the client, while still managing to handle secondary storage efficiently.

2.2 Concrete Data Models.

The storage interfaces provided by most operating systems and some language's runtime libraries are based on the view of the storage system as a mechanism for preserving concrete representations of information.

2.2.1 The Unit of Data.

A significant property of a concrete data model is its *unit of data*: the smallest unit of storage that a client can address. Typically this is a *byte*: a unit capable of representing one character of text. Other interfaces use a *word* as the unit,

packing a number of bytes into each word. This has implementation advantages on machines with a word addressed architecture, but requires the client to deal with the packing and unpacking of characters from words. Apart from that, byte and word oriented interfaces are much the same. Examples of byte and word oriented file system interfaces include the UNIX file system [Ritchie 74] and the Cambridge File Server [Dion 81] respectively. Examples of concrete byte oriented interfaces in runtime libraries include the standard C Library [Kernighan 78], and various BCPL libraries [Richards 80] [Middleton 79] [Wilkes 82].

The other common unit of data is the *record*. At an abstract level, records are modeled as a collection of named and typed *fields*. In a concrete model, the internal structure of records is largely irrelevant. The common concrete model of a record is as a *fixed group* and a number of occurrences of a *repeating group*. When there are no repeating groups, records are said to be *fixed length*. Records with repeating groups are said to be *variable length*, though this does not necessarily imply that the length of a given record can be altered. Most commercial operating systems provide a record oriented storage interface, often with a variety of different record formats. This allows the client to choose a format suitable for a particular application, but tends to make it more difficult to provide general purpose utilities. Examples of record oriented interfaces include RMS under VMS [DEC 80], the MVS Data Management Services [IBM 80] and Record Manager under CDC's NOS operating systems. Many of the older database systems produced at the time of the CODASYL report [CODASYL 71] are examples of record oriented interfaces.

2.2.2 Addresses and Keys.

Data units can be identified by address or record number. An *address* can be an absolute or relative disc address with an offset from the start of the block. For example in MVS access methods, an `actual` address contains absolute cylinder and head numbers, while a `relative` address is a logical track and block offset from the start of the file. Other interfaces, including all non-record oriented interfaces, use logical byte or word offsets as addresses.

When the storage interface uses record oriented addressing, the client may need to carry out non-trivial calculations to work out the address of a particular record. This is avoided if the interface allows records to be referred to by *record number*.

Record numbering is typically restricted to fixed length records or records with a fixed maximum length as in CDC and VMS **Relative** files.

The other way of identifying data units is by *key*. Keys come in two varieties; identifiers and data keys. *Record identifiers* are tokens typically generated by the storage interface when a record is allocated. As such they convey no other useful information. By contrast, *data keys* are client supplied values and may contain information that is intrinsically useful. However the interface merely sees a data key as a value. Data key values do not always need to be unique, and some interfaces allow multiple keys per record.

File organizations that support the retrieval of records in key order as well as randomly are commonly called *indexed sequential*. Keys are only used in record oriented storage interfaces, because a single word or byte rarely conveys enough information to justify the overhead of assigning it a key. Since auxiliary data structure need to be maintained, file organizations with keys generally occupy more space and are more expensive than those with only sequential or address based access[1].

2.2.3 Files and File I/O Operations.

Most concrete storage interfaces present a view of a file as a sequence of data units. Sequences can be either vector-like or list-like. New records can only be inserted at vacant positions in a vector-like sequence, while with a list-like sequence they can be inserted at any point. Sequences can be enumerated in an order defined by the addresses, numbers or keys for the records.

By contrast, some file organisations do not allow enumeration of records in any sensible order. Hash tables can be enumerated in an order defined by hashed key values. Organisations which simulate heaps often cannot be enumerated at all.

There are two modes of access to a file; *sequential access* and *random access*. With sequential access the client enumerates the file one data unit at a time,

1. MVS Direct Access Method is an exception to this rule which proves the point. Since it has no auxiliary index structure, record retrieval by key under BDAM is done by a linear search of the disc. This can tie down a channel for a long time, blocking other I/O activity.

performing operations on data units. Unoccupied record slots are generally (but not always) skipped by the storage interface. In many cases the client can backspace (sometimes only once) to return to a data unit that has already been enumerated. Sequential operations are defined in terms of the current data unit, so that the client might read, update or delete the current data unit or insert a new data unit before or after the current one. With random access operations, the client supplies an address, record number, key value or key pattern which is used to find a data unit. A random access operation is often just used to set the current data unit for subsequent sequential operations.

2.2.4 Memory Mapped Storage Interfaces.

Most concrete storage interfaces treat data in central memory as separate from data in the file system. The client uses the storage interface requests to explicitly move data between these two spaces. The alternative is to make explicit I/O unnecessary by having primary and secondary memory appear to the client as one level in the storage hierarchy.

Many computers have hardware for virtual to physical address translation. This hardware is frequently used to provide programs with their own virtual machines. It can also be used to support demand paging between primary memory and a swapping device. Additionally, address translation hardware can be used to implement a storage interface by mapping a disc file into a client's virtual address space. Such a file system interface is provided by the Pilot [Redell 80] [Xerox 84]. The Multics operating system [Organick 72] takes the notion a stage further by using a uniform segmented architecture for both the address space of client programs and the file system.

There are two ways to use a memory mapped storage interface. The client can set up a window into a file and move it around to view different parts of the file. The semantics are similar to those of a conventional buffered interface. Alternatively, the client can map an entire file into its address space and manage the data it contains using primary memory techniques. While the latter approach makes it easier to store complicated data structures, it introduces some extra problems.

First, the client needs to avoid wasting disc space by storing garbage along with the persistent data. In the absence of a language level compacting garbage collector, the client has to keep transient and persistent data separate. A second problem which arises with non-segmented architectures is that it is not always possible to map a file into a client's address space at the same virtual address. This means that the client may need to use relative pointers within the memory mapped file which can be a nuisance even in a language like Mesa [Xerox 84b] which supports them. Finally, the client generally has no control over the way that the paging system writes out "dirty pages". The state of a simple memory mapped file after a crash is hard to predict. Consistent file update requires the use of mechanisms like shadow paging which introduce extra overheads.

Mapped files can have a considerable impact on overall system performance because of competition for physical memory pages. The algorithms that the client uses to refer to mapped data must take care to avoid thrashing. In Pilot a client can inform the paging system that particular virtual memory pages are going to be needed soon or are no longer required. This allows the paging system to adjust its strategy as appropriate.

Not all memory mapped storage interfaces rely on address translation hardware. For example, the BCPL paged heap package [Jordan 79] can be viewed as simulating a segmented address space. The disadvantage of this particular interface is that the client needs to explicitly probe the segments to make sure they are in primary memory, and has to lock segments to stop them being paged out at the wrong time. Other examples include the concrete storage interfaces that underlie PS-Algol [Atkinson 82b] and arguably persistent Poly [Matthews 84].

2.2.5 General Points.

Concrete storage interfaces have their place where efficiency and speed of access or update are of overriding importance. The cost of efficiency is extra effort required in program development and maintenance, and often lack of generality and portability. The programmer loses the ability to define permanent data in abstract terms and the assistance of a type checker in detecting errors early. This is especially so in systems where the storage interface is a set of assembler macros.

Considerable knowledge is often needed to make effective use of a concrete storage interface. The programmer needs to select the most appropriate record type from the many possible, and decide whether to use buffered or unbuffered I/O. Unbuffered I/O requests need to be made in multiples of the physical disc block size aligned on disc block and memory page boundaries. Finally, the client often needs to select file size parameters when creating a new file so that space can be allocated efficiently. It would be better if the application programmer did not need to consider any of these issues.

It is the author's opinion that no single concrete model is suitable for all purposes. However, it is desirable that concrete storage interfaces should have some common properties. The most important of these are that an interface should hide differences in underlying hardware and details of disc allocation as far as possible. Interface consistency and simplicity are also important, especially when a storage interface provides a variety of record types or file organisations.

2.3 Storage Interfaces in Programming Languages.

2.3.1 Conventional Interfaces.

Many programming languages have storage interfaces similar to the concrete interfaces described in section 2.2.3. Such interfaces can be provided using programming language constructs or standard runtime libraries.

An example of such an interface is the `FILE` data type and the associated standard procedures in Pascal [Jensen 75]. A `FILE` is a sequence of units of a given Pascal type. The `FILE` variable defines a buffer variable of the type of the data unit which the client uses at the lowest level when reading and writing data. Low-level file I/O is done using the 4 standard functions `get()`, `put()`, `reset()` and `eof()`. There are also higher level `read()` and `write()` routines for `text` files for which Pascal allows variable numbers of arguments as a special case.

This style of interface has a number of problems. File typing is not generally well enough defined or implemented to detect incompatibilities that might arise between different versions of programs and files. Files produced on one system

may not be compatible with a second system in spite of the formal types being the same. This second problem means that file portability is generally achieved by encoding the data in a textual form which is both expensive and a potential source of errors.

2.3.2 Checkpointing as a Storage Interface.

Many functional and logic programming languages provide a storage interface based on *checkpointing*. In such an interface, the state of a computation is saved by copying the program's workspace to disc. State is restored either by executing the saved image, or by loading the state into a base system and resuming execution at the appropriate point. For example, Franz Lisp [Foderaro 83] provides a `savelisp` function which saves the entire program state as an executable image. Similar facilities are provided by other Lisp systems [Pitman 83] [Weinreb 83].

The main problem with checkpointing as a storage mechanism is that the client's entire computation state is dumped. Checkpoint files contain a large amount of redundant information. Systems including some implementations of Prolog [Pereira 83] keep the code of (say) the interpreter separate from a client's state so that only the client's state needs to be dumped. This does not help in the case where only a small part of the client's state has changed since the last checkpoint.

A second problem with checkpointing is the lack of resilience of checkpoint files. Minimal integrity checking is performed when a dump is taken. Thus, if a bug in client code damages some of the data structures, the corruption is likely to propagate to the checkpoint file undetected. When corruption is detected in a checkpoint file, the only realistic option is to rebuild from backups. For this reason, an alternative means of saving the client's state is essential, even if it is only a script of the user's input. A similar need arises when upgrades are made to the base system (or client program).

Finally, a checkpoint file is only usable by the program that produced it, and except in the case where the file is used read-only, by only one instance of that program at a time.

2.3.3 Data Flattening.

Most concrete storage interfaces, and interfaces provided by conventional programming languages provide little support for the storage of complex objects which use pointers. The main difficulty with storing pointers in the file system is that pointer values depend on the location, and it is not always possible (or desirable) to restore a data structure at its original location.

One way to deal with pointers is to write the data structure in a *flattened* form replacing pointers with location independent references. If the host language is garbage collected, there should be enough information around at runtime to distinguish pointers from other data. This makes it possible to write a storage interface that can flatten and store arbitrary data structures. Otherwise, flattening is still possible, but only with the assistance of the programmer.

A data flattening storage interface that requires minor programmer assistance is the `gc_dump` / `gc_read` package for some implementations of the CLU programming language [Liskov 81]. To save a cluster of a given type, the client invokes the type parameterized `gc_dump` routine giving the name of the file for the dump. The reverse process is performed by the `gc_read` routine which is also type parameterized. The only client assistance required is the inclusion in each cluster of a standard `_gcd` operation which simply applies the same operation to the cluster's representation. If the CLU runtime system made available more runtime type information, the `_gcd` operation would be unnecessary. A similar mechanism [Hamilton 85] is used for marshaling the arguments and results of remote procedure calls.

The problem with the CLU `gc_dump` / `gc_read` package is that it allows violation of the type system in the same way as Pascal files. Since there is no abstract type information available at runtime the `gc_dump` operation can only dump the representation of the data structure. When loading from a dump file, `gc_read` has no other type information whatsoever, so it has to assume that the encoded structure corresponds to the abstract type which it is expected to return.

The other sort of data flattening storage interface is one in which a non-trivial amount of help from the client program is needed. An example of this is the

"persistent data" package [Smith 84] for Modula-2 used in various VLSI design programs. Unlike CLU, the Modula-2 runtime system cannot distinguish pointers from other data or determine the size of an object. Thus, the client program must register a scanning routine for each **Type** of object to be dumped. This routine enumerates the data fields and the pointers in an object supplied as an argument, giving the **Type** for each pointer. A scanning routine can be written to omit redundant fields to save space, and possibly to upgrade to a new representation.

Another example of this sort of data flattening interface (though it is primarily intended for data transmission) is given in [Herlihy 81]. A client defines an internal and an external representation for each cluster and provides **encode** and **decode** operations. The paper mentions the possibility of versions of a cluster with a common external representation and different internal representations. As an extension to Herlihy's mechanism, [Hamilton 84] proposes a new primitive CLU type `flotsam` which is a discriminated union to which new variants can be added after a client has been compiled. Assuming a global registration scheme for `flotsam` tag values, Hamilton's scheme should solve the worst of the type safety problems of Herlihy's mechanism and allow multiple external representations for clusters.

Data flattening allows the client to handle transient and persistent objects similarly without many of the disadvantages of checkpointing. In cases where no client assistance is required, the client program can be internally and (theoretically) externally type safe. The flexibility to upgrade representations can only be had by sacrificing internal type safety and requiring the client to translate between different forms. As with checkpointing, data flattening is not suited to cases where the client needs to make small updates to a large permanent data structure.

Data flattening storage interfaces provide I/O operations whose semantics are based on copying. This causes problems when dumping objects with arbitrary connectivity. Loops and shared subnodes within a single object being dumped can be handled correctly at relatively small additional cost. This is not true for subnodes when subnodes are shared by separate objects. To preserve sharing in a data flattening interface, the client must treat a collection of objects that share subnodes as a single object when dumping and restoring.

2.3.4 Persistent Storage.

Persistent storage systems lie roughly between checkpointing and flattening interfaces. As with checkpointing, the client can treat a file as an extension of memory without worrying about explicitly copying the objects and the resultant problem of shared subobjects. On the other hand, persistent storage systems are more selective about the information that they dump, allowing small updates to be made at realistic cost. Persistent data can be independent of a single program, and concurrent access by different program instances is feasible, even if it is not usually implemented.

A good example of a persistent storage interface is that provided by the PS-algol language [Atkinson 84]. In PS-algol, persistence is achieved by transparently migrating a program's data objects between main memory and one or more databases. To start up, the programmer calls the **open.database** routine and is returned a pointer to a **root** object. As the client follows pointers outwards from the **root**, the objects referred to migrate into memory where they can be read and modified at will. When the programmer calls the **commit** routine, changes to existing persistent objects, and new locally created objects are propagated to the database(s) depending on whether or not they are accessible from the **root**.

The type system in PS-algol provides a single primitive **pntr** type which can refer to any type of object. Type checking occurs at runtime when the pointer is de-referenced and the object is used. The PS-algol type system is based on structural type equivalence rather than type name equality. Complete type information is stored in the database so that references to persistent objects from different client programs can be type checked.

Unknown to the programmer, PS-algol pointers actually have a local and a persistent form. The local form is an offset into a table maintained by the runtime system containing the actual memory address, the object's type and other information. When the program dereferences the persistent form, the runtime system hashes the persistent pointer to get a table offset. If necessary, the object is loaded into memory and a new table entry is allocated. The table offset is then substituted for the persistent identifier. When **commit** is called, any newly created local objects that are to be saved are allocated persistent identifiers. All

database objects in memory are then written out with local pointers replaced with persistent pointers. The PS-algol runtime and database algorithms are described in [Atkinson 82] and [Atkinson 82b].

In PS-algol, every reference to an object is made indirectly through the object table. In the VAX/UNIX implementation of the persistent Poly system [Matthews 82] [Matthews 84] pointers to objects in memory are normal addresses and are dereferenced directly. Persistent pointers are encoded as invalid addresses so that attempts to dereference them result in a hardware detected address fault. The Poly runtime system catches the resultant UNIX signal, fetches the object into memory, replaces the persistent pointer with the object's address and resumes. Since the cost of handling a UNIX signal is considerable, the Poly system tries to avoid unnecessary address faults by sweeping the stack and the database area, translating persistent pointers to objects that have already been loaded. Thus, Poly trades off lower costs for dereferencing local pointers against higher costs of handling persistent pointers. Another aspect where the Poly persistence system differs from PS-algol is that Poly distinguishes between mutable and immutable types, and can avoid writing most persistent objects that have not changed during a run.

2.4 Databases and Database Management Systems.

Database systems differ from most other storage systems in that they treat the data as separate from the programs that manipulate it, and go to considerable lengths to cut down on unwanted interdependencies. A *Data Base Management System (DBMS)* provides data management services for databases shared by a number of applications. The DBMS provides the client with abstract data storage and manipulation facilities. Concurrency control and access control facilities are also typically provided.

This section briefly outlines some of the commonly used database data models and some of the properties of DBMS's. The database terminology used is taken from [Tsichritzis 77].

2.4.1 Database Data Models.

The *network database model* is a loosely defined model which draws from graph theory. The primitives in a network database are records, types and links. A *record type* which is a collection of named fields corresponds to an attribute relationship. Duplicate record instances are in general allowed. A *link* represents an N:M association between entity sets. Links can be either information carrying or non-information carrying. *Information carrying* links give information that is not otherwise represented, and must therefore be constructed explicitly. *Non-information carrying* links duplicate information present in the fields of records.

The database is viewed as a network of nodes and arcs connecting the nodes, where the nodes are instances of record types and the arcs are instances of links. Record processing involves starting with the records of one record type, selecting on the values of record fields and following links to other record types.

The *hierarchical database model* is a special case of the network model. In the hierarchical model, links can only represent 1:N relations, with one *parent* record type having links to one or more *child* record types. Furthermore, when a database of record types and links is expressed as a data structure diagram, the diagram must form an ordered tree called a *hierarchical definition tree*. The head of the tree is the *root record type* which has no parent record type.

Data selection with the hierarchical model is done by *tree traversal* or by *hierarchical selection*. In the former, the tree is traversed selecting records of a given record type that match a given qualification. The client may restrict the search to the children of a selected parent record, or to scan all records. With hierarchical selection, a record type is treated as a hierarchical set and records are selected on the basis of the values of fields and the parent child relationships.

The hierarchical model is more restrictive than the network model in the data that it can represent. However, the data to be represented can be simplified by the process of normalization, which among other things reduces N:M associations to 1:N associations.

The *relational database model* [Codd 70] is the first model to have a formal mathematical basis. The model uses *relations* to represent both associations and attribute relations, and formally defines the basis for the manipulation of data with a *relational algebra*.

A relation is a set of N-tuples, where each tuple represents a relationship between a collection of attribute instances. Unlike the network model, the tuple instances for a given relation are unique. A relation expresses the attribute relationships of a given object type in an obvious way. Associations between objects are expressed as relations whose elements are the keys of objects. It is not necessary to store the equivalent of non-information carrying links in a relational system since the relations can be derived "on the fly" using the relational algebra.

Relational algebra defines a collection of *relational operators*. These operators all take relations as arguments and return relations as results. There are 3 simple operators for eliminating unwanted information from a relation. *Restriction* (R[Aθv]) and *selection* (R[AθB]) give subsets of the tuples in a relation which satisfy a simple condition θ between an attribute value, and respectively a literal or another attribute of the same tuple. *Projection* (R[A]) gives a relation with some of the attributes of the original after duplicate tuples have been eliminated. Three operators, *union* (R∪S), *intersection* (R∩S) and difference (R–S) perform set operations on compatible relations. The final 3 operators combine relations in other ways. The *cross-product* (R⊗S) operator produces a relation consisting of all possible unique combinations of two relations. The join operator (R[AθB]S) gives all combinations of the tuples in two relations for which a condition θ holds between a given attribute from both relations. The division operator (R[A ÷ B]S) expresses the "for all" condition.

2.4.2 Data Definition in DBMS's

The most difficult part of setting up a DBMS data base is defining the data it holds. A data definition or *schema* [CODASYL 71] describes various properties of the data. In discussions of the architecture of an ideal DBMS [ANSI 75] there are typically 3 levels of schema (figure 2-1), describing different aspects of the database.

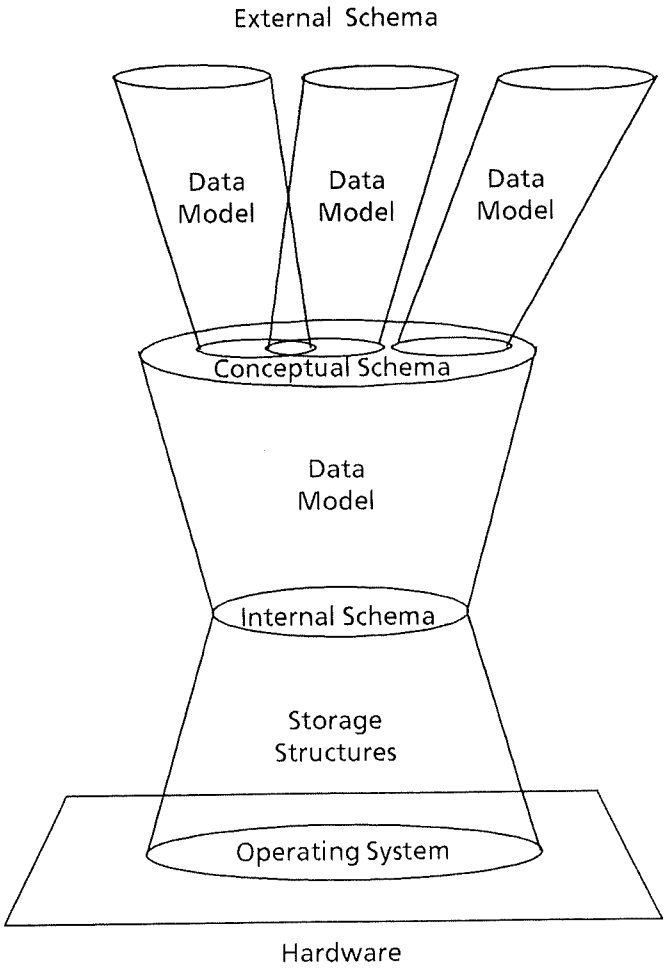

Figure 2-1

The central *conceptual schema* is the logical description of the entire database. Written in some *data description language* or DDL, it specifies a global data model for the entire database. A DDL allows the Data Base Administrator to name and describe record types, specifying such things as

- the domain and units of measurement for data values,
- integrity constraints,
- locking rules, and
- data access control rules.

An *external schema*, typically written in the same DDL, describes a particular application's view of the database. The external schema are intended to give

clients *logical data independence* by isolating them from changes to the conceptual schema.

The internal schema describes how the global data model is mapped onto the storage structures of the DBMS. The internal schema, written in a *storage definition language* or SDL, describes such things as

- the storage medium or device to be used,
- how record types are mapped to files,
- file ordering and indexing, and
- buffer allocation strategy.

Changes can be made to the internal schema to take account changes in hardware and to improve the performance of the DBMS without affecting applications. Separating the internal schema from the conceptual schema gives the applications *physical data independence*.

2.4.3 Data Manipulation Facilities in DBMS's.

A DBMS typically supports two forms of data manipulation facilities. Interactive query systems give the user direct access to the database for ad hoc enquiries. While interactive query is important, it is not relevant to this thesis. The other data manipulation facility provided is the interface used by application programs.

A program level interface to a database in a DBMS takes three forms. The simplest form is a library of routines called from the host programming language. The second approach to the client / DBMS interface is to embed *database manipulation language* (*DML*) statements in the host programming language. To do this it is necessary to extend the syntax of the host programming language: something that may be frowned upon by language purists. The result allows database operations to be expressed in a more natural and concise fashion. This approach also allows the client / database interface to be type checked (as far as the host language allows this).

Most of the complexity of database interfaces embedded in a host language is due to the incompatibility of procedural host languages and non-procedural database

models. This is particularly apparent in database selection. A selection which is specified non-procedurally, gives a collection of records which must be returned to the client program so that it can deal with each one procedurally. The solution adopted by most DBMS's is to use a *cursor* to step through the selected records one at a time.

The third approach to interfacing the client with the database is to write the client program entirely in a DML. This avoids type incompatibility between the DML and separate host languages, and removes the problem of "mixed metaphors" programming described above. The SQL language used IBM's DB2 system [Date 84] is an excellent example of a combined DDL and DML that can be used on its own or embedded in PL/1, COBOL, FORTRAN or Assembler programs.

2.4.4 Data Evolution in DBMS's.

Database systems generally provide support for evolving data requirements in two ways. The first way is to provide tools for converting the database from an old format to a new one. This is not enough by itself, since client programs would also need to be converted to use the new format data. The use of schemas and subschemas helps to avoid this problem. There are however other problems.

Converting a database from one format to another is typically both expensive in machine resources and time consuming. In a system where the data must be available at all times, the latter is a special problem. While schemas and subschemas help before and after reformatting, they are no help while reformatting is in progress. Instead it is necessary to take a snapshot of the database and keep a log of transactions that occur while conversion is underway.

Most DBMS's process database requests using an interpreter so that application programs don't have to be recompiled whenever the database is changed at the level of the conceptual or internal schema. IBM's DB2 system is unique among commercial DBMS's in that the SQL statements are compiled to machine code. A client program with embedded SQL is passed through a precompiler to extract SQL statements, and insert appropriate type declarations and calls to the runtime supervisor. The extracted SQL source is stored by DB2, then compiled and optimized for the current structure of the database. When the client program is

run, the DB2 runtime supervisor checks that the database layout has not changed. If necessary, the SQL statements are recompiled transparently before link loading.

3 - The Basic Entity System.

3.1 What is an Entity?

An abstract object can be considered as having three key properties;

- the type or mode of the group of components which represent the object,
- the invariants for the component values, and
- the actual values of the components.

For example, a **date** can be modeled as having 3 integer components **day**, **month**, and **year**. The **year** has no constraints on its value, the **month** must have a value in the range 1 to 12, and the value of the **day** component depends on both the **year** and **month** values. Manipulations of a **date** must conform with the component type rules and must not violate the value invariants.

If a client program has direct access to the fields of an data item, it is difficult to ensure that the invariants are maintained. In most programing languages, the invariants of a data structure are not reflected in the type definition and therefore cannot be checked. One way to avoid this problem is to hide an object's representation behind a set of procedures which carry out the necessary operations. Only the object's operation procedures need to know how the object is represented and how operations are performed.

In some programing languages, an *abstract data type* (as distinct from a concrete or representational data type) formally defines data items in terms of the *operations* which are used to manipulate them. In this general context, an instance of an abstract data type is called an *object*, and the style of programing using objects is referred to as *object oriented*.

Abstract data typing provides the programmer with an additional means of expressing abstractions which makes it easier to write, debug and verify programs correct. Furthermore, encapsulating the representation of data items reduces the likelihood that a client program will corrupt the data. This is very important in a file system where an object's lifetime is independent of a single program instantiation.

Encapsulation of data using abstract data types is central to the entity system. Objects in the entity system are called *entities*. An entity is characterized by three distinct properties; its class, its implementation and its representation.

The first property of an entity is its *class*. A class is an abstract data type which defines the operations that can be applied to an entity. Operations are specified in terms of the formal types of the arguments and results, and their external semantics. The entity system uses strong data typing, so the class of a given entity cannot be changed.

The second property of an entity is its *implementation*. The implementation is the body of code that manages the state which represents an entity. It provides procedures for performing the entity operations defined by the class, hiding details such as the entity's representation format and the operation algorithms from the client. A given entity implementation will manage a number of entities of the same class. However, there can be a number of implementations for a given class, each using different representation layouts or operation algorithms. A client program may simultaneously make use of different implementations of a class without being aware of it.

The third property of an entity is its *representation*. An entity's representation includes state held in secondary storage or cached by the implementation, and the local variables of an implementation procedure that is performing an operation. The representation is defined and managed by the entity's implementation, and is only accessible using the operations defined by the entity's class.

To recapitulate, an entity is an instance of an abstract data type known as a class. It is managed by one of a number of type managers for the class which are known as implementations. The client of an entity sees it in terms of the interface presented by the class. This is illustrated in figure 3-1.

Splitting the client interface from the implementation has two important benefits. The first is that improvements to the algorithms or representation *that do not entail a change to the object's interface* can be made by producing a new implementation. It is not necessary to change or even recompile existing client programs to use the new implementation. In addition, it is not necessary to

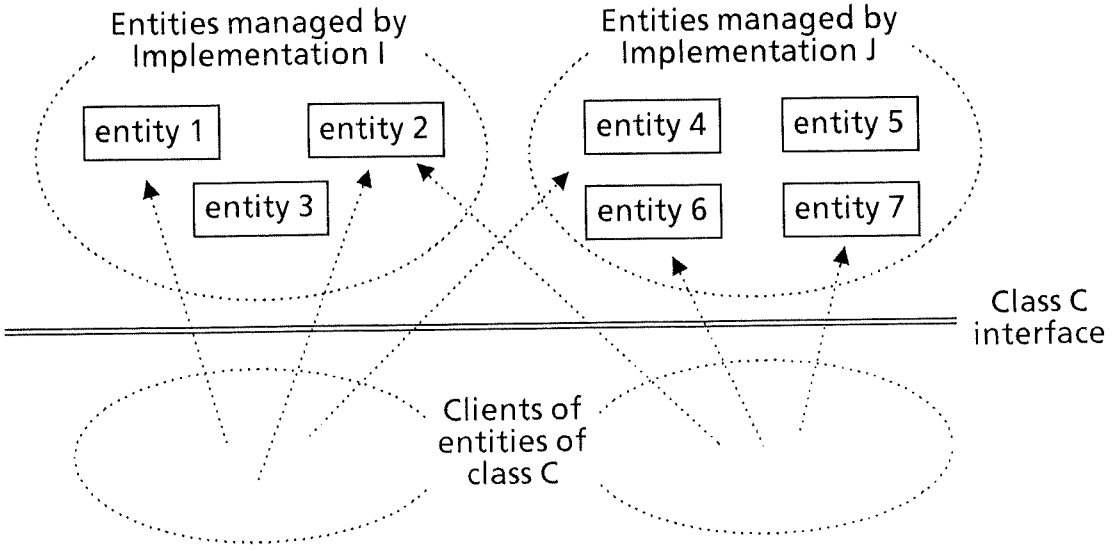

figure 3-1

convert existing entities to the new implementation, since client programs can use a mixture of old and new implementations. If it is desirable that old entities be upgraded to use a new implementation, the changeover can be made gradually. New entities can be created with the new implementation and existing entities can be recreated as and when convenient.

A second benefit of splitting the class and implementation is that it allows the programmer to write a number of implementations of a given class, each tuned to work best in different situations. Suppose there is a class of entities which store simple ASCII text. The best method of representing a given piece of text depends on its size and the degree to which this is likely to vary. Algorithms that are best for small text entities are not likely to be good for large ones. The entity system gives the programmer the option of writing different text implementations tuned for these two cases and others. While a client creating a text entity needs to choose an appropriate implementation, a client using an existing text entity is not aware of which implementation is used.

Entities exist in two forms; passive and active. Entities in the *passive* form are held in secondary storage. Entities in the *active* form reside partly in primary storage. It is only possible to apply operations to active entities.

Passive entities are signified by entity identifiers. An *entity identifier* or *id* is a unique and unforgeable token for an entity in its passive form. The same entity

identifier refers to the entity for as long as it exists. Bound to each entity id is an *entity triple* consisting of a *class id*, an *implementation id* and a *representation id*. Class and implementation ids are unique tokens signifying respectively, an entity's class and implementation. They may well be the identifiers for entities holding the class definition and the implementation code, but this is not essential to the model. The class and implementation ids are fixed when an entity is created and cannot be changed.

A representation id signifies that part of an entity's representation which persists on secondary storage when the entity is made passive. The exact meaning of a representation id is known only to the entity implementation. It may be the entity id for a *storage object:* an entity whose purpose is to provide lower level data storage. Alternatively, it could be a private token which is understood by the implementation alone.

Figure 3-2 illustrates the relationships between the components of a number of passive entities. **E1** through **E5** are the identifiers for entities of class **C**. The entities with identifiers **E1** and **E2** are managed by the implementation **I** which uses tokens **T1** and **T2** to signify the entity representations. The remaining entities of class **C** are managed by implementation **J** which uses storage objects of class **C-S** to hold the representations. Thus the representation ids for entities **E3** through **E5** are identifiers for the storage object entities **S3** through **S5**.

An active entity is signified by an entity handle. An *entity handle* is effectively a capability which allows a client to request operations on an entity. There can be a number of handles for a single entity, each conveying different rights to perform operations to the respective clients. The semantics of the respective rights, and access rules in general are part of the class specification. Entity handles are required for all operations on entities with the exception of some of the generic operations discussed in the next section.

Each class defines an entity handle type that is incompatible with other handle types, so that handles and operation invocations can be statically type checked. An entity handle is typically an opaque pointer to a data structure private to the entity system. The handle data structure includes an *operations table* which holds pointers to operations procedures provided by the implementation. Indirection

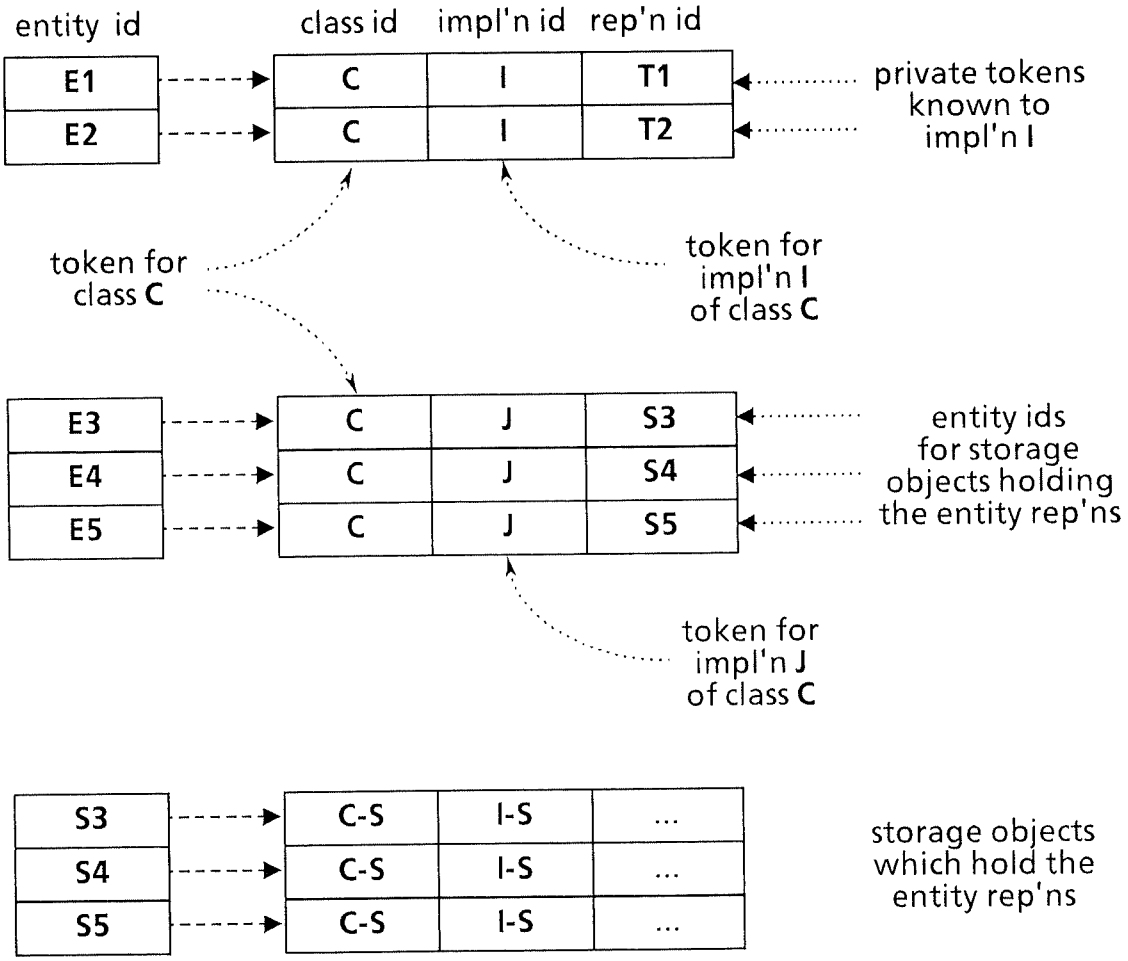

figure 3-2

through the operations table is what allows a client to use entities with different implementations without being aware of it.

The entity handle data structure also refers to the entity activation. An *entity activation* is the part of an active entity's representation which persists between operations, but not when the entity is made passive. In the standard model, there is exactly one activation for an active entity which exists as long as there are handles for the entity. An activation can be thought of as the "global frame" for an active entity.

An entity's implementation can use an activation for any number of purposes. If the entity's passive representation is held in a storage object, the handle for the object will be held in the activation. It can be used as a cache for the entity's permanent representation. It can hold lock variables and semaphores that the implementation uses to synchronize concurrent operations on the entity. Finally

it can hold state associated with individual entity handles, such as the rights that the handle conveys, or the state of an enumeration in progress.

Figure 3-3 illustrates the relationships between entities, handles and activations. There are two active entities **E1** and **E2** with class **C** and implementation **I**. There are two handles **E1/1** and **E1/2** for the first entity, the first conveying "read-write" access rights and the second "read-only". A third handle **E2/1** refers to the second entity. The operations tables in the private handle structures are shown as holding references to the appropriate implementation procedures, and the associations with the activations. Each activation is shown as including a handle for a storage object (**S1** or **S2**) containing the permanent representation of the entities. The private handle data structures are shown as straddling the double

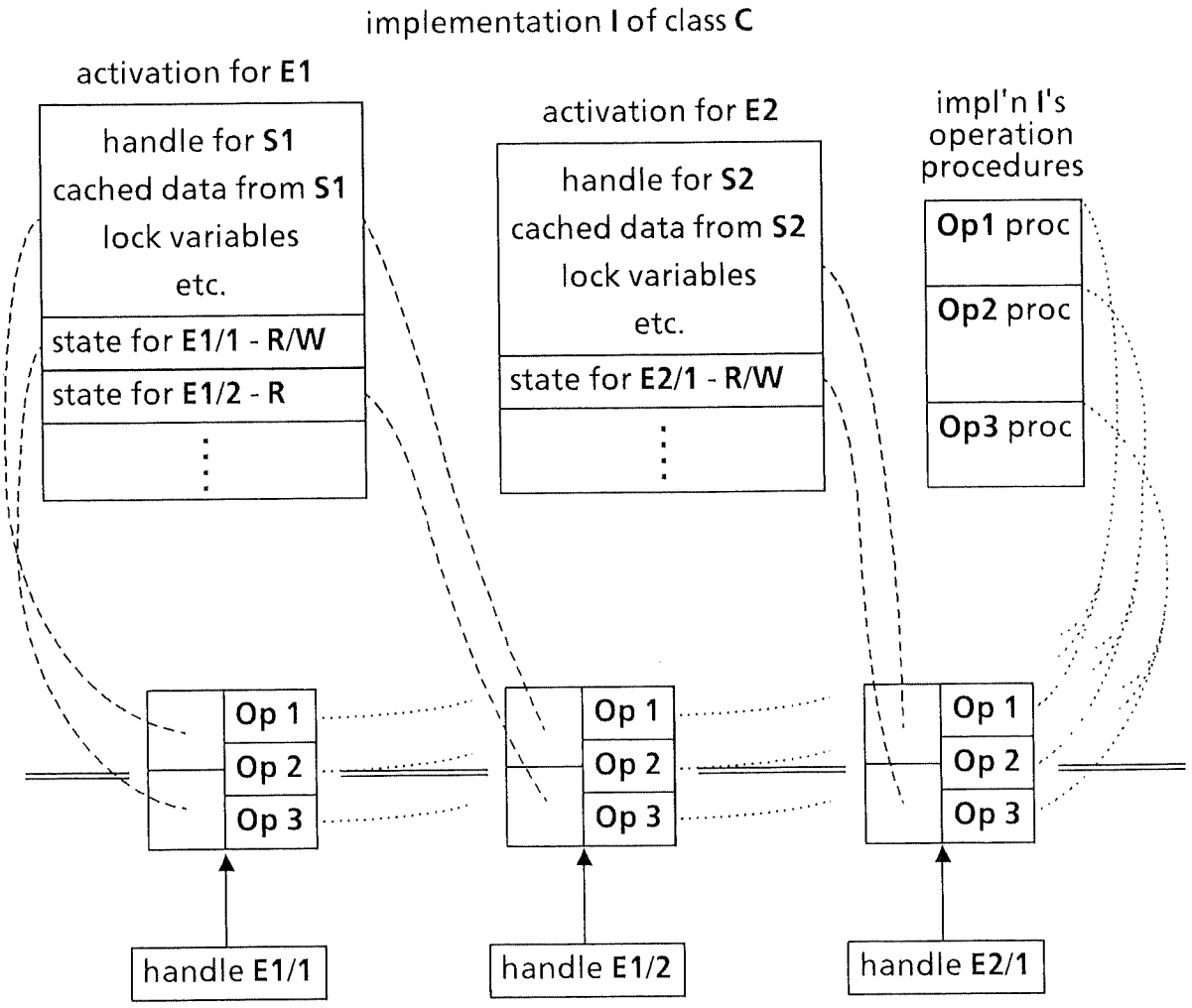

figure 3-3

line to indicate that they form the interface between the implementation and the clients.

3.2 Entity Operations.

Operations on entities fall into 2 distinct groups. The majority of operations are *class specific*, having no meaning in the context of other classes. A few operations are *generic* in that they can be applied to entities of any class. There are typically four such generic operations for each class of entities; **Open**, **Close**, **Create** and **Delete**. The class specific operations are handled by the entity's implementation[1], while the generic operations require a call to the entity system kernel. It is assumed that the types of the entity handle, the arguments and the results can be checked when the client program is compiled.

The syntax that the programmer uses to specify an entity operation invocation will depend on the way that entity handles and other entity system data types map into the client programing language. In this thesis, examples will be given in a pseudo-code that has similarities with Mesa and CLU. In this pseudo-code, a construct of the form **handle$op** indicates that the abstract operation **op** defined by a class **C** is applied to the entity of class **C** referred to by **handle**.

In general, an invocation of an entity specific operation is expressed by pseudo-code of the form

$$res_1, \ldots\ res_m \leftarrow \texttt{handle\$operation}(arg_1, \ldots\ arg_n)$$

where `handle` is an entity handle, **operation** is the name of the operation being invoked, $\mathbf{arg_1}$ through $\mathbf{arg_n}$ are the actual arguments for the operation, and $\mathbf{res_1}$ through $\mathbf{res_m}$ are the results. The entity operation invocation results in a call of the form

$$res_1, \ldots\ res_m \leftarrow \{impl\}.\texttt{operation\_proc}(\texttt{handle}, arg_1, \ldots\ arg_n)$$

1 This is an over simplification. In an entity system implemented in a closed operating system, the kernel may well need to be involved with passing arguments and results, and with dispatching the operations.

where **{impl}** is some implementation of the class of **handle**, and **operation_proc** is an operation procedure supplied by the implementation. The value of **{impl}.operation_proc** will have been found in the operations table associated with the **handle**.

Before an entity can be used, the client must obtain a handle using the *open operation* defined by the class. An invocation of an open operation has the following form,

$$\texttt{handle} \leftarrow \texttt{\{classname\}.Open(entity\_id, arg}_1\texttt{,... arg}_n\texttt{)}$$

where **{classname}** is the name of some class, **handle** is an entity handle whose type is defined by the class, **entity_id** is the identifier for the entity to be opened, and $\texttt{arg}_1$ through $\texttt{arg}_n$ are class specific open arguments.

An open operation has a number of functions. First it checks that the entity is of the correct class. Next, it creates a handle for the entity of the appropriate type. If the entity is passive, this involves finding the entity's implementation and producing a new activation. Finally the entity implementation's open operation procedure is called with the remaining arguments to perform implementation specific initialization.

Though most of an open operation either can be or has to be performed by the kernel, the final stage is class specific. This suggests the following simple scheme for handling an open operation invocation

$$\texttt{handle} \leftarrow \texttt{Kernel.Open\_Entity(entity\_id, class\_id);}$$
$$\texttt{\{impl\}.Open\_proc(handle, entity\_id, arg}_1\texttt{,... arg}_n\texttt{)}$$

In the above, **class_id** is the class identifier corresponding to the class whose open operation is being called. As for other entity operations, **{impl}.Open_proc** is found in the entity handle's operation table.

For reasons that will become apparent in chapter 4, it is necessary to implement the invocation of an *Open* operation in a more complicated way. The class specific arguments are copied into a record and a pointer to the record is passed as an argument to the **Kernel.Open_Entity** call.

```
arg_rec.arg1 ← arg1; ... arg_rec.argn ← argn;
handle ← Kernel.Open_Entity(entity_id, class_id,
                            ADDRESS_OF(arg_rec))
```

When **Kernel.Open_Entity** has completed all it needs to do, it finds and calls **{impl}.Open_proc** directly, passing the pointer to the argument record.

An implementation's open operation procedure typically performs a number of tasks. It initializes any data structures in the activation associated with the handle. If the entity has been newly activated, it extracts the representation id from the triple and where appropriate opens the representation storage object. The class specific arguments may give some idea of what the client expects to use the handle for. The operation procedure checks that the client should be allowed access to the entity, and ensures that this does not conflict with handles for the entity that already exist. Finally, it may pre-read strategic parts of the representation from secondary storage in anticipation of the client's operation invocations.

When the client has finished using an entity, it explicitly releases the handle by invoking the *close operation* defined by the class. A close operation invocation has the form

```
handle$Close(arg1,... argn)
```

where `handle` is the entity instance to be closed, and **arg**$_1$ through **arg**$_n$ are class specific close arguments. The first thing that a close operation does is to call the implementation's close operation procedure as follows.

```
{impl}.Close_proc(handle, arg1,... argn)
```

A close operation procedure performs housekeeping tasks necessary before the handle is released. These tasks may include flushing data out to secondary store, and releasing resources associated with the handle. Depending on how the class is defined, the close operation procedure may also be the appropriate point to "commit" or "abandon" outstanding transactions[2], or perform some other actions that require class specific arguments. If the handle is the last one for the activation, there may be additional things to do.

When the close operation procedure has finished, the kernel is called to release the entity handle, and if necessary the activation. The kernel close call has the following form.

```
Kernel.Close_Entity(handle)
```

A new entity is generated by the *create operation* defined by the entity's class. An invocation of create takes the following form.

```
entity_id, handle ← {classname}.Create(impl_id, arg1,... argn)
```

In the above, **impl_id** gives the client's selection of implementation for the new entity, and $\mathbf{arg_1}$ through $\mathbf{arg_n}$ are class specific creation parameters. The results of the create operation are an entity identifier and a handle.

A create operation is performed in two parts as follows

```
entity_id, handle ← Kernel.Create_Entity(impl_id);
{impl}.Create_proc(handle, entity_id, arg1,... argn)
```

where **{impl}.Create_proc** is the implementation's create operation procedure taken from the entity handle. The **Kernel.Create_Entity** routine makes a new entity identifier and triple. Then it finds the implementation, allocates an activation for the new entity and makes the handle. The **{impl}.Create_proc** performs any implementation specific initialisation. It initialises the activation for the new entity. It allocates and initialises the entity's long term representation, and binds the corresponding representation id into the entity triple. Depending on how the operation is defined, the class specific arguments can include initialisation parameters and information about the initial entity handle.

The *delete operation* is applied when an entity is about to be removed from the filing system. At the appropriate time, the entity system kernel calls the implementation's delete operation procedure as follows.

2 The entity system does not provide primitives for implementing atomic transactions. However there is no obvious reason why classes and implementations cannot be designed with these properties.

```
{impl}.Delete_proc(handle, kill)
```

This procedure releases any resources held by the entity, and deals with any interdependencies with other entities. In some situations it is reasonable for an entity's implementation to take steps to prevent the entity from being deleted. To avoid problems with undeletable entities, the kernel needs to be able to suppress entities that are behaving pathologically. If an entity is about to be suppressed, the **{impl}.Delete_proc** is called with the **kill** flag **TRUE**.

3.3 Some Examples of Entities.

In this section, a number of entity classes are discussed as examples of the entity system described so far. They will also be used to illustrate a number of problems which have not yet been encountered. The example classes described are

- timestamp entities,
- text entities,
- directories, and
- Modula-2 program entities.

The first (trivial) example is the class of **Time_Stamp** entities. A timestamp is simply a record of the time. When a significant event occurs, the client program may record the time by applying the **Take_Stamp** operation. At a later time the timestamp may be read back using the **Read_Stamp** operation. In the notation of the previous chapter, the client interface to a timestamp is as follows

```
ts_handle$Take_Stamp()
time ← ts_handle$Read_Stamp()
```

where **ts_handle** is an entity handle whose type is defined by the **Time_Stamp** class, and **time** is a numerical time value. The following operation procedures would need to be provided by an implementation of the **Time_Stamp** class.

```
{impl}.Take_Stamp_proc(ts_handle)
time ← {impl}.Read_Stamp_proc(ts_handle)
{impl}.Open_proc(ts_handle, entity_id)
```

```
{impl}.Close_proc(ts_handle)
{impl}.Create_proc(ts_handle, entity_id)
{impl}.Delete_proc(ts_handle, kill)
```

where **time** and **ts_handle** are as above, and **entity_id** is the identifier for a timestamp entity.

In a simple implementation of the timestamp class[3], the value of an active timestamp might be cached in memory. **Open_proc** would read the stamp value into memory, **Read_Stamp_proc** would return a copy of the cached stamp, **Take_Stamp_proc** would update the cached stamp, and **Close_proc** would flush the cached stamp back to the passive storage area. **Create_proc** would simply initialize the passive representation to a null value, and **Delete_proc** would be a dummy routine.

In some situations, it is preferable for a timestamp value to be written to stable store the moment it is updated. To this end, a second implementation can be defined with the appropriate properties. Timestamps could then be created with which ever implementation is more appropriate, and programs which use the entities would never need to know the difference.

The **Time_Stamp** class could have been defined to provide another operation of the form

```
date, day_of_week, time_str ← ts_handle$Stamp_Strings()
```

where **date**, **day_of_week** and **time_str** together make up human readable string representation for a timestamp value. At first sight this seems quite sensible, since producing a human readable representation for the value of a given a timestamp entity is likely to be a common activity. However, since every implementation of the class would need to support the time to string conversion, it is better to provide a library routine of the following form.

```
date, day_of_week, time_str ← Time_Strings(ts_handle)
```

3 The Time_Stamp example class is somewhat contrived. In practice timestamps would normally be a part of a larger entity and would not have a separate implementation.

A more general way of providing this sort of derived operation will be discussed in chapter 4.

The next example is of a class of **Text** entities. Entities of this class could be the source code of computer programs, unformatted documents or sections of documents, command scripts: anything which can be represented as a series of characters. By way of illustration, the **Text** class is defined to support atomic update.

Most programs which use text objects will want to refer to them in terms of a character stream. However, for the sake of efficiency the class of text objects is defined to provide a buffered interface onto which streams can be retrofitted. The operations of the class can be defined as follows.

```
buffer, nos_read ← text_handle$Read(nos_chars)
text_handle$Write(buffer, nos_chars)
text_handle$Seek(position)
text_handle$Set_End_Of_Text()
position ← text_handle$Tell()
position ← text_handle$Tell_End_Of_Text()
```

The **Read** and **Write** operations transfer data starting at the current position[4], with **Write** automatically extending the entity. The **Seek** operation resets the current position within the text, and the **Set_End_Of_Text** operation truncates or extends the entity to the current position. The **Tell** operations return the current position and the position of the end of text respectively. The class definition for text entities could define additional parameters for the **Open** and **Close** operations as follows

```
text_handle ← Open_Entity(entity, text_class_id, mode)
Close_Entity(text_handle, commit)
```

4 Here it is assumed that the operation invocation mechanism does not lose messages. If this were not so, the Text interface could be defined to have idempotent semantics which are generally less convenient for the client.

where **mode** is a collection of flags specifying read access, write access, truncate on open and so on, and **commit** is a flag which allows the client to commit the changes made or to cancel them. It should be noted that both access control and atomicity must be provided by the implementation of a text entity.

A third example is the class of **Directory** entities. In a conventional file system, the directory is the primary (often the only) mechanism for naming and associating a number of related files. In the entity system, special purpose classes can provide whatever structuring is appropriate to a particular application: the final example is intended to illustrate this point. In some situations though, the conventional directory model is the most appropriate way of organizing information. This example is an attempt to model the high level properties of the CHAOS filing system [Wilkes 79].

A **Directory** entity consists of a collection of named entries providing connections to other entities. Each directory entity handle has an associated set of access rights {**C**, **V**, **X**, **Y**, **Z**}. These rights are requested by the client and checked by the implementation when the directory is opened. The **C** right allows the client to create new entries in the directory. The {**V**, **X**, **Y**, **Z**} rights determine the access the client has on individual entries. The exact meaning of these rights depend on the context, but they may be thought of as representing access groups or levels of access.

Each directory entry consists of a name, an entity id, an access matrix for the entity and a set of (say) 8 entry access bits. Three of the entry access bits {**D**, **A**, **U**} control respectively deletion of the entry, alteration of the access matrix and updating of the entity identifier. The remaining 5 access bits are defined by the class of the entity held in the entry. The access matrix is a 4×8 matrix of boolean values with a row for each of {**V**, **X**, **Y**, **Z**} and a column for each access bit.

When the client invokes a directory operation for a given entry, the vector of rights {**V**, **X**, **Y**, **Z**} is multiplied logically by the access matrix to give an 8 bit access mask. The result is a mask which is ANDed with the entry's access bits to give the client's effective access. This is illustrated in figure 3-4. When a client whose handle gives {**V**, **X**} access to the directory requests an operation on an entry whose access matrix and access bits are given, the effective access to the entry is

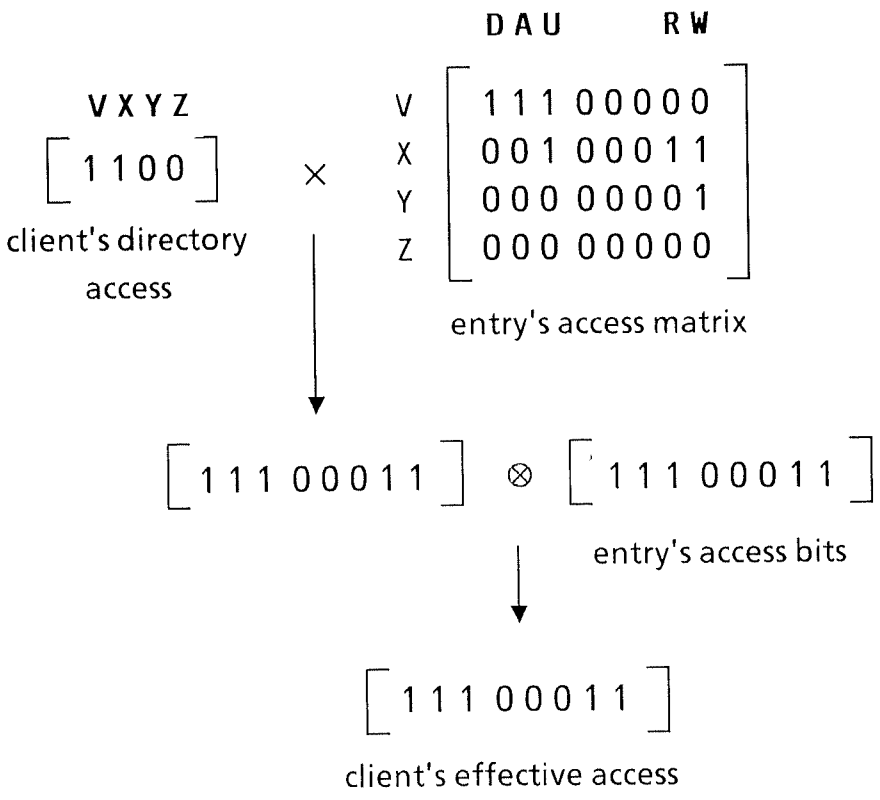

figure 3-4

{D, A, U, R, W}. Similarly, just {Z} access to the directory gives the client no access at all to the entry.

The class specific access bits represent the client's access to the entity held in the entry. When the entity is opened, they are passed to the entity's implementation as an open argument. Figure 3-4 shows two access bits; R and W. The interpretation for an entity of another class would be different. A directory entity controls access to entities by performing entry open operations on the client's behalf. This is difficult to express cleanly in most languages.

The operations provided by the directory class can be described as follows.

```
handle ← dir$Open_Entry(entry_name, class,
        requested_access, arg1,... argn)

dir$Insert_Entry(entity_id, entry_name,
        access_matrix, entry_access)
```

```
dir$Update_Entry(entity_id, entry_name, entry_access)

dir$Alter_Matrix(entry_name, new_matrix)

dir$Delete_Entry(entry_name)

dir$Alias_Entry(entry_name, alias_dir, alias_name,
        access_matrix, entry_access)

found, entry_access, access_matrix, times ←
        dir$Examine_Entry(entry_name)

found, name, entry_access, access_matrix, times ←
        dir$Examine_Next(first)

dir_access, times ← dir$Directory_Info()
```

The **Insert_Entry** operation creates a new entry called **entry_name** which contains the given **entity_id**, with **access_matrix** and **entry_access**. The **Update_Entry**, **Alter_Matrix** and **Delete_Entry** operations respectively replace the entry's entity id, replace the access matrix and delete an existing entry. **Alias_Entry** allows entity sharing and moving without breaching the access control system. An entry called **alias_name** is inserted into the directory **alias_dir** with access determined by masking **entry_access** with the effective permissions from the original entry. When an appropriate entry is present, **Examine_Entry** returns information about it. The **Examine_Next** operation allows a client program to scan through a directory looking at all of the entries. Finally, **Directory_Info** returns the timestamp values for the entire directory and the directory access that is given by the client's handle.

A directory entity has a number of timestamps. Associated with the directory as a whole there are timestamps for the directory's creation and for the directory's last update and access. In addition, each directory entry has an individual last update timestamp. These timestamps are accessible through various examine operations described above. However, this sort of interface is unnecessarily restrictive.

Suppose we want to write a client program whose sole purpose is to display timestamp values. When this program is applied to a free standing timestamp entity, the operation to extract the time value from the entity will be of the form

```
time ← ts_handle$Read_Stamp()
```

while for a directory entity, the operation will be either

```
..., times ← dir_hand$Examine_Entry(entry_name)
```

or

```
..., times ← dir_hand$Directory_Info()
```

depending on which of the many timestamps is required. For other classes the interface is likely to be different again. The timestamp display program would also need to know how to open each class of entity. This would be just about bearable if it were not for the fact that new classes may be being defined all the time, requiring continual changes to this and other utility programs. This illustrates the need for support for *polymorphism* in the entity system. This issue is discussed in chapter 4.

The directory entity also illustrates the need for support for multiple handles and nested operation requests on a single entity. Suppose that the client wants to alias an entry within a directory; e.g. the CAP command **"alias .a .b"**. This maps onto an implementation procedure call of the form

```
Alias_Entry(current_dir, "a",... current_dir, "b",... )
```

which may then invoke the operation.

```
current_dir$Insert_Entry("b",... )
```

The result is two overlapping operations on the entity handle **current_dir**. Multiple handles for a single entity arise when a client requests an operation on the directory structure that involves following a loop. For example, if the component **"*"** in a pathname indicates a link to the **root** node of a filing system, and the user is selected to the root directory, then following pathnames of the form **".a.b.*.a"** or **".*.a"** is likely to result in multiple handles for the **root** directory entity.

An example which illustrates the use of the entities as a structuring mechanism is **M2_Program**, the class of Modula-2 program entities. The previous two examples of entity classes are essentially a rehash of conventional filing system facilities. The classes provide no features that are not already provided by conventional

systems (without the benefit of typing and support for multiple implementations of course). The same cannot be said for the following example.

A computer program may be viewed as a complex data object with a large number of heterogeneous components. In the case of Modula-2, these include

- source code for the **DEFINITION** and **IMPLEMENTATION** modules,
- symbol and code reference tables for the modules,
- relocatable binaries for the **IMPLEMENTATION** modules,
- the executable image of the program,
- compiler and linker directives for producing the executable image,
- information about dependencies on external subroutine libraries,
- test programs and benchmarking data,
- internal and external documentation, manual entries, help files, etc.,
- change logs.

In a conventional system, the components of a program are organised in an ad hoc fashion using filename suffixes, and other conventions that are liable to break down. Dependency information in the form of a **Makefile** is typically inaccurate or incomplete.

In the entity system, a class of **M2_Program** entities can be defined which ties together the components in a well defined way. Each module of a program would be represented by an **M2_Defs_Module** or **M2_Impl_Module** subentity. Other components such as the executable image, the linker directive and program documentation would be represented by the entity itself. The **M2_Program** class would provide operations for reading and writing the top level components, initiating a link-edit and for gaining access to the subentities.

An **M2_Module** sub-entity would contain the source code, compilation directive, and dependency list for the module as well as the derived object code, symbol tables and debugger information. Operations would be provided for reading and

writing the source code and the compilation directives. Another operation would be used to request recompilation. The dependency list would be a list of pairs of module names and **M2_Module** entity ids for other modules in the program and shared library modules. The compiler would check that the dependencies correspond to the source code. If the compilation was successful, it would also update the derived components using operations provided for that purpose. Other operations would be used by the linker and debugger to read the object code, symbol table and so on. These operations would check that the derived components are up to date, and automatically initiate recompilation if necessary.

With an object as complex as this, it can be difficult to decide on the best way to provide the required functionality. For example, a programing environment requires version management at both the program and module levels, but it is unclear how this should be split between the program entity and a separate version management utility. This sort of question can often only be answered after a process of experimentation. Unfortunately, the entity system described to date allows the programer one chance to get the class definition correct. If the interface is defined badly, the programmer has to live with the consequences. A method for resolving this problem is proposed in the next chapter.

3.4 Comparison with Other Languages and Systems.

The three best established examples of object based type systems in programing languages are Simula 67, Smalltalk and CLU. In Simula 67 [Birtwistle 73], an object type is called a *class*. A class declaration gives the external interface to an object and the implementation of the interface. The external interface includes all of the variables declared as *class parameters* and in the `class` body, so that Simula 67 classes do not fully hide the implementation of an abstraction from the client. A class declaration can be *prefixed* with another class so that the class being declared *inherits* the variables and actions of the *superclass*. This allows classes to be composed from other classes by the technique known as *data specialization*.

In Smalltalk [Goldberg 83], a `class` defines the *messages* an object accepts, the *instance variables* and the *methods* which implement the classes messages. Classes can inherit messages, instance variables and methods from a *superclass* in a similar way to Simula 67's class prefixing. The Smalltalk type system is polymorphic in that a message can be sent to any object whose class or

superclasses provides an appropriate method. Runtime type checking is used to ensure that an object has a method to deal with the messages it receives. Smalltalk implements operations on two objects by sending a message to the first object with the second object as a message argument.

In CLU [Liskov 81], a *cluster* defines the abstract interface, an object's concrete representation and the implementation of the operations defined by the interface. CLU uses *type parameterization* rather than subclassing as the means for constructing cluster types. The CLU type system is designed for compile time[4] checking and does not support polymorphism.

None of these languages support multiple implementations of a class. Indeed in both CLU and Simula 67, dyadic operations assume that both operand objects have the same representation. Smalltalk comes closest to supporting multiple implementations. It is possible to define new classes that are externally equivalent to existing classes in all but name. This is quite adequate for providing objects tuned for different purposes. However it is not ideal as the method of upgrading implementations for two reasons. First it is not transparent to the client, since the object's class will be different. Second it will tend to lead to a proliferation of classes.

The author postulates that the Smalltalk language could be extended to separate the protocol description from the implementation description and allow multiple implementations for a given class. It seems that this would not interfere with Smalltalk's way of handling dyadic operations.

Unlike Simula 67, Smalltalk and CLU, classes and implementations are defined separately in the entity system. Since a class can have multiple implementations, dyadic operations need to be implemented in the entity system either using the Smalltalk approach, or as implementation independent procedures provided by the class. The latter could be treated as library routines.

4 In practice, inter-cluster type checking is defered to the link edit phase in order to cope with type parameterisation. As a result, relinking a large program is very expensive.

The entity system does not support composition of classes using type parameters or subclassing. While such features are desirable in an object oriented programming language, adding type parameters or subclasses to the entity system would cause severe problems for clients written in non-object oriented languages. Besides, the entity system is intended to supplement existing programming languages in a particular area rather than replace them. The basic entity system (described in this chapter) does not support polymorphic types. However, operation projection described in the next chapter does provide a degree of polymorphism.

In many object oriented operating systems such as Eden [Almes 83] and CAP-3 [Herbert 78] [Pardoe 85], objects are automatically activated and passivated without the client program's knowledge. While this makes writing client programs much simpler, it has a number of disadvantages. Automatic activation means that the client cannot inform the object's implementation of the sort of operations that will be requested. Information of this nature has to be passed to the object in separate operation requests. Alternatively, the implementation has to make conservative assumptions about the expected uses of the activation, and initialize accordingly. The policy used in CAP-3 is to avoid unnecessary activation by activating the smallest possible unit of storage required for the immediate situation. The effect is that attempting to use a complex object results in a number of kernel operations to activate object components.

A second disadvantage is that automatic passivation requires a fast asynchronous garbage collector for object activations. The garbage collector needs to be able to passivate objects quickly before locks that they hold and other system resources they are tying down interfere with the smooth running of the system. A fast asynchronous garbage collector would be expensive, and could have the side effect of causing unnecessary object passivation and reactivation.

The entity system also requires a garbage collector, albeit for a different reason. When a client program terminates or an entity is passivated, there is a possibility that it might fail to close other entity handles which it possesses. The kernel can often detect this if it keeps track of the entity handles used by each program and implementation. However, there is a possibility of entity activations (directly or indirectly) holding handles for themselves, and an entity handle garbage collector is needed to deal with this problem. It can be important that such a loop of

handles is broken at the correct point. While no general scheme is available for finding the correct point, the heuristic of breaking the loop at the oldest handle is likely to be appropriate most of the time.

When tidying up entity handles, the kernel is not in a position to apply an entity's normal close operation since it cannot supply the class specific arguments. The solution is for each implementation to provide a generic *abort* operation procedure of the form.

```
{impl}.Abort(handle)
```

Such a procedure would be similar to the same implementation's close procedure except in respect of the class specific arguments. In the case of entities whose update semantics use the transaction approach, an abort procedure would roll-back the transaction in progress.

Another area that warrants discussion is the relationship between entity handles and activations. In earlier versions of the entity system, a distinct entity activation was used for different handles for the same entity. This was simple in that an entity implementation had to deal with a single handle only. However, since this meant that there was no sharing of temporary data between the handles for an entity, correct management of cached data was a problem. Furthermore, if these versions of the entity system had been multi-threaded it would have been very difficult for an implementation to synchronize concurrent operations on different handles.

Since all handles for an entity now use the same activation, the programer can potentially use the implementation language's concurrency primitives to synchronize entity operations. The implementation can use a shared activation for caching information common to all handles for an entity. Sharing an activation reduces the overhead of opening the second and subsequent handles, since the kernel does not need to create a new activation, and the implementation open procedure can avoid repeating a lot of the initialisation.

Chapter 4 - Projections and Perspectives.

4.1 Projections and Perspectives.

The main problem with the entity system as described in chapter 3 is the rigidity of the type scheme. For example, the timestamp display program in section 3.3 had to use a number of interfaces to access the timestamp values; one for free standing timestamps, another for the creation and update timestamps on a directory, a third for the corresponding directory entry timestamps and so on.

A similar problem arises when a class definition needs to be changed to correct some deficiency in the interface. After changing the class definition, the programmer must update all programs that use the class and some or all of the existing entity implementations. In addition, it may also be necessary to convert the representations of existing entities with the old version of the class.

This degree of rigidity in the type system is acceptable in a small self contained program where changing an interface affects that program and nothing more. It is a different matter with large systems and when persistent objects are involved. A minor change to (say) a client's interface to a conventional operating system often requires that a large number of applications are recompiled. A change to an interface provided for a class of persistent objects is likely to involve data conversion as well as program conversion. To get around problems of type rigidity, the entity system provides a dynamic operation coercion scheme called *projection*[1]. This section describes projection and presents some illustrative examples.

In section 3.2, an entity was described as being characterized by 3 properties; the class, the implementation, and the representation. The class that is a property of an entity is better described as the *base class* or *natural class* of the entity. Now we introduce the concept of the perspective class through which the client views the entity. Previous descriptions assumed that the perspective class used by the

1 The idea of a projection mechanism for the entity system was originally proposed in [Wilkes 81]. While the mechanism proposed there was simple operation renaming which is akin to projection in the database sense, entity system projection has evolved into something considerably more powerful.

client is the same as the natural class of the entity. When this is not the case, operations which the client requests from the point of view of the perspective class are *projected* into operations for the base class.

The mechanism for projecting perspective class operations into base class operations is set up when the client program opens an entity. The client invokes the open operation supplied by the perspective class and the perspective class identifier is passed as an argument to the **Kernel.Open_Entity** call:

```
handle ← Kernel.Open_Entity(entity_id,
                persp_class_id, arg_ptr)
```

If **persp_class_id** differs from the base class identifier in the entity triple, the kernel goes through a process known as *projection resolution*. If resolution succeeds, the handle returned by the **Open_Entity** call is a handle for the perspective class which allows the client to invoke perspective operations. These invocations are projected into the appropriate base class operations on the underlying entity.

Projection is intended to be transparent to the client program. For instance, the timestamp display program should not need to distinguish between free standing timestamp entities, and timestamps in directory, program or any other entities. The program opens them all with the **Time_Stamp.Open** operation and sees a **Time_Stamp** handle which provides a **Read_Stamp** operation.

The restrictions on projections are as follows:

- It must be possible to resolve the projection at open time using the arguments supplied in the open call.
- The semantics of an operation in perspective space must be compatible with the semantics of the corresponding natural operation.
- A projection must be independent of the implementation of the natural class. A projection of a base class as a perspective class must work for all implementations of the base class.

4.2 Mechanisms for Projection and Projection Resolution.

In the previous section, the concepts of projection and projection resolution were outlined. In this section, the basic projection mechanism is described. Various projection resolution techniques are presented and discussed with some simple illustrations.

The projection resolution mechanism is built around routines called *resolution functions*. A resolution function has the following general form.

$$\texttt{handle} \leftarrow \texttt{resolution\_fn(proto\_handle, } \texttt{arg}_1, \ldots \texttt{ arg}_n)$$

Proto_handle is a collection of information about the base entity being opened, and the perspective which the client has asked for. When the time comes, this information will be sealed to form the handle. The class specific open arguments are passed in $\mathbf{arg}_1$ through $\mathbf{arg}_n$.

A typical resolution function first decides whether it is capable of resolving the projection given the client's open arguments. If it is, it then builds an operation table for the handle using operations procedures from the entity's implementation and from the resolution function itself. At some point the entity's open operation procedure is called. Finally the resolution function calls the kernel to transform the proto-handle into a full handle for the perspective class. This handle is returned to the client.

The components of a *proto-handle* include the perspective class identifier, and information about the immediate entity such as its class and implementation identifiers. At various stages in the resolution process it may also contain a pointer to the entity activation, a template operation table supplied by the implementation, and the operation table which the resolution function is building. The information that constitutes a proto-handle is heavily dependent on the way that the entity system is implemented.

A given resolution function may well fail to produce a suitable projection. When this occurs, the entity system kernel tries other resolution functions in turn until

one of them succeeds. If all possible candidate resolution functions are tried and none of them succeed, the client's open request fails.

The need for using the more complicated method of passing open operation arguments should be apparent by now. A resolution function needs access to the class specific open arguments, yet it must be called from the kernel. In languages like Modula-2, CLU and Mesa, the only reasonably type safe way to pass generic arguments through the kernel is to assemble them in a record and pass the address of the record. Thus a resolution function actually looks more like the following

```
handle ← resolution_fn(proto_handle, arg_rec_ptr)
```

where **arg_rec_ptr** is the pointer to the record of open arguments passed to **Kernel.Open_Entity**.

A number of techniques are available to the programmer writing resolution functions. The simplest of these is *operation substitution* or *renaming*. Suppose we define an **Option** class with three class specific operations

```
handle$Set_Option(string)

string ← handle$Get_Option()

time ← handle$Time_Set()
```

The semantics of these operations are obvious. **Get_Option** and **Set_Option** respectively read and update the option value, and Time_Set returns the time at which the option was last updated.

The formal argument and result types and the operation semantics for the **Option.Time_Set** and **Time_Stamp.Read_Stamp** operations are compatible. The projection of an option entity as a (read-only) timestamp conceptually renames **Time_Set** as **Read_Stamp**. In practice, the resolution function substitutes the **Time_Set** operation procedure from the **Option** entity's implementation for the **Read_Stamp** operation in the operation table being built. Thus whenever the client program invokes the operation it views as **Read_Stamp**, it is the **Time_Set** procedure that is actually called.

In a given situation, there may be many possible mappings from the client's perspective class onto the natural class. For instance, suppose that the option class also provides the operation

```
time ← handle$Time_Created()
```

which returns the time when the option entity was created. An open invocation of the form

```
handle ← Kernel.Open_Entity(option_entity_id,
                Time_Stamp.class_id)
```

does not provide sufficient information for resolving the projection. The client could be referring to either of two different timestamps within the option entity. To get around this problem, the perspective class open operation can be defined to take a *selector string* argument which is used to select between alternative projections.

```
handle ← {persp_classname}.Open(entity_id,
        selector_string, arg1,... argn)
```

The extra **selector_string** argument is also present in the arguments for the corresponding perspective open operation procedure. Thus an option entity is opened as a timestamp using one of the following invocations.

```
handle ← Time_Stamp.Open(option_entity_id, "created")

handle ← Time_Stamp.Open(option_entity_id, "set")
```

The resolution function would assemble an entity handle with either **Time_Created** or **Time_Set** substituted for the **Read_Stamp** operation depending on the selector string.

A resolution function for projecting options as timestamps is given in figure 4-1. This illustrates both substitution and the use of a selector string. The resolution function checks that the client has not asked for write access to the timestamp. To prevent attempts to write the timestamp, the resolution function substitutes an error trap for the **Take_Stamp** operation. Then it substitutes the appropriate operation procedure from the option entity implementation for **Read_Stamp**, and the **Close_proc** operation procedure for timestamp **Close** operation.

```
/* Open argument record type defined by class */
Time_Stamp.Open_Args: TYPE = RECORD [
     selector:    String
     mode:        {Read_Only, Read_Write}]

FUNCTION Project_Opt_As_TS(p: Proto_Handle,
        args: POINTER TO Time_Stamp.Open_Args
     ): Time_Stamp.HandleType
   BEGIN
   IF args↑.mode ≠ Read_Only THEN
     /* The Take_Stamp operation cannot be projected */
     EXCEPTION(Projection_Failed,... )
   ENDIF
   /* Substitute Error_Trap for Time_Stamp.Take_Stamp */
   IF args↑.selector = "created" THEN
     /* Substitute Base_Impl.Time_Created_proc
        for Time_Stamp.Read_Stamp */
   ELSIF args↑.selector = "set" THEN
     /* Substitute Base_Impl.Time_Set_proc
        for Time_Stamp.Read_Stamp */
   ELSE
     EXCEPTION(Projection_Failed,... )
   ENDIF
   /* Substitute Base_Impl.Close_proc
     for Time_Stamp.Close */
   RETURN (/* Seal p as Time_Stamp handle */)
   END Project_Opt_As_TS
```

figure 4-1

The third projection resolution technique is the use of *projection operations*. A projection operation is an operation defined by the natural class of an entity which is applied during projection resolution. It allows the resolution function to make use of information stored in the entity, without introducing any dependencies on the entity implementation. The arguments and results of a projection operation are defined by the class as for other operations. In principle, there is nothing to stop projection operations being invoked directly by the client.

As an example of the use of projection operations, suppose that a text editor attempts to access one of the text components of a program entity, using a

selector string to give the name of the component. Some components have fixed names like **"linker-directives"** and **"modification-log"** that can be built into the resolution function. Others like the source code components, will have names that can only be determined by accessing the program entity being opened. In the latter case, the resolution function needs to use a projection operation to find the correct component.

Suppose that the program class provides a set of operations compatible with the text entity operations. A further program operation is of the following form

```
success ← handle$Select_Text_Component(name)
```

selects the component to which the program / text operations refer. A resolution function for resolving a program as a text object could be written as in figure 4-2.

```
FUNCTION Project_Prog_As_Text(p: Proto_Handle,
     args: POINTER TO Text.Open_Args): Text.Handle
BEGIN
   handle:    Text.Handle
   /* Substitute Base_Impl.Read_Text_proc for Text$Read */
   ...
   /* Substitute Base_Impl.Close_proc for Text$Close */
   handle ← /* Seal p as Text_Handle */
   IF Base_Impl.Select_Text_Component(
         handle, args↑.selector) THEN
      RETURN handle
   ELSE
      handle$Close(...)
      EXCEPTION(Projection_Failed)
   ENDIF
END Project_Prog_As_Text
```

figure 4-2

The resolution function makes a text handle, substituting program operations for text ones as necessary. Then it calls the **Select_Text_Component** procedure to switch to the appropriate text component[2]. If this succeeds, the text handle is

2 There is a possible problem with typing here. The resolution function has a text handle but `Select_Text_Component` is an operation on a program handle. If this matters, the resolution function should assemble a program handle as well.

returned to the client otherwise the text handle is closed, and the resolution fails.

The representation of the text components of a program entity is implementation dependent (naturally). Components such as compilation directives may be held in the entity's storage object, while components such as the source code may be represented as free standing text entities. In the former case, the implementation of the **Select_Text_Component** operation might cache the text component in the activation ready for the text manipulation operations. In the latter case, the implementation for **Select_Text_Component** would open the relevant subentity, and the text manipulation procedures would degenerate into operation requests on the subentity handle. This suggests an alternative projector operation which returns a completed text handle as follows.

```
text_handle ← prog_handle$Open_Text_Component(name, ...)
```

The final technique for projection resolution is *operation filtering*. Unlike the previous techniques, filtering incurs significant overheads whenever an operation is invoked. Supporting operation filtering also makes the kernel more complicated. On the other hand, this technique allows the entity system to provide facilities that would be very hard to reproduce in more conventional file systems and in other object based systems.

In the last chapter, a class of directory entities was described. Consider the problem of projecting such a directory entity as a timestamp. While the operations **Directory_Info** and **Examine_Entry** provide access to a directory's timestamps, these operations are not type compatible with the timestamp operation **Read_Stamp**. This means that the operation substitution technique is not directly applicable. The resolution function could apply **Directory_Info** or **Examine_Entry** as a projector operation, but none of the mechanisms discussed so far will return the time value in response to a client's **Read_Stamp** request. Besides, this approach is incorrect, since it would give the values of the timestamps at the time the directory was opened rather than the time at which the **Read_Stamp** request was made.

The solution is for the resolution function to substitute one of its own local procedures for the **Read_Stamp** operation. It then opens the directory entity as a directory, and stores the handle in the *projection activation* along with other information supplied at open time. When the client later invokes the

Read_Stamp operation, the substituted procedure invokes the **Directory_Info** or **Examine_Entry** operation on the directory. The timestamp value is then returned to the client.

Given that the operation which returns a directory's creation and last update timestamps has the form

```
current_perms, creation_time, update_time ←
        dir_handle$Directory_Info()
```

and the corresponding timestamp operation is,

```
time ← ts_handle$Read_Stamp()
```

a resolution function capable of projecting just a directory's creation and update dates is illustrated in figure 4-3.

Operation filtering relies on being able to use a projection activation to store state for the handle after the resolution function has returned. In the above example, this state includes the global variables **dir_handle** and **sel_create**. Unlike entity activations, there is a separate projection activation for each entity handle. A projection activation is created every time a resolution function is called by the kernel. If the resolution function fails, or if it does not substitute any of its local procedures, the activation is released on return to the kernel. Otherwise, the projection activation is kept until the entity handle is closed.

The motivation for shared entity activations does not apply to projection activations. Cached data is best held in the underlying entity activation rather than the projection activation, since the best caching scheme will depend on the entity implementation. Furthermore, since the entity could simultaneously be accessed by other routes, there is a risk that the cache will get out of date. Synchronization of operations on projected handles has to be carried out in the underlying entities for the same reason.

```
/* Resol'n function globals (the projection activation) */
dir_handle:    Directory.Handle
sel_create:    BOOLEAN

FUNCTION Dir_As_TS(p: Proto_Handle,
        args: POINTER TO Time_Stamp.Open_Args
     ): Time_Stamp.Handle
BEGIN
   ...
   IF args↑.selector = "created" THEN
      sel_create ← TRUE
   ELSIF args↑.select = "updated" THEN
      sel_create ← FALSE
   ELSE
      EXCEPTION(Projection_Failed,... )
   ENDIF
   dir_handle ← /* Seal p as Directory.Handle */
   /* Substitute Filter for Time_Stamp$Read_Stamp */
   ...
   /* Substitute Close for Time_Stamp$Close */
   RETURN /* Seal p as Time_Stamp.Handle */
END Project_Dir_As_TS


FUNCTION Filter(ts_hand: Time_Stamp.Handle): Time
BEGIN
      create, update: Time
   VOID, create, update ← dir_handle$Directory_Info()
   IF sel_create THEN
      RETURN create
   ELSE
      RETURN update
   ENDIF
END Filter


PROCEDURE Close(ts_hand: Time_Stamp.Handle)
BEGIN
   dir_handle$Close()
END Close
```

figure 4-3

4.3 Projection Ambiguity.

The examples presented so far have been single level projections, requiring only one resolution function. The power of the projection mechanism does not become apparent until more complicated examples are considered. This section presents such an example, and discusses a serious problem with projection which the example brings to light.

Objects in most filing systems can be referred to by a textual *name*. The syntax and semantics of object names vary considerably, reflecting differences in the filing system structure and the designer's personal tastes. In the typical filing system, directories hold object name information in parallel with structural information. In an entity system, the structural information can be represented by entities of many classes. Furthermore, entity projection can produce structural viewpoints that are not explicitly defined by a base class; for example, a program entity can also be viewed as a collection of text objects. Object naming in the entity system needs to reflect both the natural and projected structural viewpoints.

A name can be associated with any visible component of the entity system, whether it is a complete entity or a component of an entity. The names of some components such as directory timestamps are bound into the code of resolution functions or entity implementations. Others, such as directory entry names and program source component names, are derived from data stored in entities.

Name to component mapping can be performed as part of the projection resolution process. A component name is passed as the selector string in the entity open operation invocation. A resolution function may interpret the name immediately, possibly with the help of projector operations. Alternatively it can save the component name in the projection activation for use in filter operations.

A sequence of component names can be thought of as a path through the entity filing system. The same sequence of names concatenated with separators into a single string forms a *pathname*. A resolution function handles a pathname by

stripping off and interpreting the first component, and recursively calling the kernel entity open routine with the rest of the pathname as selector string.

The sample resolution function in figure 4-4 can handle a single step in a multi-

```
FUNCTION Dir_As_Anything(p: Proto_Handle,
     args: POINTER TO Any_Open_Args): Any_Handle
   BEGIN
      head, tail:    STRING
      dir_handle:    Directory.Handle
      handle:        Any_Handle

      head, tail ← Strip_Off_Head(args↑.selector)
      dir_handle ← /* Seal p as Directory */
      USE HANDLER Dir_Error_Trap
      handle ← dir_handle$Open_Entry(
           head, p$Persp_Class_Id, tail,... )
      dir_handle$Close(...)
      RETURN handle
   END Dir_As_Anything;

HANDLER Dir_Error_Trap(exception: EXCEPTION)
   BEGIN
      IF exception = Directory_Error(Entry_Not_Found) THEN
         dir_handle$Close(...)
         RAISE Projection_Failed(...)
      ELSE
         RESIGNAL
      ENDIF
   END Dir_Error_Trap
```

figure 4-4

stage projection; the step being the projection of a directory as a directory entry. The resolution function splits the `selector` string at the leftmost component separator, then invokes the directory operation **`Open_Entry`**, using the head component as a directory entry name, and passing the rest as the selector string for **`Open_Entry`**. If the implementation for the directory entity fails to find the named entry, it raises the exception **`Entry_Not_Found`**. The exception handler catches this, tidies up the directory handle, and signals that the resolution function has failed.

A sketch of an implementation of the **Open_Entry** operation is given in figure 4-5 The implementation procedure searches for an entry with the required name.

```
TYPE Directory_Entry = RECORD [
        entity_id: Entity_Id
        ...
     ]

PROCEDURE Open_Entry(
     dir_hand: Directory_Handle, entry_name: STRING,
     persp_id: Class_Id, selector: STRING,... ): Any_Handle
  BEGIN
     entry: Directory_Entry
     entry ← Find_Entry(entry_name)
     ...
     RETURN Open_Entity(
          entry.entity_id, perspective, selector,... )
  END OpenEntry
```

figure 4-5

After checking permissions for the entry, it calls the kernel routine to open the entity, passing through the client's perspective and (in this case) the tail string from the resolution function. If the tail string is null and the perspective id matches the natural class id, the kernel will perform a simple open. Otherwise, the kernel starts on the next stage of the resolution.

It is not always immediately obvious how a projection should be resolved. Consider the following sample filing system structure in figure 4-6. Suppose that a client program tries to open the directory entity **x** as a timestamp handle. There are a large number of selector strings that the client could supply; **"created"**, **"A"**, **"A.inserted"**, **"B.updated"**, **"B.C"** and **"B.C.updated"** are just a few of the possibilities. Given just the perspective class and selector string, the kernel cannot determine à priori the sequence of resolution functions that will successfully resolve the projection.

All resolution functions are specific to one base class and a subset of the perspective classes. This limits the number of possible resolution functions at each stage. The kernel's resolution routine tries each of these functions in turn until it finds

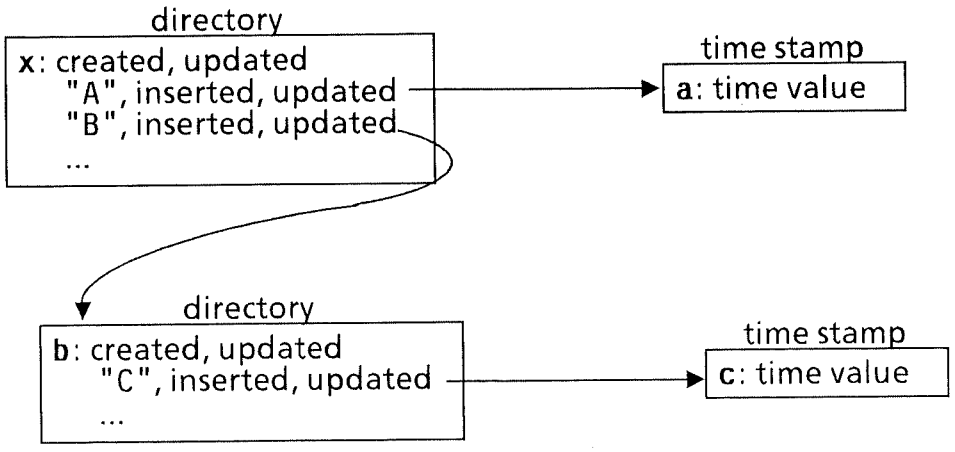

figure 4-6

one which succeeds. By using this strategy at each stage, the kernel performs a depth first scan of the tree of possible projections until one that works is found.

In the example above, the only applicable resolution functions for the first stage are **Dir_As_TS** for the directory's created and updated stamps, **Dir_As_Entry_TS** for the inserted and updated stamps associated with a given directory entry, and **Dir_As_Anything** for free standing timestamps entities entered in the directory. If the client supplies the selector string **"B.C.updated"**, the projection search tree might look something like figure 4-7.

The problem with this scheme is that it allows ambiguous names. In the sample directory structure, **"B.updated"** could refer to either the **"updated"** stamp of directory **b** or the **"updated"** stamp of the **"B"** entry in directory **x**. Further ambiguities arise if a timestamp entity is inserted into one of the directories with the name **"created"**, **"updated"** or **"inserted"**.

The naming ambiguity reflects inconsistencies between the base class and the different projections. The static component name **"updated"** is used in two places to mean slightly different things. It can be avoided by changing the names of the entry timestamps to (say) **"entry-inserted"** and **"entry-updated"**. The other problem is that directory entry names can clash with static component names defined by resolution functions. A somewhat unsatisfactory way of

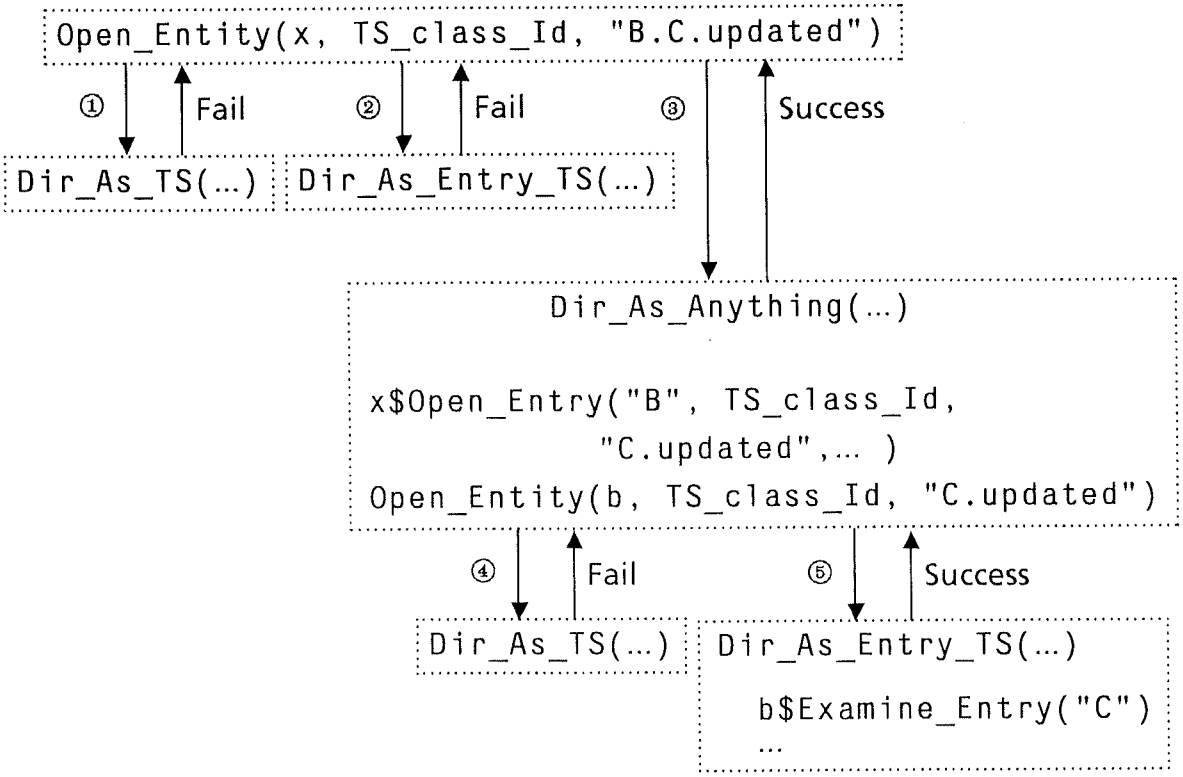

figure 4-7

removing this ambiguity would be to choose static component names which do not conform with the syntax rules for directory entry names.

This is just an example of the general problem of projection ambiguity. In a given situation, there may be a number of different possible resolutions for a given projection, each giving an entity handle with different semantics. Without extra information, there is no way that the kernel can tell which is the correct projection to use. It would seem that ambiguity problem is inherent to this form of type coercion. In any case, some way is needed to avoid practical ambiguity problems.

The client program may be in a position to specify which resolution function is to be used at a given point. If so, it needs to pass the name of the resolution function to the kernel. One option is to encode the name in the selector string using an escape sequence. If the kernel finds an escaped resolution function name at the beginning of the selector string it applies the function without searching for any other. Thus in the example above, a selector string of **"!Dir_As_Entry_TS!B.updated"** would unambiguously give the timestamp for the last update of the entry **"B"**, and **"!Dir_As_Anything!B.updated"** would give the timestamp

for the last change to the directory **b**. This scheme relies on the kernel understanding the selector string which implies that it is a general rather than class specific open argument.

In the absence of any other information, the order in which possible resolution functions are tried in an ambiguous case determines the projection which results. Thus, another way that a client can affect the resolution process is by altering the order in which resolution functions are tried.

Resolution function ordering information is held in a data structure called a *projection index*. A projection index holds an ordered list of resolution functions for each perspective / base class combination. At each stage in the resolution process, the kernel finds the appropriate list and tries the functions in the given order until one of them succeeds. If the list is empty, or if none of the resolution functions succeeds, the resolution fails. In a multi-stage resolution, the kernel backs up to the previous stage and tries the next alternative.

The order in which the kernel tries resolution functions also affects the efficiency of projection resolution. Ideally, the correct resolution function will be the first one tried in the majority of cases. Otherwise, a lot of time could be wasted applying the wrong resolution functions.

The previous discussion assumes that each stage is carried out by separate resolution function calls. This need not be so. A resolution function can also handle a number of alternative projections or a sequence of projection stages. In the timestamp example, `Dir_As_TS` and `Dir_As_Entry_TS` can be written as a single resolution function which handles both cases. Similarly, the `Dir_As_Anything` resolution function could be rewritten to follow the path through a number of directory entries in one application.

Bundling resolution functions avoids the kernel calling resolution functions that cannot possibly succeed, and reduces the number of resolution function calls in general. Since setting up a resolution function call can be expensive, such optimizations may be well worth doing. The disadvantage is that the client loses control of the resolution process. When a number of stages in the resolution process are handled by a single function, the client cannot get around ambiguity

problems by altering the order of resolution functions in the projection index or by specifying the resolution function by name.

Chapter 5 - An Experimental Entity System.

This chapter describes an experimental implementation of the entity system. The purpose of the system was to try out the entity system concepts. Though some of the concepts outlined in chapters 3 and 4 have progressed since work stopped on the experimental entity system, most of the key areas were implemented.

5.1 The Scope of the Experimental System.

An initial prototype of the entity system was implemented in a few months in BCPL for LSI-4 computers in the Cambridge Processor Bank [Needham 82]. It ran under the TRIPOS operating system, using the Cambridge File Server to store entity representations. Something close to a basic entity system without projections was implemented along with some sample classes and command programs. However, the prototype ran into address space problems, and became too complicated to debug using the tools provided by TRIPOS.

A number of lessons were learned from the LSI-4 prototype. The most obvious ones were the need for a typed programming language, a stable operating system substrate and symbolic debugging. It was clear that a full scale entity system would need a large address space. Finally there was a need for a fast interface to the permanent storage medium.

The experimental entity system runs in a single user process on a VAX 11/750 computer under the 4.1bsd and 4.2bsd versions of UNIX. Passive entities and other entity system state is held in a simulated disc file store. The file store is implemented on top of a file in the UNIX file system and is accessible from UNIX for debugging purposes. UNIX is used for editing, compiling and linking, and when the entity system is running, it provides a stable substrate from which to run debuggers. The benefits of a powerful and stable support environment cannot be over emphasized.

The experimental entity system is mostly coded in Modula-2 [Wirth 80], with a little VAX assembler code. Modula-2 is a simple strongly typed language suitable for operating systems work. Its main advantages over languages such as C and

BCPL are that it provides both strict type checking, and modular separate compilation facilities. CLU [Liskov 81] has proper support for object based programming, but suffers from the disadvantage of requiring garbage collection. Besides, an object based file system implemented in a non-object based language can be generalized to an object based language while the reverse is not necessarily so. At the time the decision was made, a significant point in favor of Modula-2 was that the language was actively supported at the Computer Laboratory.

The experimental entity system was implemented with the following limitations:

- it runs as a single UNIX process in a single address space,
- there is provision for only one client process, and
- only one programming language (Modula-2) is supported.

This a consequence of the fact that there was limited time available, and that the system needed to be simple enough to allow easy experimentation with the kernel and its interfaces.

The experimental entity system does not satisfy many of the requirements mentioned in section 1.3. The class definition machinery needed to fully satisfy the type safety and extensibility requirements 1) and 2) was never completed. The security requirement 4) is not satisfied as a consequence of the entity system kernel, entity implementations and the client program all running in a single UNIX process. The multiple client language requirement 5) is simply not addressed. Finally, requirement 6) is not satisfied as a consequence of the above.

Limiting the scope of the system avoids many problems. The kernel and entity implementations do not normally have to worry about synchronization of concurrent requests. Since there is only one addressing domain and one programming language, parameter passing is considerably simplified. However, in its final form the system tries to simulate firewalls between the kernel, entity implementations and the client's code, and hence to show that an entity system can be implemented in an environment where the firewalls exist.

The remainder of this chapter describes the state of the experimental entity system when development ceased. Section 2 describes the permanent storage

interface and its implementation. Sections 3 and 4 describe the basic entity system kernel and the projection mechanism. Section 5 presents some of the programs and classes written to exercise the experimental kernel. The final section presents a number of lessons that were learned from the experimental implementation.

5.2 A Standard Storage Interface.

There are advantages in having a universal storage interface which all entity implementations use for long-term storage. In the experimental entity system an attempt was made to define such an interface. The interface is based largely on the Cambridge File Server (CFS), [Dion 80] [Dion 81] with a number of changes in the light of experience with the file server itself, and with the LSI-4 prototype entity system.

The *Standard Storage Interface* (or *SSI*) models information as being either links or ordinary data. Ordinary data is treated as uninterpreted 8 bit bytes. An SSI *link* is a unique identifier encoded as a 128 bit number. It contains a *link type* field which gives the meaning of the link, and other fields depending on the type. Four link types (see figure 5-1) are defined by the SSI: storage object identifiers, class

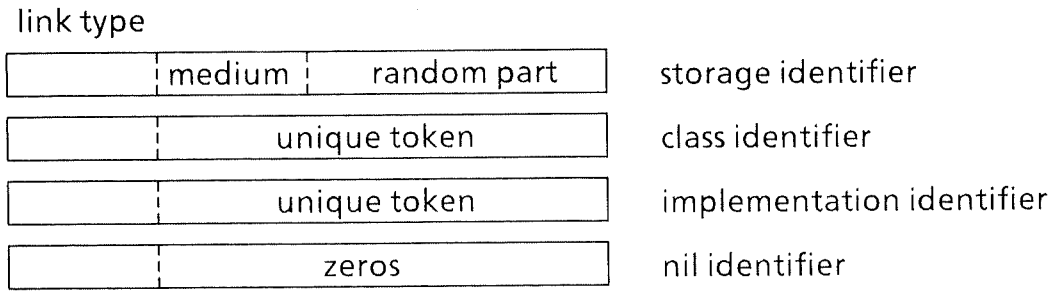

figure 5-1

identifiers, implementation identifiers and the null link. A storage object link has a function similar but not identical to that of a **PUID**[1] in the CFS model. In particular, the SSI does not guarantee that a storage object identifier is "permanently unique", and therefore it is incorrect in this system for an SSI client to hold identifiers outside of the SSI. This makes it possible to archive at the level

1 In the CFS, a PUID which stands for Permanent Unique Identifier is the identifier for a file or index. Clients are allowed, and in some cases expected, to bind PUIDs into their code.

of the storage object. As far as the SSI is concerned, class and implementation identifiers are opaque tokens.

A *storage object* is the vehicle the SSI uses for the permanent storage of link and data information. A storage object (see figure 5-2) is modeled as a pair of vectors,

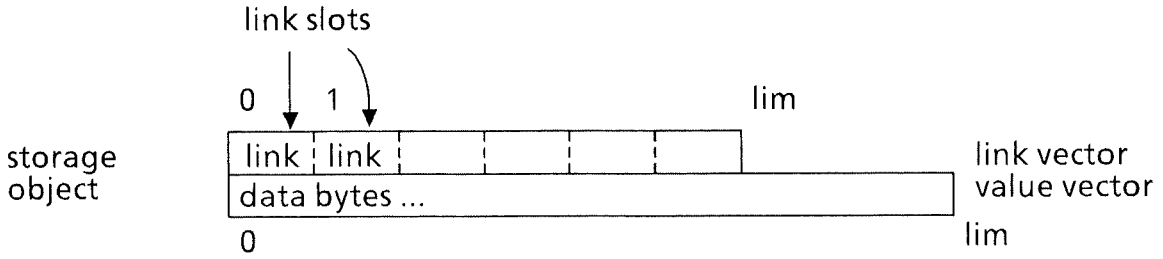

figure 5-2

one of bytes of data and the other of links. The links and bytes of data in the vectors are indexed from zero up to a bound set by the client. Either or both of the vectors can be empty. Combining the functions of CFS index and file in an SSI storage object was motivated by the observation that CFS applications which use indices for structuring typically have a file in parallel with each index to store associated information.

Operations on a storage object require a *storage handle* as a parameter[2]. A storage handle is the client's capability to use a storage object, and conveys both the subrange of the storage object that the client has access to, and the associated access rights. A handle for an entire object is obtained by applying the **SSOpen** operation to the object's link as found in the link vector of a "parent" storage object. A handle for a subvector of an object's link or data vectors (figure 5-3) is obtained by applying **SSRefine** to an existing handle. Operations on a refined handle refer to links and bytes relative to the start of the refinement. When a handle is no longer required, the client releases it with the **SSClose** operation.

2 An SSI storage object is not an entity, nor is a storage handle an entity handle.

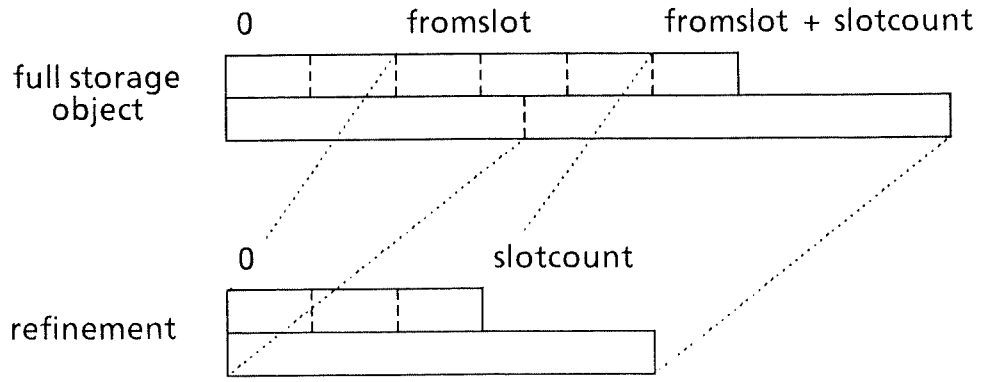

figure 5-3

The operations which deal with handles and access are as follows:

```
handle := SSOpen(parenthandle, slot, access)

refinedhandle := SSRefine(handle, frombyte, bytecount,
                 fromslot, slotcount, access)

SSChangeAccess(handle, access)

SSClose(handle)
```

The operations for transferring data and links between a client's address space and a storage object are as follows:

```
nbytes := SSGetBytes(handle, start, length, buffer)

nbytes := SSPutBytes(handle, buffer, start, length)

nlinks := SSGetLinks(handle, slot, linkbuffer)

SSGetLink(handle, slot, link)

SSPutLink(handle, link, slot)
```

The link and data vector sizes are read and updated by the following operations:

```
limit := SSFindLinkStoreLimit(handle)
```

```
limit := SSFindValStoreLimit(handle)

SSChangeLinkStoreLimit(handle, limit)

SSChangeValStoreLimit(handle, limit)
```

The SSI uses a similar mechanism to the CFS for deleting objects. An object is guaranteed to exist in the storage system so long as its link is held in the link store of an object that is accessible from the root of the file store. Therefore, when the **SSCreate** operation allocates a new storage object it stores the link in a "parent" storage object. The **SSDelete** which removes the link from a slot in a link vector may precipitate deletion. An object that has been removed from all link vectors, will continue to exist until the client(s) close the last handle.

```
SSCreate(parenthandle, slot)

SSDelete(parenthandle, slot)
```

As with the CFS, object deletion is controlled in the first instance by reference counts with potentially asynchronous garbage collection to pick up detached cyclic structures. This leads to a definition of object lifetime where the existence of an object may depend on the timing of a garbage collection. For example, suppose there is an object for which two copies of the link exist; one held in the root object and the other held in the object's own link store. A client reads the link into local memory, then removes it from the root object creating a degenerate detached cycle. The object is now in a state where its continued existence depends on whether the client uses **SSPutLink** to save the link in another link vector before a garbage collection occurs.

This non-determinism in the definition of an object's lifetime can be removed by replacing **SSPutLink** with an operation to copy a link from the link store of one storage object to another. Source and destination objects are guaranteed to exist because of the handles held by the client and therefore the object in question is guaranteed to exist. If a client has a handle for a detached cycle, all objects in the cycle are guaranteed to exist and the cycle can potentially be reattached. However, once the last handle has been closed, all objects in the cycle effectively cease to exist, since there is no way to reattach the cycle or open one of the objects.

In the entity system, the kernel needs to be informed when objects representing entities are going to be deleted so that the appropriate delete operation can be applied. The SSI therefore has the concept of a *cherished* object. When an cherished storage object is found to be inaccessible it is relinked into the link store of a distinguished storage object. The client (in this case the entity system kernel) can look in the distinguished object for deleted storage objects and deal with them as appropriate. As a safeguard, storage objects have the cherished bit removed when they are relinked. The **SSCherish** operation marks storage objects as cherished or uncherished, and **SSQueryHandle** examines the status of an object, including whether or not the object is cherished or relinked, and the numbers of links and handles which refer to it.

The SSI access control scheme is more flexible than the equivalent CFS scheme. There are three basic access modes **read**, **write** and **alter limit**, each of which controls a subset of the SSI operations. For each mode there are two access rights; the **access** right and the **lock** right. If a storage object handle has an **access** right, a client may use it to request any of the associated operations on the object or object refinement. If a handle has a **lock** right, the client is guaranteed that no other handles give the corresponding **access** right, and vice versa[3].

Though a client opening an object with **SSopen** can ask for whatever access and lock rights it needs, **SSRefine** and **SSChangeAccess** normally only allow rights to be removed. To prevent this being a problem, there are additional **access control** and **lock control** rights which allow the client to enhance a handle with enable and lock rights that it did not previously have. For example, a handle with lock control right allows the client to use **SSRefine** or **SSChangeAccess** gain new locks on the handle or a refinement. The **control** rights can never be regained once lost by a handle.

Separating access and locking allows the client to simulate most conventional forms of file and record locking. For example, single writer / multiple reader file locking can be simulated by all readers requesting **read access** and **write lock**, and writers requesting **write access** and **write lock**. To simulate record

3 These interlock rules have an interesting implication: if a handle with an **access** right is refined with the corresponding **lock** right (or vice versa), the original handle's right must be cancelled to maintain mutual exclusion. The alternative would be to fault the refinement.

locking, each client needs a handle for the full file with both **lock** and **access control**. A client locks a record by creating a refinement for the portion of the storage object which holds the record with the appropriate **access** and **lock** rights.

When a handle is produced by **SSOpen**, the client may request any rights on the object. This means that if the SSI access control scheme is to be the basis of file system protection, care must be taken to ensure that links do not "leak out". If **SSPutLink** were replaced with a copy operation, this would be less of a problem. Another thing the SSI access control mechanism does not support is waiting for locks to become available. Given that the experimental entity system only allowed one thread of control, it would have been pointless to include this facility.

The SSI as described above has been implemented in the experimental entity system. The implementation uses a conventional file in the host filing system as a virtual disc and simulates an implementation of a file system at the disc block level. Three stand-alone utilities were written for maintaining the standard storage system. The *Storage Medium Initializer* is a simple program which creates a new storage medium file. The *Storage Medium Editor* provides interactive commands for examining and changing a storage medium. The editor, which uses the SSI for all operations on the storage medium, is useful when looking for bugs in entity implementations. Finally the *Storage Medium Verifier* checks the integrity of a medium file at a number of levels, and interactively corrects inconsistencies. It has facilities for displaying virtual disc blocks and various summaries of the storage medium data structure, and includes a (synchronous) garbage collector.

5.3 The Basic Entity System.

The representation of an entity in the experimental system is an SSI storage object (figure 5-4). The first two link slots of the storage object contain the entity's class and implementation identifiers respectively. They are protected from interference by giving the client a refined storage handle which excludes the first two slots. Since opening a storage object in the SSI is an operation on a "parent" storage handle, the same is true for an entity. Therefore, a client of the kernel

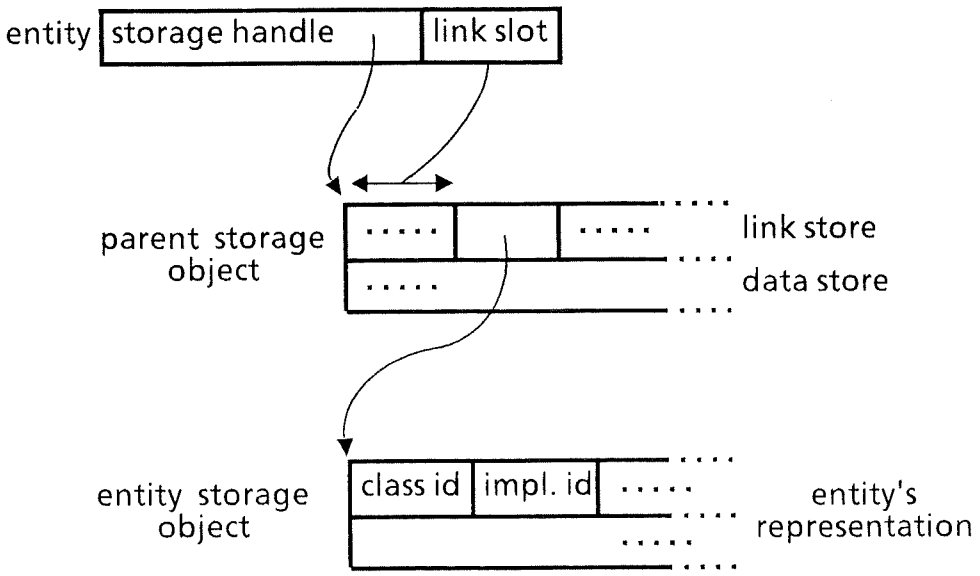

figure 5-4

refers to a passive entity using the type **Entity** which is a record containing a storage handle and a link slot number.

An entity handle is represented by the type **EntityHandle**. This is an opaque type which hides a pointer to an entity handle descriptor (figure 5-5) held in the kernel. The descriptor holds copies of the base class and implementation identifiers for the entity, the class identifier for the perspective under which it was opened, and the storage object handle passed to the client. The next, prev, parent and child fields are pointers to other entity handle descriptors used by the kernel when closing handles that the client has "dropped on the floor". There is also a pointer to an *operations table* for the entity handle, containing procedure variables for procedures in the entity implementation or resolution function which perform the class specific part of the operations.

Standard Modula-2 provides a single, statically linked instance of each **IMPLEMENTATION** module. The experimental entity system uses a dynamic module package to allow multiple instances of a module and dynamic module loading. This package is described in Appendix A. Entity implementations and resolution functions are dynamically loaded modules with a separate module instance for each activation. Dynamic loading avoids having to relink the entire system whenever an implementation or resolution function changes. As such it is

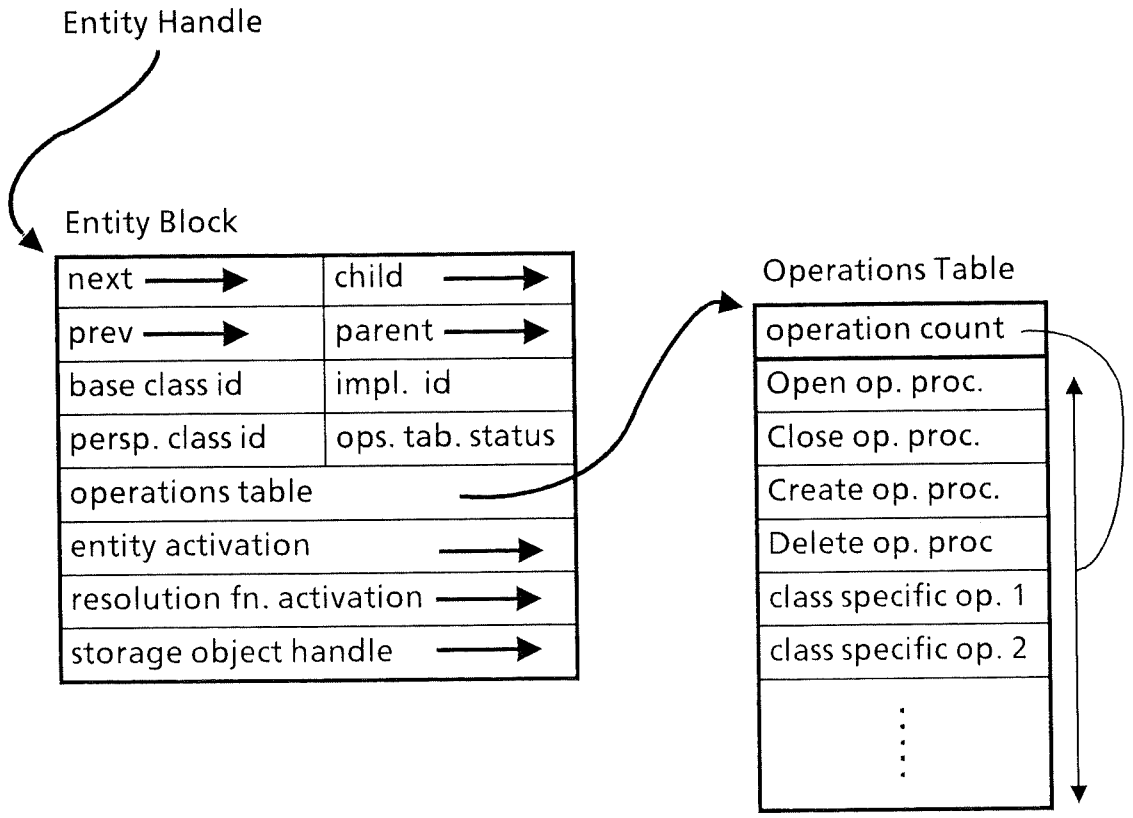

figure 5-5

essential to the entity system concept. The penalty is the cost of dynamic loading and relocation, but this can be offset by preloading and caching modules. A separate module instance is used for each activation to make entity implementations simpler and more object like. This is useful to the programmer, but is not necessary. Indeed, if entities were used to represent large numbers of small objects, the global frame table would overflow.

The first time the kernel gets a request for a given implementation, it loads the corresponding dynamic module and causes it to execute its initialisation code. The initialisation code fills a master operations table with procedure variables and calls back into the kernel passing a pointer to the table. The kernel sets up an implementation table entry (figure 5-6) to record the master module instance and operations table for the implementation and other information. The data structures for a projection activation are similar but simpler.

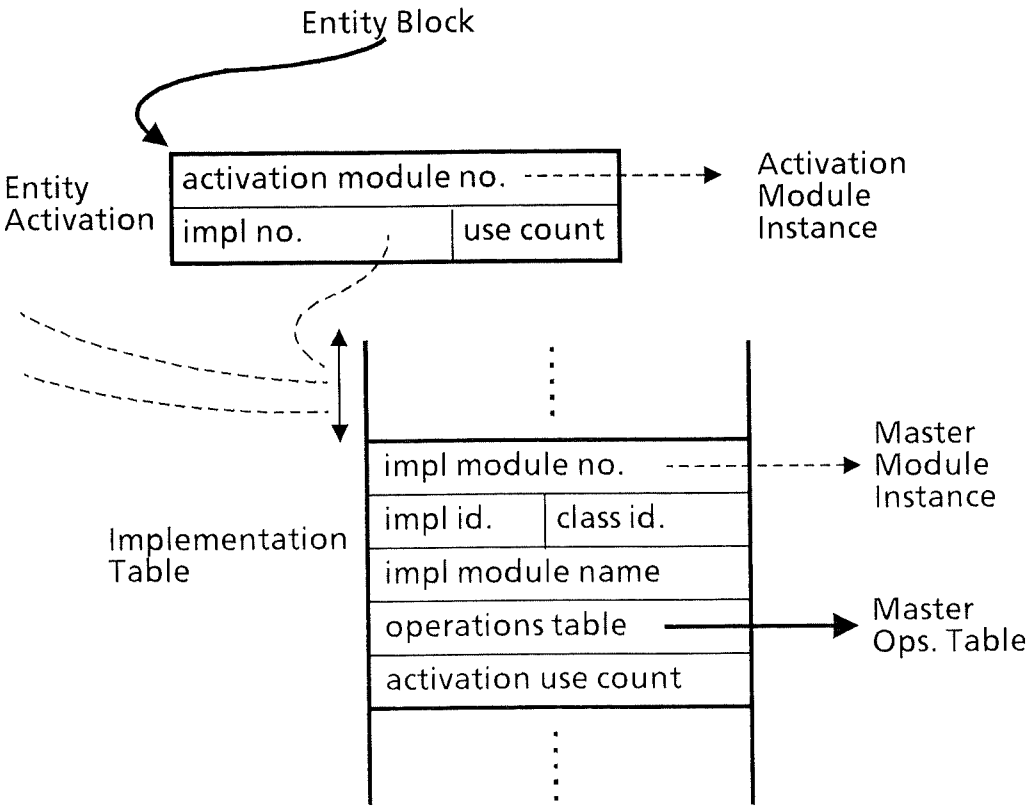

figure 5-6

In a full entity system, entity implementations and resolution functions would be stored within the entity system itself. This was never implemented in the experimental system. Instead, the dynamic module files are brought in from the Unix file system, and the binding between implementation identifiers and Unix filenames is compiled into the kernel. This avoids problems with creating a new entity file system and bootstrapping the system at start of day. A better approach to these problems is to bind the code for the crucial implementations into the kernel. Non-crucial implementations would be found using an implementation index.

When a new entity activation is required, a new instance of the implementation module is made and an entity activation descriptor is allocated. An operation table is produced for the handle, with procedure variables referring to the new instance rather than the master instance. For historical reasons, the experimental entity system does not support activation sharing between all handles for a given entity. The use count field in an activation descriptor is used (possibly spuriously)

when opening an entity to indicate whether or not a handle being assembled uses this activation of the base entity.

Since the entity kernel tries to simulate firewalls, the client is "forbidden" from breaking open a handle to get at the operations procedures. Instead, all entity operation requests are dispatched by the kernel. Passing an arbitrary collection of arguments and results in Modula-2 is a problem. One approach is to have the kernel return the operation procedure to be invoked by a stub operation procedure. This approach was not allowed in the experimental system because it involved passing a procedure from one notional address domain to another.

Other approaches are for the stub procedure either to copy the class specific parameters into a record whose address is passed, or to push them onto a stack. Both of these approaches have been tried in the experimental system; the latter using assembler coded stubs to save the parameters and to transfer control to the kernel. Both approaches work, but they are tedious to code by hand and are outside the protection of Modula-2 type system. Given time, these problems could have been solved using a translator for a class definition language.

Class specific entity operations are handled as follows. The client calls the stub operation procedure exported by the Modula-2 module which defines the class. This stub procedure takes the entity handle and appropriate operation specific parameters. It saves the parameters (in a record or on a special stack) and calls the kernel with the entity handle and an *operation index* as arguments. The kernel uses the operation number as an index into the handle's operation table to extract the operation procedure and calls it passing the entity handle and saved parameters. The mechanisms for passing class specific parameters to generic operations is similar, except that parameters passed to the kernel are different.

The kernel open routine is called with an `Entity` record, a selector string (possibly null), and the perspective class as arguments as well as any saved parameters. It first opens the entity storage object and extracts the class and implementation identifiers. The entity storage handle is then refined to exclude these slots. If the class identifier differs from the perspective class requested, or the selector string is non-null, the projection resolution mechanism described in the next section comes into play. Otherwise, the kernel finds the implementation module for the entity, and creates a new entity activation. The handle's operation table is made from

the master operation table, and all the pieces are assembled to give an entity handle. The open operation procedure is invoked passing the entity handle, refined storage handle and saved parameters and the kernel finally returns the entity handle to the client.

The kernel create routine allocates a new storage object in the slot in the parent object's link store and writes the class and implementation identifiers. Then it searches the table of loaded implementations for one which matches the implementation identifier and creates an activation and an entity handle. The implementation's create operation is invoked, and the entity handle and activation are freed. The kernel delete routine checks that the entity storage object is not referred to by some other part of the filing system. If this is the case, it makes a handle, applies the delete operation procedure, dismantles the handle and deletes the storage object. The kernel close routine is straightforward. The kernel extracts and applies the close operation procedure then dismantles the handle.

5.4 The Projection Mechanism.

The projection mechanism is driven by resolution functions called by the kernel. A resolution function typically looks at the head of the selector string, the base class and perspective identifiers and the permissions that the client has requested for the entity. If it is capable of doing so, it then directs the kernel in the assembly of an operations table using natural operations taken from the base implementation and the resolution function's own local procedures. The resolution function then instructs the kernel to seal the handle and it returns it to the kernel. If the projection is not possible, the resolution function returns the `NIL` pointer instead of a handle and the kernel may then try another resolution function.

A resolution function in the experimental entity system is a dynamically loaded module. Before a resolution function is invoked by the kernel, a dynamic instance of the module is created for the resolution function activation. This activation lasts at least until the resolution function exits. If the function fails to produce a handle, or if the handle does not use any resolution function procedures, the activation is released. Otherwise, the resolution function activation is attached to the entity handle descriptor.

Taking projection resolution into account, the algorithm for opening an entity is as follows:

1) Open the entity storage object and read the class and implementation identifiers.

2) If the client's perspective matches the base class, and if the selector string is null, the entity open process is completed by following steps.

   a) Make a new entity activation, loading the implementation if it is not already loaded.

   b) Make the implementation's operation table.

   c) Refine the entity storage object handle.

   d) Assemble the entity handle.

   e) Apply the open procedure, returning the completed entity handle.

3) Otherwise, if the head of the selector string is an "escaped" resolution function name, attempt to resolve the projection as follows.

   a) Make a resolution activation, loading the resolution function if it is not already loaded.

   b) Invoke the resolution function.

   c) If no operations from the resolution function were substituted, release the projection activation.

   d) If the resolution function failed, release the entity activation, and return an error indication.

   e) Otherwise return the entity handle produced

4) Otherwise, repeat the following steps until either the name lookup fails, or the resolution function succeeds.

   a) Lookup the name of the first (or next) resolution function for the current base class, perspective class pair.

   b) Try the resolution function as 3) above.

The projection interface that resolution functions use to direct the assembly of an entity handle consists of the 4 following kernel routines.

```
SubstituteImplementationOp(persp_op_no, base_op_no)
SubstituteResolutionOp(persp_op_no, resoln_proc)
entity_handle ← SealProjectedHandle(perms, open)
entity_handle ← SealNaturalHandle(perms)
```

When the resolution function is invoked, the perspective operation table is empty. The resolution function calls the first 2 functions in the projection to set up the table. **SubstituteImplementationOp** inserts the operation procedure at offset **base_op_no** in the natural implementation operation table into the perspective operation table at offset **persp_op_no**. **SubstituteResolution-Op** inserts the procedure **resoln_proc** from the resolution function module. In both cases, the kernel massages the procedure variables so that they refer to the appropriate activation module instances.

The remaining two routines are used to complete the assembly of entity handles. **SealProjectedHandle** completes an entity handle from the perspective operations table assembled using the **Substitute...** routines. This is typically the last statement of a resolution function. **SealNaturalHandle** returns an entity handle with a copy of the base implementation's operations table referring to a completely new activation. This handle can be used in filter operations.

The **Seal...** routines can apply the open operations for the handles they produce. With **SealNaturalHandle** this happens unconditionally, while the behavior of **SealProjectedHandle** is governed by two factors. If **open** is **TRUE**, the open operation is applied regardless. Otherwise, it will only be applied if the handle being produced has operation procedures from the implementation. Open arguments are passed using the mechanisms described previously. The resolution function typically has to build an additional set of open parameters when it uses **SealNaturalHandle**.

The kernel needs to recover cleanly from the failure of a resolution function so that alternative resolution functions can be tried. As part of the recovery process, activations which have not been bound to entity handles are deallocated, refer-

ence counts on implementation and resolution function modules are decremented, and saved open arguments and operations tables are discarded. The kernel does not close entity handles that were sealed as part of a failed resolution step, since it cannot tell if the resolution function has already passed them to some other part of the entity system.

The order in which resolution functions are tried is governed by the *projection index*. The projection index is an entity containing an ordered list of resolution functions for feasible combinations of perspective and base classes. To accommodate entity system startup and changes to the projection index, the kernel provides the following routines for enabling and disabling projection resolution;

```
EnableProjections(projection_index_handle, trace)
DisableProjections()
```

The **EnableProjections** routine allows the client to select between different projection indices. If **trace** is **TRUE**, the kernel displays a trace of the resolution process as an aid to debugging resolution functions.

5.5 A Model Filing System.

This section describes a filing system with a simple user interface implemented using the experimental entity system. The filing system includes a small number of implemented classes and projections, and some client *command programs* for manipulating various sorts of entities. The user interface to the filing system and the execution environment for the commands is provided by a primitive command line interpreter.

Six entity classes have been implemented for use in the filing system; time stamps, "byte files", directories, unique identifier generators, and projection and class indices.

Timestamp entities are trivially simple, providing operations for reading and setting a time/date value stored by the timestamp. Free standing timestamps are of little practical use in a running system, since useful timestamp values are

generally components of a larger object. However, timestamp entities have proved to be useful for testing an entity system under development.

Byte file entities store information to which no special meaning is attached. The byte file interface provides a set of operations similar to those available for a UNIX file; **`read`**, **`write`**, **`seek`**, **`tell`** and **`set_end_of_file`**. Additional operations allow the client to read time stamps storing the byte file entity's creation and last update times.

Directory entities form the backbone of the user filing system. A single directory holds a number of entries consisting of a textual name, access control information, and an entity identifier. The access control model provided by directory entities is based on the CAP model [Wilkes 79] using access matrices. Operations are provided for inserting, deleting, and examining entries, and for altering access matrices. The directory class and its problems were described in section 3.3.

Unique identifier generator entities provide a source of class and implementation identifiers when new classes and implementations are defined. A unique identifier generator provides a single operation **`generate_id`**. As with time stamp entities, the implementation of an identifier generator is trivial.

Projection index entities are used by the entity system kernel as part of the projection resolution process. A projection index appears to the outside world as a collection of lists of resolution function names. When the kernel needs to resolve a projection, it applies an operation to search the index for the first resolution function for a given combination of perspective and natural classes. Subsequent operation calls return further functions in the appropriate order. Operations are provided for examining an index and for inserting and deleting resolution functions. The kernel projection mechanism must be disabled to update to the current projection index, but this restriction could be removed.

Class index entities provide the mappings between class identifiers and textual names. Operations are provided for these mappings, for listing the index and for creating and deleting classes from the index. A new class is created by supplying a class name and the identifier generator handle which is used to generate a unique class identifier.

In addition to the above implemented classes, a *listable object* perspective has been defined, to allow entities to be displayed by a "universal" print program. A listable object handle provides one operation which is called repeatedly to produce a textual image of the underlying entity. Resolution functions have been written for time stamps, byte files and directories. In the latter two cases, the selector string supplied by the client is used to select different display formats. Projection resolution is used for following directory paths as explained in an example in section 4.3.

The user interface to the model filing system is a simple command line interpreter. When the command line interpreter starts up, it finds and opens the root directory and the global projection index. After setting up the user's environment, it accepts commands from the user terminal and dynamically loads and invokes command programs.

A total of 16 command programs were written for the model filing system. `Create` makes a new entity and inserts it into the filing system. `Alias`, `alter` and `delete` manipulate the directory structure, `examine` prints out a specific directory entry, and `home` and `set` allow the user to change the current directory and move around the filing system. Byte files are manipulated by the commands `input`, `type`, `import` and `export`. Timestamps can be printed and, in the free standing case, set using the `time` command. The universal print command called `list` can be used to look at byte files, directories and timestamps. Finally, the user can update class and projection indices using the interactive command `classedit`. The class editor provides subcommands for listing indices, and creating and deleting classes, and inserting and removing resolution functions.

5.6 Lessons from the Experimental Entity System.

In section 1.3, a set of 7 requirements for an entity system were listed. As noted at the start of this chapter, the experimental entity system was intended as a test bed for ideas rather than a production system, and several of these requirements were not addressed. However, the experience gained using the experimental system has lead to insight as to what could be achieved without difficulty in a full production system, and what problems remain.

Though it was a huge improvement over BCPL, Modula-2 was not an ideal language for implementing the experimental entity system. Modula-2 does not support exceptions, and while an exception handling package was written, it was too cumbersome in practice. Poor support for arrays with runtime bounds, generic data types and runtime type checking proved to be a problem. The absence of garbage collection made data structure management more difficult for kernel and client code alike.

While the dynamic module package proved adequate, it has intrinsic problems with interfacing to static code, and implementation problems with procedure variables and submodules. When an improved version of the Modula-2 compiler and debugger for the VAX was released, the dynamic module package could not be upgraded because the runtime module structures were too different, and we were forced to continue to use the old language tools. With hindsight it would have been better to use records to represent entity and projection activations.

The experimental system runs as a Unix process with a single logical address space. Therefore passing operands and results between the client and the entity implementation does not require transfer of data across hardware protection boundaries. While this simplifies matters, it means that many questions of parameter passing were not explored experimentally. Mechanisms have been proposed ([ISO 84b], [Herlihy 81] etc.) for passing complex data structures as arguments which appear to fit the bill. These issues are discussed in the next chapter.

The experimental entity cannot tidy up entity handles after client program failure. Indeed, the kernel did not always deal correctly with projection failures. Ideally, recovery of dropped handles would be integrated with the client's garbage collector. Handles dropped when a client process crashed or exited with open handles should be aborted by the kernel.

A colleague made some modifications to the experimental system kernel to try to solve the problem of dropped handles [Barman 84]. In his scheme, an entity handle potentially *owns* a number of other entity handles. When a handle is closed, the implementation's Close procedure is called, and the kernel closes all handles that the handle owns. When an entity handle is created, it is owned by a default owner handle set up by the client. In the command line interpreter for

example, a new default owner handle is set up each time a command program is run, and is closed afterwards. If an entity handle is to be passed outside of the scope of its owner, the client can call a kernel routine to transfer ownership to another handle.

The standard storage interface proved to be unsatisfactory for writing entity implementations. Though storage access at this level is necessary when speed and efficiency is paramount, too much time has to be spent designing file layouts and writing storage allocation code. A better approach is to define a range of storage interfaces for a variety of purposes.

One aspect of the design of the experimental entity system that is clearly wrong is the method used to represent passive entities. Storing class and implementation identifiers in the link store of the storage object gives rise to a number of problems. First, it means that entity implementations must use a separate storage object for every entity no matter how small. Second, every storage object representing an entity must have a link store vector allocated, irrespective of whether the implementation would otherwise need it. Finally, any client which knows an entity identifier could break open the entity using the storage interface.

A better alternative is for an entity identifier to be a token for an entity triple managed by the kernel (figure 5-7). The kernel can make sure that an entity's

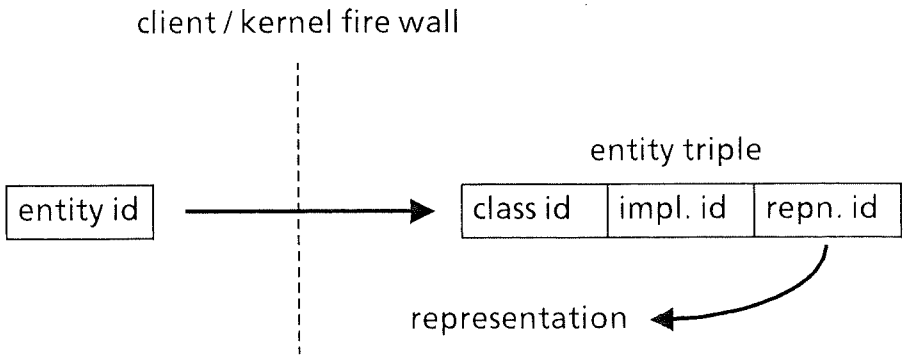

figure 5-7

representation identifier is only ever given to a bona fide activation of the entity implementation. It can also guarantee that class and implementation identifiers are never changed. Releasing an entity identifier to a client would no longer risk the integrity of the entity's representation, and would reduce the security risk. Other implications of this alternative are discussed in section 6.5.

Another of the problem areas in the experimental entity system is in the control of permissions to use entities. Irrespective of whether or not it is possible to crack open an entity as described above, possession of an entity identifier allows a client to open an entity with any permissions. If protection and privacy are required, either the implementation must validate the client using some independent means, or the client must not be given the entity identifier in the first place. The former approach means that the access control rules must be independent of the route the client took to get to an entity. This makes access control matrices in the style of CAP [Wilkes 79] impossible to implement. The latter approach means that an object holding an entity identifier must perform the open on the client's behalf. If the entity can be of any class, as is the case with a directory entity, the type of the operation which opens the subentity for the client cannot be expressed without linguistic support for generic arguments.

I now believe that the only general solution to this problem is to define entity identifiers to be capabilities which have permissions associated with them. Primitives would be needed for duplicating identifiers, determining whether an identifier has appropriate permissions, and for refining and enhancing an identifier's permissions. A directory entity would determine the access to which a client was entitled using access matrices or whatever. It would then return a suitably refined identifier for the entity so that the client could perform the entity open itself. The capability scheme used in CAP [Wilkes 79] has a small set of global permissions, while in Hydra [Wulf 81] and CAP-3 [Herbert 78], there are both global and type specific permissions. In the entity system, there does not appear to be any need for global permissions.

Chapter 6: Implementation Issues.

6.1 Language Independent Class Definition.

In the experimental entity system, a definition consists of a hand coded Modula-2 **DEFINITION** module whose implementation is coded in assembler. This would clearly not be acceptable in a fully implemented entity system. Experience with the experimental entity system shows that it is important that as much as possible should be done automatically to avoid human errors.

In any single language entity system, the host programming language is the best basis for class definitions. The programmer would specify the operations and associated types for arguments and results in the type system of the host language. The entity system would then add auxiliary declarations for entity handle types, class identifier constants and the like. Any stub procedures necessary for the operation invocation can be generated as appropriate.

An entity system which supports client programs written in many languages presents a number of hard problems. How does the user specify types in a language independent way? What does type compatibility mean between different programming languages, and how is it checked? How is a class definition translated into a form that compilers can cope with? How are value representations translated?

The easy approach is to avoid all of these problems and require the programmer to define classes in each of the target languages, and if necessary, functions for mapping values from one form to another. This approach is unsatisfactory because of the amount of clerical effort the programmer is faced with and the scope for error.

In theory, fully general language independent specification of classes must be possible. After all, when a programmer designs an interface, he or she starts with a conceptual model and translates that into a programming language. Though the model is probably fuzzy, it can be viewed as a language independent

interface specification. This argument is of little practical use without the ability to express the model, and translate it into a real programming language.

Another approach might be to design a meta-language capable of describing the type systems of all of the target programming languages. This meta-language would form the basis for tools for automatically translating the types in a class definition into equivalent types in each target programing language. The author has no evidence to suggest that this approach would work, but it is worth noting that there is considerable work in the formal specification and description of type systems. An example which could prove to be especially relevant to entity class definition is the Pebble language [Burstall 84].

The most pragmatic approach to language independent class definition is to design a *class definition language* or *CDL* based on elements of the type system common to all of the supported programming languages. Primitive types like signed and unsigned integers, characters and reals have similar meanings in most programming languages. Most languages also provide some form of array and string types, and though the methods provided for manipulating array and string values vary, the meanings of the types are roughly the same.

The Courier remote procedure call standard [Xerox 81] and the ISO's Abstract Syntax Notation One (ASN.1) [ISO 84a], [ISO 84b] are good examples of this approach. These two standards define type abstractions for primitive types like integers, characters and reals, and for records, arrays and other type constructors. Then they define a standard representation that is used for transmitting typed values between systems. Courier can deal with nested non-recursive types. ASN.1 also allows recursive types, so long as the values have a finite size. As far as they go, both Courier and ASN.1 are roughly compatible with most strongly typed programming languages.

A CDL can adopt the ideas of Courier and ASN.1 for concrete types. Possible problems with the precision of primitive types such as integers and reals can generally be solved by restricting the CDL and checking at runtime that argument values are in the required ranges. String length restrictions can be handled in the same way. In some cases type concepts are incompatible. For example, array bounds in some languages start at zero or one, while in others the lower bound is part of the array type. In this and other cases, the CDL must take a view that is

compatible with as many languages as possible, and the programmer must expect to make adjustments to his or her programming style.

Assuming that the low-level incompatibilities can be taken care of, there are still a number of problems.

Argument Passing Semantics.

In most languages, entity operation invocation is expressed as a procedure call. Different languages use a variety of semantics for procedure call argument passing; call by value and call by reference being the most common. It would be preferable if entity operations defined in a CDL allowed the same semantics. Unfortunately call by reference semantics cannot be implemented when an entity implementation has no access to the client's address space, as would be the case in many entity systems. The same is true when the client and the entity implementation are written in languages with incompatible type representations. For this reason operation arguments in a CDL should be defined to be passed using a call by value mechanism.

An entity operation can return a number of results. Languages such as Pascal and Modula-2 restrict the number and type of the results of a procedure call. In such languages, an operation invocation can be transformed so that multiple results appear in the target language to be call by reference arguments. The stub operation procedure would be responsible for copying the results returned by an entity implementation's operation procedure into the "result" arguments.

Type Hiding.

Languages such as Mesa and Modula-2 have a concept of a hidden or opaque data type. Hidden types allow a module to pass values to a client without the client being able to use it. Hidden types in a representational type system perform a similar function to abstract types by restricting access to data items and minimizing dependencies between the modules of a program. Ideally, a CDL would use abstract types and the need for hidden types would not arise. In practice, operation arguments need to have representational types for compatibility with existing programming languages.

The problem with hidden types in a CDL is that when a data item with a hidden type is passed to a language without type hiding, the type and contents of the item are revealed. This allows the client program to make use of and possibly alter the hidden data. Furthermore, the client may now depend on a hidden type in a way that was not intended by the implementor of the interface.

Privacy of hidden types cannot be addressed by a CDL, since the value of a hidden type is readable in any language which has type system loopholes. Assuming that we are dealing with undisciplined rather than malicious programmers, a possible solution is to make it difficult to write a program which binds to a hidden type. This could be done by scrambling the names in a hidden type, rearranging its components, or something similar.

Object Based Typing.

Given that most of the languages concerned have no support for abstract (object) data types, the CDL type system needs to be representational at the level of an operation argument. In an object oriented type system the implicit invariants of an object representation can be generally relied on provided that the operations are implemented correctly. If an abstract object's representation is passing to a language without abstract data types or type hiding and back, there is no way of knowing what might have happened to it. A programmer using an object based language may need to be aware of this.

Constructed Objects in CLU.

In CLU, except for the primitive data types that fit in a single machine word, all data types are represented as pointers to heap nodes. This gives the CLU type system some novel and useful properties for constructed types. For example, a given record can appear at a number of points in a CLU array of records.

From the point of view of a CDL, it is unfortunate that CLU's type semantics are incompatible with the semantics of most other languages. A CLU array of records is directly compatible with a conventional array of pointers to records. To achieve compatibility with a conventional array of records, a CLU array must be copied eliminating sharing of subnodes.

Pointer Arguments.

Though neither Courier nor ASN.1 make any provision for handling pointers, a CDL needs to be able to do so. When an entity implementation and its client are written in the same language and occupy the same address space and have a common dynamic storage manager (i.e. garbage collector), pointer passing introduces no new problems. These arise when passing pointers from one address space or language to another.

A pointer argument to an entity operation might be used as follows:

- by treating it as a token and passing it back to the originator at the appropriate time,
- by comparing it with other pointers to find the shape of a data structure,
- by dereferencing it to access the data to which it refers, and
- by interpreting it as the pointer to (say) a buffer into which data should be written.

For the first two uses, simply passing the value of the pointer itself is sufficient. The remaining uses are problematical.

The approach taken in various remote argument passing mechanisms for CLU [Herlihy 81] [Hamilton 84] is to pass the entire data structure that can be reached from a pointer. Given that the CLU type system gives all objects pointer semantics, this is the most natural thing to do. In general, the cost of transferring an entire data structure is going to be high, especially if the transfer mechanism is to correctly handles cycles and shared substructures. Furthermore this effort may well be wasted, since the client might actually use only a small part of the copied data, or none of it at all.

Passing a data structure in this way involves making a copy of the structure in the recipient's domain. The value of the resultant pointer in the recipient's domain will be different from the original value. This is a problem if the pointer is then passed back to the original address space and compared with the original pointer value. Similarly, when copies of a pointer are passed as different arguments in the same or separate operation invocations, the recipient will end up with two distinct copies of the same data structures.

While it might be possible to preserve the relationships between data structures in the two domains, doing so is likely to be costly.

An alternative to copying an entire data structure is to copy nodes of a data structure on demand using similar techniques to PS-Algol [Atkinson 84] and Persistent Poly [Matthews 84]. The worst case (where the recipient looks at the whole data structure) is likely to be considerably more expensive than copying the entire data structure in one operation. In addition, there are the situations where copy on demand could lead to unexpected results. As an example, consider what might happen if an implementation receiving a pointer tried to use it while a later operation was in progress.

The best approach is to restrict the use of pointers along the following lines. A client or implementation may dereference a pointer passed as an argument provided:

- its address space is the same as that of the pointer,
- it has appropriate access to the memory in which the data is stored,
- the respective languages have compatible type systems and representations for values, and
- the respective language garbage collector(s) can cope with the potential problems.

If any of the above conditions is not satisfied, a client or implementation is only allowed to compare CDL pointers and pass them as arguments.

Given that this restriction is made, there needs to be some other way for data to be transferred between domains. In general, the simple way of implementing cross-domain argument transfer is for each domain to provide routines which marshal data structures into a standard external format and back again. The same routines could also be called by entity client and implementation code to explicitly dereference pointers and copy entire data structures. The saving is that the recipient of a pointer makes the decision as to how much data to transfer, not the argument passing mechanism.

Garbage Collection.

Passing a pointer from one domain to another can cause problems for language level garbage collection. The trouble is that a garbage collector for one domain has no knowledge of pointers held in other domains. The ideal solution to this problem is to modify the respective intra-domain garbage collectors to handle cross-domain pointers. Unfortunately, the effort involved in making the necessary changes for a number of languages is likely to be considerable.

Fortunately, it is possible to largely avoid the problem. Suppose that a pointer to a data item in a client's address space is passed in an entity operation invocation. While the invocation is in progress, the data in the client's domain is safe from garbage collection since one of the stub operation procedure's arguments will point to it. Thus the implementation's operation procedure is guaranteed that the pointer will be valid for the duration of the call.

Use of pointer arguments after the relevant operation has finished, and use of pointer results both require care to avoid dangling pointer problems. The pointer argument dereferencing mechanism can check that the pointers are still valid. The check itself is inexpensive, but extra information must be stored with both the pointer and the node it points to.

By the time a dangling pointer is found it is probably too late to do anything about it. Dangling pointers could be avoided entirely by providing a routine with which a program registers an interest in a data item in another domain. This would tell the remote garbage collector not to remove the data item. An entity operation procedure would use this mechanism when it saves a pointer argument. A similar routine is used to say that the node is no longer interesting.

Procedure Arguments.

The entity system provides mechanisms for handling operation procedures and for invoking remote operations. It should not be difficult to generalize them to allow procedure values to be passed as entity operation arguments and called by the recipient. The only difficulty is in defining the environment in which the procedure is executed.

Entity Handle Arguments.

There are situations where it is necessary to pass entity handles as arguments in entity invocations. The `Directory.Alias_Entry` operation in section 3.3 is an example of this. Dealing with entity handle arguments present both conceptual and practical problems.

CDL arguments are passed by copying rather than by taking a reference. In the case of an entity handle, there are three possible interpretations for copying. First, a handle can be copied within a program by duplicating the pointer. The next approach is to duplicate the private handle data structure. Finally, the handle state held in the entity activation could be duplicated as well.

The first interpretation is not appropriate when passing handles across domain boundaries. The second interpretation allows entity handle copying across domains, but has unpleasant semantics. An operation on one copy of a handle affects the handle state associated with the second. When one copy of a handle is closed, the second copy turns into the equivalent of a dangling pointer. The third interpretation results in two independent handles with desirable characteristics. Copying activation data structures requires the help of the entity's implementation since it alone understands them.

Assuming that a class definition language is possible, there remains the problem of implementing it. One approach is to extend each target language compiler to understand class definitions. A modified compiler would internally transform the CDL types into the equivalent host language types, and would generate code for invoking entity operations and converting the arguments. While this would give the best results in the long term, it would mean wholesale changes to a number of compilers.

A better approach is to write a translator or translators capable of converting a class definition into source code acceptable to each target compiler. The translator would produce equivalent target language definitions for the classes types and operations, and any stub operation code necessary. It is relatively unimportant that it may be necessary to bypass the target language type checker in the stubs. Since the target language stubs would be produced automatically, the possibility for human error in the interface does not arise. This technique is used in the Matchmaker interface specification language used in Spice [Jones 85].

6.2 Class Management.

A class in the entity system is characterized by a class identifier and the definition of the class operations. In the experimental entity system, the class definition is a Modula-2 **DEFINITION MODULE** which is stored outside the entity system, and the entity identifier is a token. Information about classes such as the value of the class identifier and the size of the operations table is compiled into each entity implementation by the Modula-2 compiler which runs outside the entity system.

In a self supporting system, class definitions need to be held in the entity file system. This can be done by defining a class of entities to represent classes. A *class entity* would have operations for returning the class name and operation definitions, and operations for creating and modifying the class definition. It would also be able to give information about the operations table needed when building an entity handle. The entity identifier for a class entity would be the class identifier for the class which it defines.

In the source of a client program, classes and perspectives are referred to by name rather than by class identifier. A class name is mapped into a class identifier and definition early on in the compilation process. The target language compiler can then type check the operation invocations and entity handles. The compilation process binds the class identifier(s) into the object module. Class identifiers also need to be mapped back into names at runtime when displaying entities and debugging implementations and client programs. The mapping between names and class identifiers (and hence class definitions) is held in a data structure called the *class index*.

When a class has been bound into a client program, or entities of that class have been created, the class definition must be treated as immutable[1]. However, a class definition can effectively be upgraded by changing the class index so that the class name points to a new version of the class. As new and existing client programs are compiled, they are bound to the new class definition. Other clients and entities remain bound to the old version of class, and incompatibilities between the different versions are handled by projection. It is also necessary to

1 This is necessary to allow compile and link time type checking.

be able to recompile a program without rebinding the classes so that old programs and entities can be maintained.

A class definition entity holds the specifications for the operations provided by entities of the class and associated abstract types. Each operation has a name, type specifications for its arguments and results, and specifications for the exceptions or signals that it raises and/or the error codes it returns. The operations and types may well use types defined in other class definitions. Finally, the class definition will hold the semantic specification for the class and its operations, and any associated documentation.

6.3 Implementation and Resolution Function Management.

The requirements for naming implementations and resolution functions in the entity system are more complex than for class naming. Implementations and resolution functions can be grouped according to the class that they implement. The differences between implementations of a class can be characterized using attribute / value pairs as described in [Lancaster 83]. Implementations can be selected by giving a search expression using these pairs. Entity implementation selection occurs when a new entity is being created or an existing entity is being opened. The selection of resolution functions occurs at open time using selection criteria described in section 4.3.

When an entity is created, the implementation to be used can be found in two ways. The implementation can be selected at compile time so that the implementation identifier is bound to the client object module. Alternatively, the implementation can be selected when the create operation occurs. The latter approach means that entity creation can pick up a new implementation without the need to rebind the client. It also means that the client can select different implementations depending on the situation. On the other hand, runtime selection makes entity creation more expensive, and removes the possibility of programmer intervention when a number of implementations or none of them have the specified characteristics.

When an existing entity is opened, the kernel uses the implementation corresponding to the identifier in the entity triple. The binding between an

implementation identifier and the code needs to be "soft" so that minor changes can be made to fix bugs. In general there may be multiple minor versions of an implementation. It is up to the entity system kernel to select the appropriate version when the entity is opened. It goes without saying that minor versions of an implementation must use the same format for the long-term representation.

Applications in the entity system depend on implementations and resolution functions conforming to class specifications. The system must guard against *trojan horses* if security and privacy are taken seriously. For example, a trojan horse version of an experimental entity system directory implementation could bypass any access control mechanisms and release information from a directory that the client should not have access to. A trojan horse can also act against the interests of a client. In the case of directories, the `alias_entry` operation could misuse the handle for the second directory entity by deleting entries or passing the handle to some other process.

The trojan horse problem is hard to solve in the entity system. In general, it is impossible to prove a program is secure [Denning 83]. However, if the kernel is implemented as a security kernel it would be possible to include the security constraints in the formal class specification and prove that each implementation meets the specification. The alternative is to have someone "vet" new implementations and resolution functions for critical classes before they are released to check that they are not a security risk. This approach is vulnerable to human corruption and carelessness.

To be effective, an implementation security scheme must make it impossible for a user to use private implementations of critical classes without having them vetted. At the same time, the user should be allowed to use his own implementations and resolution functions for private and non-critical public classes.

6.4 Host Machine and Operating System Problems.

The entity system generally has to run within an existing hardware and operating system environment. This section discusses the problems in implementing an entity system in host systems which provide

- a single virtual address space for all processes
- a virtual address space for all processes with memory access control,
- disjoint virtual address spaces for each process,
- virtual address spaces for each process with shared memory facilities,
- capability based addressing,
- processes on separate machines on a local area network.

An entity system within an open system [Richards 79] [Redell 80] can never be absolutely secure. Some protection is provided if the host language has strong data typing, though most strong type systems have loopholes either by design or by accident. Error recovery can be a problem in an open system because of difficulties in finding the resources such as file locks and heap space that must be freed when a piece of code crashes. Furthermore, when a program goes haywire it may corrupt some totally unrelated data structure or piece of code. On the other hand, open systems present the least problems for passing entity operation parameters.

Provided that it is not necessary to convert arguments from one representation to another, invoking an entity operation in an open system is comparable in cost to an ordinary procedure call. Indeed, once arguments have been converted, an operation invocation is an indirect procedure call. No process switch is required.

In a single processor, single address space system with memory protection, fire-walls between components of an entity system are possible. This makes it possible to reliably recover from errors in clients and implementations, and allows an implementation to safely hold privileged information. When an unrecoverable error occurs in a client or entity implementation, the kernel applies the **Abort** operation to any entity handles held by the domain. This means that the kernel must keeps track of entity handles passed from one domain to another. If an implementation crashes, it is necessary to invalidate any associated entity handles and mark the corresponding entity triples as unsafe.

The immediate cost of firewalls is in changing the memory protection registers on each entity operation invocation. If the client and implementation are mutually

suspicious there is also the cost of copying operation arguments. An alternative is to trust the implementation to access and update data in the client's domain. When an operation's arguments include an entity handle, it would be necessary to guard against the client passing a entity handle with a bogus implementation in order to gain access to privileged information. Furthermore, it is advisable for an operation to be performed using a different stack so that the client cannot induce stack overflow, and to avoid information leaking in "dead" stack frames. Depending on the host architecture, this may also require a process switch and associated overheads.

When the architecture of the host system requires that individual clients and implementations occupy disjoint address spaces, all arguments must be copied. This sort of host environment typically equates an address space with a process, and requires that operation arguments and results are passed using an inter-process message passing mechanism. Facilities for sharing memory segments between processes can be used to avoid copying operation parameters, but only if implementation can be trusted. An entity implementation would need to use relative addressing to avoid clashing with pointers passed by the client.

Ideally, all handles for an entity share one entity activation to facilitate synchronization and caching. For an implementation to manage the entities as a pool rather than individually, it needs access to a separate global data area as well as the activations. In environments with UNIX style processes, this means either a separate process for each activation, or one process for the entire implementation containing all activations. In either case, the fact that processes are single threaded causes problems when clients make simultaneous operation requests. Requests can be queued with an implementation handling them one at time, but special precautions would need to be taken to avoid deadlock. The alternative is for each implementation or activation process to contain a mini-kernel which simulates multiple threads of control. This assumes that the host operating system allows non-blocking I/O. This approach is taken in the Eden system [Lazowska 81] where an `Eject` (Eden object) is a UNIX process which receives interprocess invocation messages and uses a runtime library to multiplex their execution.

A fine grained capability architecture offers the best solution to argument passing. The iAPX 432 machine [Myers 82] allows the use of segments to represent language level data objects, with pointers being represented by

segment capabilities. This avoids problems with pointers and allows complex objects to be passed from one domain to another without worrying about information leakage. By contrast, the capability architectures of CAP [Wilkes 79] and CAP-3 [Herbert 78] are too coarse to be used for individual nodes in a program's data structure.

An entity system in a distributed environment presents a different set of problems. Communication between machines in a local area network is orders of magnitude slower than within a single machine. When the client and the entity activation are on different machines, marshaling operation arguments may well take a fraction of the total time. The cost of communication makes it important that the entity system should keep the client and the entity activation on the same machine where ever possible.

It is simpler if there is only one activation of an entity and it is on the machine which holds an entity's permanent representation. Unfortunately, this is counter to the aim of reducing the client / activation communication costs and is likely to lead to network and processing bottlenecks. The alternative is for entity implementations to be internally distributed, and have the operation procedures propagate the changes between the entity activations on different machines. This allows an implementation to take advantage of processing resources on different machines, and makes it possible for the kernel to balance the processing and network load to avoid bottlenecks. This must be offset against the added complexity of an internally distributed implementation, and the additional communication overheads this may entail.

The entity system operation invocation mechanisms would need to be rethought to take into account the possibility of data loss and network partition. Entity classes would also need to be defined with the distributed nature of the system in mind.

6.5 Improving Entity System Efficiency.

For the entity system to be competitive with established techniques for storing structured data, it needs to have comparable performance. The performance issue was not seriously addressed in the experimental entity system, but it is

particularly bad in the area of storage efficiency. The overhead of storing a small entity is enormous in relative terms, and each entity activation uses scarce resources such as global frame table entries. Another area where entity systems in general fall behind is in the cost of opening entities and making operation requests. In this section, methods for improving entity system efficiency are discussed. These ideas are based in part on work in progress on a new entity system kernel which runs under the Xerox Development Environment.

It has already been noted that it is inefficient to hold class and implementation identifiers in the same storage object as the entity representation. Separating the entity triple from the representation means that the representation storage object does not always need a link store. More important, it allows an entity implementation to use the low level storage interface that is most appropriate to the task in hand. Even so, storing and activating each entity separately leads to storage overheads for small objects that are unacceptably high.

Storage efficiency is improved if an implementation manages a number of entities together rather than dealing with them separately. In the experimental entity system, an implementation module instance would need to handle a number of entities and associated activations. In a system with client and entity in separate address spaces, this means one address space per implementation rather than one per entity activation.

Such an implementation can store the representations of a number of entities in a single storage object. At entity open time, it translates an entity's representation identifier into an offset into the storage object to find the entity's representation. This implies that the implementation generates its own representation identifiers. The disadvantages of multiple entity implementations is that they are more complicated, and that when an implementation goes wrong or the machine crashes, the number of entities at risk is higher.

Another factor that contributes to storage inefficiency for small entities is the size of global identifiers. Global entity identifiers need to be at least 8 bytes long, and probably longer to guarantee that identifier values are never reused. Each entity triple contains three such identifiers (four including the entity identifier itself) and each external reference requires a further copy of the identifier.

When an entity is composed of a collection of subentities that are only ever used within the context of the large entity, the subentity identifiers do not need to be globally unique. Instead entities and entity identifier space can be organised as a number of nested *entity domains*. An identifier for an entity in a domain is only significant within the domain and cannot reliably stored outside of the domain. An entity can hold identifiers for entities in surrounding domains, though it will handle them differently because of the identifier size difference. In the experimental system, a domain could be represented within the storage object of an enclosing entity. External identifiers would be stored in the object's link store, and the representation of the identifiers within the inner domain would contain the corresponding offset into the link vector.

There are many possible benefits from using nested entity domains. Identifiers used in collections of small entities can be handled more efficiently, making it feasible to use entities for fine grain objects. Identifier management and garbage collection within a domain can be independent of all but surrounding domains. This has obvious advantages in a distributed system.

The cost of an entity operation invocation depends on the environment in which the entity system is implemented. As far as efficiency is concerned, the best environment is one where the client and the implementation are in the same address space and use the same representations for parameters. When the client and the entity implementation have to be separate processes in different address spaces, operation arguments must be moved from one address space by copying. At least two process switches are needed to transfer control between the client and implementation. Finally there is the direct and indirect cost of changing the address mapping. In such an environment, it appears that the only way to cut down operation invocation overheads is to put the client and entity implementation in the same address space.

Breaking down the firewalls is an unacceptable risk when either the client or the implementation is dealing with sensitive data. Therefore it is still necessary to support operation invocation in other address spaces. The address space containing the client and entities also needs to service operation requests from outside. Indeed, the address space may have to persist after the client program has exited

to deal with requests on outstanding external handles. All of these complications can be handled by a mini-kernel resident in the address space.

Chapter 7 - Conclusions.

This thesis has described a model for a file system which extends data typing beyond the bounds of a single program. In doing so, the model addresses a number of requirements that do not arise in the type system of a conventional programming language. These include resilience in the face of bugs and the ability to evolve file types and representations with time.

The entity system has an object oriented type system, so that an entity provides both the data and the algorithms which maintain it. This simplifies application programming, and helps to insulate the file system from the effects of erroneous client programs. An entity is characterized by a class, an implementation of the class and a representation which the implementation is responsible for managing. New implementations can be brought into service without recompiling the client programs. Indeed, since a client can transparently use entities with the same class and different implementations, it is not necessary to convert existing entities to a new format.

The entity system provides a mechanism known as projection which amounts to a dynamic type coercion scheme. Projection provides the flexibility needed to allow class interfaces to evolve with a minimum of disruption to existing programs and objects. It can also be used to provide some of the capabilities of superclasses and subclasses in Smalltalk and parameterized types in CLU. The main problem with projection is that it is necessary to hand code the resolution functions to set up a projection.

The thesis has described an experimental entity system and its associated low-level "standard" storage interface. A number of conclusions were drawn from this work. The most important one is that the central ideas of the entity system, namely separation of class and implementation and the projection scheme, do appear to be sound. Indeed, projection may turn out to be important in its own right as the means of building complex systems. The experimental system also provides positive evidence that a full entity system could be implemented for both personal computers and timesharing systems.

On reflection, various flaws are apparent in the experimental entity system. The idea that all implementations can be served by a single low-level storage interface proved to be false. The strategy of storing an entity's class and implementation identifiers in the storage object holding the representation caused problems that had not been foreseen. It demonstrated that an access control mechanism is best supported by the core of the system by making identifiers full capabilities with access rights.

Chapter six discussed a number of areas which were not investigated using the experimental entity system. Section 6.1 dealt with programming language independent definition of classes. The idea of a class definition language was proposed, and various alternatives for the CDL's type system were discussed with reference to the problems of implementing them. The author hypothesizes that a CDL could be designed and suitable translation software implemented which would allow an entity system to be used with most strongly typed languages.

Sections 6.2 and 6.3 dealt with some of the infrastructure needed in a self supporting entity system. Assuming that the problems of multi-language class definition can be solved, class management is largely a matter of binding names to classes. Implementations need to be characterized according to their properties. The need to protect against "trojan horse" implementations and resolution functions was also discussed.

Section 6.4 outlines how an entity system might be implemented within various machine and operating system architectures. One conclusion is that passing operation arguments is likely to be expensive in a system with memory protection and mapping hardware, unless the implementor is prepared to trust entity implementations to behave correctly. A fine grained capability architecture would allow complex argument passing without compromising security. It was shown that both UNIX-like systems and distributed systems present new problems, but it is conjectured that they are not unsolvable.

The entity system has more inherent overheads than are encountered within most programming languages. These overheads are especially apparent in storing small objects and in making operation requests. In section 6.5, some methods are proposed which would help to reduce these overheads. These include the use of domains for closely related collections of entities, implementations which manage

a number of entities, and having a trusted client and the entities it uses share the same address space.

As a general conclusion, it seems that the entity system model satisfies the original requirements. Much work is still needed in a number of areas, including exploring the infrastructure issues and class definition. The problems of entity system implementation in a multi-user system like UNIX and in a distributed system also need further work.

The author is currently working on a "full scale" implementation of the entity system for a Dandelion workstation running the Xerox Development Environment (XDE). The XDE entity system already supports multi-processing and implementations that manage a number of entities. Storage objects are entities in their own rights, and two different storage object classes are already available. It is planned to add support for entity domains, and for entity recovery and garbage collection in the near future. The new entity system is going to be used as the vehicle for programming environment research by the author and other people.

The last example of section 3.3 is a rough sketch of how an entity could be used to represent the large scale components of a Modula-2 program such as the source and object code, symbol tables, dependency lists and documentation. A colleague is using a similar approach for organizing Mesa programs. An associated system browser operates on entities representing Mesa program components such as source, object and DF files[1]. "Annotation" entities are bound to these components containing related information such as documentation, compilation and display instructions. An event mechanism is proposed for notifying annotations when the corresponding component entities change.

The author is planing to use the entity system to represent Mesa program source code. It is observed that in a conventional programming environment, a lot of time and effort is expended in converting programs in a textual form into parse trees. From a number of points of view, it would be better if the primary representation for the program was the parse tree itself. The author plans to represent a Mesa parse tree using entities. A syntax directed editor will be written

1 The XDE DF or "Describe File" software is a set of programs for managing the versions of the component files that make up an XDE system, utility or package.

which operates directly on the parse tree, and the Mesa compiler will be modified to accept the parse tree as input. Another possibility is the use of parse trees rather than source text as the basis of finding the differences between versions of a program.

Using highly structured typed objects rather than textual files to represent information has the disadvantage of making the information more difficult to display and edit. It is believed that projection can play an important part here. A suitable perspective class of "display objects" could be defined, and associated resolution functions could be written to extract and format the information held by various classes of entity. The perspective class would be used by a generalised entity viewer to allow the user to examine any entity for which suitable resolution functions existed. This could be taken a step further to provide a general purpose entity editor. Another application in this line is a generalised tool interface for applying entity operations.

References.

[Almes 83] Almes, G. T., et al,
The Eden System: A Technical Review,
Technical Report 83-10-05. Dept. of Computer Science,
University of Washington

[ANSI 75] American National Standards Institute
Interim Report ANSI / X3 / SPARC Study Group on Data Base Management Systems.

[Atkinson 82a] Atkinson, M. P. et al
Algorithms for a Persistent Heap,
Report CSR-109-82 University of Edinburgh.

[Atkinson 82b] Atkinson, M. P. et al
CMS - A Chunk Management System,
Report CSR-110-82 University of Edinburgh.

[Atkinson 84] Atkinson, M. P. et al
PS-algol Reference Manual,
Report PPR-4-83 University of Edinburgh.

[Barman 84] Barman, H. J.
Report for Year 1983-1984.

[Birtwistle 73] Birtwistle, G. M. et al
SIMULA BEGIN
Petrocelli / Charter.

[Burstall 84] Burstall, R & Lampson, B.
"A Kernel Language for Abstract Data Types and Modules" in *Proceedings of the International Symposium on the Semantics of Data Types*, Sophia Antipolis, France 1984.

[CODASYL 71] CODASYL DBTG
CODASYL Data Base Task Group, Conf. Data Sys. Languages,
ACM.

[Crawley 82] Crawley, S. C.
The Testbed Entity System,
Programming Environment Research Group Note. Cambridge University Computer Laboratory.

[Date 83] Date, C. J.
An Introduction to Database Systems [2 volumes],
Addison-Wesley.

[Date 84] Date, C. J.
A Guide to DB2,
Addison-Wesley.

[DEC 80] Digital Equipment Corporation
An Introduction to VAX-11 Record Management Services,
AA-D024C-TE Digital Equipment Corporation.

[Denning 83] Denning, D. E. R.
Cryptography and Data Security.
Addison-Wesley.

[Dion 80] Dion, J.
Fileserver External Specifications. Version 8,
Systems Research Group Note. Cambridge University Computer Laboratory.

[Dion 81] Dion, J.
Reliable Storage in a Local Network,
Technical Report 16. Cambridge University Computer Laboratory.

[Foderaro 83] Foderaro, J. K. et al
"The FRANZ LISP Manual" in *ULTRIX-32 Supplementary Documents Vol 2,*
AA-BG67A-TE Digital Equipment Corporation.

[Goldberg 83] Goldberg, A. & Robson, D.
Smalltalk: The Language and its Implementation,
Addison-Wesley.

[Hamilton 84] Hamilton, K. G.
Transporting Abstract Types,
Mayflower Group Note. Cambridge University Computer Laboratory.

[Hamilton 85] Hamilton, K. G.
A Remote Procedure Call System,
Technical Report 70. Cambridge University Computer Laboratory.

[Herbert 78] Herbert, A. J.
A Microprogrammed Operating System Kernel.
Ph. D. Thesis, Cambridge University.

[Herlihy 81] Herlihy, M. & Liskov, B.
A Value Transmission Method for Abstract Data Types,
Report of Laboratory of Computer Science, MIT.

[Hughes 83] Hughes, J. W. and Powell, M. S.
DTL: A Language for the Design and Implementation of Concurrent Programs as Semantic Networks.
Software Practice & Experience. Vol 13 (December 83)

[IBM 80] International Business Machines
OS/VS2 MVS Data Management Services Guide Release 3.8,
GC26-3875-1 International Business Machines Corporation.

[ISO 84a] International Standardization Organisation
Specification of Abstract Syntax Notation One (ASN.1).
ISO/OSI working document. ISO/TC 97/SC 16 N1795.

[ISO 84b] International Standardization Organisation
Encoding Rules for Abstract Syntax Notation One.
ISO/OSI working document. ISO/TC 97/SC 16 N1796.

[Jensen 75] Jensen, K. & Wirth, N.
Pascal User Manual and Report,
Springer-Verlag.

[Jones 85] Jones, M. B., Rashid, R. F. & Thompson, M. R.
"Matchmaker: An Interface Specification Language for Distributed Processing" in *Proceedings of the Twelfth Annual Symposium on Principles of Programming Languages*, ACM.

[Jordan 79] Jordan, M. J. & Singer, D. W.
BCPL Paged Heap, Program documentation.
Cambridge University Computer Laboratory.

[Kent 78] Kent, W.
Data and Reality,
North Holland.

[Kernighan 78] Kernighan, B. W., & Ritchie, D. M.
The C Programming Language,
Prentice-Hall.

[Kerschberg] Kerschberg, L., Klug, A. & Tsichritzis, D.
"A Taxonomy of Data Models in Systems for Large Databases" in *Proceedings of the 2nd International Conference on Very Large Data Bases*,
Elsevier/North Holland.

[Lancaster 83] Lancaster, J. N.
Naming in a Programming Support Environment,
Report TR-312, Laboratory for Computer Science, MIT.

[Lazowska 81] Lazowska, E. et al
"The Architecture of the Eden System" in *Proceedings of the Eighth Symposium on Operating Systems Principles*,
ACM SIGOPS Operating Systems Review Vol. 15 No. 5.

[Liskov 81] Liskov, B. et al
CLU Reference Manual,
Springer-Verlag.

[Matthews 82] Matthews, D. C. J.
Poly Report,
Technical Report 28. Cambridge University Computer Laboratory.

[Matthews 84] Matthews, D. C. J.
Private Communication.

[Middleton 79] Middleton, M. D.
A Proposed Definition of the Language BCPL.

[Mitchell 81] Mitchell, J. G.
Thoughts as a Basis for Programming Environment Tools,
Programming Environments Research Group Note. Cambridge University Computer Laboratory.

[Myers 82] Myers. G. J.
Advances in Computer Architecture,
John Wiley.

[Needham 82] Needham, R. M. & Herbert, A. J.
The Cambridge Distributed Computing System,
Addison-Wesley.

[Organick 72] Organick, E. I.
The Multics System: An Examination of Its Structure,
MIT Press.

[Pardoe 84] Pardoe, J. B. D.
Private Communication.

[Pardoe 85] Pardoe, J. B. D.
Private Communication.

[Pereira 83] Pereira, F. editor
C-Prolog User's Manual - Version 1.4d,
SRI International.

[Pitman 83] Pitman, K. M.
The Revised MACLISP Manual,
Report, Laboratory for Computer Science, MIT.

[Redell 80] Redell, D. D. et al
Pilot: an Operating System for a Personal Computer,
Communications of the ACM. Vol 23 (February 1980).

[Richards 79] Richards, M. et al
TRIPOS - A Portable Operating System for Minicomputers,
Software Practice & Experience Vol 9 (June 1979).

[Richards 80] Richards, M. & Whitby-Strevens, C.
BCPL - the Language and its Compiler,
Cambridge University Press.

[Ritchie 74] Ritchie, D. M. & Thompson, K.
The UNIX Timesharing System,
Communications of the ACM Vol 17 No. 7 (July 1974).

[Singer 81a] Singer, D. W.
Classes as the basis of a programming environment.
Rainbow Group Note. Cambridge University Computer Laboratory.

[Singer 81b] Singer, D. W.
The first attempt PEAT Kernel,
Rainbow Group Note. Cambridge University Computer Laboratory.

[Smith 84] Smith, L. D.
The Management of Persistent Data in Modula-2,
VLSIC Design Aids Group, Acorn Computers UK Ltd.

[Tsichritzis 77] Tsichritzis, D. C. & Lochovsky, F. H.
Data Base Management Systems,
Academic Press.

[Tsichritzis 82] Tsichritzis, D. C. & Lochovsky, F. H.
Data Models,
Prentice-Hall.

[Weinreb 79] Weinreb, D. & Moon, D.
Lisp Machine Manual - 2nd Preliminary Version,
Report of Artificial Intelligence Laboratory, MIT.

[Wilkes 81] Wilkes, A. J.
Projections and Perspectives,

Programming Environments Research Group Note. Cambridge University Computer Laboratory.

[Wilkes 82a] Wilkes, A. J.
Hades - A Command Environment that Supports Structure,
Software Practice & Experience, Vol. 12 (1982).

[Wilkes 82b] Wilkes, A. J.
A Portable BCPL Library,
Technical Report 30. Cambridge University Computer Laboratory

[Wilkes 79] Wilkes, M. V. & Needham, R. M.
The Cambridge CAP computer and its operating system,
Elsevier - North Holland.

[Wirth 80] Wirth, N.
Modula-2,
Institut für Informatik, Eidgenössische Technische Hochschule, Zurich.

[Wulf 81] Wulf, W. A., Levin, R. & Harbison, S. P.
HYDRA/C.mmp: An Experimental Computer System,
McGraw-Hill.

[Xerox 81] Xerox Corporation
Courier: The Remote Procedure Call Protocol,
XSIS 038112. Xerox Corporation.

[Xerox 84a] Xerox Corporation
Pilot Programmer's Manual,
XDE3.0-5001 Xerox Corporation.

[Xerox 84b] Xerox Corporation
Mesa Language Manual (version 3.0),
XDE3.0-3001 Xerox Corporation.

Appendix A. Dynamic Modules in Modula-2.

The experimental entity system uses a dynamic module package to support dynamic loading of implementations, resolution functions and command routines for the command line interpreter. To explain how the dynamic module package works, it is necessary to describe how the Modula-2 runtime system used for the entity system works.

Implementations of Modula-2 based on Lilith M-code use a *global frame table* (GFT) to bind together modules (figure A-1). Each implementation module

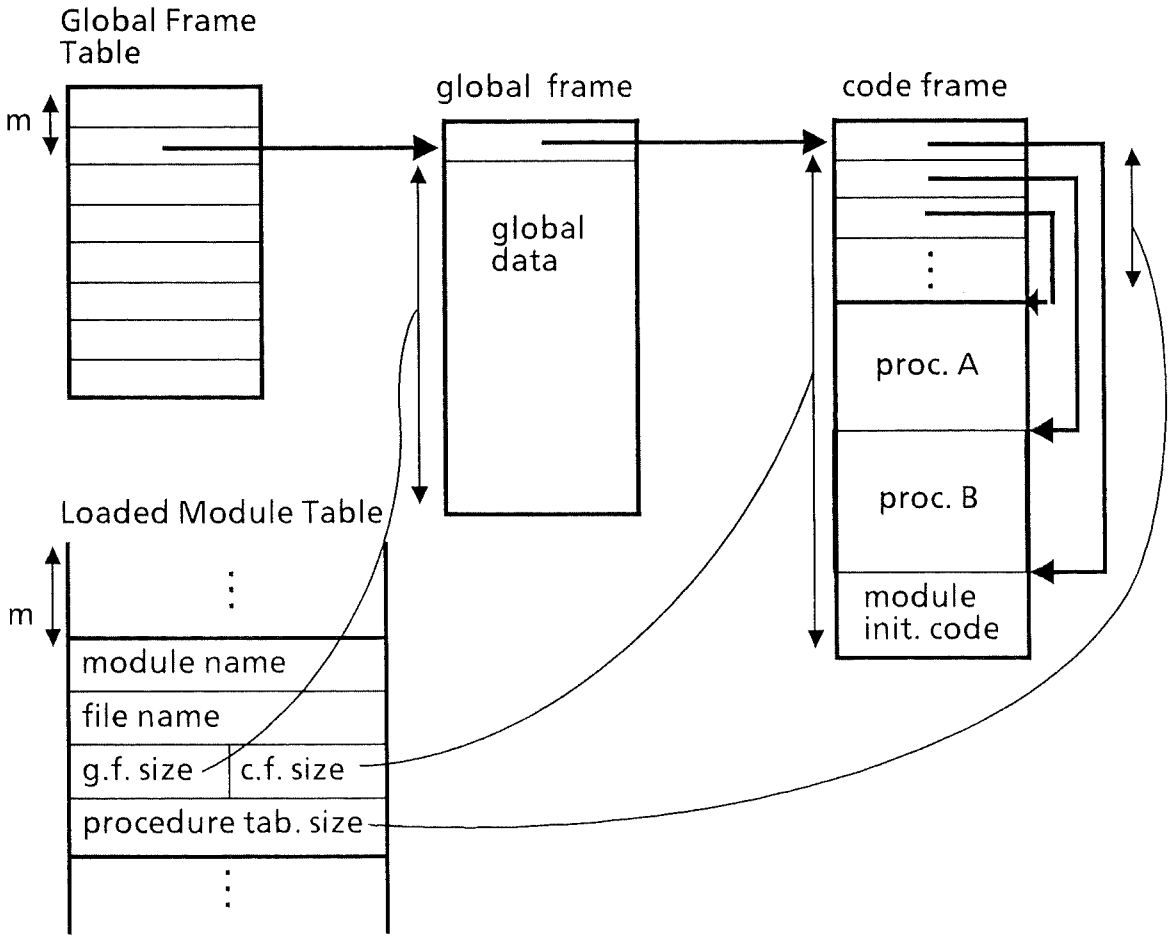

figure A-1

consists of a *global frame* which holds the module's global variables, and a *code frame*. The code frame starts with a *procedure table* which has pointers to all of the module's procedures. A non-local procedure call sets a register to point at the called module's global frame, and indirects through the procedure table to get to

the procedure. Calls using procedure variables are done the same way, and therefore procedure variables consist of a GFT number and a procedure number. Only calls to constant procedures in the current module are translated into absolute (or PC relative) call instructions. The *Loaded module table* is used to map module names into GFT slots, and find the sizes of global frames to duplicate them.

The dynamic module package can be used in 3 ways. The simplest of these is to dynamically load an implementation module in place of a stub implementation module. References to procedures and global variables in the dynamic module can then be made as if the module was statically linked.

A second technique is to load a module at a new slot in the GFT. Such a module cannot be accessed from outside using the normal Modula-2 import / export mechanism, since such references have to be resolved at link time. This means that the global variables cannot be accessed, and that procedures can only be accessed using procedure variables. It is necessary to go through some contortions to make the relevant procedure variables known outside of the dynamic module. One way this can be done is by the loaded module's initialisation code calling back on a procedure exported statically by some other part of the system with the relevant procedure variables as arguments.

Finally, the package allows the client to create a new instance of an existing static or dynamically loaded module. This is done by duplicating the module's global data frame and allocating a new GFT slot. This has the same external reference problems as the previous technique. Unfortunately, one thing a duplicated module instance cannot do is generate a correct procedure variable because the compiler generates code which uses a constant module number bound into the code frame when it is relocated. Indeed, the dynamic module package does not even invoke the module initialisation code for a copied module, since initialisation has already been done. The solution in this case is to massage procedure variables from the original module, replacing the old module number with the new one.

The dynamic module package has a two other limitations. First there is no interface checking, so there is nothing to stop incompatible implementations from being loaded. This would have been possible to implement, but it would

have involved referring to information that was not present in memory. The second limitation is that there is no way of specifying dependencies between modules so that shared submodules can be handled neatly. As it is, modules and collections of modules to be dynamically loaded in one operation are *prelinked* into a single file. It would be desirable to have mechanisms for loading submodules when they are needed and for avoiding the premature unloading of submodules that are shared by many modules.

Appendix B. Glossary of Entity System Terminology.

Abort operation: This operation defined by all *classes* is applied to *entity handles* which have not been closed by their *clients*. It gives the *implementation* a chance to restore an *entity* to a consistent state after the client has terminated uncleanly.

Active: An active *entity* is an entity in a form which allows *clients* to invoke *operations* on it.

Base class: The base class of an *entity* is the entity's true type. This is the *class id* for the base class which is bound into the *entity triple*.

Class: A class in the entity system is the object oriented interface that an *entity* presents to its *clients*. It is thus the abstract type of an *entity*.

Class definition: A class definition is the type definition for a *class*. It includes the types and semantic specifications of the *operations* which may be applied to all *entities* of the *class*. It also includes the type definition for *entity handles* for entities of the class.

Class id: This is an *identifier* for a *class*. Since there are 1-1 relationships between a class, a class id and a *class definition*, the *kernel* checks *entity* types at run time by comparing class ids. In some entity systems, a class id is the identifier for the entity which holds the class definition.

Client: A client is a piece of code which uses a given interface. It may be a user program, a utility program, an entity *implementation* or a *resolution function*, or even the entity system *kernel*.

Close operation: This operation defined by all *classes* releases an *entity handle*, and deactivates the corresponding *entity* if there are no other handles for it.

Create operation: This operation defined by all *classes* creates a new *entity* using an *implementation* selected by the *client*. Class specific create arguments tell the entity implementation how to initialize the

new entity. The result of the create operation is an *entity identifier*.

Delete operation: This operation defined by all *classes* is applied to an *entity* when it is about to be deleted so that it can release any resources it uses. Typically the delete operation is applied by the *kernel* when an entity becomes inaccessible.

Entity: This is the term for a first class object in the entity system. An entity is strictly typed persistent object, characterised by a *class*, an *implementation*, and a *representation*. An entity may exist in *active* and *passive* forms.

Entity activation: An entity activation holds the short term *representation* of an active *entity*. This includes information cached from the entity's permanent representation and state associated with transactions in hand. It may also include the state associated with individual *entity handles*. The *kernel* arranges that there is never more than one activation of each entity at any given time. The *implementation* can therefore use the activation to synchronize *operations* on different handles.

Entity handle: An entity handle is a short term capability for an *entity* in its *active* form. It may be thought of as a capability to use the entity. In general, a client needs an entity handle to invoke an *operation* on an entity. A number of handles for a given entity may exist at any time. Individual handles may carry access rights and other state associated with the client.

Entity id: An entity id is an *identifier* for an *entity* in its *passive* form. It may be thought of as a capability to refer to an entity. Entity identifiers do not carry any fine grained access control information.

Entity triple: A *passive entity* is conceptually stored as a triple which consists of a *class id*, an *implementation id* and a *representation id*.

Generic operations:
A small number of *generic operations* are defined by all *classes*. These operations perform standard tasks, but may also take class specific arguments. Key parts of all of the

generic operations are carried out the *kernel*. The common generic operations are *Open*, *Close*, *Abort*, *Create* and *Delete*.

Handle: A handle is a short term capability to use an object.

Identifier or Id: An identifier is a "magic number" which represents an object for its entire lifetime. The creator of an identifier may encode useful information in the value of an identifier. Such information is hidden from the outside world by the type system, but should not be relied upon to be secure.

Implementation: An implementation is the type manager for a number of *entities* of a given *class*. It provides a set of procedures which perform the *operations* defined by the class. The *operation procedures* are responsible for the management of each entity's *representation*, for access control and for the synchronization of concurrent operations on a single entity. There may be a number of different implementations for a given class, each managing a disjoint subset of the entities of that class. Thus, a given entity is managed by one implementation.

Implementation Id: This is an *identifier* for an *entity implementation*.

Kernel: The kernel of the entity system is responsible for overall entity system control. It manages the most important low level data structures and provides a number common services used by other parts of the system.

Open operation: Each *class* defines an open operation which takes an *entity identifier* and class specific arguments and returns an *entity handle* for the class. The class specific arguments allow the *client* to tell the *implementation* about the intended use of the handle. If the *perspective class* implied by the open operation is different from the *base class* of the entity, the kernel uses *projection resolution* to set up the entity handle.

Operation: An operation is a abstract manipulation applicable to *entities* of a given *class*. An operation is characterised by an operation name and an operation type. The latter consists of the formal

names and types of the arguments and results of the operation. The specification of an operation also includes a formal or informal description of its external semantics. Operations are defined as part of the *class definition*.

Operation invocation:

An *entity client's* request to perform an *operation* is called an operation invocation.

Operation procedure:

This is a routine which performs an *operation* on an *entity*. An operation procedure conforms with the *operation type*, and when called it should perform the operation on the entity according the specified semantics. Operation procedures are provided by *implementations* and *resolution functions*.

Operations table: Associated with every *entity handle* typically is a structure called the operations table which holds pointers to the *operation procedures*. When the client invokes an operation on a handle, the operations table gives the procedure to be called. This extra level of indirection is needed to support *projection*.

Passive: A passive *entity* is an entity held entirely in stable storage. A passive entity must be *activated* before it can be used by *clients*.

Perspective class: A *client* views an *entity* as having a perspective class. In the simple case, the client's perspective class and the entity's *base class* are the same. Otherwise, the *open operation* provides the client with an *entity handle* that coerces a client's perspective class *operation invocations* into base class operations on the entity itself.

Projection: Projection is the term for run time coercion of *operations*. The *client* invokes operations in the context of a *perspective class*. These are projected into operations in the context of the entity's *base class* that can be carried out by *operation procedures* from the *implementation*. Operation projection is transparent to the *client* and *implementation* of the entity.

Representation: The representation of an *entity* consists of two parts. The long term part persists when an entity is *passive*. The short term part is associated with an *entity activation*, and with individual *entity handles*. An entity's representation is only accessible to the entity's *implementation*.

Representation Id: A representation id is the *identifier* for an *entity's* long term *representation* that is held in the *entity triple*. It may be the identifier for a *storage object* which holds the representation.

Resolution: Resolution is the process of deciding on a *projection* and setting up the mechanisms for carrying it out. Resolution occurs at *entity open* time when the *client's perspective* and the *entity's base class* are different. Resolution can fail when none of the available *resolution functions* can produce an appropriate projection. Resolution failure is analogous to a run time type error.

Resolution activation:
A resolution activation holds the *resolution function's* global variables. It is used by a resolution function's *operation procedures* for holding information relating to the *entity handle*. There is a resolution activation for every entity handle that uses resolution function operations.

Resolution function:
A resolution function is a routine which does the *perspective class* specific part of *projection resolution*. A resolution function builds and returns an *entity handle* compatible with the perspective class. The entity handle will have an associated *operations table* containing *operations procedures* from the *implementation* and the resolution function.

Selector string: This is a string passed to the *kernel* and to *resolution functions* when *opening* an *entity*. The *client* may use it to give more information in cases where there is more than one way to resolve a projection.

Storage object: A storage object is an object which provides a low level data storage interface for use by *entity implementations*. When storage objects are *entities* there may be a number of *classes* of storage object suitable for different purposes.

Stub operation procedure:

In some entity system implementations, the client calls a stub operation procedure to *invoke* an *entity operation*. A stub operation procedure, which is independent of the *implementation* of the entity, is typically responsible for collecting the arguments and dispatching them to the *implementation* using an ordinary or remote procedure call, and then for returning the operation results to the client.