University of Glasgow
Department of Computing Science
Lilybank Gardens
Glasgow G12 8QQ

University of St. Andrews
Department of Computational Science

North Haugh
St Andrews KY16 9SS

A Context Sensitive Addressing Model

Persistent Programming
Research Report 27

Alan Dewle

A Context Sensitive Addressing Model

A. J. Hurst
Department of Computer Science,
Australian National University,
GPO Box 4, Canberra, A.C.T. 2601
Australia

Introduction

The growth of computer systems has been marked by rapid developments in the field of hardware technology and productivity, and relatively slow advances in the field of software technology and productivity, particularly the latter. Two aspects of this phenomenon are relevant to this discussion.

One is the now almost universal use and acceptance of high level languages to improve programmer productivity and software performance. This has been in spite of arguments advanced against the use of high level languages on the grounds of "inefficient use" of the underlying hardware. The demonstrated improvements in software production and maintainence, and the improved manageability of large software systems that arise from the use of high level languages have largely dispelled such arguments. Indeed, it is now widely accepted that the way forward in software design is to decrease the emphasis paid to the "efficiency" of software execution, and look more closely at the "efficiency" of software construction.

The second aspect has been the thought that in order to reduce software complexity and inefficiency, functional capability should be transferred from the software domain to the hardware domain. This idea has prompted such significant architectural developments as virtual memory, asynchronous I/O processing, complex instruction sets, zero address architectures and machine support for block structured addressing. A more recent commercial development has been the appearance of capability machines, which, with their vast address spaces, support a longer term view of program execution and data support.

One significant software development that is closely tied to the appearance of capability machines is the concept of data persistence [AtkiM83]. This notion argues that the lifetime of data objects should not be constrained by the lifetime of the program execution that creates them. Such a concept eliminates much of the programmer's task, as he does not now have to spend time and code converting data from internal to external storage representations. A large part of the I/O burden on programs is removed, as a persistent data object maintains its representation outside of the program which created it, and therefore can be easily retrieved and accessed by subsequent programs.

A consequence of persistent object management is that object-address bindings become far more important than in conventional systems. While it is possible to manage these bindings in a localized system (because all bindings are known, or can be deduced from the structures themselves), once the bindings become distributed, management is far more difficult.

For example, suppose an object a has been created. It will be allocated some storage, and thus the address y of this block of storage is bound to the object. If this address y is used to reference the object a, we must ensure that y remains bound to a for the lifetime of references to a. Alternatively, if a is rebound (by, say, a compacting garbage collector), we must update all references to a that use the now out-of-date address y. While this is possible in a closed system, if ever y is exported from the system, we are immediately constrained to preserved the binding y to a for the lifetime of a, or alternatively that any reference to y is automatically remapped to the new address of a. Indeed, we must also ensure that if subsequent references to y cannot be excluded, an error mechanism is invoked if an attempt to reference a through address y is made. This effectively rules out the re-use of y as an identifier for any other object, and is the primary motivation for capability style architectures, which can guarantee the non-reuse of addresses.

Whilst the capability machine is an ideal vehicle upon which to implement persistent languages, its large address space is also a disadvantage. It has been demonstrated in the construction of large software systems that complexity can only be conceptually handled when the system is broken into smaller, more manageable segments. It is argued that, while some of the management of the large address space can be done automatically, a localization of address space, supported by the architecture, will more closely model the structures required by the software designer, with attendant benefits of reduced system complexity.

The context sensitive architecture is a proposal to provide a localized addressing mechanism, so that address space management may be decoupled from address space bindings. It is argued that the addressing of objects is shown to be better served by the management of small, local address spaces, which bind to a large persistent stable store mechanism [BrowA86]. The size and distribution of these address spaces is entirely at the discretion of the programmer or compiler writer, who could provide an address space for anything between a complete system and a Pascal record style data structure. This can be provided for the same cost as existing (virtual) memory management schemes, and provides a more congenial environment for the construction of persistent programming environments.

Note that it is not the intention of this paper to reject previous addressing mechanisms. Rather, it should be seen as an attempt to build upon the strengths of such designs, and to improve the ability of an architecture to deliver the addressing resource in a form suitable for the application being programmed. In particular, little or no new technology or hardware is involved: it is proposed to implement a pilot scheme using existing commercial machines. What is involved is a recasting of the distribution of address conceptualization, but not functionality, between the hardware and software domains.

Objects

The objects that we wish to deal with in computing can be classified broadly into two groups. On the one hand are the objects that can be represented with relatively few bits of information, and are fixed in size. On the other hand are the objects whose representation is not known *a priori*, whose size is more than a few bits, or whose length may change dynamically. Note that we may include in this group objects traditionally regarded as primitive types, such as strings, because the requirement that a string's size may change dynamically forces a memory management policy upon us.

Into the first group go the basic types: boolean, integer, real and pointer. In the second group goes everything else: vectors, structures, procedures, and strings. These objects are included in this group because they all involve some form of address or storage management. Such storage management entails an explicit binding between the object itself, and the store address, in order to access the object in question. Those objects whose representation or length may change dynamically clearly require the ability to rebind the object identifier to the object representation, and hence store address. And even where the size and representation of an object is known at compile time, it is more efficient to store its representation in some fixed location, and pass its address around as a handle to the object, than to pass the entire object around. It is just this phenomenon that leads to the use of descriptors for structured objects in most high level language architectures and on machines like the B6700 [OrgaE73] and MU5 [KilbT68]. Because of this representation, all these objects must be manipulated by passing pointers to the object, and from this representation arises the main difficulty in the persistent management of objects. A pointer cannot be a reference within an address space, if the possibility of passing that pointer outside of the address space exists.

For this reason, we introduce a new object: the *context*. The context is a local address space, containing a collection of objects of interest. Note that we allow the possibility of recursive contexts, that is, contexts may contain objects that are themselves contexts. Within the local address space defined by

a context, objects are identified by an index into the space. This index is persistently bound to the object, that is, it cannot be re-used for any other object within the context, for the lifetime of the context. These indices are called object identifiers, to distinguish them from the addresses used at the representational level. We shall call addresses store identifiers, to highlight this subtle, but important distinction. The thesis of this paper can then be simply stated:

> *objects should only be retrieved by object identifier, never by storage identifier.*

Context Sensitive Computer Architecture

A computer architecture is itself a binding. It binds decisions made by the computer architect into hardware, or at least microprogram code. Some of these binding decisions may be deferred, but in general, most of them are static bindings in the sense that they are made when the architecture is defined. For example, the size of an integer data type is usually fixed by the architecture; the programmer may be free to choose from a limited range of integer types, but he is not free to choose either the length or representation of an integer type.

The binding of store identifiers to objects is made at compile time, but in reality is forced by the early binding of the addressing mechanism in the design of an architecture. In particular, the provision of a large, flat address space, and the lack of suitable address space structuring mechanisms, means that the compiler writer has little choice but to bind his objects directly into the machine address space. What is desirable is an architecture that allows bindings to be made at arbitrary stages of the program and data structure creation process, depending upon the price or functionality required by the programmer, or the application.

The B1700 architecture provides an environment where many binding decisions about architectures may be deferred. Its philosophy is that

> *... the effort needed to accomodate definability from instruction to instruction is less than the effort wasted from instruction to instruction when one system design is used for all applications.* [WilnW72a]

It attempts to provide not a general purpose architecture, but a microprogrammable architecture designed for the specific purpose of emulating special purpose architectures. All languages on the B1700 are implemented by compiling then into a specially tailored intermediate code, which is then interpreted by an interpreter written in this emulator specific microcode.

The B1700 is also a notable exception for another reason: it provides, in the hardware, the ability to change integer lengths and representations, dynamically. However, neither this type binding, nor the binding of intermediate code architectures, is fully exploited: all interpreter systems for the B1700 bind both architecture and integer representation at interpreter construction time, which is not a great deal later than in conventional architectures.

It was in attempting to defer this architectural binding that the Context Sensitive Architecture was first proposed [HursA82]. The context sensitive architecture took as its design philosophy the theme that

> *The binding between architecture and language is too static and inflexible; a more desirable system is one in which bindings between machine and language objects can be made at any stage of the system process, even up to the instant of use of an object.*

The binding between source and target language is normally made by a compiler. The target language is in turn bound by the design of the target architecture. On the B1700, the target language is bound

by the design of the compiler, not the architecture, and thus occurs at a later stage of the system design. In a context sensitive architecture, the target architecture can be bound *during execution* of the program, and thus occurs at the latest possible stage.

For example, suppose a program is characterized by alternating bursts of I/O activity and compute-bound activity. When executing the I/O phase, a context sensitive architecture can use an instruction set tailored to I/O operations. When executing the compute phase, I/O instructions are not needed, and a different, arithmetic oriented architecture might be employed. Or the width of a data type (such as integer) might be chosen interactively, as a program execution proceeds, or perhaps as further accuracy is required. Of course, some discretion in the use of these mechanisms must be observed, as the indiscrimant use of such dynamic bindings may entail a severe performance penalty.

One result demonstrated by such an approach is the reduction in static and dynamic program size over existing architectures, and indeed over sizes claimed for the B1700 itself [WilnW72b]. In particular, the peak dynamic memory bandwidth required (a bottleneck in von Neumann style architectures) could be reduced by up to 50% [LokaC84].

The context sensitive mechanism can be used to implement the Direct Execution architectures of Flynn [FlynM83]. In a direct execution architecture, the architecture changes at procedure invocation time. It is argued that such decisions should be decoupled from source level artefacts, and should instead be bindings made quite independently of other structures, and orthogonal to other bindings. In this way, the software system designer can choose bindings that reflect what he wants, rather than what the computer architect thought he might want.

In principle, any binding made by an architecture can be deferred in a context sensitive architecture. In this paper, we shall be looking at just one such binding: the object-address binding.

Addressing Paradigms

The flat address space.

A problem suffered by many architectures is that they provide an address space which is at the same time too flat and too small. A flat address space suffers from the disadvantage that it does not model the way in which programs are written. A compiler must often go to some effort to map object references into this flat space, and sophisticated run time support systems are often required to maintain an addressing environment.

A further constraint is the size of the address space. Many architectures have gone through some revision process in order to extend the original address space, as software development outstrips the resources that are supplied by the hardware. The development of the VAX-11 architecture from the PDP-11 architecture is one such example.

Paged virtual memory is a technique that partly overcomes a limited address space. By providing an extended address word, and a mapping mechanism between the CPU and real store, the programmer sees a "virtual" space that can be much larger than the actual "real" space provided on the machine. This mechanism is so powerful that it is now used on many modern computer architectures.

The most serious criticism of the flat address space is however, the problem of maintaining object-address bindings, as outlined above. We shall return to this point.

The capability address space.

Capability machines provide very large address spaces, much larger than could ever be realistically used, and exploit this large address space to provide a further range of resources, such as protection and secure typing of data. With such a large range of addresses, there is no longer any problem about reusing addresses, and a new address may be assigned to every new object created during the lifetime of the system. Allied to this is the need for some sort of hardware address mapping mechanism, so that object identifiers (drawn from capability address space) can be dynamically mapped onto store identifiers. However, the introduction of large address spaces requires a correspondingly large address word (up to 128 bits), and does not entirely solve the problem of a flat address space.

The segmented address space.

In order to provide an address space with more structure, segmentation is used. This is implemented by dividing the address word into two components, thus giving a two dimensional address space. The distinction between paging and segmentation is often misused, or misunderstood. A paged system does not provide a two dimensional address space. For example, addresses 1FFF and 2000 (hexadecimal) will always be contiguous in a paged system, but not necessarily so in a segmented system, as they may reside in different segments. An object may reside across page boundaries, but never across segment boundaries.

Segmentation is a powerful mechanism, and was used to significant advantage in the Multics [CorbF65] system to provide both protection and dynamic binding mechanisms. It has also been used as a means of realising virtual memory, such as in the B6700 and MU5 designs. More recent systems, such as MONADS-PC [AbraD85] and Poppy [CockW85] provide segmentation as a component of capability-based designs. Our model of context addressing owes much to existing segment designs.

The distributed address space.

The fundamental weakness with all these systems, however, is that they all place too much reliance on a knowledge of the address space. In order to reference an object, its address must be known. Within a local environment, this is not a particularly serious problem. Addresses can be made known to the system for the purpose of storage management and garbage collection. If an object is moved around in store, its address can be updated. The binding between object and address, usually established at compile time, may be modified by the operating system, as long as the references to the object (the uses of its address) are known to the operating system. Thus there is a system wide management of addresses, and uniform treatment of address space.

The availability of cheap computing power has lead to a growth of interest in distributed systems, where each site on a network may have its own addressing domain. Traditionally, such systems make a distinction between local address space, and distributed address space. Accessing distributed space may be more costly than accessing local space, either because of hardware constraints, or because of programming constraints. Different address paths are required, and often these differences are carried up into the software design.

Once the system becomes distributed, the management of object-address bindings becomes exceedingly difficult. An address passed out of a local environment must either be preserved so that the binding between object and address is maintained, or it must be remapped when re-presented to the environment.

Note that by "distributed", we do not mean just a distribution in space. Distribution in time of object-address bindings will also lead to similar problems of address preservation. Recent work in persistent programming [AtkiM84] has shown that a "stable store", in which the binding between objects and addresses is preserved over different executions of the same (or different) programs, is desirable.

A related problem to the preservation of object-address bindings arises in connection with the modularity of large software systems. In order to avoid re-compiling such large systems every time a change is made to some (potentially small) component of the system, the system is sub-divided into modules of manageable size and complexity. When an individual module has been changed and re-compiled, it is linked into the rest of the system by adjusting all external references to point to the new location of existing objects.

This problem is just one of extended identifier-object bindings, where identifiers are bound to objects at object creation time, and the identifiers must be rebound to the new addresses assigned to the objects as they are re-linked together. In other words, a system that provides a persistent name space can also be used to provide re-linking and modular construction mechanisms. Such a result has indeed been demonstrated by Atkinson and Morrison [AtkiM85].

Contextual Addressing

We may paraphase the problem of object addressing as the problem of identifying objects within some domain. Let Θ denote an object within some such domain. Then we can denote the object domain by $\{\Theta\}$, the set of such objects. A particular object Θ can be identified by attaching (*binding*) an identifier θ to it. We shall represent this binding by the notation

$$\theta \approx \Theta$$

read as "θ binds Θ". From the identifier θ we can retrieve or *reference* the object Θ (but not necessarily vice versa). The identifer θ is called an *object identifier*. If $\{\theta\}$ denotes the set of such identifiers, then provided that the identifiers are mutually distinct:

$$\forall\, i,j \quad i \neq j \Rightarrow \theta_i \neq \theta_j$$

we can use the set of identifers as unique references to the objects, and write this labelling as a mapping

$$\{\theta\} \rightarrow \{\Theta\}$$

read as "the identifier set θ maps to the object domain set Θ". We shall define a domain of objects identified in this way as a *context*. A context may be considered as a convenient grouping of objects, related by some theme. That theme may have some higher level meaning, or it may be just that the objects have a common identifer domain.

The actual form of θ has not been specified: any symbolic convention can be used. In programming use it is normally a character string. However, any mutually distinct symbols (according to some grammar) may be used, and in particular, since the objects Θ are unordered, we may impose an ordering

$$i \approx \Theta_i$$

upon them, and use the cardinal number i as an identifer for object Θ. The identifer i we shall call a *local identifier*, as it can only be distinct within a particular domain, or *local context*.

The actual process of addressing an object, given an identifier i, requires the extraction of the object Θ from the object domain $\{\Theta\}$ so that $i \approx \Theta$ is satisfied. For non-ordered domains of i, this requires

a search of the identifier space, but for ordered domains this search is acceptably fast. In particular, for ordered, contiguous domains, the search time is of order unity. This form of addressing is known as *random access*.

When the objects are words in store, then the local identifiers are *addresses*, and we obtain the conventional model of addressing a computer store. But not all computing objects are words. We wish to impose some higher level structure upon this memory organization, so that it models more closely the structure of arbitrary computing objects. Usually, this task is performed by a compiler: it translates the local identifiers of the source program into the local identifiers of the target memory. Source identifiers are represented as character strings, memory identifiers as addresses. The compiler chooses representations for the objects (based upon some translation rules according to the primitive structuring), and selects local identifiers from the store context. There are therefore three levels of addressing:

1 The *source* domain. Local identifiers are the source identifiers. Let $\{S\}$ be the set of source identifiers. The programmer determines $\{S\} \rightarrow \{\Theta\}$.

2 The *object* domain. Local identifiers are *object identifiers*. These are not contiguous, but are *selected* store context identifiers. Let $\{O\}$ be the set of object identifiers. The compiler determines $\{O\} \rightarrow \{\Theta\}$

3 The *store* domain. Local identifiers are *store identifiers*, or addresses. Let $\{A\}$ be the set of store identifiers. The compiler writer determines $\{A\} \rightarrow \{\Theta\}$. Note that $\{A\} \supset \{O\}$ and $\{O\} \rightarrow \{A\}$.

Two distinct domains, object and store, share a common identifier domain. This causes problems whenever a reference to an object must be passed out of the local context to some other context. Normally, such object reference passing is done by passing a store address (store local identifier). But if the representation of objects is to be changed for any reason, such as for garbage collection, program swapping, or virtual memory management, we must be careful to remap the store local identifier. This is realized by a variety of mechanisms, depending upon the application.

For example, when garbage collecting, all object identifiers must be marked or tagged as such, so that they may be rebound to the new store identifiers after completion of the collection. Virtual memories provide two address spaces, real and virtual, and hardware performs a mapping from virtual to real so that any alteration in the use of real addresses as store identifiers is hidden from the use of virtual addresses as object identifiers.

Such an organization is all very well if we are concerned with only one address space, or we can map all objects into the one address space. However, when we wish to distribute our address spaces, either in space, such as with a multiprocessor organization, or in time, such as with a persistent organization, it is not convenient or realistic to flatten the combined object domains into one flat address space. This is because each sub-address space has local management considerations that should not impinge upon the management of other address spaces. What is required is some mechanism for structuring the address domains so that local addressing does not affect external addressing of the object.

Accordingly, we propose a model of addressing called the *contextual model*. Each addressing sub-domain is free to manage store identifiers in any way appropriate to its application. These addressing domains are called *contexts*. Outside a context, the only addressing information visible is object identifiers. Store identifiers are specifically not exported from a context. Of course, an object may itself be structured, and have some nested addressing structure. Whether this internal structure is revealed is a matter for local management.

Fundamentally, this model is similar to the segmentation model of addressing. Like segmentation, it requires a flat address space to identify the contexts themselves. Like Multics, it provides a dynamic binding between object referents and the objects themselves. It differs primarily in its organization of addresses, and in the way in which addresses are passed between contexts. Because we proscribe the use of local addressing information, the only non-local addresses allowed within a context must be contextual addresses.

The contextual addressing model is hierarchical, but addresses are not constrained to be tree structured. Local identifiers may be created within a context for referencing arbitrary objects in arbitrary contexts. However, since there is always a unique creator for any given object, each context can be placed on a tree that represents the ancestral tree of creation. We can therefore identify a distinguished path for an arbitrary structuring of contexts that represents the ancestral path of creation. This path must be tree-structured.

A context contains a linear address space, of local identifiers i. Let us denote a particular context A as the group of objects $\{\Theta_a\}$ with local identifiers a, $0 \le a < |A|$ where $|A|$ is the number of objects in A.

Within this context can be objects that are themselves contexts. These objects are called *nested contexts*. Let B be a nested context in A, $B \in \{\Theta_a\}$. Then

$$\exists a : a \approx B$$

Now, if we wish to access an object Θ_b with local identifier b within the context B, we can reference it from within the context A by writing

$$a \downarrow b$$

where the operator $\downarrow$ is a *qualified identifier* operator, and serves to link object references through nested contexts. A phrase such as $a \downarrow b$ is called a *qualified identifier*.

Note that such addressing is entirely relative. The qualified identifier $a \downarrow b$ will refer to object Θ_b only in the context A. Within context B, the local identifier a may be bound to an entirely different object from B. The correct reference from within context B is of course just b.

Where multiple address spaces exist in this way, we must be careful to distinguish exactly which address space is meant. We shall use *within* an address space or context to mean the address required to distinguish between objects contained by a context, and *outwith* an address space or context to mean the address required to distinguish between objects of which the current context is a member. In figure 2, local identifiers d and e are within context B, and identifiers a, b, c are outwith context B.

Directed Graph Representation of Contexts

Contexts as we have just described form a directed graph. Objects are represented as nodes on this graph, and local identifiers select edges from nodes that correspond to context objects. Two types of edges are possible:

a) edges that select objects whose representation lies in the domain of the context represented by the node from which the edge emanates.

b) edges that select objects whose representation lies in some other domain.

Edges of the first kind may represent local (nested) addressing information only, and thus an addressing environment formed with edges of this kind is tree structured. Edges of the second kind

must represent contextual addresses only, and an addressing environment formed with edges of arbitrary kind is a directed graph. Leaf nodes are objects other than context objects.

Flat Addressing

Edges radiating from a node form an addressing domain, corresponding to the local identifiers that they represent. Objects within this addressing domain can therefore reference each other directly, by the direct use of the appropriate identifier. This is called *flat addressing*, as it occurs in a single, linear, flattened address space. This is the form of addressing used in von Neumann style architectures.

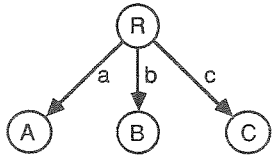

Figure 1: Flat Addressing

For example, in figure 1, R is a context containing objects A and B, with local identifiers a and b respectively. A can address B directly, and vice versa. If a third object C is introduced, it is addressable directly from either A or B via identifier c.

Context Addressing

Where objects are not in the same address space, contextual addressing must be used.

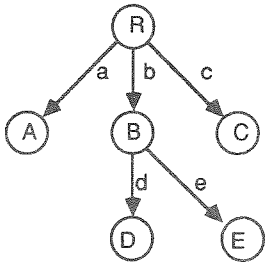

Figure 2 : Contextual Addressing

In figure 2, R is a context containing objects A , B and C , with local identifiers a , b and c as before. Now however, B is a context containing objects D and E , with local (with respect to B) identifiers d and e. Within context R, objects A , B and C can be directly addressed, but objects D and E must be addressed indirectly, as $b \downarrow d$ and $b \downarrow e$ respectively. From any object Θ in the system, addressable objects are those objects whose nodes are reachable from the node containing the address of the ob - ject Θ . That is, they must lie in an address space outwith Θ, and formed by arbitrary concatenation of qualified identifiers. Clearly, if we wish to address an object Θ' not reachable from Θ, we must extend the mechanism in some way.

This is done by introducing the parent address operator, $\uparrow$. Successive applications of the parent address operator serve to remove the addressing context one step closer to the root of the context tree

(always following the context creator, not other potential context paths). For example, in Figure 2, we could address C from D as $\uparrow c$. The local identifier c is interpreted in the context R, not B. Strictly speaking, this is not an operator, but a local identifier, whose binding is always to the parent context. However, it behaves rather like an operator, serving to recast the meaning of the following identifier from the current context to the parent context. If used as an identifier, the above reference to C would be written more correctly as $\uparrow\downarrow c$.

Alias Addressing

Addressing as provided by the hierarchical scheme above is a powerful and conceptually simple model. But it is restrictive. Non-local addresses can become unwieldy. If we must provide a long chain of qualified identifiers for each access, the process of dereferencing an object could be expen - sive. Sometimes this cost must be paid: if an object is not in the current addressing domain, there is probably a good reason for this, such as it is only reachable over a transmission line, or there is a low speed data path between the two contexts.

This is not always true. A context can be created to model some logical structure, rather than a phys - ical structure, and it is important that the cost of accessing such logical structures be kept as low as possible. Thus we relax the hierarchical constraint, and allow contexts to directly address other, non-nested contexts. This corresponds to allowing non-tree edges in the directed graph.

Essentially, a local model of the non-local addressing domain is set up, and in this local context are stored the links necessary to establish the non-local addressing domain. Local references to the non-local object can now be represented as local references, and the cost of representing the complete contextual address information is paid only once. Such an address is called an *alias address*. If the aliased context is implemented in the same address space, an alias address is just a pointer into the aliased context. Only where the aliased context is implemented in another address space is the address path to reach that context required to be stored as the representation of the aliased address, and the cost of indirect address have to be paid.

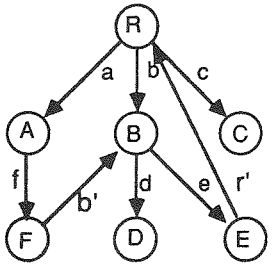

Figure 3 : Alias Addressing

For example, figure 3 shows two such aliases, b' from context F, and r' within context E. Objects within F may address B as b', rather than as $\uparrow\uparrow b$, and objects in E may similarly refer to r', rather than $\uparrow\uparrow$. Unix afficionados will note that this mechanism is identical to the file link structure.

It is possible to establish *multiple parents* once we allow aliasing. For example, in figure 3 C may set up an alias to context D. Then D will have the real parent B and alias parent C. Only the real parent can be reached with the parent address operator.

Examples of Context Addressing

Let us look at how context sensitive addressing can be used in practice.

First of all, note that block structured addressing is just a special case of context addressing, where we are constrained to use a purely hierarchical tree of contexts. Each block is represented as a context, and objects declared in that block will have a representation stored in the context's address space, and will be identified from other objects in the block by a local identifier, corresponding exactly to the ordinal position of the declaration within the block. A master context is used to hold all currently addressable blocks, and the local identifier within this context exactly corresponds to the lexical level, or static declaration depth, of the block to which it is bound. Above this, a further context is used to hold all blocks currently in existence, and models the stack used to implement block structured addressing. Figure 4 shows a conventional display and stack based addressing scheme, with a set of display registers to indicate the current addressing environment. Frames are stored on a stack, which is represented as a linear address space.

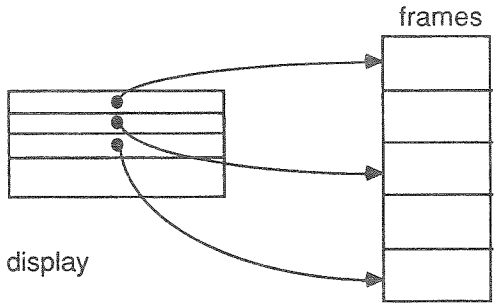

Figure 4 : Conventional block structure implementation.

Figure 5 shows how this addressing environment would be represented as a set of contexts.

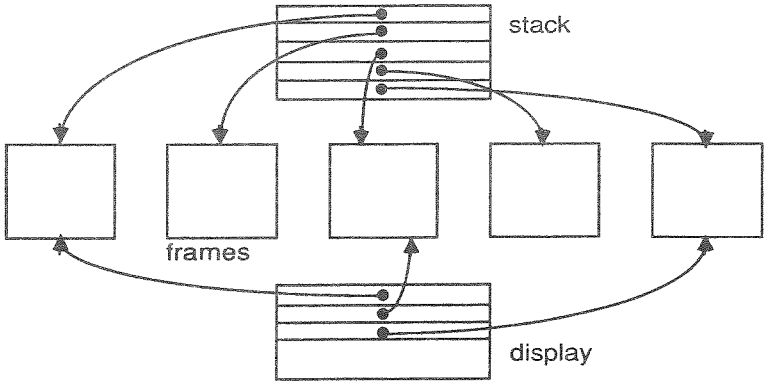

Figure 5 : Block Structured Addressing with Contexts

One of the primary applications for contexts is in supporting persistent program representations. In order to manage the large set of objects that may accumulate, some structure must be imposed upon these objects. A hierarchical structure is too restrictive, and so the contextual model can be used to advantage. For example, a suite of programs for some application area may be stored in the persistent database. When a user opens this database, he may want some of the programs and not others. Or he may want to replace some of the routines by his own versions. The context model allows him to set up an environment containing just those objects relevant to the application. Different environments can be explored by simply changing the bindings of objects within a context.

An analogy with real life may be helpful here. When asked by someone to supply some information, we might answer "It is in the manual". Which manual is implied from the context of the question. If there is more than one manual in the context of the question, we might extend our "flat" address space by further addressing information, such as "the User's Manual", or "the Reference Manual", etc. But the questioner might not have these objects in his context, so we will have to provide him with sufficient addressing information for him to find the right context. Hence we will build up some contextual addressing information, as in "the User Manual in the library", or "the User Manual in the Computer Centre library", or even "the User Manual in the University of Adelaide's Computer Centre library". Implicit in such a sequence of address widenings is that there is ultimately some common address space for both questioner and answerer. The advantage of the contextual model is that the wider addressing information can be factored out of each reference, and bound once, or as many times as required, without changing the representation. In the above analogy, this corresponds to binding the local reference "manual" to the complete address string, and then subsequently referring to just "the manual".

Such addressing patterns are not new in computing. File directory structures, as exemplified by the Unix model, provide exactly such a hierarchical domain. Electronic mail addresses are more subtle, yet include just such a model. What is new here is that we propose such an address model for object addressing at the hardware level. Implicit in this model is the notion that all contextual addresses are dynamic bindings of object identifiers to objects, just as virtual memory and capability systems are dynamic bindings of logical addresses to physical addresses. Static bindings can only be made to object identifiers, not to store addresses.

Implementation

A context is simply an abstraction of address spaces, and just as compilers can map object identifiers into store identifiers, a context can be implemented as a mapping upon a conventional random address space. Where a context is used to implement an abstract object, this is the implementation normally used. Where the context is used to implement remote objects, the address used will reflect the addressing information required to reach the remote object. At all times the address of a context must indicate a path to the addressed context from the addressing context.

There is nothing particularly difficult about implementing the context objects themselves. They are represented as linearly indexed tables, where each entry in the table describes one object. These entries are accordingly called *object descriptors*. For scalars, and objects of less than 64 bits in length, the descriptor can contain the value of the object itself. For larger objects, the value of the descriptor is a pointer (into a local address space) to the object's representation. Because the representation is stored in a local space, and because the pointer to this space can never be visible outside the context, this local space can be managed without reference to any external data structures.

For example, suppose A is a context containing B, a string, and C, a context representing a table of strings. We could represent these objects graphically as shown in figure 6.

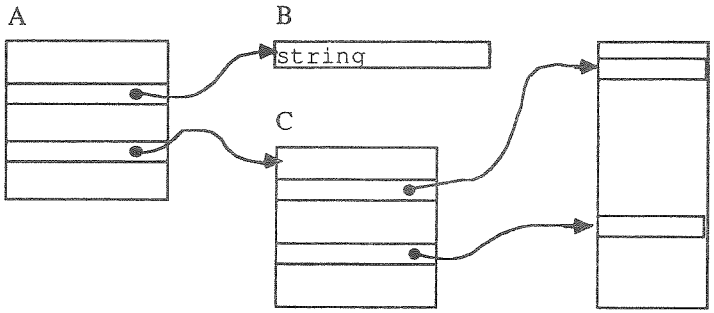

Figure 6 : Context Implementation

A has a local address space in which is stored a descriptor for B and a descriptor for C. As well, there are various address spaces used to store the representations of B and C. At this point, no assumption has been made about how these address spaces are implemented. Now suppose further that A and B are implemented in semiconductor store, and C is implemented as a disk store. All that is required of our context implementation is that the pointer from the descriptor for C contains addressing information for A to access the disk store, as well as the address of the representation of C in disk space.

Note that this does not violate our requirement that local addressing information cannot be exported from a context. The object C is in A's address space, which has been extended from the single semiconductor store to include the two address spaces. Local addresses within the context C reside within the representation of C, and are not accessible outwith C. Thus C may have a management strategy associated with it, which may rebind objects within the disk space, as long as the descriptors for these objects (wholly within C's representation) are properly updated. C itself cannot be altered by this management strategy, as it lies in A's address space.

Name Spaces

The context addressing model is also intended to provide a solution to the implementation of name spaces. A name space is a collection of identifier - object bindings, and is best thought of as a run-time representation of a symbol table. A symbol table is a structure used by a compiler to establish and record source identifier - store identifier bindings, and it has two basic operations: insertion and lookup. Insertion occurs upon declaring a new identifier in the current scope, and lookup occurs whenever an identifier is referenced. Most implementations of compilers discard the symbol table information once the source program has been compiled, but a few, notably those providing run-time symbolic debugging or monitoring mechanisms, maintain the symbol table information in some form or another. Where a compiled program may be linked with another, some of the symbol table information may be preserved in order to resolve those linkage-time bindings. Name spaces are simply an extension from the compile-time and linkage-time symbol tables into the run-time environment. That is, the binding time of identifier - object pairs is (potentially) delayed to the latest time possible.

To build a name space using contexts, we provide a context for the objects themselves, called the *object context*, and a separate context, called the *name context*, to hold the names. These could be stored as strings. The mapping between the names and objects is defined by the local identifiers used

in each context, so that it is 1:1. Other schemes could also be used. To resolve a binding between name and object, the name is searched in the name context, and the local identifier used in that context will be the local identifier in the object context.

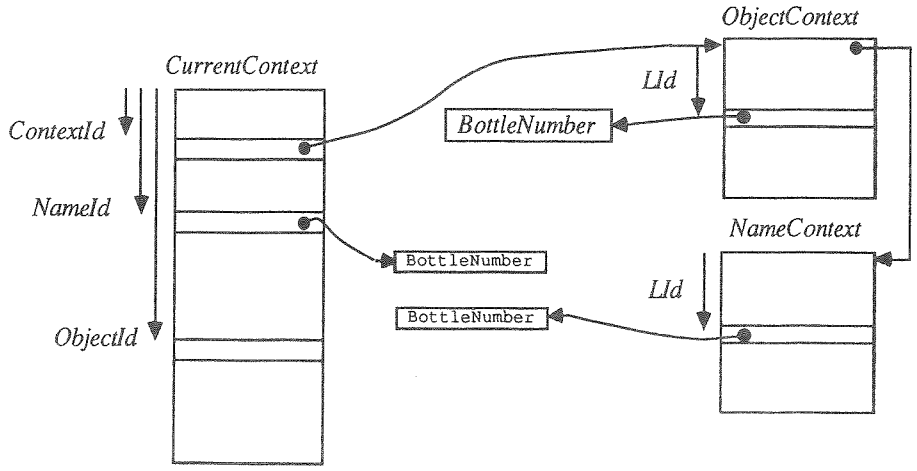

Figure 7 : Resolution of Name Space bindings.

For example, figure 7 shows how a reference might be resolved. Identifiers in *Italic* represent local identifiers, while identifiers in `Courier` represent strings. Let *NameId* be a local identifier (in the *CurrentContext*) bound to the object "`BottleNumber`" (a string). Similarly, let *ContextId* be a local identifier bound to the context object *ObjectContext*. Finally, note that *ObjectId* is a local identifier whose value is to be set to the context address for the object *BottleNumber* in context *ObjectContext*.

In order to perform the binding, suppose there is an instruction "**bind** *Object* **to** *Name* **in** *Context*", which has the effect of binding a local identifier *Object* to the object in *Context* that has source identifier *Name*. In the above example this would become "**bind** *ObjectId* **to** *NameId* **in** *ContextId*", which executes by searching the name context *NameContext* (associated with *ContextId*) for the string *NameId*, and on succesful match, replaces the value of *ObjectId* with the context address <*ContextId*, *Lld*>, where *Lld* is the local identifier found for the match. This object may be used subsequently to retrieve and modify the object *BottleNumber* by dereferencing the context address. If no match is found, a run time error is signalled. The mechanism can be extended to allow arbitrary context addressing.

For collections of persistent objects, called databases, resolution of references into the data base can be done symbolically, either by the compiler, where the object name is known at compile time, or by the system itself, at run time, if the name is only known dynamically. Note in particular that there is little overhead involved, and a static binding, besides being only performed once, does not incur any run-time penalty. In fact, once the binding is resolved, references to the object proceed just as quickly, regardless of whether the binding was done statically or dynamically.

Further Research

Details in a paper such as this are necessarily sketchy. In some circumstances, this lack of detail corresponds to detail yet to be defined, and work is under way on describing a complete implementation of a context sensitive architecture, employing the features described above.

Note that contexts avoid naming all objects in a single flat address space. However, it can be argued that this does not really solve anything, as now the contexts must be named in an address space that is itself flat as far as the contexts are concerned. That is, each context must agree on a common address space in order to pass contexts around.

It is thought that this is true where efficient implementations of address spaces must be employed. Two aspects of this phenomenon are relevent. Where contexts reside in a common system, they may be realized as objects in an underlying flat space. However, it is not essential that they are. On the other hand, where objects must reside in separate address spaces, then it is essential that they use a common address space in refering to each other, so that subsequent references (in either time or space) map consistently onto the same objects.

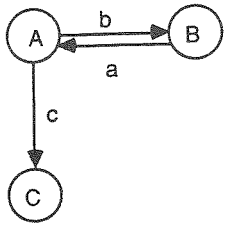

Figure 8 : Distribution in time

For example, in figure 8, A passes the address c to B. If B subsequently uses address c (in A's sense), it must refer to C.

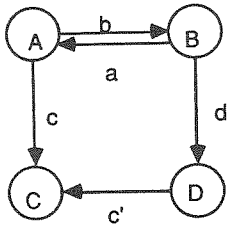

Figure 9 : Distribution in space

In figure 9, *A* passes the address *c* to *B*. If *B* passes address *c* (in *A*'s sense) to object *D*, then *D* must regard the local identifier *c* as a reference to *C*. This implies that either the contextual information be extended as an object is passed around, or that contexts have a common address domain. An implementation will demonstrate the relative merits of this.

Conclusions

Delayed bindings provide a powerful mechanism for the implementation of many dynamic objects in computing. However, implementers are often loath to use them, because of suspected inefficiencies in their use. We argue that delayed bindings can be supported at the computer hardware level, without a significant performance penalty.

A model of context sensitive addressing has been developed. This model is a powerful one, and subsumes existing forms of addressing. It provides directly at the architectural level a closer representation of the mechanism of addressing computing objects, and a means for implementing the structure of such objects. In particular, it is a convenient architecture for the exploration of appropriate structures for the support of persistent programming.

Acknowledgement

This work was done while the author was on leave at the University of St. Andrews, and was supported by SERC grant no. GR/D 47790. The author is indebted to Ron Morrison and the members of the PISA Project team at St. Andrews for their encouragement and stimulating company.

References

AtkiM83 M. P. Atkinson, P. J. Bailey, K. J. Chisholm, W. P. Cockshott & R. Morrison, *An approach to persistent programming*, The Computer Journal, vol. 26, no. 4, pp.360-365, 1983.

BrowA86 A. L. Brown, *A Persistent Stable Store*, to be published.

OrgaE73 E. I. Organick, *Computer System Organization: the B5700/B6700 Series*, Academic Press, 1973.

KilbT68 T. Kilburn, D. Morris, J. S. Rohl & F. H. Sumner, *A System Design Proposal*, Information Processing 68, p491, North Holland, 1968.

WilnW72a W. T. Wilner, *The Burroughs B1700*, AFIPS, 1972.

HursA82 A. J. Hurst & C. J. Lokan, *Context Sensitive Computer Architecture*, Proc. of ACSC-5, Perth, 1982.

WilnW72b W. T. Wilner, *Burroughs B1700 Memory Utilization*, Proc. FJCC 1972, pp.579-586, AFIPS, 1972.

LokaC84 C. J. Lokan, *Static Analysis of Programs for Context Sensitive Architecture*, Ph. D. Thesis, ANU, 1984.

FlynM85 M. J. Flynn & L. W. Hoevel, *Execution Architecture: The DELtran Experiment*, IEEE Trans. Comp. vol. C32, no. 2, pp.156-175, Feb 1983.

CorbF65 F. J. Corbato & V. A. Vyssotsky, *Introduction and Overview of the Multics System*, Proc. FJCC 1965, vol. 27, part 1, pp.185-196, AFIPS, 1965.

AbraD85 D. A. Abramson & J. L. Keedy, *Implementing a Large Virtual Memory in a Distributed Computing System*, Proc. 18th Annual Hawaii International Conference on System Sciences, pp. 515-522, 1985

CockW85 W. P. Cockshott, *The Persistent Store Machine*, Persistent Programming Report 18, Dept. of Computing Science, University of Glasgow, Glasgow G12 8QQ, 1985.

AtkiM84 M. P. Atkinson, P. J. Bailey, W. P. Cockshott, K. J. Chishom & R. Morrison, *Progress with Persistent Programming*, In *Database – Role and Structure*, P. M. Stocker, M. P. Atkinson, & P. M. D. Gray, Eds., Cambridge University Press, Cambridge England 1984.

AtkiM85 M. P. Atkinson & R. Morrison, *Procedures as Persistent Data Objects*, ACM Trans. Prog. Lang. Systm., vol. 7, no. 4, pp.539-559, Oct 1985.

Bibliography

Copies of documents in this list may be obtained by writing to:

The Secretary,
Persistent Programming Research Group,
Department of Computing Science,
University of Glasgow,
Glasgow G12 8QQ
Scotland.

or

The Secretary,
Persistent Programming Research Group,
Department of Computational Science,
University of St. Andrews,
North Haugh,
St. Andrews KY16 9SS
Scotland.

Books

Davie, A.J.T. & Morrison, R.
"Recursive Descent Compiling", Ellis-Horwood Press (1981).

Atkinson, M.P. (ed.)
"Databases", Pergammon Infotech State of the Art Report, Series 9, No.8, January 1982. (535 pages).

Cole, A.J. & Morrison, R.
"An introduction to programming with S-algol", Cambridge University Press, Cambridge, England, 1982.

Stocker, P.M., Atkinson, M.P. & Grey, P.M.D. (eds.)
"Databases - Role and Structure", Cambridge University Press, Cambridge, England, 1984.

Published Papers

Morrison, R.
"A method of implementing procedure entry and exit in block structured high level languages". Software, Practice and Experience 7, 5 (July 1977), 535-537.

Morrison, R. & Podolski, Z.
"The Graffiti graphics system", Proc. of the DECUS conference, Bath (April 1978), 5-10.

Atkinson, M.P.
"A note on the application of differential files to computer aided design", ACM SIGDA newsletter Summer 1978.

Atkinson, M.P.
"Programming Languages and Databases", Proceedings of the 4th International Conference on Very Large Data Bases, Berlin, (Ed. S.P. Yao), IEEE, Sept. 78, 408-419. (A revised version of this is available from the University of Edinburgh Department of Computer Science (EUCS) as CSR-26-78).

Atkinson, M.P.
"Progress in documentation: Database management systems in library automation and information retrieval", Journal of Documentation Vol.35, No.1, March 1979, 49-91. Available as EUCS departmental report CSR-43-79.

Gunn, H.I.E. & Morrison, R.
"On the implementation of constants", Information Processing Letters 9, 1 (July 1979), 1-4.

Atkinson, M.P.
"Data management for interactive graphics", Proceedings of the Infotech State of the Art Conference, October 1979. Available as EUCS departmental report CSR-51-80.

Atkinson, M.P. (ed.)
"Data design", Infotech State of the Art Report, Series 7, No.4, May 1980.

Morrison, R.
"Low cost computer graphics for micro computers", Software Practice and Experience, 12, 1981, 767-776.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"PS-algol: An Algol with a Persistent Heap", ACM SIGPLAN Notices Vol.17, No. 7, (July 1981) 24-31. Also as EUCS Departmental Report CSR-94-81.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Nepal - the New Edinburgh Persistent Algorithmic Language", in Database, Pergammon Infotech State of the Art Report, Series 9, No.8, 299-318 (January 1982) - also as EUCS Departmental Report CSR-90-81.

Morrison, R.
"S-algol: a simple algol", Computer Bulletin II/31 (March 1982).

Morrison, R.
"The string as a simple data type", Sigplan Notices, Vol.17,3, 46-52, 1982.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"Progress with Persistent Programming", presented at CREST course UEA, September 1982, revised in "Databases - Role and Structure", see PPRR-8-84.

Morrison, R.
"Towards simpler programming languages: S-algol", IUCC Bulletin 4, 3 (October 1982), 130-133.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Problems with persistent programming languages", presented at the Workshop on programming languages and database systems, University of Pennsylvania. October 1982. Circulated (revised) in the Workshop proceedings 1983, see PPRR-2-83.

Atkinson, M.P.
"Data management", in Encyclopedia of Computer Science and Engineering 2nd Edition, Ralston & Meek (editors) January 1983. van Nostrand Reinhold.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Algorithms for a Persistent Heap", Software Practice and Experience, Vol.13, No.3, 259-272 (March 1983). Also as EUCS Departmental Report CSR-109-82.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"CMS - A chunk management system", Software Practice and Experience, Vol.13, No.3 (March 1983), 273-285. Also as EUCS Departmental Report CSR-110-82.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"Current progress with persistent programming", presented at the DEC workshop on Programming Languages and Databases, Boston, April 1983.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"An approach to persistent programming", The Computer Journal, 1983, Vol.26, No.4, 360-365 - see PPRR-2-83.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"PS-algol a language for persistent programming", 10th Australian Computer Conference, Melbourne, Sept. 1983, 70-79 - see PPRR-2-83.

Morrison, R., Weatherill, M., Podolski, Z. & Bailey, P.J.
"High level language support for 3-dimension graphics", Eurographics Conference Zagreb, North Holand, 7-17, Sept. 1983. (ed. P.J.W. ten Hagen).

Cockshott, W.P., Atkinson, M.P., Chisholm, K.J., Bailey, P.J. & Morrison, R.
"POMS : a persistent object management system", Software Practice and Exerience, Vol.14, No.1, 49-71, January 1984.

Kulkarni, K.G. & Atkinson, M.P.
"Experimenting with the Functional Data Model", in Databases - Role and Structure, Cambridge University Press, Cambridge, England, 1984.

Atkinson, M.P. & Morrison, R.
"Persistent First Class Procedures are Enough", Foundations of Software Technology and Theoretical Computer Science (ed. M. Joseph & R. Shyamasundar) Lecture Notes in Computer Science 181, Springer Verlag, Berlin (1984).

Atkinson, M.P., Bocca, J.B., Elsey, T.J., Fiddian, N.J., Flower, M., Gray, P.M.D.
Gray, W.A., Hepp, P.E., Johnson, R.G., Milne, W., Norrie, M.C., Omololu,
A.O., Oxborrow, E.A., Shave, M.J.R., Smith, A.M., Stocker, P.M. & Walker, J.
"The Proteus distributed database system", proceedings of the third British National Conference on Databases, (ed. J. Longstaff), BCS Workshop Series, Cambridge University Press, Cambridge, England, (July 1984).

Atkinson, M.P. & Morrison, R.
"Procedures as persistent data objects", ACM TOPLAS 7, 4, 539-559, (Oct. 1985) - see PPRR-9-84.

Morrison, R.,Bailey, P.J., Dearle, A., Brown, P. & Atkinson, M.P.
"The persistent store as an enabling technology for integrated support environments", 8th International Conference on Software Engineering, Imperial College, London (August 1985), 166-172 - see PPRR-15-85.

Atkinson, M.P. & Morrison, R.
"Types, bindings and parameters in a persistent environment", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 1-24 - see PPRR-16-85.

Davie, A.J.T.
"Conditional declarations and pattern matching", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 278-283 - see PPRR-16-85.

Krablin, G.L.
"Building flexible multilevel transactions in a distributed persistent environment, proceedings of Data Types and Persistence Workshop, Appin, August 1985, 86-117 - see PPRR-16-85.

Buneman, O.P.
"Data types for data base programming", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 291-303 - see PPRR-16-85.

Cockshott, W.P.
"Addressing mechanisms and persistent programming", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 363-383 - see PPRR-16-85.

Norrie, M.C.
"PS-algol: A user perspective", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 399-410 - see PPRR-16-85.

Owoso, G.O.
"On the need for a Flexible Type System in Persistent Programming Languages", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 423-438 - see PPRR-16-85.

Morrison, R., Brown, A.L., Bailey, P.J., Davie, A.J.T. & Dearle, A.
"A persistent graphics facility for the ICL PERQ", Software Practice and Experience, Vol.14, No.3, (1986) - see PPRR-10-84.

Atkinson, M.P. and Morrison R.
"Integrated Persistent Programming Systems", proceedings of the 19th Annual Hawaii International Conference on System Sciences, January 7-10, 1986 (ed. B. D. Shriver), vol IIA, Software, 842-854, Western Periodicals Co., 1300 Rayman St., North Hollywood, Calif. 91605, USA - see PPRR-19-85.

Atkinson, M.P., Morrison, R. and Pratten, G.D.
"A Persistent Information Space Architecture", proceedings of the 9th Australian Computing Science Conference, January, 1986 - see PPRR-21-85.

Kulkarni, K.G. & Atkinson, M.P.
"EFDM : Extended Functional Data Model", The Computer Journal, Vol.29, No.1, (1986) 38-45.

Buneman, O.P. & Atkinson, M.P.
"Inheritance and Persistence in Database Programming Languages"; proceedings ACM SIGMOD Conference 1986, Washington, USA May 1986 - see PPRR-22-86.

Morrison R., Dearle, A., Brown, A. & Atkinson M.P.; "An integrated graphics programming environment", Computer Graphics Forum, Vol. 5, No. 2, June 1986, 147-157 - see PPRR-14-86.

Atkinson, M.G., Morrison, R. & Pratten G.D.
"Designing a Persistent Information Space Architecture", proceedings of Information Processing 1986, Dublin, September 1986, (ed. H.J. Kugler), 115-119, North Holland Press.

Brown, A.L. & Dearle, A.
"Implementation Issuses in Persistent Graphics", University Computing, Vol. 8, N0. 2, (Summer 1986) - see PPRR-23-86.

Internal Reports

Morrison, R.
"S-Algol language reference manual", University of St Andrews CS-79-1, 1979.

Bailey, P.J., Maritz, P. & Morrison, R.
"The S-algol abstract machine", University of St Andrews CS-80-2, 1980.

Atkinson, M.P., Hepp, P.E., Ivanov, H., McDuff, A., Proctor, R. & Wilson, A.G.
"EDQUSE reference manual", Department of Computer Science, University of Edinburgh, September 1981.

Hepp, P.E. and Norrie, M.C.
"RAQUEL: User Manual", Department of Computer Science Report CSR-188-85, University of Edinburgh.

Norrie, M.C.
"The Edinburgh Node of the Proteus Distributed Database System", Department of Computer Science Report CSR-191-85, University of Edinburgh.

Theses

The following Ph.D. theses have been produced by members of the group and are available from the address already given,

W.P. Cockshott
Orthogonal Persistence, University of Edinburgh, February 1983.

K.G. Kulkarni
Evaluation of Functional Data Models for Database Design and Use, University of Edinburgh, 1983.

P.E. Hepp
A DBS Architecture Supporting Coexisting Query Languages and Data Models, University of Edinburgh, 1983.

G.D.M. Ross
Virtual Files: A Framework for Experimental Design, University of Edinburgh, 1983.

G.O. Owoso
Data Description and Manipulation in Persistent Programming Languages, University of Edinburgh, 1984.

Persistent Programming Research Reports

This series was started in May 1983. The following list gives those produced and those planned plus their status at 15th December 1986. Copies of documents in this list may be obtained by writing to the addresses already given.

PPRR-1-83	The Persistent Object Management System - Atkinson,M.P., Chisholm, K.J. and Cockshott, W.P.	£1.00
PPRR-2-83	PS-algol Papers: a collection of related papers on PS-algol - Atkinson, M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.	£2.00
PPRR-4-83	The PS-algol reference manual - Atkinson, M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R	Presently no longer available
PPRR-5-83	Experimenting with the Functional Data Model - Atkinson, M.P. and Kulkarni, K.G.	£1.00
PPRR-6-83	A DBS Architecture supporting coexisting user interfaces: Description and Examples - Hepp, P.E.	£1.00
PPRR-7-83	EFDM - User Manual - K.G.Kulkarni	£1.00
PPRR-8-84	Progress with Persistent Programming - Atkinson,M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.	£2.00

PPRR-9-84	Procedures as Persistent Data Objects - Atkinson, M.P.,Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.	£1.00
PPRR-10-84	A Persistent Graphics Facility for the ICL PERQ - Morrison, R., Brown, A.L., Bailey, P.J., Davie, A.J.T. and Dearle, A.	£1.00
PPRR-11-85	PS-algol Abstract Machine Manual	£1.00
PPRR-12-86	PS-algol Reference Manual - third edition	£2.00
PPRR-13-85	CPOMS - A Revised Version of The Persistent Object Management System in C - Brown, A.L. and Cockshott, W.P.	£2.00
PPRR-14-86	An Integrated Graphics Programming Environment - second edition - Morrison, R., Brown, A.L., Dearle, A. and Atkinson, M.P.	£1.00
PPRR-15-85	The Persistent Store as an Enabling Technology for an Integrated Project Support Environment - Morrison, R., Dearle, A, Bailey, P.J., Brown, A.L. and Atkinson, M.P.	£1.00
PPRR-16-85	Proceedings of the Persistence and Data Types Workshop, Appin, August 1985 - ed. Atkinson, M.P., Buneman, O.P. and Morrison, R.	£15.00
PPRR-17-85	Database Programming Language Design - Atkinson, M.P. and Buneman, O.P.	£3.00
PPRR-18-85	The Persistent Store Machine - Cockshott, W.P.	£2.00
PPRR-19-85	Integrated Persistent Programming Systems - Atkinson, M.P. and Morrison, R.	£1.00
PPRR-20-85	Building a Microcomputer with Associative Virtual Memory - Cockshott, W.P.	£1.00
PPRR-21-85	A Persistent Information Space Architecture - Atkinson, M.P., Morrison, R. and Pratten, G.D.	£1.00
PPRR-22-86	Inheritance and Persistence in Database Programming Languages - Buneman, O.P. and Atkinson, M.P.	£1.00
PPRR-23-86	Implementation Issues in Persistent Graphics - Brown, A.L. and Dearle, A.	£1.00
PPRR-24-86	Using a Persistent Environment to Maintain a Bibliographic Database - Cooper, R.L., Atkinson, M.P. & Blott, S.M.	£1.00
PPRR-26-86	Exception Handling in a Persistent Programming Language - Philbrow, P & Atkinson M.P.	£1.00
PPRR-27-87	A Context Sensitive Addressing Model - Hurst, A.J.	£1.00
PPRR-28-86b	A Domain Theoretic Approach to Higher-Order Relations - Buneman, O.P. & Ochari, A.	£1.00
PPRR-29-86	A Persistent Store Garbage Collector with Statistical Facilities - Campin, J. & Atkinson, M.P	£1.00
PPRR-30-86	Data Types for Data Base Programming - Buneman, O.P.	£1.00
PPRR-31	An Introduction to PS-algol Programming - Carrick, R., Cole, A.J. & Morrison, R.	£1.00