C** and Eiffel**: languages for distribution and persistence

Vinny Cahill, Chris Horn, Andre Kramer, Maurice Martin and Gradimir Starovic *
Trinity College,
IRL - Dublin 2.

November 21, 1990

Abstract

This paper discusses in what way a programming language may be extended for distribution and persistence. We specifically concentrate on C++[1] and Eiffel[2]. Our attention is on the programming models, together with rationale and incurred costs. The extended languages are supported by a language independent execution environment which offers distributed invocation, concurrency, mapping and unmapping of objects, dynamic link loading, load balancing and garbage collection. However we do not here consider in detail the internal mechanisms of our support environment.

Parts of the work presented here have been partially supported by the CEC, under Esprit project 2071, Comandos.

An extended version of this paper has been submitted for publication: PLEASE do not reference this paper without the explicit permission of the authors.

This version of the paper was presented at the microkernel applications workshop, held at the OSF research institute, Grenoble 27-29th November 1990.

1 Motivations

Together with a number of other institutions in the Esprit Comandos project, we are constructing an execution space for persistent and distributed programming[16][7]. The layer produced by the project is for Unix and microkernels such as OSF-1 Mach and Chorus. It is language independent, and provides object management services such as distibuted invocation, persistence, dynamic link loading, concurrency and synchronisation, load balancing and garbage collection. One of our major concerns is that popular languages should be supported in our environment: consequently for a given language, we may either have to extend it with distribution and/or persistence facilities, or map its own support for these facilities onto our execution layer. We discuss in this paper our designs for evolving in particular C++[23] and Eiffel[18].

The widely accepted mechanism to extend a programming language to enable distributed applications to be written is the remote procedure call; with an associated stub generator for that language and runtime support to provide a generic interface to the appropriate network protocol stack. The attractiveness of RPC is that it is now well understood and accepted, because it isolates a programmer to a reasonable degree from the specialised understanding required to use a particular network [3]:

> "Our hope is that by providing communication with almost as much ease as local procedure calls, people will be encouraged to build and experiment with distributed applications."

In this paper we concentrate on the issues in extending the invocation mechanism of a programming language, for both distributed invocation and invocation of objects potentially resident on persistent storage. While we are confident that our support is applicable to a range of programming languages, here we focus specifically on both C++ and Eiffel. In extending these languages for distribution and persistence our concern is, in the first instance, to maintain their model of synchronous invocation of objects while maintaining the general philosophy of making cost explicit.

We do nevertheless acknowledge that there are some well recognised limitations to the classical RPC model. We accept the contribution that proposals such as futures[9][25], promises[15], REV[22] and trading[10] can make but, as a first step, we are more

*e-mail: horn@cs.tcd.ie
[1] AT&T v2.0
[2] v2.2

concerned with supporting the execution models of our chosen languages.

We have emphasised object oriented languages, not only because of their current popular appeal, but also from our own pleasant experiences in building applications and tools from them. We believe that they are a promising approach to designing and building distributed applications.

2 Transparency

A number of research projects have explored the possibility of hiding the boundaries between local and remote virtual address spaces, so making cross context invocation syntatically equivalent to intra context invocation [4][5][12]. The proponents suggest that programmers are relieved of distribution concerns, and can write (re-useable) code which is independent of its actual use in a distributed application. In turn, application builders and system administrators can decide how best to structure an application and its data over several nodes, including the ability to modify this mapping without having to rewrite, recompile or even relink the applications.

The contrary approach is to make the programmer blatently aware of the difference in cost between local and potentially remote invocations. This style is advocated in, for example, Argus[14]. It is also a natural consequence of the addition of a standard RPC package to an existing language: remote procedures can be invoked somewhat similarly to local ones, but the programmer is acutely aware that she is invoking a remote procedure. One argument in favour is that a programmer is frequently in an excellent position to supply advice to the execution environment and should therefore be able to specify which interfaces could be RPC invokable, and which should always be locally resolved without RPC calls.

At the level of our language independent execution environment, our philosophy in respect of transparency in any particular programming language is to be neutral. However when extending any specific language – such as C++ or Eiffel – above our execution environment, our approach is to try to reflect the philosophy of that language. We believe our execution environment can support both transparent and non-transparent distribution. We do however note that prototypes which have explored the transparent approach are, to our knowledge, limited in each case to a single language, with a tight coupling of its runtime support.

One approach towards providing a transparent distributed environment which is receiving some attention is distributed shared memory: its application to C++ is reported, for example, in [5]. Our feelings about DSM systems are quite mixed: on the one hand they provide a transparent environment, and allow even a language as complex as C++ to be reasonably transparently supported; on the other, despite initial experiments such as [8][27] and [24], we remain to be convinced that DSM can scale, address heterogeneity, fault tolerance and multiple users.

Our own execution environment currently does not normally provide a mechanism for reading and writing arbitrary bytes of an address space context over a network – there is no assumption about the availability of distributed shared memory. Thus mechanisms are restricted to invocations and (distributed) thread creation.

3 The basic model

We start here by considering the various issues and alternatives for accessing objects in a distributed environment, and proceed in section 3.1 below to give the solutions we are adopting.

In both C++ and Eiffel, accesses to objects can be in two ways: by pointer (including also constant pointers such as C++'s references)[3] or directly (ie "inline" – a member object in C++ or a direct object from an expanded type in Eiffel). Since we do not assume distributed shared memory, in our distributed environment a member/direct object cannot be remote from its encapsulating object.

If, as a result, the only mechanism for accessing remote objects is via pointers, the next issue is whether, and if so how, pointers to remote objects should differ from those to local objects. By *remote objects,* here we mean objects that may potentially be in another context. *Local objects* are on the contrary guaranteed never to be separated into a different context from those objects which hold pointers for them.

If pointers to remote and local objects are fully interchangeable, then a transparent model with respect to distribution results. For C++, but less so in Eiffel, the programmer is normally aware of the cost of each language construct[4]: thus for these languages we feel on the contrary that the cost of remote access should be apparent. How then should local and remote pointers be syntatically distinguished ?

[3] Although note that in Eiffel, pointers are not explicit.

[4] But note that in Eiffel, an attribute access is indistinguishable from a parameterless routine access of the same name.

One possibility is to distinguish remote pointers by, for example, using an "@" rather than a "*" in C++, or by using "smart" pointers. As a result for each class, there can be both local and remote pointers for its instances. Remote instances of a class require that RPC stubs are used to marshall arguments and dispatch invocations for each remotely invokable routine[5]. If any class can potentially have remote instances, RPC stubs must always be available for it. Our view is that this can sometimes be an unnecessary overhead, particularly during compilation – generating these stubs when compiling the class definition.

A further issue is assignment between local and remote pointers. In general a local pointer cannot be updated with the value of a remote pointer: the object referenced by the remote pointer may naturally be remote, and a subsequent access to it via a local pointer would not use the RPC stubs. It is conceivable to insert a runtime check to test for proximity, and to allow the assignment if the target object is actually local. However this would be complicated by the object consequently migrating away.

In a non object oriented world, simply associating "remoteness" with typing information would presumably be adequate to prevent the two categories of assignment above: remote types would not be assignment compatible with local types, and vice versa. However, for C++ and Eiffel at least, the situation can arise due to subtyping: in such languages an instance of a subclass can be used for an instance of one of its superclasses. Hence in principle a pointer to a remote derived class could be assigned to an instance of a non-remote base class, and vice versa. One option would be to prevent mutual subclassing of the two categories of classes: this would then lead to two completely separate class hierarchies[17]. We feel that this would be unfortunate since it would discourage re-use of code – for example a non-remote class could not easily be converted into a remote one via the obvious derivation.

We argue that inheritance between non-remote and remote classes and vice-versa should be provided: however assignment of local and remote pointers should be normally mutually incompatible.

3.1 Accessing remote objects

Based on the above discussion, we believe that – for Eiffel and C++ at least – it is wise to allow the designer of a class to specify whether or not remote instances of it are possible. RPC stubs are then generated only for classes which may have remote instances. We call such a class a *global class*. Note that an instance of a global class is only potentially remote; it may actually be local. It is also possible that an instance of a global class may migrate from one node to another in the distributed system – its location is not necessarily fixed.

Given a pointer to an instance of a global class, what categories of features can be accessed ? In C++, a public member of a class can be any designator – eg a function, a simple type, a class instance, or a pointer (or array) to a type or class instance. Obviously member functions should be accessible via a remote pointer, provided that the arguments and results of the function can be marshalled by rpc stubs (cf section 4): we call such functions *global functions*.

In Eiffel, an exported feature can be a routine, or an expanded or reference attribute. The basic types are all expanded classes. Thus an Eiffel routine can be a *global routine* in our environment if it can be marshalled.

In the absence of a mechanism to read or write arbitrary bytes of a remote address spaces (eg by de-referencing and offsetting from a remote pointer) we advocate that in addition to global functions and global routines, only member instances (in C++) and instances of expanded classes (in Eiffel) of global classes should be accessible via a pointer to a global instance: we call such a member a *global member object* or *global direct object*. That is we impose no restrictions on what comprises the public interface to a global class: however given a pointer to an instance of that class, only its global functions and global member objects are accessible. Functions which are to be global must be explicitly marked; global member objects are global by virtue of their class:

```
global class A {
  ...
public:
  ...
}

class Z {
  ...
public:
  ...
}


global class B {
  ...
```

[5] We discuss exactly what categories of features can be supported for marshalling later below in section 4.

```
public:
  A part1;              //visible via B*
  Z part2;              //invisible via B*
  global void f1(....); //visible via B*
  void f2(....);        //invisible via B*
  int y;                //invisible via B*
  char* str;            //invisible via B*
}
```

Naturally this is recursive: class `Z` may have public member parts which themselves are (inline) instances of global classes. Thus given a pointer `B *bptr`, a "part-of" navigation chain can be formed by `bptr->pz.py.px...` which ultimately must denote a global function or an instance of a global class. If the object referenced by `bptr` really is remote, then invoking one of its own global functions, or one of the global functions of its public parts, will result in a remote invocation. Accessing one of its (nested) public parts so as to form an address (eg to obtain an lvalue) is also legimate since our execution environment detects pointers to remote objects (whether or not they be nested as parts within another remote object).

Our rules for C++ references match those for pointers: given a reference to an instance of a global class, only the global functions and global member objects can be used from the reference. Note too that for the purposes of these access rules, we classify C++ non-class entities (eg `int`s) as non-global entities. In Eiffel, as noted above, the basic types are (non-global) expanded classes: however since expanded classes cannot be derived, a non-global expanded class cannot be subclassed to form a global one.

It should be noted that global functions have the semantics of virtual (in C++), or redefined or deferred (in Eiffel), functions. That is, for example in C++, if a global function is called via a pointer to a base class, which actually references an instance of a derived class, then the global function in the derived class will be called.

3.2 Derivation

In addition to the access rules above, we impose the further limitation that when assigning a value to a pointer to a global class, that value must also be a pointer to, or the address of, a global class. A complimentary rule is imposed on pointers to non-global classes[6].

Thus given

[6]However we do not outlaw C++ type casts, and leave the programmer to suffer the consequences should she use them.

```
global class A {
 ....
}

class B : public A {
 ....
public:
 A a;        // global member
}

global C : public B {
 ....
public:
 A aa;       // global member
 B b;        // local member
 global int fn(....);
}

global int C::fn(....};
{ A *ap;
  B *bp;
  C *cp;
  ....
  ap = cp;       // legal
// ap = bp;      // illegal
// bp = cp;      // illegal
  ap = &(cp->aa);// legal
//ap = &(bp->a); // illegal
//bp = &(cp->b); // illegal
}
```

3.3 Encapsulation

When considering the rules for pointer assignment and dereferencing, we must also take account of the current object. If the current object is an instance of a global class, then every access to its instance data is categorised (for the purposes of assignment compatibility checking) as an access via a remote pointer – after all, C++'s `this` or Eiffel's `Current` is then a pointer to a global class.

In C++, the current object may access the private (as well as the protected and public) instance data of another object of the same class, given the value of, or a pointer to, such an object. In Eiffel, the current object can only access the exported features of another object, even if that other object is from the same class. However selective exporting can be used (possibly to the current class itself) so as to allow access to more features than are generally visible. In our extensions, such rules must be moderated by the access rules for remote pointers. Thus, in a non-global class, the rule for access to peer instances are unchanged. However for a global class, given a

pointer to one of its peers, the current object can only access the global functions and global member objects of that peer (using the same rules as previously). For C++, the same access rules also apply to friend classes and functions.

```
global class B {
 ....
}

global class A {
    int n;
    global int fn(A&);
    B b;
    B *bp;
public:
    global void f1(....);
    global int f2(....);
}

global int A::fn(A& aref)
{   // aref.fn, aref.b, aref.f1
    // and aref.f2 are all available.
    // aref.n and aref.bp are hidden.
    // However n and bp are available
    // to the current object
}
```

Although it is certainly feasible to be more restrictive and outlaw peer and friend access altogether, we believe it is legitimate to have non-public features which are (possibly remotely) accessible by collaborating instances and classes. However the programmer still retains control over the cost of remote access, and incurs the additional overheads only for those features which she has distinguished.

3.4 Cloning

Both C++ and Eiffel allow a copy of an object to be obtained: in Eiffel, this can be a shallow or deep clone; in C++ the copy is by default a member by member copy of the instance data, but the default for a class `X` can be overridden by providing `operator =(X&)` for general assignment, and `X(X&)` for initialisation by assignment.

For global classes, we currently impose the restriction that such copying is not supported. Once again, our basic problem is the lack of arbitrary byte access to remote contexts: copying an instance of a global class may well require access to a remote instance.

4 Marshalling

The normal way of adding RPCs to a programming language is to provide interface definitions, from which a tool can generate RPC stubs for both the client and server[3][10]. In languages which provide support for abstract data types – including the object oriented languages – it is obviously natural to consider a tool which takes an ADT definition in that language – for example a "flattened" Eiffel class definition – and likewise generates client and server stubs. A more friendly programming environment results.

A major question is how closely the RPC package can match the type system of the host language. The basic types, such as `int`s[7] are usually relatively simple to marshall into an RPC packet, possibly translate between host (and compiler) conventions, and unmarshall on delivery. However pointers are notoriously difficult – yet in many object oriented languages, including C++ and Eiffel, pointers are a common representation for object identifiers. It is highly desireable that object identifiers are RPC marshallable for any distributed object oriented language. A related problem is the transmission of copies of objects since the location of object pointers within the instance data must be located during RPC marshalling. C++ structs are no different in this regard, but unions are impossible without additional code supplied by the programmer herself.

In passing object instances, there may be "hidden" data added by the compiler, which must also be RPC marshallable. For example, a C++ class may contain pointers to so-called "vtbls", which serve to dispatch virtual function calls and so implement dynamic binding. Likewise Eiffel objects contain mechanisms for handling redefined or deferred routines. Such compiler generated information must be identified and rebuilt at the recipient.

For our extended versions of Eiffel and C++, pointers[8] to global classes can be passed as arguments and results within global routines. However pointers to non-global classes cannot be. Further, copies of instances of global classes also cannot be passed (cf section 3.4). Copies of instances of non-global classes in principle can be, however our initial implementation does not support this.

Values of basic types (such as `int`s) can be passed, but pointers to and references for basic types cannot be passed for C++.

Yet a further problem in C++ is arrays. Since an array is (only) represented by its first element,

[7] But sometimes with the exception of floating point types.
[8] as well as C++ references.

an RPC marshalling routine cannot know the actual length of the array unless additional information is given by the programmer – and this information is not mandatory in conventional C++. Thus we do not currently support marshalling of C++ arrays, with the exception of a pointer to a `const char`, which by convention represents a null terminated character string.

In Eiffel, a marshalling routine can determine the (dynamic) length of an array, and act accordingly.

There are some further nuances in C++. For example, pointers to class member functions are legitimate values for argument transmission: we can support these by marshalling a code address (and in fact other supplementary information).

4.1 Cross language calls

Our execution environment allows invocations from objects written in one language to those of another, whether or not these all be locally in the same address space context. Naturally in so doing, issues arise due to the mis-match in type systems of the languages concerned. Our basic approach here is to further restrict as necessary the values which can be marshalled for a particular language in a call which will be cross-language. To help us to this end, a distributed service called the *type manager* is being built above the basic execution environment. The type manager records type information registered in respect of interfaces at compile time and work on it is being led by our colleagues at Glasgow within the project[7].

5 Persistence

Many modern languages, if not actually incorporating persistence into their definition, include some support for persistence in their libraries. For example, the Eiffel libraries include classes `STORABLE` and `ENVIRONMENT`: any class which includes `STORABLE` in its inheritance hierarchy can have persistent instances; if `ENVIRONMENT` is used, all object creations done while an environment is "open" will be persistent. Likewise for example the C++ Interviews library[13] includes class `persistent`, somewhat akin to Eiffel's `STORABLE`.

One of the pioneering developments in persistent languages was PS-Algol[20]. A major postulate of this project was that persistence should be orthogonal to type. As a result the persistence of an object is not statically determinate. When an object is created, it will not become persistent unless it becomes referenced by another persistent object. It will persist as long as it is so referenced: eventually it may become disconnected and be discarded. Thus persistence is a dynamic attribute that can be gained and lost at execution time: further, not all instances of the same type may necessarily persist.

The advantages of this approach are summarised in [1]: the saving of coding effort and space to transfer data to and from files or a dbms; a conceptually simple, single program view of data; and benefitting from any type safety of the programming language when applied to persistent data.

Although we also adopt this approach to persistence, it should be noted how it relates to our approach to distribution. For distribution, we separated global classes from non-global ones: all instances of a non-global class are always local to objects which hold references to them; a local instance can never dynamically become remote. However, an instance of a global class may or may not be remote (and this degree of proximity may change dynamically due to migration). Thus we have one category of classes which behave similar to "normal" (C++ or Eiffel) classes with respect to distribution, and a second which are more costly to use.

In the same way that, even in the best case[9], a potentially remote object is more costly than a local one, also a potentially persistent object is more costly to use than one which is always non-persistent. The cost of a persistent object is chiefly detecting whether or not it is currently loaded – even in the best case (ie the object is already loaded) this test must still be taken[10].

Thus we divide classes into two further categories: those for which persistent instances never exist, and those for which some of their instances may persist. Note that in the second case, persistence is determined (and may change) at runtime, depending on what objects are transitively reachable (at a given consistent point in time) from a set of root objects.

Thus in principle there are four categories of classes:

	local	potentially remote
transient	$\mathcal{A}$	$\mathcal{B}$
potentially persistent	$\mathcal{C}$	$\mathcal{D}$

[9] Costs are reported later in section 10.

[10] It is conceivable to consider memory violations as a basis for this test, thus relegating it to the hardware MMU: however it seems difficult to do this in a language independent way, and notably to determine what the current object (and maybe also thread) is within the violation signal handler.

In practice we have chosen not to support category $\mathcal{B}$ classes for C++ and Eiffel in our environment. The chief reasons are simplicity and cost: a programmer can decide that a class should behave exactly like a "normal" one – non-distributed, non-persistent – and with no consequent costs; or she can choose to add the ability to persist on a per-instance basis, with some additional resultant costs; and finally she can choose to add the potential to be remote, and incur the heaviest cost penalty.

Automatic (in C++) and local (in Eiffel) instances of category $\mathcal{C}$ and $\mathcal{D}$ classes are not supported (as in [21]). However arrays of both, including an array of instances of global classes, are supported.

5.1 Class based approaches

As noted above, class libraries exist for both C++ and Eiffel to provide persistence. In our execution environment we believe it is important that applications written using these classes should continue to function.

We believe that re-implementing these classes should be relatively straightforward. Our environment will store all objects reachable from a given persistent object, with the exception of instances of transient classes. Thus to implement the "persistence completeness" rule of Eiffel, any Eiffel class which derives from class `STORABLE` must be checked to confirm that neither it nor its dependents contain pointers to transient objects. Doing otherwise should, in our view, generate a compile time warning. A similar check should be used for any object used in `put` or `force` of class `ENVIRONMENT`, but this will in general need to be a runtime check. Implementing the keyed insertion and retrieval of this class require a persistent table, a generally useful library class in any case.

5.2 Object oriented data management

Object oriented database systems, such as propounded by[2], are receiving commercial interest as feasible stores for object technology. Alternative approaches are extending relational database technology [6]. We do not intend to enter this debate: our execution environment includes a persistent store on which such data management systems can be built.

Our storage subsystem typically uses a number of segments for each address space context. A segment can be unmapped by one context and remapped into another. However a default policy is that a segment cannot be simply unmapped without checking for any pointers into it (from other segments or from thread stacks). This as a result may limit the number of objects which can be mapped into a context[11].

While this approach is adequate and reasonably inexpensive for many purposes, it is inadequate for large collections of objects since the virtual space will become depleted. Thus to support large collections, different segments must be mapped and unmapped at the same position in a context, as required. While mapped, a segment may need to be pinned so that any pointer values into it which are temporarily established remain valid. We are currently actively investigating such an extension, together with a mechanism for atomic transactions and recoverable memory suitable for moderating accesses to large collections of objects: we intend to report on the details of this work in due course.

6 Static storage

Static storage is frequently used in C++ to achieve global information. The problems of supporting static storage in a persistent environment are well known and solvable[21]. However in a distributed environment, the problem seems harder. Since static storage is by default a part of the executable image of a linked C++ program, a naive use of static storage in a distributed environment will result in multiple copies of the static data, once at each node where the same code image is being used. Communication via the static storage area is difficult to support – once again assuming that distributed shared memory is not used.

Our current proposal is to limit static storage for C++ programs to `const` values and also `const` pointers to global class objects. In particular the latter allows a well-known object to be available for RPC use: the object identifier for that object is obtained during compilation.

For Eiffel, static storage is not available to the programmer. However `once` routines are provided which provide some measure of global information in that they will only ever be executed once, and their results cached for immediate return to any subsequent invocations. Supporting these is similar to the solution identified for persistent storage above.

[11] by virtue of limitations imposed by the underlying virtual memory system, such as the paging space available.

7 Exception handling

While Eiffel includes exception handling in its language definition, C++ does not yet provide a stable mechanism[11]. In our system, the range of exceptions that may occur from the exception are increased over those normally available, so as to include for example communication failures.

In Eiffel, our approach is simply to extend the range of exceptions exported by the kernel class `EXCEPTION`. Thus programmers using global classes can attempt to rescue failures arising due to the distributed environment. Naturally an exception due to distribution may not be successfully caught, as indeed any exception may not be caught. In this case the current "activity" is terminated (itself raising an exception).

Likewise when an incoming RPC invocation is dispatched into an Eiffel routine, that routine may fail with an uncaught exception. In this case the exception is caught by the RPC dispatcher and returned back to the caller.

With respect to C++, in the absence of exceptions from the current language definition, we are tempted to allow the definition of handlers for system related failure conditions, akin to the `extern void (*set_new_handler (void(*)())) ()` of `<new.h>` which allows failure of `new` to be caught. Programmer level exceptions arising from a call to an object written in another language (eg a call to an Eiffel object which returned an Eiffel exception) could also be mapped into a single failure category, with an associated handler.

8 Concurrency

Omitted in this version of the paper.

9 Binding and naming services

A concern in RPC environments is the binding of a client of an interface to a potential provider. Binding environments like the DECdns, recently adopted by OSF as an integral part of their DCE offering[19], and the more sophisticated ISA trader[10] allow decisions to be deferred until runtime, potentially making successful "marriages" on a best effort basis.

As may be apparent, our execution environment does not rely on the availability of such brokerage services. Objects are named by object identifiers (eg C++ pointers) and mechanisms are provided to ensure that the references remain valid even should the designated objects migrate (eg as a result of load balancing within our execution environment), or persist and later be mapped on demand. Object identifiers are returned by Eiffel's `Create` and C++'s `new`. They can be subsequently communicated by direct assignment or by parameter and result transmission during function calls. Because of the dynamic binding presented by the class hierarchies, a "client" may actually be bound to a "server" which offers a more elaborate interface than the client actually requires.

Nevertheless it may be useful to associate human readable names with object identifiers, and to support a looser degree of binding. In our view this is best achieved as an application in our environment - eg by a set of (eg C++) classes which combine to provide a naming service, and which can benefit from the persistence and distribution mechanisms provided by the execution environment. An X.500 like directory service has already been prototyped in this way by our colleagues in Bull within the project[7].

Note too that the concept of trading should be possible in our environment. A "server" can export a binding to an interface which it is proffering. This interface could in principle be an augmented public description of a C++ or Eiffel class. Likewise a client could obtain an object reference from a trader, which should be subtype compatible with the expected interface. We hope that this approach can be explored in conjunction with the ISA project.

10 Costs

So far we have consciously led the reader away from consideration of the details of our underlying execution environment. We have done so so as to focus attention on what programming model we present to a C++ and Eiffel programmer. Nevertheless it is reasonable to give some attention to the costs incurred by our various extensions. A fuller description of the intricate internals of our environment is in preparation. Currently a prototype of our execution environment is operational above Ultrix (on Decstations and μVaxes), and retargettable compiler tools for our Eiffel and C++ extensions are well underway.

As noted in section 5, we propose three categories of classes.

Transient and non-global classes are exactly equivalent to "normal" Eiffel and C++ classes. The only restrictions on their use is avoiding pointer assignments to and from persistent and/or global instances,

and incur no additional overheads in our environment.

10.1 Non-global but persistent classes

Non-global classes which can have potentially persistent instances incur, for each (non-member or non-direct) instance, a space and time overhead. They each have a header which includes the size, class identifier and language identifier of the object. It also includes a pointer to a set of functions (actually a C++ vtbl) which our execution environment can up-call so as to manage the object in a language specific way[26]. The most important of these is the `nextptr` function which can be used to iterate through the object identifiers within the instance data of the object.

Apart from the space cost of the header, and the occasional use of the up-calls, the main cost in using a persistent object is detecting whether or not it is currently mapped. As noted in section 5.2 we use segments to store clusters of related persistent objects. An object identifier for an as yet not mapped object occupies the same storage space as normal (eg a 32bit pointer in most C++ and Eiffel implementations), unlike for example [21].

When mapping a segment, one possible option is to ensure that all pointers into that segment are immediately made valid. In our environment, achieving this would require the whole incoming segment (and not the whole context) to be scanned: however this can be slow, particularly if only a few objects from the segment are required. It would certainly be unacceptable in a data management application. Hence an alternative is to only "register" the requested object from the segment, with the consequence of an explicit check on each pointer de-reference[12].

In summary, once the target object denoted by a pointer is mapped, the pointer refers directly to the object (as in normal C++ or Eiffel) but there can be a residual test (which will succeed) to confirm that the object is present.

10.2 Global and persistent classes

The final category is global classes which can have persistent instances. The overheads of these in addition to those incurred by non-global persistent classes are essentially the RPC stubs required to marshall parameters at the client side, and to unmarshall and dispatch the invocation at the called side. Each (source) class thus has two forms: one form containing RPC stubs for a client; the other containing the dispatching code and original class code. The *stub-class* and *true-class* are actually subclasses of the *interface-class* in which any global functions appear as virtual, or deferred, functions. These three classes are automatically produced from the original source class definition.

[12] In practice this may be optimised to checking when a pointer is available in a new scope.

11 Conclusions

In this paper we have summarised our approaches to extending the Eiffel and C++ programmer's model for distributed and persistent programming. We have noted that these extensions are being implemented above a language independent execution environment, and have given an indication of the costs involved in each extension. Implementation of the extended support for Eiffel and C++ is underway.

12 Acknowledgements

Within TCD, we would greatly acknowledge in particular the work and input of Sean Baker, Ann Barry, Alexis Donnelly, Ahmed El-Habbash, Dominic Herity, Jooli Ooi, Annrai O'Toole, Mark Sheppard, Brendan Tangney, and Bridget Walsh. Finally all the TCD team owe our special thanks to the work of Neville Harris as TCD project director.

We are also conscious of our indebtedness to our many colleages outside of TCD with whom we have collaborated in Esprit Comandos. We would in particular express my appreciation to Jose Alves Marques, Roland Balter, Richard Cooper, Dominique Decouchant, Michel Gien, Paulo Guedes, David Harper, Sacha Krakowiak, Reinhold Kroger, Roger Lea, Helmut Meitner, Micheal Mock, Xavier Rousset de Pina, and Gerard Vandome.

We acknowledge that Unix is a trademark of Unix Systems Laboratories, Inc. Chorus is a trademark of Chorus Systèmes.

13 Appendix

We here include an attempt to describe in C++ of the additional rules we introduce for global classes (cf section 2). We give class definitions for both class `global` and class `nonglobal`. Our global classes in principle then use class `global` as their base class; likewise non-global classes use `nonglobal`. Unfortunately we cannot capture completely in C++ the semantics we desire.

```
class global{
private:
   global(const global&);
   // no init. by assignment
   global& operator=(const global&);
                  // no assignment
   ~global();     // no autos or temps
protected:
   global();      // construction allowed
                  // in subclasses
}

class nonglobal{
private:
   nonglobal(const global&);
  // no init. by assignment
   ~nonglobal();  // no autos or temps
protected
   nonglobal();   // construction allowed
                  // in subclasses
}
```

And so given, for example:

```
class A : public global{ // A global class
public:
         // assume these are global functions
   f(A&);         // legal
// g(A);          // wrong: memberwise copy
}

class Z : public nonglobal{//non-global class
public:
  void fn();
  A aa;           // a global member object
}

void Z::fn()
{
  A* pa = new A;   // a remote pointer
  Z* pz = new Z;   // a local pointer
  Z& rz = *pz;     // legal
  pa->f(*pa);      // legal
//A temp;          // illegal - no autos.
//*pa = *pa;       // illegal - no copying
//*pz = *pz;       // should be legal - #1
//*pz = rz;        // should be legal - #2
//pa->g(*pa);      // should be illegal- #3
}
```

At both #1 and #2, a non-global object is assigned a member-wise copy of a non-global object. This is legal by our rules, but would be disallowed by the C++ above since class `Z` contains a member whose class `A` derives from class `global` – and class `global` disallows member-wise copying. To model our rules in C++, we would need to amend the code above to indciate that `operator=` of class `global` can be used by (the friend) class `Z`.

At #3, an instance of a global class is passed by value (member-wise) as an actual parameter. This is disallowed by our rules, but for the C++ code above only a warning message is produced.

In practice, the system wide garbage collector would conceptually be a "friend" of both the `global` and `nonglobal` classes, and would be responsible for calling destructors.

References

[1] M. Atkinson, R. Morrison and G. Pratten "Designing a Persistent Information Space Architecture", *Proceedings of IFIP'86,* pp115-119, Dublin, Sept. 1986.

[2] M. Atkinson, F. Bancillon, D. DeWitt, K. Dittrich, D. Maier, S. Zdonik "The Object Oriented Database System Manifesto", *Deductive and Object-oriented Databases,*, Elsevere Science Publishers, 1990.

[3] A.D. Birrell and B.J.Nelson "Implementing Remote Procedure Calls" *Xerox report CSL-83-7*, October 1983.

[4] A. Black, N. Hutchinson, E. Jul and H. Levy, "Object structure in the Emerald system", *University of Washington*, Seattle, TR 86-04-03, April 1986. (also in proceedings of OOPSLA '86, Portland Oregon, Sept 1986 reproduced in ACM SIGPLAN Notices, Vol 21, No. 11, Nov 1986).

[5] J. Chase, F. Aamdor, E. Lazowska, H. Levy and R. Littlefield "The Amber System: Parallel Programming on a network of multiprocessors", Distributed Shared Memory Design", *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pp211-223.

[6] The Committee for Advanced DBMS Function "Third-generation Database System Manifesto", *Proceedings of Object oriented Database Conference*, Windermere, July 1990.

[7] Comandos Consortium, "The Comandos Guide", *Esprit project 2071*, 1990.

[8] A. Forin, J. Barrera and R. Sanzi "The Shared Memory Server", *Usenix Winter Conference*, pp229-243, 1989.

[9] R. Halstead "Multilisp: a language for concurrent symbolic computation", *ACM Transactions on Programming Languages and Systems*, Vol 7, No 4, Oct 1985.

[10] ISA Consortium, "The ANSA Reference Manual", *Esprit project 2267*, 1990.

[11] A. Koenig and B. Stroustrup "Exception Handling for C++ (revised)", *Proceedings of the 1990 Usenix C++ Conference,* pp149-176, San Francisco, April 1990.

[12] S. Krakowiak, M. Meysembourg, H. Nguyen Van, M. Reveill, C. Roisin and X. Rousset de Pina "Design and Implementation of an Object Oriented, Strongly Typed Language for Distributed Applications", *Journal of Object Oriented Programming*, pp11-21, Sept/Oct 1990.

[13] M. Linton, J. Vlissides and P. Calder "Compusing User Interfaces with InterViews", *CSL-TR-88-369, Stanford University,* November 1998.

[14] B. Liskov and R. Scheifler "Guardians and Actions: Linguistic support for robust, distributed programs" *ACM Transactions on Programming Languages and Systems*, July 1983, Vol 5, No 3, pp381-404.

[15] B. Liskov and L. Shrira "Promises: Linguistic Support for Efficient Asynchronous Procedure Calling" *Proceedings of the SIGPLAN-88 Conference on Programming Language Design and Implementation*, June 1988, pp260-267.

[16] J. Alves Marques, R. Balter, V. Cahill, P. Guesdes, N. Harris, C. Horn, S. Krakowiak, A. Kramer, J. Slattery and G. Vandome, "Implementing the Comandos architecture", *Proceedings of the Esprit'88 Conference*, North-Holland, pp1140-1157, Nov. 1988.

[17] J. Alves Marques and P. Guedes "Extending the Operating System to Support an Object Oriented Environment", *Proceedings of OOPSLA'89*, ACM Press, pp113-122, October 1989.

[18] B. Meyer "Object Oriented Software Construction", *Prentice Hall*, 1988.

[19] "OSF Distributed Computing Environment Rationale", *Open Software Foundation*, May 1990.

[20] Persistent Programming Research Group, "PS-algol reference manual - third edition", *Persistent Programming Research Report 12*, Department of Computing Science, University of Glasgow and Department of Computational Science, University of St. Andrews, November 1986.

[21] J. Richardson and M. Carey, "Persistence in the E language: Issues and Implementation", *Software – Practice and Experience,* Vol 19(12), pp1115-1150, Dec. 1989.

[22] J. Stamos and D. Gifford "Implementing Remote Evaluation", *IEEE Transactions on Software Engineering*. Vol 16, No 7, July 1990.

[23] B. Stroustrup "The C++ Programming Language" *Addison-Wesley*, 1987.

[24] V.O. Tam and M. Hsu "Fast Recovery in Distributed Shared Virtual Memory Systems", *Proceedings of the 10th Conference on Distributed Computing Systems*, pp38-45, June 1990.

[25] E. Walker, R. Floyd and P. Neves "Asynchronous Remote Operation Execution in Distributed Systems", *Proceedings of the 10th Conference on Distributed Computing Systems*, pp253-259, June 1990.

[26] M. Weiser, A. Demers and C. Hauser "The Portable Common Runtime Approach to Interoperability", *Xerox PARC*, March 1989.

[27] S. Zhou, M. Stumm and T. McInerney "Extending Distributed Shared Memory to Heterogeneous Environments", *Proceedings of the 10th Conference on Distributed Computing Systems*, pp30-37, June 1990.