Distributed Object Interoperability via a Network Type System

Vincent F. Russo and Patrick A. Muckelbauer and Robert E. Minnear

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907 USA

Abstract

Object-oriented programming techniques are increasingly gaining attention as a solution to some of the software engineering problems plaguing the construction of large software projects. Unfortunately, object-oriented interfaces are usually only enforced and usable through language mechanisms making it impossible for disjoint components to interact in a loosely coupled distributed environment. We are investigating the possibility of creating a runtime notion of an object’s interface and allow the dynamic querying of objects for their conformance to that interface. Our project also provides a mechanism to dynamically generate proxies to objects in different protection domains so that once this conformance is confirmed, operations can be invoked on the object in a mechanism compatible with the programming language in question. This is done without violating the encapsulation and enforced interfaces provided by the distributed objects.

1 Introduction

The use of object-oriented programming techniques is increasingly gaining attention as a potential solution to some of the software engineering problems plaguing the construction of large software projects[MeY87, HO87, JF88]. Object-oriented programming and design techniques stress modularity and encapsulation through narrow and rigidly defined interfaces as a way of achieving low coupling between individual software components and high cohesion in the implementation of each component. A component has high cohesion if all of its subcomponents are related strongly. Coupling measures the interdependencies between components. High coupling makes it difficult to separate, understand, maintain and reuse the individual components[GJM91]. Low coupling is also a desirable feature for distributed systems because it can help facilitate object migration, substitution, replication and reliability. It would seem, therefore, that object-oriented programming should be an ideal approach to use in the construction of distributed systems. Unfortunately, object-oriented interfaces are usually only accessible and enforced through specific languages or homogeneous systems. This severely limits the degree to which disjoint, unrelated components can interact in a loosely coupled, multilingual, distributed environment.

As part of the Renaissance project at Purdue University, we are focusing on overcoming these limitations. In particular, we are building a system which provides the ability for unrelated applications written in a variety of programming languages to obtain access to objects distributed throughout a network of machines and implemented in a variety of manners. To limit our scope, we focus primarily on the client side of such a system, but we will not ignore server issues. We are attempting to achieve our goals by creating a first class description of an object’s interface which has a runtime realization and is independent of any programming language or system. We provide translators for this description into a variety of languages and systems.¹ These descriptions are used to generate interfaces to remote objects dynamically at runtime. They also allow flexible but complete interface specification providing both increased encapsulation and documentation.

¹We hope to even make it possible to map non-object-oriented substrates into our systems. For example, providing object-oriented access to files resident on an NFS file server.
2 Background

The main focus of object-oriented programming is to consider system components as interacting collections of data that can only be accessed through a predefined set of operations. Throughout the rest of this proposal we will use the term object to refer to such a collection, the term method to refer to one of the operations and the term signature to refer to an object’s entire set of methods. Methods are invoked to alter or access the data of an object. In a true object-oriented system, method invocation is the only way in which object data are accessed. We will refer to a piece of code which uses an object by invoking methods as a client of that object.

A fundamental feature of object-oriented programming is the notion of a class as a generator, or template, for object creation. A class specifies the interface and implementation of objects created from its template. Objects created from a given class are termed instances of that class. Most object-oriented languages provide a mechanism for classes to inherit portions of the signature, and implementations of some of the methods, from other classes called parents. This is how code sharing is implemented in class-based object-oriented languages. Classes which inherit from other classes are termed subclasses of those classes. A subclass’ signature is a copy of the parent’s signature plus any additional methods the subclass chooses to add. Likewise, a subclass can provide implementations for the new methods (or defer that decision to further subclasses) and provide new implementations of existing methods in the parent’s signature. These new implementations only affect instances of the subclass. The parent’s implementations are still used when the methods are invoked on instances of the parent classes. Inheritance is useful for code and interface sharing, for factoring code into common places, and for incremental development and documentation [HO87, Sny86, JF88, Mey87]. Inheritance is so useful in the context of class construction that it is viewed by many as an essential feature of object-oriented programming [Weg87].

The other feature distinguishing object-oriented programming is polymorphism achieved through late binding of method invocations to their implementations by target objects. This feature usually relies on some form of signature conformance checking to verify proper uses of objects. Signature \( S_x \) is said to conform to signature \( S_y \) (written \( S_x > S_y \)) if every method in \( S_y \) is also found in \( S_x \) (\( S_x \) may have additional methods as well), and for every method in \( S_y \), each of the following conditions hold.

1. The corresponding method in \( S_x \) has the same number of parameters.
2. The parameters of \( S_y \’s \) version of the method conform to those of \( S_x \’s \) version.
3. The result of the method in \( S_x \) conforms to the result of the corresponding method in \( S_y \).

With these rules, conformance can be viewed as a substitution rule for objects. If \( S_x > S_y \) then an object with signature \( S_x \) can be used wherever one with signature \( S_y \) is expected. This can also be thought of as narrowing the view of the object from \( S_x \) to \( S_y \).

For the purposes of this paper, we will distinguish two uses of the term signature. As described above, every object has a signature defining all the methods the object makes available to its clients. This signature is usually specified by the class from which the object was instantiated. We term this the object’s concrete signature. In addition, for each program variable there is a signature that describes objects the variable can reference. We term such a signature an abstract signature.

Type checking object-oriented programs amounts to verifying whether the concrete signature of an object conforms to the abstract signature of variables to which a reference to the object is assigned. In both statically and dynamically typed languages, a class specifies the concrete signature of its instances. Statically typed languages such as C++[Str86], Eiffel[Mey88] and Trellis/Owl[SCB+86], explicitly code abstract signatures in the program, allowing conformance to be checked at compile time. These abstract signatures are usually specified as classes; candidate objects are assumed to be instances of these classes or their subclasses. It should be noted that this restriction often limits conformance to a subclass relation, not the strict signature conformance defined above. In dynamically typed languages like Smalltalk[GR83] and CLOS[DG87] variables have no declared types. Rather, conformance checking is deferred until method invocation time.

In our work we focus primarily on clients of objects distributed through different protection and implementation domains, and not on the way the objects themselves are implemented. For this reason, we feel that while classes are an invaluable object implementation tool, they are an inappropriate mechanism for describing distributed objects to clients. We feel using a signature-only description mechanism over classes is essential to reduce the coupling between objects and clients. Classes include implementation details which are irrelevant to client code. Rather than requiring the
client code to know the class definition of an object in order to access the object, we allow programmers to specify only the abstract signature they expect the object to have. We allow this specification in a language independent manner which is mapped by a translator into the target language.

As a simple motivation for our approach, consider reading files from a remote system providing file objects to clients. The abstract signature supplied by the programmer of file objects would likely consist of methods for reading (read), writing (write) and positioning (seek) the files. However, when writing the class for file objects, the programmer will likely provide methods for various system implementation functions and system information function as well. These methods would not be of interest under normal circumstances to the average user of file objects but might be important for proper system operation. Examples of such methods would be those to access or update information about the position of the file on permanent storage, or to return information about update and modifications times for the individual files[Mad92]. The desire to keep interfaces narrow to decrease coupling between components should make clear that these additional methods are superfluous to the average client program. Requiring or allowing their existence to be made apparent to client programs only increases the coupling between the client program and the implementation of the file objects. Ideally, it should be possible for the client program writer to specify exactly the abstract signature he or she needs the remote file object to have and for the system to check the conformance to this signature at the time a reference to a remote file object is created. It is uninteresting to the programmer of the client code how the methods in these signatures are implemented and what additional methods are available. Moreover, classes are inherently a language specific issue and not usually applicable in a multilingual environment.

This example emphasizes why using classes as the description of remote objects is not desirable. A class specifies the concrete signature of its instances along with the implementation of some or all of the methods in the signature. If all the information specified by a class were exported to clients, it would severely hamper the ability to modify objects. Each time implementation specific details about the class were changed, client code might need to be altered to reflect these changes. This is especially true of statically typed languages like C++ which take advantage of the class layout to optimize method lookup and invocation. Inheritance only serves to complicate the problem further since information about multiple classes might be needed by clients[RL89, Ben87].

3 Motivation

The approach we have taken in our system is a direct outgrowth of previous work on the Choices [Rus91a, CJR87, CRJ87, RJC88, RC89] and Renaissance[Rus91b] object-oriented operating systems[Rus91a]. By object-oriented operating system we mean that system entities are represented as objects that are instances of representative classes. Such objects include encapsulations of hardware devices, traditional operating system entities such as processes and files, system data structures managing resource allocation, system policy modules, and, in particular, low level operating system data structures such as page tables and device control registers.

The way these systems are implemented is not what motivates our work. Rather, it is the way they provide interfaces to application programs which convinces us this work is necessary. All operating systems must provide a set of application interface primitives to allow delayed binding of application requests to the operating system functions that implement desired services, and to insure integrity of system data and functions. The main feature distinguishing an object-oriented operating system from traditional operating systems is that it should provide its primitives as method invocations on system objects. This implies that method invocations in an object-oriented operating system have one of three characteristics:

1. **Intra-system methods**: methods invoked within the system (between system objects)

2. **Intra-application methods**: methods invoked within the application (between application objects)

3. **Inter-domain methods**: methods invoked on system objects by applications.

Intra-system and intra-application method invocations are implemented as dictated by the languages in which they are programmed. Inter-domain invocations must cross any barrier the system provides to support protection of system data from aberrant or malicious application programs, and support dynamic binding of application requests for system services to the system objects implementing those services. Unprotected object-oriented operating systems
can implement this with an extra indirection on invocations of system object methods. A protected object-oriented operating system must implement method invocation on system objects in such a way that privilege levels can be crossed in a transparent manner.

Besides a mechanism to allow invocation of methods on system objects, an object-oriented operating system needs a naming mechanism to provide applications with references or capabilities to system objects in the first place. This is similar to the way that references to servers must be obtained in message-passing systems. One solution is to provide every application with a predefined reference to a name server object that maps symbolic names to references. Queries to this object return references to other objects, which in turn provide specific system services. The name server defines the set of all system services available to an application as a set of objects. The classes of these objects, in turn, define the application interface of the operating system. This interface can be extended or reduced dynamically by adding objects to, and removing objects from, an application’s name server.

Object-oriented operating systems may impose static or dynamic typing on system object references.Typing can be integral to the implementation of the name servers. In a dynamically typed object-oriented system, queries to a name server require only the name of the object and return a reference to the object if it exists no matter what the type of the object. Type checking is deferred until the object’s methods are invoked. Statically typed systems do not allow method invocations to untyped references. The type of the object being referenced must be confirmed before method invocations are allowed. Support for static typing can be achieved in two ways. First, name servers can provide untyped references to objects, but the references are not usable until they are narrowed to a particular type. After narrowing, a new reference is constructed that may be used as the target of a method invocation with no further checks. If the object is not of the proper type, the narrow operation should fail. Second, untyped references can be prohibited. The type of the object being looked up can be included as an argument to the name server query operation. The name server will only return a reference if an object of the given name exists and it is of the type requested. In this way, once a reference is obtained, it can immediately be treated as a typed reference and method invocation sends can proceed normally. This is how both Choices and Renaissance implement their interfaces.

The problem with the current implementation of these systems which motivated our work is that they both use classes as the mechanism to describe kernel objects to application programs. The tight coupling which this imposes between applications and the system forces recompilation of applications each time that a system class it uses is changed. While this restriction may be acceptable on a single system which changes infrequently, we feel that as we move to distributed systems and allow applications to access objects on remote machines, it would be entirely too restrictive.

4 Approach

In a multilingual, multisystem environment, it is essential to provide access to a canonical representation for signatures which is independent of machine/system/language. Our description supports signatures to describe objects, and in addition supports a set of primitive types. Primitive types are simple data without methods. Types as integer, character, array of <type>, and aggregates (structures) are examples. For the remainder of this paper, the term object unqualified will be used to refer to objects with methods and the term data to refer to instances of primitive types. Providing for simple data items is motivated by the success of languages like C++ at providing support for object-oriented programming while not penalizing traditional operations such as integer arithmetic, and by the observation that many method arguments are integers, characters or strings.

A client program using our description language must be written in a style which accesses all distributed objects through method invocations. No objects can be passed by value. Rather all objects are passed by reference. All data arguments to methods are passed by value or value-result. We feel that passing objects by value is really an object mobility issue and not a typing issue. Passing objects by value would involve moving class and implementation information between protection domains. This information is intentionally outside the scope of our interface descriptions.

Every object accessible to clients using our description scheme has a signature describing name, return type, and argument types of every method the object will accept. For example a buffer object might have the following signature:

signature Buffer =
  size() : integer;
  characterAt( integer ) : character;
  putCharacterAt( character, integer ) : void;
Signatures can be used to specify the conformance required for arguments of other signatures as well. For example, the Buffer signature can be used as a description of the arguments to methods of a File signature.

```
signature File =
    read( Buffer ) : integer;
    write( Buffer ) : integer;
end
```

Because of the importance of determining the type of an object, all objects implicitly support the signature method. This method returns a reference to a Signature object describing the object's concrete signature. Signatures are themselves first class entities. Each signature is accessible through the signature Signature described below.

```
signature Signature =
    conformsTo( Signature ) : boolean;
end
```

The conformsTo method tests whether the signature conforms to another signature.

Our approach focuses on providing a first class, runtime representation of signatures that can be compared for conformance and integrated with language notions of objects and methods where present. Each object participating in our system is able to be queried for its signature description as an object. We provide the ability for objects representing programmer defined signatures to be created and checked for conformance with the signature objects provided by system defined objects. Initially, we are integrating this representation into the C++[Str86] programming language so that the C++ type checking system can check statically for correct uses of remote objects signatures[GR91]. However, we do not wish to limit ourselves to object-oriented languages. We will also integrate our notion in a procedural manner into traditional programming languages.

In a distributed object environment, accessing objects in different protection domains requires some sort of remote method invocation mechanism. Proxies[Sha86] are a well expected solution to this problem. Proxies map the language notion of procedure call or method invocation transparently into a network transfer to the remote object and then back. Proxies must transfer arguments to, and results back from, the node containing the remote object. In our system, we generate proxies dynamically at the time object references (variables) are bound to actual objects. This generation is based on, and guided by, the signature of the object.

A benefit of generating proxies dynamically is the ability to implement some or all of the proxy’s methods as create-on-fault. Create-on-fault for a proxy’s method means to delay the actual implementation of the method until the first invocation. At that time the actual implementation of the method is constructed and added to the proxy. Thus only those methods actually called need to be constructed and the work of constructing them is amortized over a set of method calls.

Proxies must rely on an underlying transport mechanism for moving data between protection domains. If the domains are simply separate addresses spaces on a single node, this transport mechanism may be a simple shared memory scheme or a light weight remote procedure call mechanism built on top of shared memory. If the domains are on physically separate nodes, a true remote procedure call over a network is used[BN84].

### 4.1 Object Attributes

Besides simply specifying the interface provided by objects with signatures, we allow objects in the system to carry attributes which can dynamically affect the generation of proxies. The goal of supplying attributes is to provide additional information about individual objects not directly attainable from their concrete signatures in an attempt to improve efficiency. The ability to generate methods during the lifetime of a proxy also means as attributes are discovered method implementations can be altered to take advantage of the additional information. Attributes are similar to POOL’s properties[Ame90], except we use attributes as an object discrimination scheme rather than a class or signature discrimination scheme. We anticipate attributes initially being as simple as (name, value) pairs, perhaps evolving into more complicated descriptions.

Instead of creating simple stubs to marshal arguments to and from remote objects, attributes are used to create proxies that attempt to gain improved efficiency. In the absence of attributes, all proxies degenerate into simple stubs. We use attribute information to both reduce network traffic and improve remote method invocation response time. Consider the following signature for files:

```
signature File =
    read( Buffer ) : integer;
    write( Buffer ) : integer;
```
No special caching information may be inferred from this signature. However, a per-object attribute for read only files could provide semantic information indicating that the return value for the method `numberOfBytes` is constant and could encode the value. For a read only file the proxy implementation of the method `numberOfBytes` can be resolved locally, thus reducing network traffic and improving method invocation response time.

4.2 Current Status

The first phase of this project is to create a signature-based interface to the Renaissance kernel. This phase includes making all application accessible kernel objects support a method which returns a signature object representing the objects concrete signature. We are in the process of altering the kernel to make kernel defined signature objects available to applications by suppling a signature name server which maps symbolic names to signature objects. We also provide the ability for application to create objects representing abstract signatures. Any signature object can then be used in conformance checking. In addition to these modifications, the new Renaissance kernel provides a dispatch routine which will allow dynamic invocations of kernel object methods.

Phase two of this project will be to add support for distributed objects. This support will be provided by a simple remote procedure call protocol to access the remote domain's dispatch routine. The protocol will include a machine independent representation for primitive types and support for the exchange of object attributes. To test this phase we will create objects on Renaissance and Unix hosts and access them using the application level interpreter built during phase one. Performance numbers will be gathered to profile the system in an attempt to identify bottlenecks. We will then investigate new attributes and transport mechanisms to reduce the effects of these bottlenecks.

Parallel with the latter half of phase two we will begin phase three, the integration of signatures into an object-oriented programming language. C++ will be used due to its availability and our previous work on integrating the signature concept into the language[GR91].

4.3 Evaluation

We will measure the success of our system by three criteria:

1. The ease in which signatures are mapped into programming languages.
2. The ease in which signatures are mapped into Renaissance and other systems providing objects.
3. The level of transparency achieved by both of these mappings.

We fully expect to encounter numerous hurdles and unforeseen problems during the remainder of this project. One problem we do foresee is the possibility that of our notion of signatures and conformance may be different than the languages into which they are integrated. For example, languages that support inheritance for conformance checking mechanism rather than using signatures. This may cause conformance checks to fail in apparently arbitrary ways that are rooted in language or system related details.

One of our design goals is to treat the transport mechanism and the proxy implementation as orthogonal issues. This could prove much more difficult then we anticipate.

Finally, we have not addressed problems of garbage collection and storage reclamtion. Cross references between different domains can make it difficult to support an efficient garbage collection scheme. Our desire to remain multilingual will serve only to aggravate the problem.

5 Related Work

Traditional Distributed Systems. Clients in traditional distributed systems such as V[Che88, Che84], Mach[A+86], and Chorus[RA88] acquire system services from servers by explicitly sending messages to ports or processes. While the servers in such systems can be considered objects and message sending analogous to invoking methods, such systems do not provide a runtime representation for signatures and therefore cannot perform type checking at the time objects are bound to references. All type checking must be done at message send time.

Clients in remote procedure call (RPC) based systems[BN84] such as SUN RPC[Sun85] acquire system services by invoking local functions which transparently access remote services. Most RPC systems provide the notion of a program and a set of procedures to call within a program which is analogous to an object with methods. However, RPC systems do not provide a runtime representation for signatures therefore they exhibit many of the same problems as the message based systems described above.
Matchmaker and Rpcgen. A common approach to address some of the goals we address in our work is to build an interface description translator like Matchmaker [JR86] or Rpcgen [Sun85]. These systems provide a procedural based interface description language to traditional distributed systems like Mach. Matchmaker and Rpcgen proxies are generated statically at compile time from the source of the description. They provide a convenient mechanism for varying programming languages to incorporate remote procedure calls. However, neither Matchmaker nor Rpcgen provides a runtime representation of the interfaces it describes. Therefore client code cannot perform type checking at the time objects are bound to references. Again, all type checking must be done at procedure invocation time and none of the advantages of late binding of proxy generation can be achieved.

Plan 9. Plan 9 [PPTT90, PPTT91] is a general purpose, distributed system implemented on a variety of platforms and networks. Like Unix it provides only one abstraction for objects, namely, a byte stream (file). All access to objects must be through this single interface. Although a substantial number of objects may map into the this interface, we feel it is far too restrictive to mandate to clients a single abstraction for objects. Plan 9 does, however, provide in effect support for additional methods on an object by using the write interface as a message passing mechanism between client and object. However, the system provides no type information for what messages are accepted by the remote objects and what the arguments to the messages are.

Distributed Smalltalk. Distributed Smalltalk [Ben87] is an attempt to extend the single user, uniprocessor version of smalltalk to a distributed environment with support for object migration. The work on Distributed Smalltalk revealed that class inheritance makes migration extremely difficult and inheritance does not appear well suited for distributed systems. This work also revealed that the extra implementation details in a class are very difficult to extend to a distributed environment. Their results further enforce our belief that signature-based rather than class-based types are the proper way to describe distributed objects.

Emerald and Eden. A common approach for supporting distributed objects is to design a language or system specifically for distributed objects. Although this may seem logical it has the severe drawback of constraining a user to a particular language including the language’s notion of objects and conformance. Our work, on the other hand, attempts to provide a mechanism for describing distributed objects using signatures and then mapping this description into different languages so users can write distributed programs in them.

A good example of the single language approach is Emerald [RTL+91]. Emerald is an object-based language for the design of distributed programs. One of the major goals of Emerald is the support for fine grained mobility or migration [JLHB87]. Fine grained mobility is the ability for objects large or small to move from node to node within the system.

Objects within Emerald are comprised of the following components:

1. A unique network identifier.
2. The data local to the object.
3. A set of methods on the object (a signature) and a copy of the code to implement those methods.
4. An optional process.

Objects with a process are termed active objects and those without are termed passive objects. During the migration of an object the data, code, and optional process must be copied to the new node. Since code is copied during migration Emerald is limited to running on a set of homogeneous machines.

The most striking similarities between our work and Emerald are in type definitions and conformance. Just as we do, Emerald allows the creation of abstract signatures and uses the identical conformance rules for type checking. The major difference between our work and Emerald is caused by Emerald’s support for fine grained mobility and the design decisions necessary to make it viable. Due to object mobility, Emerald is a monolingual distributed object language running on a set of homogeneous machines. Emerald does not scale well to moderately large networks for several reasons. First, the creation of a unique network identifier becomes increasingly more difficulty as the network grows. Second, Emerald relies on a broadcast technology for locating objects which becomes proportionally more difficult to support as the number of networks grows in the system. We have intentionally not addressed these issues, relying instead on our transport level to solve these problems.

The Eden system [LLA+81] predated Emerald and has many of the same features. Eden’s relationship to our work is similar to Emerald’s.
6 Conclusion

To the extent that our research is successful, it will provide programmers of distributed systems an object-oriented framework within which to construct such systems. In addition, it will bring a language independent notion of object and type (signature) closer to reality. Finally, providing the ability to separate implementation details from interface details in distributed systems will allow the construction of less coupled and more cohesive distributed software.

References


