Lingua Franca: An IDL for Structural Subtyping Distributed Object Systems

Patrick A. Muckelbauer and Vincent F. Russo
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907
{muckel|russo}@cs.purdue.edu

Abstract

Recently the trend has been towards applying object-oriented techniques to address problems of building scalable and maintainable distributed systems. Object-oriented programming increases modularity and data abstraction by supporting encapsulation through narrow, rigidly defined and strongly enforced interfaces to objects. Unfortunately, object-oriented interfaces and mechanisms are usually only accessible and enforced through language mechanisms or strict programming conventions. This severely limits the degree to which disjoint, unrelated components can interact in a multilingual, loosely coupled distributed system. An accepted solution to the language dependence problem is the use of high-level interface description languages (IDLs). IDLs provide a description mechanism for an object's interface that is independent of any programming language. In this paper we describe an interface description language and runtime support system that uses structural subtyping rules rather than the traditional interface name equivalence rules for conformance checking. We argue that the choice of structural subtyping rather than interface name equivalence leads to a less coupled and more extensible distributed system.

1 Introduction

As part of the *Renaissance* project at Purdue University we are researching tools and techniques for building scalable, maintainable and extensible distributed systems. The difficulty of building such systems is due mainly to the high degree of heterogeneity in a distributed environment. Large scale distributed systems must contend with multiple architectural platforms and the differences in their data representations, multiple languages and the differences in their notion of types and values, and multiple vendors providing different implementations and versions of components.

In order to build a successful and scalable distributed system in such an environment we believe it is necessary to minimize the *coupling* between distributed components[4]. Coupling measures the interdependencies between interacting components. Low component coupling is a desirable feature of any system because it decreases the difficulty of separating, understanding, maintaining and reusing individual components[10]. Over time, it is inevitable that some components of a system will need to evolve. High component coupling increase the likelyhood that, in order to maintain inter-operability, changes to support this evolution will require modifications to other dependent components. These modifications may, in turn, require modifications to other dependent components and so on. In highly distributed systems this problem will be worsened since numerous programmers, often working for different vendors, will develop individual components. Furthermore, these programmers are often unaware of many component inter-dependencies.

Recent trends have been towards applying object-oriented techniques to address such problems in distributed and non-distributed systems. Object-oriented programming increases program modularity and data abstraction by supporting encapsulation through narrow, rigidly defined and strongly enforced interfaces to *objects*[4]. An object consist of an encapsulated state and a set of operations, or *methods*, to modify or access the state. The *interface* of an object is the collection of all its methods. The *only* way to modify or access an object's state is to *invoke* a method of the object's interface. We term an autonomous collection of objects a *domain*. The increased abstraction and encapsulation provided by the object-oriented model helps reduce component

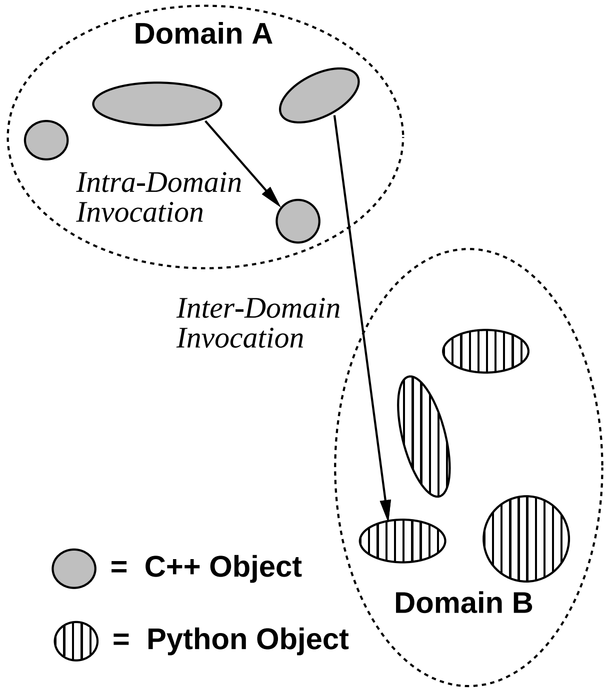

Figure 1: Inter/Intra-domain Method Invocations

coupling across separately developed domains.

Like other recent work[19], we use a distributed object system model in which interaction between clients and servers are implemented as method invocations on objects in separate domains. I.e., clients access a server by invoking methods on remote references to objects in the server's domain. Each domain is self-contained and usually implemented in a single programming language (or at least a single consistent object model). Intra-domain object method invocations are supported by the language(s) in which the domain is implemented. Our work is to develop tools and libraries that provide support for inter-domain object method invocations. This interaction is illustrated in Figure 1.

The modularity and encapsulation provided by the object-oriented model would at first glance seem to be ideal for distributed systems. Unfortunately, object-oriented mechanisms are usually only accessible and enforced through language support or strict programming conventions. This severely limits the degree to which disjoint, unrelated components can interact in a multilingual, loosely coupled distributed system.

An accepted solution to the language dependence problem is the use of high-level interface description languages (IDLs)[19]. IDLs provide a mechanism for specifying an object's interface independently of any programming language. Translators map these specifications into a target language's notion of objects and interfaces. The generated language specific modules are used by programmers to implement server objects (usually by inheriting from automatically generated parent classes) and/or to generate typed references to remote objects.

Components in a distributed system that wish to interact need to agree on common interface descriptions in order to guarantee the proper use of objects. The checking of this agreement, which is termed *conformance* checking, defines rules for the substitutability of objects, i.e., when a server object provides the interface that a client domain is expecting. A common approach to checking conformance is to define a server object's *type* as a named interface that it exports. Client requirements, also specified as named interfaces, are compared to server objects' types during conformance checking. I.e., conformance is defined to to be equivalence between interface type names (or hierarchies of interface type names). Consequently, interacting domains must agree a priori upon the names of the interfaces and their definitions.

As the conformance rules for our system, we chose instead to use structural subtyping rules. In this paper we will argue that the choice of structural subtyping over name equivalence leads to less coupled and more extensible systems.

The remainder of the paper will first cover some background material and then introduce our system by discussing the *Lingua Franca* Interface Description Language and the implementation of the runtime support libraries. We then discuss the differences between name based conformance checking and structural subtyping and detail the conformance rules we have chosen. We follow with discussions of the performance of method invocation and runtime conformance checking. Finally, we end with a comparison to related work and a summary of the advantages and disadvantages of structural subtyping.

2 Background

A distributed system is an environment that allows programs to interact across address space and machine boundaries. This is accomplished by supporting access to values within a remote domain through *remote references* to that domain. Operations on remote reference allow accessing and modifying remote values. A program providing remote values is termed a *server*. A program accessing those values is termed a *client* of the server.

Most distributed systems introduce some notion of *types* for describing the server values accessible through remote references. Types provide structure to a system by placing constraints on the set of values servers can export and clients can reference. Checking conformance to these constraints guarantees that remote references are used in a consistent manner and amounts to testing whether the type of a server value conforms to the type of a client's remote references to the value.

The inter-operability between a client and server is determined by the conformance rules chosen for the system. The conformance rules provide a substitution rule for types. If a type t_x conforms to a type t_y then a value of type t_x exported by a server can be referenced in a type consistent manner by a client expecting a value of type t_y. Specifically, t_x is said to be a *subtype* of t_y and t_y is said to be a *supertype* of t_x.

Conformance checking in a distributed system has several advantages: it catches errors early, it guarantees a sense of correctness to the system by not allowing clients and servers to interact in an inconsistent manner, and it allows greater execution time efficiency of remote accesses to be achieved[6].

Client programs bind to server values dynamically at runtime. Since the types of these server values are not known at compilation time, a static analysis of the client domain is not sufficient to check the conformance between the types expected by the client's remote references and the types of the remotely referenced values. Instead, distributed systems must rely on runtime checks to test conformance and guarantee type consistency. Before a remote reference can be used, the referenced value's conformance to a type specified within the client must be checked. We term this operation *narrowing*. Once a reference has been narrowed, static analysis of the client's code can further guarantee type consistency. The narrow procedure takes a remote reference and a local representation of a type. If, using a set of chosen rules, the type of the remote value conforms to the locally specified type, a new typed remote reference to the value is generated. The server value can then be accessed in a type-safe manner through this new typed reference with no further runtime checks. The narrowing operation usually returns a NULL reference, or perhaps raises an exception, to indicate the conformance check failed. There are several important consequences of the narrow procedure. First, there must be some runtime representation for types, secondly, a client must be able to retrieve or query a remote value for its runtime type information, and thirdly, a sound set of conformance rules must be defined. Conformance rules are sound if the rules do not lead to type inconsistences.

3 System Overview

A detailed specification of our rules for conformance are discussed in Section 4.3 but, briefly, our notion of an object's "type" is simply its interface specified as a set of named methods and their argument and result types. Client requirements, specified as interfaces *expected*, are compared at runtime to exported server interfaces using the standard subtype notion of method subset matching with covariance of return types and contravariance of arguments. I.e., interface I_x *conforms* to interface I_y if and only if every method in I_y is also found in I_x (I_x may have additional methods as well), and for every method in I_y, each of the following conditions hold.

1. The corresponding method in I_x has the same number of parameters.
2. The parameters of the method in I_y conform to the corresponding parameters in the method in I_x.
3. The result of the method in I_x conforms to the result of the corresponding method in I_y.

The names given to the interfaces (but not the individual methods in the interface) are irrelevant to the conformance check and are local only to a particular application.

The major tools we have implemented to support our system model are the *Lingua Franca* Interface Description Language and a set of runtime libraries to support inter-domain method invocations between domains described with *Lingua Franca* programs. Using our tools, a server object implementor describes the object interfaces exported by his/her domain in *Lingua Franca* and uses a translator to convert these descriptions into the target programming language in which the domain will be implemented. *Separately*, client programmers describe their requirements with *Lingua Franca* descriptions specifying the interfaces their applications need from server domain objects. Our approach differs from others in that the two descriptions are separate and can be independently modified, extended, versioned and even named. The conformance between a client and server interface is determined at runtime during narrowing when runtime representations of these two interfaces are structurally compared. The runtime support libraries implement the necessary runtime type information, the runtime conformance checking (narrowing), and object invocations across distributed modules. The libraries make no assumptions about the underlying hardware. They define a canonical representation for values and provides mappings to and from the canonical representation and the target hardware representations.

It is obvious that the approaches we use to obtain hardware, language, and implementation independence are very similar to the approaches used in other distributed systems, and we have leveraged heavily from their experiences. The main focus of our work is how to support *interface independence* and *versioning* in a way that minimizes component (client and server) coupling. Interface independence (low interface coupling) allows the minimization of the effects changes to a server's interface have on the clients of the server. Interface independence is achieved through the subtyping relations defined by the chosen conformance rules. The more flexible the conformance rules, the more tolerant the system is to changes to an object's interface. Versioning is the ability to join interfaces (versions) while still conforming to clients using the individual interfaces. I.e., it should be possible to allow a single server to provide multiple versions of a service with possibly different semantics. The extent to which both of these can be achieved depends on the conformance rules used and mechanism for specifying interfaces.

3.1 Lingua Franca

The *Lingua Franca* type description language provides a simple but powerful mechanism for describing types in a distributed environment. By keeping *Lingua Franca* simple we hope to reduce the burden of learning a new language. Simplicity is achieved by starting with a small number of basic types and using a set of powerful *type constructors* to build more complicated types. The design of *Lingua Franca* attempts to adhere to the principal of *Data Type Completeness*: wherever a type is allowed in a type constructor, any type may be used without exception, including other (inline) type constructors. This allows a rich set of types to be described using only a few constructors and reduces the number of rules and the overall complexity of the language.

Lingua Franca supports both *interface types* that describe object interfaces and *data types* that describe non-methoded values. *Lingua Franca* also allows an *Anything* type that is the infinite union of all types (i.e. the top of the type lattice). Type *Anything* allows all types, whether interface or data, to be treated uniformly.

Types are created from a small set of primitive types and a set of type constructors for describing more complex types such as sequences, records, and, most importantly, object interfaces. Translators map *Lingua Franca* programs and the types they describe into specific target languages producing language specific types. For interface types, the language specific types (usually a class) are used by programmers to construct servers (via subclassing of automatically generated parent classes) and/or to generate typed references to remote objects after narrowing.

Lingua Franca separates issues and details of how *Lingua Franca* types are represented in a target language from the specification of those types. A set of default mappings to implementation types is built into each translator. For example, sequences are translated into arrays in C++[23]. Programmers can also provide directives to the translator to control the representation of *Lingua Franca* types within a target language. These directives are of the form:

`Implement` S `as` T

where S is a *Lingua Franca* type and T is a language specific type. Supporting customizable translation provides for a clean separation of type and representation and leads to a simplier language with greater flexibility. For instance, *Lingua Franca* does not support separate type constructors for lists and arrays, but instead, only supports a type constructor for sequences of a type. Whether a sequence is represented as a linked list or an array in the target language is unimportant at the type *specification* level (i.e. within *Lingua Franca*) but can be selected with the `Implement` operation. The representation type is not part of the conformance check and can be different in a client and server.

Lingua Franca defines a single set of semantics for passing arguments and results during method invocation. Interface types (objects) are passed by reference. Data types are passed by value. Passing methoded objects by value is really an object mobility issue and involves moving the implementation and state of the object (and its closure) between programs. This information is currently outside the scope of *Lingua Franca*. Data types are passed by value so that they may be accessed by the remote domain efficiently. While it might be useful to allow programmers the choice of alternate semantics such as pass by reference or copy over/copy back, in practice this has not been necessary and the added flexibility does not seem to warrant the complexity it would add to the language's syntax and semantics, conformance rules, and runtime support libraries.

3.1.1 Grammar

A program written in *Lingua Franca* consist of statements for binding values to identifiers. Subsequent uses of an identifier in the program refer to its value and are permissible anywhere its value is. Two kinds of values within a *Lingua Franca* program are allowed: integer and type. An integer is represented as an arithmetic expression in infix notation with the standard rules of precedence. A type is created from a set of primitive types and the set of type constructors for creating more complex types. The syntax for a *Lingua Franca* program is

program ::= [<*statement*>] *

statement ::= <*type-stmt*>
| <*integer-stmt*>
| <*recursive-stmt*>

type-stmt ::= type <*ident*> = <*type*>
[, <*ident*> = <*type*>] * ;

integer-stmt ::= integer <*ident*> = <*expr*>
[, <*ident*> = <*expr*>] * ;

The *type-stmt* declares a set of identifiers, refered to as *type identifiers*, and binds a type value to each of them. Similarly the *integer-stmt* declares a set of identifiers, refered to as *integer identifiers*, and binds an integer value to each of them. The *recursive-stmt* is for building recursive types and is discussed in Section 3.1.3.

Identifiers share the same namespace and may not be redefined. Consequently, an identifier always refers to a same value. In this respect, identifiers should be thought of as a naming mechanism for values and not as a variable with a value. Finally, an identifier must be defined before it is used (the only exception to this rule is within a recursive statement).

Figure 2 lists the entire set of primitive types in *Lingua Franca* and the set of values they define. Records are built using the *record of* type constructor. Discriminated unions are built using the *case of* type constructor and are very similar in form to *datatypes* in ML[16]. A tag or discriminate field for a case type is not necessary as it is implicit in the declaration. Field names within the *case of* and *record of* type constructors must be unique.

Monomorphic sequences are built using the *sequence of* type constructor. Both fixed and variable length sequences can be built by optionally supplying a length to the *sequence of* type constructor.

Type	Description
Integer	a 4 byte scalar value
Byte	1 octet of raw data
Character	an ASCII character
Nil	the single element Nil
String	sequence of ASCII characters
Boolean	TRUE or FALSE
Float	IEEE floating point numbers

Figure 2: Primitive Types in *Lingua Franca*

An object interface is described using the *interface of* type constructor. All methods names of an interface must be unique. Several alternatives were considered for method overloading. Those based on the structure of the types of the arguments/results led to complex rules that did not warrant the small gain in programmer usefulness. Finally, the names of method arguments are optional and for documentation only. As this implies, the names of arguments are not used in conformance checking.

Lingua Franca does not have a separate type constructor for enumerated types, but rather, an enumerated type is treated as a special case of a discriminated union. An enumerated type is described with the *case of* type constructor where the type of the data associated which each tag name is `Nil`. For example the the enumerated type $(red, green, blue)$ is specified as

```
case of
     red : Nil;
     green : Nil;
     blue : Nil;
end case
```

The complete syntax for a type specification is shown in Figure 3.

3.1.2 An Example

The small code fragment detailed below is a sample *Lingua Franca* program that describes a set of interfaces necessary to implement a simple file system. The program defines two interfaces: `FileServer` and `File`. The `FileServer` interface provides the typical operation found on a directory. In particular, opening, creating, removing, and aliasing files. The `File` interface provides methods for reading and writing files.

type ::=	<*type-identifier*>	| <*primitive-type*>
	| <*infinite-union-type*>	| <*sequence-of-type*>
	| <*pointer-to-type*>	| <*case-of-type*>
	| <*record-of-type*>	| <*interface-of-type*>
primitive-type ::=	Integer | Byte | Character | Nil | Float | String | Boolean	
infinite-union-type ::=	Anything	
sequence-of-type ::=	sequence of [<*expr*>] <*type*>	
pointer-to-type ::=	pointer to <*type*>	
case-of-type ::=	case of	
	[<*name*> : <*type*> ;] *	
	end case	
record-of-type ::=	record of	
	[<*name*> : <*type*> ;] *	
	end record	
interface-of-type ::=	interface of	
	[<*name*> (<*argument-list*>) : <*type*> ;] *	
	end interface	
argument-list ::=	[<*argument*> [, <*argument*>] *]	
argument ::=	[<*name*> :] <*type*>	

Figure 3: Syntax for Type Specifications

```
type Buffer = sequence of Byte;
type File = interface of
   read( pos : Integer, len : Integer )
         : Buffer;
   write( Buffer, pos : Integer )
         : Integer;
end interface;
type FileServer = interface of
   open( name : String )
         : pointer to File;
   create( name : String )
         : pointer to File;
   remove( name : String )
         : Boolean;
   link( name : String, File )
         : Boolean;
end interface;
```

3.1.3 Recursive Types

Intuitively, a recursive type in one that is constructed by referencing itself. Let S be the set of types used to construct T. The closure of T is S unioned with the closure of each element of S. By definition, the closure of a primitive type is the empty set. A type T is defined to be recursive if it is in its own closure. In *Lingua Franca*, type identifiers must be defined before they are used. Consequently, recursive types can not be described without additional rules allowing, at least, the ability to refer to a type in its own construction.

A possible solution is to allow a type identifier to remain unbound during its own definition. This allows types that are self recursive to be described but cannot describe mutually recursive types. The types t_1 and t_2 are mutually recursive if t_1 is used in the construction of t_2 and conversely t_2 is used in the construction of t_1. What is needed is the ability to refer to unbound types or, in terms of *Lingua Franca*, to use an identifier before defining it. This is the purpose of the recursive statement. The recursive statement declares a set of type identifiers and, within this statement, allows them to be used before they are defined. The recursive statement has the syntax:

recursive-stmt ::= recursive <*type-stmt*>

Care must be taken so that the value described by a type has a finite representation. For example, if no restrictions are placed on the use of undefined identifiers within a recursive statement the following type, which describes values with no finite representation, could be specified:

```
recursive type
  Infinite = record of
               field1 : Integer;
               field2 : Infinite;
             end record;
```

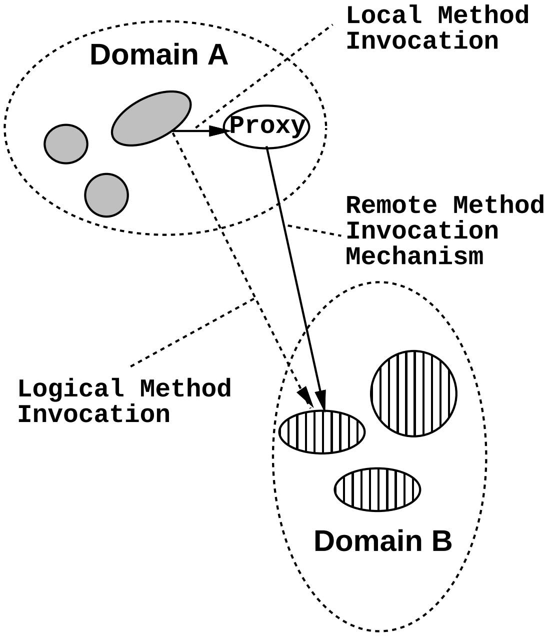

Figure 4: Proxy Object

To prevent this problem from occuring, the use of an undeclared type identifier within a recursive statement is limited to:

1. the type used in a pointer type constructor
2. an argument to or result of a method in an interface type constructor

3.2 The Runtime Support Libraries

In our system, all interaction between client and server domains are through method invocations on remote objects. All objects have a globally unique identifier. These identifiers can be thought of as *handles* to the objects. Passing an object between domains amounts to passing the object's handle. Transparent use of these handles is achieved through the use of *proxy* objects[22]. As shown in Figure 4, a proxy is a local representation of a remote object and maps the language notion of procedure call or method invocation transparently into a remote method invocation. In our system, the local proxy object is an instance of a class generated automatically from a *Lingua Franca* interface description. The code implementing the proxy, including the necessary argument marshalling and unmarshalling, is also automatically generated from this description.

The runtime system supports a universal operation applicable to all values described by *Lingua Franca* programs: the `signature` operation. This operation is used to retrieve the type information needed during narrowing. The `signature` operation returns a *signature object* that describes the type of the value to which it was applied (i.e. all values are self-describing). Signature objects and the data they contain are automatically generated from a *Lingua Franca* program by the language translators. The signature object provides a complete description of the type. For an interface type this includes the names, return types, and argument types of all its methods. For a record, or union type, this includes the names and types of each field. The signature object describing the type of a value provides sufficient information to create at runtime a new *Lingua Franca* program describing the type of the value. The new *Lingua Franca* program may differ syntactically from the original, but, the value's type in the new and original program will be structurally equivalent. The signature objects provide the necessary runtime type information to perform conformance checking. As an optimization, results of a conformance check are cached using the handles of the signature objects compared by the check.

The *Renaissance* system provides support for *Lingua Franca* derived programs running on UNIX systems or on a native object-oriented operating system kernel. Translators are currently supported to map *Lingua Franca* programs into C++[23] for compiled applications and Python[25] for interpreted applications. Additional translators of *Lingua Franca* to other programming languages are being investigated along with the corresponding runtime libraries. The *Renaissance* system and tools provide support for inter-domain object method invocations (transparently) using *Lingua Franca* object interface descriptions. To promote interoperability and scalability, no assumptions are made about what language is used within a domain or on what operating system or architecture the domain in running. For example, the current prototype supports UNIX application domains invoking methods of objects that are part of an object-oriented operating system kernel running on a separate machine (and vice-versa). We implemented this operating system as a proof of concept. The operating system's services (kernel, file system, network, devices, etc.) are all *Renaissance* domains.

The transport mechanisms used to perform cross-domain messaging are implemented as part of the *Renaissance* runtime support libraries and are modular and based upon the domains of the client and server[17]. Communication protocols have been designed and built to support machine independent, cross domain method invocations using shared memory and UDP/IP packet transport. In cases where client and server turn out to be in the same domain, the object invocations are optimized to be local language method invocations, i.e., there are no proxies for properly conforming objects in the same address space.

4 Conformance Rules

We explore two approaches to conformance in a distributed system to see how they effect interface independence and versioning. The first approach, used by conventional large-scale distributed systems, we call *inheritance subtyping* and bases conformance on a hierarchy of type names. The second approach, used in our distributed system, is called *structural subtyping* and bases conformance on the structural compatability of the server's type and the client's requirements. We will show the choice of structural subtyping over inheritance subtyping to be more flexible.

For simplicity, all examples that follow use the *Lingua Franca* IDL described earlier. For illustrations of interface inheritance it is necessary to introduce additional syntax. The '+' type constructor will be used to denote interface inheritance and will only be used in examples using inheritance subtyping. Also, for the purposes of clarity in certain examples underscores in names will be used which, strictly speaking, *Lingua Franca* does not allow.

4.1 Inheritance Subtyping

A common approach to checking conformance is for each object to describe its interface with a name and to have agreement between interacting components as to what the names "mean". A mechanism is usually provided for interfaces to *inherit* from other interfaces. Interfaces that inherit from other interfaces are termed *subclasses* (for historical reasons) of those interfaces and this relationship applies transitively. A subclass' interface is a copy of the direct parent's interfaces plus any additional methods the subclass chooses to add. For inheritance subtyping, conformance is defined by the inheritance relationship between interfaces. Interface I_{sub} conforms to interface I_{super} if and only if either I_{sub} is I_{super} or I_{sub} is transitively a subclass of I_{super}. It is interesting to note that inheritance subtyping is really an extension of name *equivalence* that allows types to have a finite set of names.

Changes to the system are described through new named interfaces that are added to the mutually agreed upon hierarchy. As long as the new type of a server inherits from its old type, the clients of the server need not be modified. Interface independence is achieved only if the inheritance relationship described above exists between the interfaces.

Versioning of a server is easily handled with interface inheritance. Each version of the server is described by a separate named interface, and a server multiply inherits the versions of the service it is willing to provide. During narrowing, the client supplies the name of the version needed and if the server has inherited this version it conforms.

A potential problem with this solution is that several multiply inherited versions may have methods with the same name but with different semantics or different arguments. Fortunately, there exist rules for method overloading that allow for such conflicts. Method names can be disambiguated on the names of the types of arguments and/or the client can specify the name of the interface to which the method belongs.

4.2 Structural Subtyping

Rather than using an interface hierarchy for conformance, structural subtyping considers the type of an object as simply its interface specified as a set of named methods and their argument and result types. Client requirements, specified as a set of methods *expected*, are compared at runtime to server object interfaces using the standard subtype notion of method subset matching with covariance of return types and contravariance of arguments. The complete rules we have chosen for interface conformance are discussed in Section 4.3, but it is important to note that a client can chose to over specify the requirements of the arguments (it provides more than is expected by the server), and may under specify the requirements of results (it accepts back less than the server specifies).

Interface independence is achieved as a consequence of the subtyping rules used. Servers' interfaces may change and, as long as the changes do not effect the clients' compatability with these servers, the clients need not change. To reduce the dependency between clients and servers, clients should describe their requirements with the narrowest interface possible. This means specifying only the methods needed, overspecifying the arguments to these methods, and specifying their result types with the minimum requirements possible. This is not as easy to achieve with inheritance based subtyping.

At first glance versioning seems straight forward; an object can support multiple versions by describing it's interface as the union of the versions it wishes to provide. But again, there is the problem with method name clashes between versions. In the case of structural subtyping, it is not possible to rely on the names of interfaces or the names of the types of the arguments to aid in the disambiguation. Our solution is to consider an object's type not as a single interface but rather a set interfaces (i.e. a set of versions). During narrowing, each of the server object's interfaces is tested for conformance to the client's interface. If at least one of the interfaces conforms, the server object conforms to the client requirements. The interfaces can be sorted by version number from most recent to least recent. If the multiple versions conform to the client's interface the most recent one is chosen. Note, the type of the object is not the union of the interfaces but rather their disjoint union.

4.3 Specific Lingua Franca Conformance Rules

As mentioned already, we have chosen to use structural rather than name based conformance rules. Roughly speaking, if one type has "at least" as much as another type, the former conforms to the later. For example consider the following types *Sub* and *Super*

```
type Super = record of
                 s : String;
                 i : Integer;
             end record;
type Sub =   record of
                 i : Integer;
                 s : String;
                 b : Boolean;
             end record;
```

Since all the fields of type *Super* are in type *Sub*, it is easy to see that a value of type *Sub* can be used anywhere a value of type *Super* is expected. In this case, *Sub* conforms to *Super*.

Formally in our system, a type α conforms to type β if and only if one of the following is true:

1. β and α are interface types and for each method μ_β of β there exist a method μ_α of α such that the following conditions hold:
   (a) μ_α and μ_β have the same name and number of parameters
   (b) the parameters of μ_β conform to the parameters μ_α
   (c) the result of μ_α conforms to the result of μ_β
2. α and β are primitive types and α equals β
3. α is the type "sequence of ϵ_α" and β is the type "sequence of ϵ_β" and ϵ_α conforms to ϵ_β
4. α is the type "sequence of N_α ϵ_α" and β is the type "sequence of N_β ϵ_β" and ϵ_α conforms to ϵ_β and $N_\alpha = N_\beta$
5. α is the type "sequence of N_α ϵ_α" and β is the type "sequence of ϵ_β" and ϵ_α conforms to ϵ_β
6. α is the type "pointer to ρ_α" and β is the type "pointer to ρ_β" and ρ_α conforms to ρ_β
7. α and β are case types and for each field ϕ_α of α there exist a field ϕ_β of β such that the following conditions hold:
   (a) ϕ_α and ϕ_β have the same name
   (b) the type of ϕ_α conforms to the type of ϕ_β
8. α and β are record types and for each field ϕ_β of β there exist a field ϕ_α of α such that the following conditions hold:
   (a) ϕ_α and ϕ_β have the same name
   (b) the type of ϕ_α conforms to the type of ϕ_β
9. β is the *Anything* type (i.e. all types conform to *Anything*)

Our rules for conformance assume that arguments are passed without side effects. This is true given our chosen semantics for argument passing. For this reason, it is only necessary to test the contravariance relationship for arguments. If, at a latter stage, the language is extended to support pass-by-reference or pass-by-value-return then the conformance rules will have to be modified so that the the covariance relationship for arguments with side-effects is tested as well.

The rules for conformance are recursive and when applied to recursive types equate to testing that the infinite expansions of the types structurally conform. Unfortunately, this implies the rules are not finitary. However, finitary algorithms exist that test the rules of structural conformance even in the presence of recursion[2]. To understand how this works, we first define the representation of a type to be a directed graph where the nodes of the graph represent types and edges are pointers to types used in the source node's construction. The nodes contain the necessary information to perform the conformance test. E.g., a structure node contains a name for each outgoing edge. Every type can be represented as a graph with a finite number of nodes. Cycles in the graph indicate recursion. The algorithm to test structural conformance takes two nodes in the graph and returns TRUE or FALSE whether the first argument structurally conforms to the second argument. A high-level outline for the algorithm is:

1. If F conforms to T then return TRUE.
2. Assume F conforms to T. Future recursive calls to the algorithm with (F,T) as the argument will return TRUE immediately in Step 1.
3. Verify F conforms to T by testing the conformances rules using the data within the nodes and return either TRUE or FALSE. This checking may make recursive calls to this algorithm but recursive calls with (F,T) as the argument will return immediately with TRUE.

A detailed look at an equivalent algorithm is provided in [2]. The algorithm guarantees the conformance rules are never verified for a pair of types more than once. Since the number of nodes in a type graph is finite, the number of verifications of the conformance rules are finite, and the algorithm will terminate even in the presence of recursion. In the absence of recursion steps 1 and 2 of the algorithm are not necessary.

4.4 Comparison of Interface and Structural Subtyping

The problem with inheritance subtyping is that it increases coupling between clients and servers. Clients must specify their requirements as a named interface and can only inter-operate with server objects that are in a inheritance relationship with this interface. Often, a client may have to over specify its actual requirements resulting in a situation where changes to an object's interface unduely effect the inter-operability between the client and server. The following is an example of this problem which we term the *false interface dependency* problem:

```
type VarOpaque = sequence of Byte;
type File = interface of
    read( Integer ) : VarOpaque;
    write( Integer, VarOpaque )
        : Integer;
    length() : Integer;
    kind() : case of
        FILE: Nil;
        DIRECTORY : Nil;
    end case;
end interface;
```

```
NewFile = interface of
    read( Integer ) : VarOpaque;
    write( Integer, VarOpaque )
        : Integer;
    length() : Integer;
    kind() : case of
        FILE: Nil;
        DIRECTORY : Nil;
        PIPE : Nil;
    end case;
end interface;
```

In this example a file system has been extended to support named pipes. Unfortunately, `NewFile` cannot be placed in a subclass relationship with `File`. Without going into a rigorous proof, consider what might happen if `NewFile` were a subclass of `File`. Clients expecting an object of type `File` could be given objects of type `NewFile`. This leads to type errors since clients only expect the kind() operation to return either FILE or DIRECTORY and invocations returning PIPE would not be expected. Consequently, clients written to use objects of type `File` can not inter-operate with servers exporting objects of type `NewFile`. This is true even if the actual requirements of the client would allow it to inter-operate with both `File` and `NewFile`. For example, a print server that takes a `File` object for printing may actually only require the read and length operation but, because the print server over specified its requirements as taking a `File` object, it cannot inter-operate with objects of type `NewFile`:

```
type PrintServer = interface of
    print( File ) : Boolean;
end interface;
```

The `File` and `NewFile` interfaces can be used to illustrate another problem; that the compatability between clients and servers may be falsely dependent upon the inheritance hierarchy. For example, `NewFile` could not be made a subclass of `File`, but the converse is not true; `File` could be made a subclass of `NewFile`. This would allow clients using the `NewFile` interface to inter-operate with objects of type `File`. However, this requires retroactively modifying the hierarchy which is not practical since it is globally shared among cooperating domains. Consequently, even if the print server is rewritten to accept objects of type `NewFile`, it will not inter-operate with objects of type `File`. We term this second problem the *false hierarchy dependency problem*.

There is a subtle difference between the two problems. In the false interface dependency problem, interfaces can not be placed in an inheritance relationship. In the false hierarchy dependency problem, it is not that the interfaces can not be placed in an inheritance relationship, but, rather they are not for what ever reason. By not placing interfaces in an inheritance relationship, clients may be unnecessarily incompatable with servers.

Both of these problems are solved trivially using structural subtyping. The print server need only specify its requires as a narrower view of `File` that objects of both type `File` and type `NewFile` conform to:

```
type VarOpaque = sequence of Byte;
type PrintServerFile = interface of
    read( Integer ) : VarOpaque;
    length() : Integer;
end interface;
type PrintServer = interface of
    print( PrintServerFile )
        : Boolean;
end interface;
```

Using structural subtyping the print server will work with both `File` and `NewFile` objects.

Appendix A further illustrates the differences between structural subtyping and interface inheritance by discussing a scheme to approximate structural subtyping with interface inheritance. The net result is that interface subtyping is only minimally effective at achieving the expressive power of structural subtyping especially in the presence of recursive types and contravariance of arguments.

4.5 Problems with Structural Subtyping

One possible disadvantage to using structural subtyping is the loss of the semantic information that hierarchies of interface names provide. We address the problem in *Lingua Franca* by supporting the augmentation of objects with semantic attributes. Semantic attributes are treated as type predicates on the object. For example, `sorted` or `FIFO`. In addition to supplying an interface when narrowing an object reference, a client may supply a list of semantic attributes for checking semantic conformance. For example, a client's interface to a file system might contain `read` and `write` methods. Many different file systems types could support this as a subset of their interface. For example, `ReplicatedFile`, `CompressedFile`, `EncryptedFile`. Clients which do not care which type of file they use, can simply narrow references they obtain to their notion of a file. An application which needs encrypted files, can specify the `encrypted` attribute during the narrowing (assuming the `EncryptedFile` objects are the only type of file objects that possess this semantic attribute).

The argument might be made that the different file types would also have additional methods. For example, `EncryptedFile` might provide a `password` operation. These additional methods could also be used by the client to distinguish between the different types of files. A client interested in an encrypted file could specify this by including the `password` operation in the interface used during narrowing. This technique actually implies the way in which semantic attributes are implemented: as nullary methods automatically added to the client's interface during narrowing and to the interface of objects specified to possess them. Attributes are still useful when the semantic information can not be inferred from the interface. For instance, consider the interfaces `stack` and `queue` that both have the same set of methods: `add` and `remove`. In this case, where a client is unable to distinguish between a stack and queue object based interface alone, the semantic attributes `LIFO` and `FIFO` could be used.

Of course attributes are not necessary at all. As eluded to earlier, attributes can be implemented as nullary methods on interfaces (e.g. an `ImAFIFO()` method). However, it is our opinion that the semantic information of an object and its interface are conceptually different and a clear distinction between them should be made.

5 Performance

A complete analysis of the performance of our system is beyond the scope of this paper and is detailed elsewhere[17]. However, it is interesting to look at two facets of the system to put it in perspective with other work.

5.1 Performance Of Cross Domain Method Invocation

Figure 5 shows the method invocation times for a null method (no arguments/no results) for various combinations of *Renaissance* clients and servers. These data and all the following were gathered on SPARCStation IPX's and SPARCStations 10's with 32Mb of memory on an otherwise idle Ethernet. The test includes using a *Renaissance* kernel resident server, a user-level application running on that kernel and a UNIX user-level application server, as well as clients in all three of these same domains. UNIX clients and severs were run on different machines, the kernel and its applications were run on

Clients	Kernel	*Server* Application	UNIX
Kernel	0.29μs	501μs	1999μs
Application	51μs	626μs	1959μs
UNIX	2544μs	2295μs	3137μs

Figure 5: Method Invocation Times for Various Client/Server Domain Combinations

	Kernel	Application	UNIX
1	16μs	414μs	2914μs
2	29μs	437μs	2970μs
3	45μs	460μs	2970μs
4	62μs	460μs	2976μs
5	81μs	487μs	3020μs

Figure 6: Reference Narrowing Times for a Kernel Server Object with 5 Methods

the same machine.

The affect of our support for customized transports in the runtime support libraries is evident in the speed of the *Renaissance* kernel to kernel method invocation times (it is optimized to a C++ virtual function call) and the application to kernel time (it is optimized to used shared memory). The remaining cases use UDP/IP as a transport and also pay the cost of data canonicalization.

5.2 Performance of Run-Time Conformance Checking

Figures 6 and 7 show the expense of narrowing a reference to a client's interface for various types of client applications contacting a kernel server and a UNIX server respectively. These number were gathered with the conformance checking cache disabled, and therefore these times reflect the actual time to run the conformance checking algorithm, including transport overheads. The leftmost column lists the number of methods in the client interface. All the methods in all the interfaces were null methods, so these times represent the minimum cost of narrowing references in each case. Argument/result conformance checking will add to the expense depending on the complexity of the types involved.

	Kernel	Application	UNIX
1	2503μs	2585μs	4282μs
2	2474μs	2542μs	4170μs
3	2503μs	2584μs	4164μs
4	2519μs	2572μs	4470μs
5	2536μs	2642μs	4218μs

Figure 7: Reference Narrowing Times for a UNIX Server Object with 5 Methods

6 Related Work

Clients in traditional distributed systems such as V [8, 7], Mach [1], and Chorus [21] 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 run-time representation for types and therefore cannot perform conformance checking. Also, in these systems the burden of data canonicalization and type checking is placed on the programmer.

Our approach is obviously related in overall philosophy to a number of other systems that use an interface description language. Object-based systems including OMG's CORBA[19], Microsoft's OLE2[15], IBM's SOM/DSOM[12] and Sun's Spring system[11] all provide an IDL for describing objects and translators for mapping these descriptions into target languages. Likewise, remote procedure call[5] (RPC) based systems such as SUN RPC[24] provide an IDL for describing services but in terms of procedures not objects. Clients in RPC-based systems acquire services by invoking local functions that transparently access remote services. Most RPC-based systems provide the notion of a program and a set of procedures to call within the program and is analogous to an object with methods. Although, object-based and RPC-based systems appear very similar, there are two important differences. First, unlike object-based systems where interfaces are first class types, most RPC-based systems do not support procedures (or programs) as first class types. Secondly, RPC-based systems use name equivalence for conformance checking and do not have a hierarchy of type names. What separates our work from the others systems is the use of an IDL that uses structural subtyping rather than names of interfaces for conformance testing.

Distributed object languages, such as Argus [14], Distributed Smalltalk [3], and Emerald [20], and distributed object systems, such as Clouds [9] and

Eden [13], not only provide a notion of objects and type conformance but also provide features such as concurrency and atomicity, replication, persistence, fault tolerance, and migration. Unfortunately, the requirements placed on these systems to support these features makes it difficult for them to scale and interoperate with one another. In our system, such features are considered object-specific and would be provided by the implementation of the object and hidden from the client behind the object's interface. We emphasize the accessing of remote objects independent of their particular implementation and provide a framework in which systems can interoperate.

7 Conclusion

We believe the use of structural subtyping to be superior in achieving the high degree of autonomy between domains and the low coupling between software components necessary for a scalable, extensible distributed system. With the structural subtyping our system provides there are no global type hierarchies to share among domains, but rather, all objects are self describing. These descriptions, available to the programmer via the signature objects, are used for conformance testing and documentation. The lack of a global type hierarchy allows new service types to be easily added to the system and existing service types to be easily modified.

The structural subtyping conformance rules we have chosen are more flexible and natural than inheritance based name conformance rules. Conformance tests that fail using inheritance may succeed using structural subtyping. Also, programmers can make use of the flexible and dynamic nature of our conformance test. For instance, an object, for security purposes, can selectively export different subsets of methods by "pretending" to have different interfaces during the narrowing process. This subset may be determined at run-time based on the client's identity.

The self-describing nature of our system has several advantages. First, the distributed environment is more manageable. As software evolves and clients become incompatible with servers, the run-time type information can be used to determine how they are incompatible and what changes need to be made to the client. The run-time type information also provides on-line documentation about an object's interface and can aid in resource discovery. Second, our system supports statically and dynamically typed programming equally well. A C++ translator for *Lingua Franca* has been built that maps *Lingua Franca* type descriptions into C++ types. Once an object is narrowed, the static type checking of C++ guarantees its proper use. Also, a World Wide Web based object browser/invoker was easily constructed by mapping dynamically discovered object interfaces into HTML documents and constructing a HTML form based translator that invokes dynamically discoved interface methods.

Using structural subtyping for run-time conformance checking is not without cost. The run-time type information imposes some size penalty on individual objects, but much of this information can be shared between objects of the same type. The greatest expense lies in the run-time costs of the conformance checking algorithm that were discussed in Section 5.2.

Our work provides programmers with an object-oriented framework within which to construct distributed object systems. In addition, it brings a language-independent notion of object and type (interface) closer to reality. Finally, providing the ability to separate implementation details from interface details, and semantic information from syntactic, allows the construction of less coupled, more extensible distributed software.

Appendix A: Approximating Structural Subtyping Using Interface Inheritance

To further illustrate the difference between structural subtyping and interface inheritance is interesting to try to approximate structural subtyping with interface inheritance. For this section we will use the following symbols:

- $I_x :< I_y$ if and only if I_x structurally conforms to I_y
- $I_x = I_y$ if and only if $I_x :< I_y$ and $I_y :< I_x$ (this is also termed *structural equivalence*)
- $I_x < I_y$ if and only if I_x is a subclass of I_y (i.e. I_x inherits from I_y)

To approximate structural subtyping for an interface I_s we mean that for each interface I_c where $I_s :< I_c$ there exist an interface $I_{c'}$ such that $I_s < I_{c'}$ and $I_c = I_{c'}$. I.e., for each client requirement that the server structurally conforms to there must exist an interface in the hierarchy equivalent to it. We assume the interface hierarchy is comprised of a finite set of interfaces and therefore the best we can hope to do is approximate a finite set of client requirements. However, it can be shown that there are potentially an infinite number of distinct client interfaces that a server conforms to[18]. Two interfaces are distinct if they are not structurally equivalent.

The general approach is to enumerate all possible structural supertypes (client requirements) of an interface and use multiple inheritance to build up the interface hierarchy. To guarantee that the proper structural subtyping relationships are approximated in the resulting hierarchy, the following invariant should hold: *if an interface is a structural subtype of another interface then it must be a subclass of the interface.*

The scheme we use for the approximation takes each interface I_s and gives a name to all possible combinations of the methods of I_s. This is done by creating a separate interface for each method in I_s and using multiply inheritance to create the possible combinations of them. For example consider the following interfaces `A` and `B`:

```
type B = interface of
    b1() : Nil;
    b2() : Ni;
end interface;
type A = interface of
    a1() : B;
    a2() : Nil;
    a3() : Nil;
end interface;
```

The resulting interface hierarchy for `A` and `B` would be:

```
type B_b1 = interface of
    b1() : C;
end interface;
type B_b2 = interface of
    b2() : C;
end interface;
type B = B_b1 + B_b2;
type A_a1 = interface of
    a1() : B;
end interface
type A_a2 = interface of
    a2() : Nil;
end interface
type A_a3 = interface of
    a3() : Nil;
end interface
type A_a1_a2 = A_a1 + A_a2;
type A_a1_a3 = A_a1 + A_a3;
type A_a2_a3 = A_a2 + A_a3;
type A = A_a1_a2 + A_a1_a3 +
        A_a2_a3;
```

Clients can now specify their requirements as the precise set of methods needed for `A` and `B`. Care must be taken when defining the interfaces and their inheritance relationships so that invariant holds. If `A` were defined as:

```
type File = A_a1 + A_a2 + A_a3;
```

then `A` would not inherit from `A_a1_a2`, the invariant would not hold, and objects of type `A` could not be used where objects of type `A_a1_a2` are expected.

This approximation scheme has severe limitations. It does not handle contravariance of arguments or covariance of results. Contravariance allows clients to over specify the requirements of an argument. E.g., there are an infinite number of distinct interfaces for this over specification. Approximating contravariance requires giving a name to each of these possible interfaces which violates the restriction that a finite number of interfaces be used in the approximation. Covariance allows clients to under specify their requirements. By re-appling our approximation scheme to the results of the method, it is possible to approximate covariance. For each method, a set of methods is created one for each possible supertype of the result. Each of these are placed in a separate interface and multiple inheritance is used to create the proper structural subtyping relationship. Reconsidering the interfaces `A` and `B` again, and applying the scheme to the results of methods, yields the following changes to the previous hierarchy:

```
type A_a1_b1 = interface of
    a1() : B_b1;
end interface
type A_a1_b2 = interface of
    a1() : B_b2;
end interface
type A_a1 = A_a1_b1 + A_a1_b2 +
            interface of
                a1() : B;
            end interface;
type A_a2 = interface of
    a2() : Nil;
end interface
type A_a3 = interface of
    a3() : Nil;
end interface
type A_a1_b1_a2 = A_a1_b1 +
                  A_a2;
type A_a1_b1_a3 = A_a1_b1 +
                  A_a3;
type A_a1_b2_a2 = A_a1_b2 +
                  A_a2;
type A_a1_b2_a3 = A_a1_b2 +
```

```
                 A_a3;
type A_a1_a2 = A_a1 + A_a2 +
               A_a1_b1_a2 +
               A_a1_b2_a2;
type A_a1_a3 = A_a1 + A_a3 +
               A_a1_b1_a3 +
               A_a1_b2_a3;
type A_a2_a3 = A_a2 + A_a3;
type A = A_a1_a2 + A_a1_a3 +
         A_a2_a3;
```

A graphical picture of the resulting hierarchy is shown in Figure 8. The solid lines and ovals are from appling our scheme to `A`, the dashed lines and ovals are from re-appling our scheme a second time to the results of the hierarchy. One thing that is immediately apparent is the relatively complicated hierarchy that results from such a seemingly innocent problem. The solution suffers from the need to pollute the namespace with the addition interface names and a complex inheritance scheme.

This scheme can iteratively be applied to the resulting hierarchy until all supertypes of an interface are enumerated. However, there is a serious short coming to this approach: for recursive interfaces the scheme can potentially be applied infinitely many times. It can be shown that recursive interface potentially have an infinite number of distinct client interfaces it structurally conforms to, but only a finite number of these client requirements (the ones named) can be approximated. For example consider the following recursive interfaces:

```
type Directory = interface of
    openDirectory( String )
        : pointer to Directory;
    list() : sequence of String;
    ...
end interface;

type NewDirectory = interface of
    openDirectory( String )
        : pointer to NewDirectory;
    list() : sequence of String;
    ...
end interface;

type ClientDirectory = interface of
    openDirectory( String )
      : pointer to ClientDirectory;
    list() : sequence of String;
end interface;
```

Assume `NewDirectory` cannot be placed in a subclass relationship with `Directory` and the client requirements are specified by `ClientDirectory`. Applying our scheme a finite number of times will not yield an interface hierarchy that approximates the `ClientDirectory` interface. However, both `Directory` and `NewDirectory` are structural subtypes of `ClientDirectory`.

We have presented only one approximation scheme. One that fully approximates non-recursive types when contravariance is ignored. This is possible because all possible structural supertypes of a non-recursive type can be enumerated (ignoring contravariance). There are others, but they all have the same general form. A finite set of client requirements are enumerated and interface inheritance is used to establish the proper structural subtyping relationships between them. Consequently, they all have the same problems and limitations: they clutter the namespace, they add complexity to the hierarchy, and they fail to fully approximate interfaces in the presence of recursion. For this reason, these approaches are of mostly theoretical interest. In practice, the programmer of the interface decides which client requirements to support usually by subclassing from a small number of existing interfaces or by creating a small number of new interfaces. The problem with this is the programmer of the *server* determines what are the possible *client* requirements.

References

[1] Mike Accetta et al. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the USENIX Conference*, pages 93–111, June 1986.

[2] Roberto M. Amadio and Luca Cardelli. Subtyping Recursive Types. *ACM Transactions on Programming Languages and Systems*, 14(4):575–631, 1993.

[3] John K. Bennett. The Design and Implementation of Distributed Smalltalk. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 318–330, 1987.

[4] Edward V. Berard. *Essays on Object-Oriented Software Engineering*, volume 1. Prentice Hall, Englewood Cliffs, New Jersey, 1993.

[5] Andrew Birrell and Bruce Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1), February 1984.

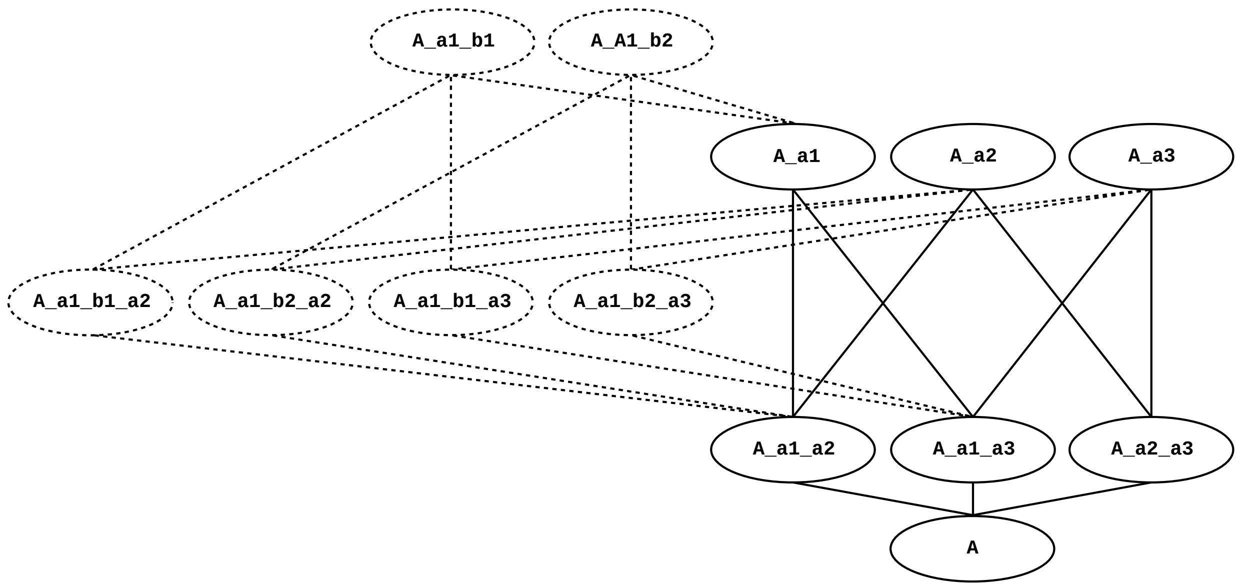

Figure 8: A Multiple Inheritance Hierarchy for interface A

[6] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[7] David R. Cheriton. The V Kernel: A Software Base for Distributed Systems. *IEEE Software*, 1(2):19–42, April 1984.

[8] David R. Cheriton. The V Distributed System. *Communications of the ACM*, 31(3):314–333, March 1988.

[9] Partha Dasgupta, Richard LeBlanc, and William Appelbe. The Clouds Distributed Operating System:Functional Description, Implementation Details, and Related Work. Technical Report GIT-ICS-87/42Functional Description, Implementation Details, and Related Work, Georgia Tech, 87.

[10] Carlo Ghezze, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering.* Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[11] G. Hamilton, M. Powell, and J. Mitchell. Subcontract: A Flexible Base for Distributed Programming. In *Proceedings of the ACM Symposium on Operating System Principles*, December 1993.

[12] IBM. *SOMobjects Developer Toolkit Users Guide Version 2.0*, 1993.

[13] E. Lazowska, H. Levy, G. Almes, M. Fischer, R. Fowler, and S. Vestal. The Architecture of the Eden System. In *Proceedings of the ACM Symposium on Operating System Principles*, pages 148–159, 1981.

[14] Barbara Liskov. Distributed Programming in Argus. Technical Report Programming Methodology Group Memo 58, MIT, October 1987.

[15] Microsoft Corporation. *OLE2 Programmer's Reference*, volume 2. Microsoft Press, 1994.

[16] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* The MIT Press, Cambridge, Massachusetts, 1990.

[17] Patrick A. Muckelbauer and Vincent F. Russo. Efficient Remote Method Invocation in a System Utilizing Structural Subtyping. Technical report, Department of Computer Sciences, Purdue University, June 1995.

[18] Patrick A. Muckelbauer and Vincent F. Russo. Lingua Franca: an IDL for Structurally Subtyping Distributed Systems. Technical report, Department of Computer Sciences, Purdue University, June 1995. This is and expanded form of the current paper.

[19] OMG. *The Common Object Request Broker: Architecture and Specification*, 1991.

[20] Rajendra K. Raj, Ewan Tempero, Henry M. Levy, Andrew P. Black, Norman C. Hutchison, and Eric Jul. Emerald: A General-Purpose Programming Language. *Software – Practice and Experience*, 2(1):91–118, January 1991.

[21] M. Rozier, V. Abrossimov, and W Neuhauser. CHORUS-V3 Kernel Specification and Interface, Draft. Technical Report CS/TN-87-25.10, CHORUS Systems, February 1988.

[22] Marc Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle. In *Proceedings of the 6th. International Conference on Distributed Computer Systems*, May 1986.

[23] Bjarne Stroustrup. *The C++Programming Language*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[24] Sun Microsystems. *Networking on the SUN Workstation*, 1985.

[25] Guido van Rossum. *Python Reference Manual*. CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, 1994.