Interoperability

PETER WEGNER

Brown University, Providence, Rhode Island ⟨pw@cs.brown.edu⟩

Interoperability is the ability of two or more software components to cooperate despite differences in language, interface, and execution platform. It is a scalable form of reusability, being concerned with the reuse of server resources by clients whose accessing mechanisms may be plug-incompatible with sockets of the server. Plug compatibility arises most literally with electrical appliances that require both static compatibility of shape and dynamic compatibility of voltage and frequency. If there is no direct match, interoperability of electrical appliances can be achieved by adapters and transformers. The client-server software paradigm is a plug and socket paradigm with static compatibility specified by types and dynamic compatibility by protocols. As with electrical appliances, incompatibility of software plugs and sockets can be mediated by adapters.

Static compatibility between calling and called procedures in strongly typed languages can be determined by type checking. Type compatibility guarantees that the plug of the caller fits the socket of the server but not that the service provided is that expected by the caller. But some of the most powerful systems of interoperation, like UNIX pipes and http World-Wide Web protocols, are typeless. Interface definition languages with static type-compatibility protocols promote safety and efficiency, but the complexity and implementation cost of typed systems for interoperation is unacceptably high in today's technology.

Type differences between data representations, like that between integers and floating-point numbers, can be handled by polymorphism and coercion. Clients and servers from different software platforms or programming languages can talk to each other through *mediators* that convert data formats. Mediation in information systems as an organizing principle for interoperation of heterogeneous components is reviewed in Wiederhold [1995]. Mediation may in turn be viewed as a form of coordination: the growing body of research on coordination languages [Ciancarini et al. 1995] has interoperation of software components as one of its primary goals. The use of "megaprogramming languages" to communicate among heterogeneous interfaces of large components (like city transportation systems) is yet another framework for interoperation [Wiederhold et al. 1992].

In focusing on client-server interoperability, we exclude data interoperability at the level of raw legacy data. The problem of effectively querying and mining (extracting the semantics of) the large volume of acquired signal processing data is of great practical importance but beyond the scope of this paper.

The basic unit of interoperation in the client-server paradigm is the procedure. But procedure-level interoperation is not a sufficient condition, though it is a necessary one for interoperation of software components [Nierstrasz and Tschichritzis 1996]. Software components may require larger-granularity units of interoperation, since the correspondence between client and server operations may not be one to one. Moreover, interoperation may require preservation of temporal as well as functional properties (order constraints on operations or coordination of inputs from multiple input streams). Such protocol constraints cannot be captured by functional correspondences of individual operations.

Interoperability can be realized for a wide range of differences in data formats and for recognized differences of representation, like that between Cartesian and polar coordinates of points. But the general problem of reusing procedure functionality is unsolvable, since functional equivalence is undecidable.

The two major mechanisms for interoperation are interface bridging and interface standardization:

— *Interface Standardization:* Map client and server interfaces to a common representation
— *Interface Bridging:* Two-way map between client and server

Interface standardization is more scalable because m client and n servers require only $m + n$ maps to a standard interface, compared with $m * n$ maps for interface bridging. However, interface bridging is more flexible, since it can be tailored to the requirements of particular clients and servers. Interface standardization makes explicit the common properties of interfaces, thereby reducing the mapping task, and it separates communication models of clients from those of servers. But predefined standard interfaces preclude supporting new language features not considered at the time of standardization (for example, transactions). Standardized interface systems are closed while interface bridging systems are open. Architectures for standardized interfaces may be considered a special case of interface-bridging architectures in which the bridge from clients to servers is replaced by two half bridges from clients to the standard interface, and from the standard interface to the servers.

The common object request broker architecture [Object Management Group 1995] realizes interface standardization by an object request broker (ORB) that serves as an adapter/transformer. The ORB handles communication among application objects, object services (system objects), and common facilities (library objects). Services provided by an object are specified in an interface definition language (IDL) and stored in an interface repository. The IDL specification serves as a target for client requests and a source for server requests.

CORBA provides half bridges from clients to the ORB and from the ORB to servers. Clients may invoke a service through a static stub or through a dynamic invocation created from the IDL specification at run time. The ORB validates client requests against the IDL interface and dispatches them to the server where arguments are unpacked (unmarshalled), methods are executed, and results are returned. Server-side software includes object adapters that bind object interfaces and manage object references, and a server skeleton that uses the output of object adapters to map operators to the methods that implement them.

Microsoft's Component Object Model COM/OLE [Brockschmidt 1995] realizes interoperability through a binary (machine language) standard that specifies multiple interfaces by a pointer to a function table and objects by a principal interface I-unknown for accessing a directory of interfaces through a "queryinterface" function. COM/OLE objects have multiple interfaces that share a common data structure. The multiple-interface model for objects is more scalable than the hierarchical inheritance model of object-oriented programming, providing greater flexibility for both interoperation and extension of object functionality. Class-based inheritance is less scalable than multiple interfaces as an organizing principle for interfaces, both because complex objects are naturally modeled by multiple interfaces and because hierarchies are too restrictive a structuring principle. Use-case models [Jacobson 1991], as well as COM/OLE, elevate collections of interfaces that share a state into a primary structuring principle of software design.

COM/CORBA interoperability, which aims at a compatibility between Microsoft and CORBA-compliant components, is being pursued by a broad industry consortium under the auspices of the object-management group (OMG). Proposals for COM/CORBA interoperability aim to realize interoperation by interface bridging. Clients send a request via a surrogate object to a state that marshals the arguments and sends them across a bridge to a server skeleton that ummarshals the arguments and manages the implementation of operations by methods of the target object. The combination of interface standardization within CORBA and interface bridging to COM provides a balance between closed efficiency and open extensibility expressed by the metaphor of closed islands connected by bridges.

The paradigm shift from algorithms in the 1960s and 1970s to interaction in the 1990s is refocusing attention from inner processes of execution to interactive processes of mediation, coordination, and interoperation among software components. Though interactive processes are inherently less amenable to formal analysis than algorithms [Wegner 1995], heuristic methods of interoperation will play an increasingly important role in the software technology of the 21st century.

REFERENCES

BROCKSCHMIDT, K. 1995. *Inside OLE 2,* 2nd ed. Microsoft Press.

OBJECT MANAGEMENT GROUP. 1995. *CORBA: Architecture and Specification,* (July). Revision 2.0. Object Management Group.

CIANCARINI, P., NIERSTRASZ, O., AND YONEZAWA, A. 1995. In *Proceedings of ECOOP '94 Workshop on Coordination Languages. Lecture Notes in Computer Science 924.* Springer Verlag, New York.

JACOBSON, I. 1991. *Object-Oriented Software Engineering.* Addison-Wesley/ACM Press.

NIERSTRASZ, O. AND TSICHRITZIS, D., EDS. 1996 (especially Chapter 3 by Dimitri Konstantas) *Object-Oriented Software Composition.* Prentice Hall, Englewood Cliffs, NJ.

WEGNER, P. 1995. Interactive foundations of object-based programming. *IEEE Computer* (Oct.).

WIEDERHOLD, G. 1995. Mediation in information systems. *ACM Comput. Surv.* (June).

WIEDERHOLD, G., WEGNER, P. AND CERI, S. 1992. Towards megaprogramming. *Commun. ACM* (Nov.).