Software Interoperability

By

Cameron T. Howie, John Kunz, and Kincho H. Law

CIFE Technical Report #117
December, 1997

STANFORD UNIVERSITY

If you would like to contact the authors please write to:

c/o CIFE, Civil and Environmental Engineering Dept.,
Stanford University,
Terman Engineering Center
Mail Code: 4020
Stanford, CA 94305-4020

SUMMARY

CIFE TECHNICAL REPORT # 117

December 1997

Header: Software Interoperability

Title: Software Interoperability

Authors: Cameron T. Howie, John C. Kunz, Kincho H. Law

Publication Date: December 1997

Funding Sources:

- Name of Agency: Kaman Sciences
- Title of Research Project: Study of Software Interoperability

1. **Abstract:** Brief Summary of Research Objectives and Results:

A significant development in computing in the 1990s has been the move toward more distributed computer systems through network communication facilities. Software interoperability has become a major issue for organizations wishing to take advantage of private and open distributed computing facilities. Software interoperability technologies seek to smooth the integration of both legacy and new applications, even on heterogeneous hardware and software platforms. This report summarizes the significant issues and terminology used in the field of software interoperability, presents a range of diverse systems and methodologies used to integrate knowledge among programs, and concludes with a new view of interoperability and a novel way of addressing it.

2. Subject:

• What is the report about in laymen's terms? Information Technologies (IT) that enable computer programs (applications) to communicate engineering data with each other across networks.

• What are the key ideas or concepts investigated?

• What is the essential message? Current IT supports interoperability, but it requires some effort to develop and support software integration platforms.

3. Objectives/Benefits:

• Why did CIFE fund this research? Kaman Sciences funded this research at the request of the U.S. Air Force and the Electric Power Research Institute (EPRI)

• What benefits does the research have to CIFE members? This research gives a model of how AEC companies can develop interoperable systems to within their firms and across the clients and vendors that are part of their supply chains.

• What is the motivation for pursuing the research? Put many technologies into perspective with relation to each other and identify how they would work in the AEC industry.

• What did the research attempt to prove/disprove or explore? Different technologies that would support software applications interoperability.

4. Methodology:

• How was the research conducted? The research includes a survey of relevant IT methods, and analysis of their features, and a comparison of how they support AEC industry needs.

• Did the investigation involve case studies, computer models, or some other method? Technology survey and analysis

5. Results:

• What are the major findings of the investigation?

• What outputs were generated (software, other reports, video, other) EPRI Technical Report EPRI RP9000-32

6. Research Status:

• What is the status of the research? Software prototype systems are currently being developed and tested

• What is the logical next step? Develop and test an interoperability architecture in the AEC industry.

• Are the results ready to be applied or do they need further development? Modest additional technology integration is needed for use, but the amount is well within the capability of most software and many larger AEC firms.

• What additional efforts are required before this research could be applied? Modest additional technology integration is needed for use, but the amount is well within the capability of most software and many larger AEC firms.

ABSTRACT

A significant development in computing in the 1990s has been the move toward more distributed computer systems through network communication facilities. Instead of viewing computers as individual devices, users want to integrate these physically separate but electronically connected resources into a single logically unified computational environment. Software interoperability has become a major issue for organizations wishing to take advantage of private and open distributed computing facilities. Software interoperability technologies seek to smooth the integration of both legacy and new applications, even on heterogeneous hardware and software platforms. Improved integration not only benefits users within a company but also facilitates the sharing of knowledge with other disciplines and organizations, enhancing the workflow processes. Many companies also have huge investments in legacy systems that they hope to avoid rewriting by using interoperability tools to bring such applications into the network-driven environments of today. This report summarizes the significant issues and terminology used in the field of software interoperability, presents a range of diverse systems and methodologies used to integrate knowledge among programs, and concludes with a new view of interoperability and a novel way of addressing it.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

A significant development in computing in the 1990s has been the move toward more distributed computer systems through network communication facilities. Instead of viewing computers as individual devices, users want to integrate these connected resources into a single computational environment. Both hardware and software vendors are developing means to support distributed computing at both a system and application program level. Interaction in the 1990s is focusing on mediation and cooperation among software components [Wegner 1996].

The need to integrate computer applications by means of an interoperability methodology is greatest in distributed environments that are heterogeneous in their composition of hardware and software resources. By contrast, integrating code in a homogeneous network environment is usually achieved through the design of a distributed operating system. Although some environments are necessarily homogeneous, most organizations and virtually all industries have heterogeneous computing environments.

Therefore, software interoperability is becoming a major issue for organizations wishing to take advantage of private and open distributed computing facilities. Interoperability is usually defined as "the capability with which two or more programs can share and process information irrespective of their implementation language and platform." Already, many Information System managers and application developers are considering systems that support the "client-server" model of resource utilization. In this context, software interoperability technologies seek to smooth the integration of both legacy and new applications, even on heterogeneous hardware and software platforms. Improved integration not only benefits users within a company, but also facilitates the sharing of knowledge with other disciplines and organizations, enhancing the workflow processes. Many companies also have huge investments in legacy systems that they hope to avoid rewriting by using interoperability tools to bring such applications into the network-driven environments of today.

This report summarizes basic interoperability principles and the different approaches that researchers have taken to provide software interoperability. In addition to surveying these methodologies, the report concludes with an outlook on the future of software interoperability and a novel means of integrating applications in an industrial environment.

2 LEVELS OF SOFTWARE INTEROPERABILITY

Figure 2-1 shows how software interoperability between different applications can be modeled at different levels of abstraction. This section describes these levels of abstraction and how they relate to the evolution of interoperable applications.

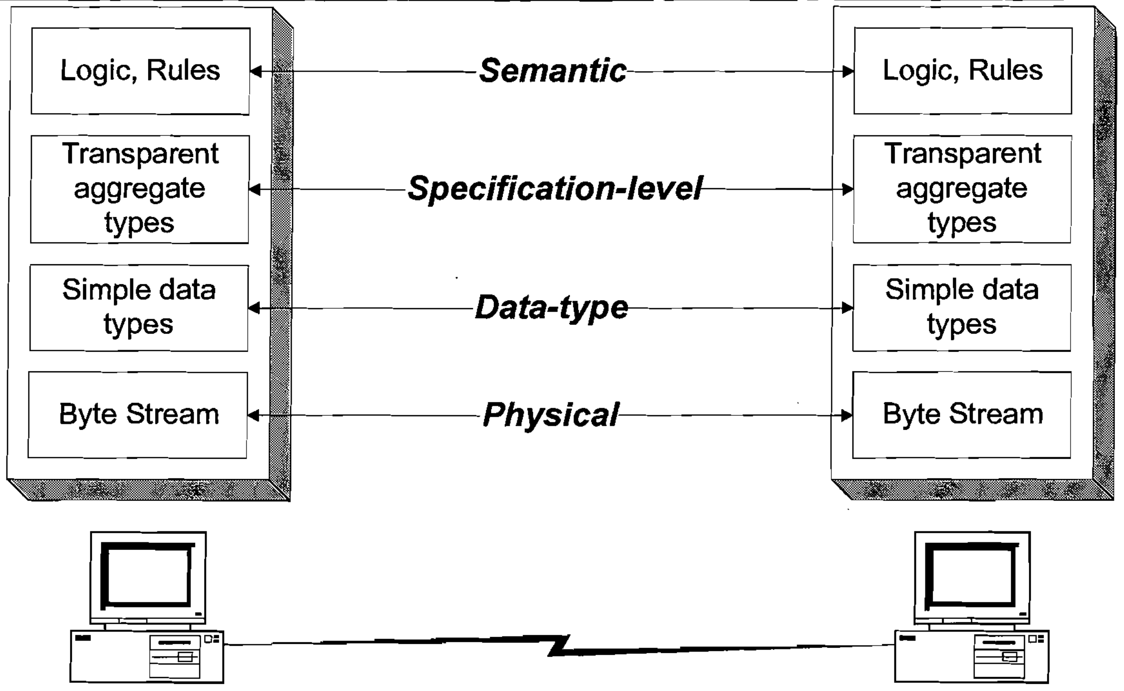


Figure 2-1: Information exchange between two software applications can be modeled at different levels of abstraction. Historically, distributed systems have focused in the physical and data-type levels.

2.1 Physical interoperability

Interoperability, even today, is usually achieved by transferring technical details through shared media such as paper drawings and electronic files on floppy disks. Such approaches date back many years to times when paper and archive media were the only means of transferring information between programs. Lack of adequate tools today ensures that many users persist with sharing at the level of physical media (via “sneaker ware”.) It is also not unusual for users to transfer information from one application to another manually, reading the output from one program and manually reentering it as input to another program. This approach to interoperability is both time consuming and costly to users, as information must frequently be entered into downstream applications using different interfaces and representation formats.

2.2 Data-type interoperability

The advent of computer networks meant that users could share information among programs directly without having to extract information out from an electronic format prior to transferring it. Many protocols have since been developed to support structured exchange of information among distributed and disparate programs. The most popular of these protocols is the Internet Protocol (IP), which is driving the interconnection of many small networks, many of which support other protocols, into a global Internet. Thus IP can be considered an interoperability protocol at the level of "wide area" networking. Moreover, hardware interfaces for connecting desktop computers to office networks are now widely available and affordable. Wireless and other telecommunication channels also support interconnection of computing devices.

Given an abundance of networking facilities, most research has focused on achieving software interoperability at the level of "data-type" compatibility. This allows data to be shared by different objects and procedures independently of the programming languages or hardware platforms used. The type model is defined using the specifications based on the structures of the objects, and type compatibility is achieved by overcoming representation differences (such as byte ordering and floating point format) in how machines implement such objects, usually by translation into and out of a universal representation format. Most systems today provide application programming interfaces (APIs) to encapsulate such differences when various programs must interoperate in a networked environment, at least at the level of simple data types intrinsic to most popular programming languages.

2.3 Specification-level interoperability

Specification-level interoperability [Wileden et al 1991] encapsulates knowledge representation differences at the level of abstract data types as well as simple types. Figure 2-2 shows an example where a table (of rows and columns) might be represented using arrays or list structures or both.

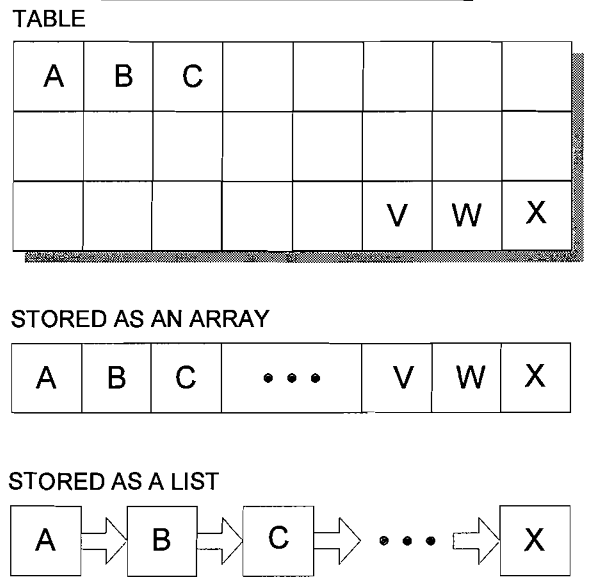


Figure 2-2: Specification-level interoperability hides the details of aggregate data types (such as lists and arrays) used in the shared knowledge model. This figure shows how a table can be stored using a single-dimension array or a linked list. Applications sharing the data need not know these underlying details and can treat the entity as a table.

Interoperability at the specification level enables programs to communicate at higher levels of abstraction and increases the degree of information hiding, affording application designers greater flexibility in how they implement the underlying structures beneath the interoperability interface.

2.4 Semantic Interoperability

Modern symbolic models explicitly represent “function” (design intent) and predicted “behavior” as well as “form” (structured description). Current research on “type” and “specification” level interoperability among programs normally considers only form at the integration interface. For example, in passing a centrifugal pump entity from one application to another, functional semantics would have to include the view that such a device is an “energy provider” to the material stream it is pumping, and behavioral semantics would include modeling its performance characteristics (e.g., as a function of its operating speed). At present there is very little work on semantic interoperability, almost all of it in the database integration community [Nadis 1996]. Systems that translate semantics across different databases are only at prototype stages. There is currently very little work in the area of interoperability where programs exchange information at a semantic level, especially in disciplines such as engineering where it is common for experts in a design project to use different abstract views of shared entities. One example is the Integrated Knowledge Framework (IKF) being developed by EPRI to achieve common semantics and a single abstract view across all aspects of power plant management, operations and maintenance [EPRI 1996].

3 INTEROPERABILITY ISSUES

There are two models within an interoperability architecture: "execution" and "type". The former concerns the execution control of integrated programs, while the latter deals with how multiple programs access and manipulate a shared data entity. This section describes the major issues pertaining to these models.

3.1 Procedure call versus message passing

A Remote Procedure Call (RPC) is used to invoke execution of a subroutine in a second program that is "registered" with the calling program. Given a network address of the callee, the calling program can invoke the subroutine across a network interface. Figure 3-1 depicts how RPCs enable access to remotely defined procedures. Although asynchronous RPC engines do exist, almost all application development currently uses a synchronous model that requires both the client and server processes to be up and running.

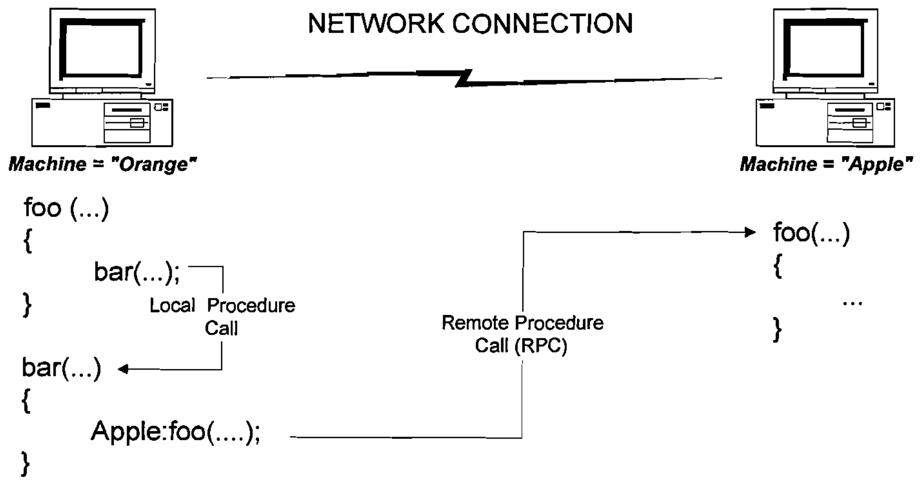


Figure 3-1: Remote Procedure Calls (RPCs) extend the paradigm of local procedure calling to provide access to remote procedures using the same syntax. The "client" application running on the machine with the network name "Orange" invokes the software defined in the "server" application running on the machine identified as "Apple". Although asynchronous RPC mechanisms exist, almost all RPC-based applications use a synchronous communication model that means that both programs (the client and the server) must run concurrently on the distributed machines. In this figure, the distinction between a local and a remote call is made using a machine name ("Apple") as a prefix to the procedure name. In practice the explicit notation is not required as low-level code in the calling application (usually generated automatically by a "stub-code" generator) will hide these details from the application programmer.

RPCs are the foundation of most client-server systems today. For example, typical database systems have client programs send queries for information to the server machine via a procedure-based interface. Where the interoperability is managed on the local machine only, Interprocess

Communication (IPC) facilities (such as "shared memory addresses") within the host operating system can be used to reduce the computational overhead of the call. Procedure calling mechanisms provide a synchronous execution model as both the caller and callee programs must be running at the time of the interaction. While good for high performance, performance of RPC-based interoperability is limited by slow and unreliable networks.

Message passing places the request sent from the calling program into a "queue" managed by the callee. This execution mode is therefore asynchronous, offering an e-mail-like store-and-forward mechanism so that the callee application need not be running to field the request (that must then be buffered in persistent storage). It is best used for loosely-coupled interoperability (e.g., workflow applications) where there is no need for real-time performance. Many "agent"-based systems use a message passing communication model. While messages are a natural abstraction for humans, they often prove very difficult to implement and debug given that programs are procedure oriented (hence the popularity of RPCs), and asynchronous systems are difficult to design, debug and maintain because much of the system behavior is "in flight" on the network – a situation exacerbated when complex message structures (like software agents) are used.

3.2 Systems Interface definition

The systems interface used for the exchange of interoperable information can be predefined or left to the application designer. Predefined interfaces force applications to comply with a "universal" interface in order to simplify the interoperability issues – changes can be made to underlying code but programs are constrained to the given API. By contrast, greater flexibility is provided if the methodology supports interoperability through the use of an interface definition language (IDL). Software developers are no longer constrained to a predetermined API; however, the more general facility increases the system's complexity as it is the interface between two programs that governs their interoperability behavior.

3.3 Execution intermediaries

Distributed systems and standards are frequently designed so that procedure calls and messages are handled by an "intermediary" program (e.g., CORBA). This controller connects the "sending" application with the "receiving" one and might be implemented as a stand-alone server (an Object Request Broker, for example) or as a module that is tightly or loosely coupled with the client application through an API. These alternatives are illustrated in Figure 3-2.

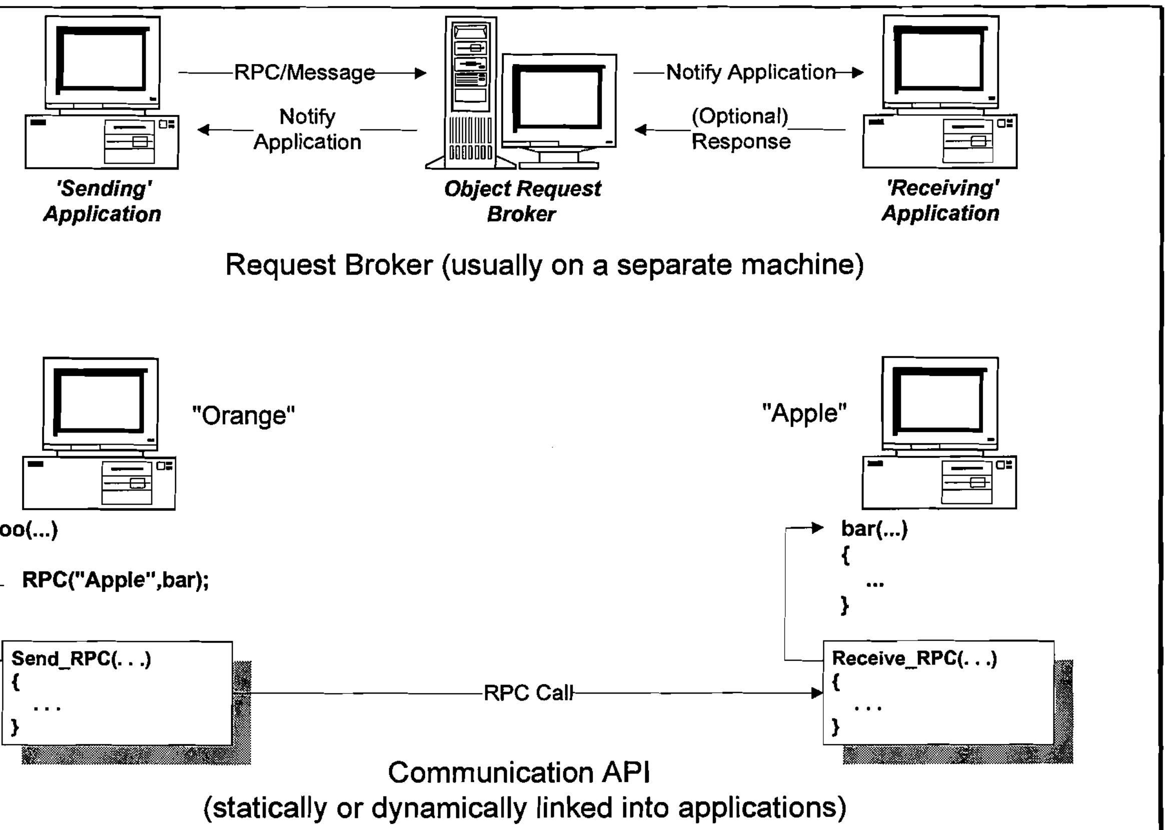


Figure 3-2: Execution intermediaries hide network communication requirements from client application developers. For example, Object Request Brokers (ORB) set up the communication link between a client and a server application that will receive and process the client request. Any results that need to be returned to the client will pass through the ORB. Another model of execution intermediary is to use client- and server-side code stubs. These low-level communication engines, or "wrappers," are linked into the applications. They manage the placing of network requests and do any necessary data conversions to support heterogeneous hardware connections.

Intermediaries can be used to simplify the interoperability interface as they present a consistent API to the calling application and shield it from the underlying communication protocols necessary to invoke the called program. They can also be instructed to communicate with other applications under certain conditions, e.g., when monitoring a transaction it may be necessary to "inform" an administrative service of the completed transaction.

3.4 Data-type compatibility

Software interoperability requires that there at *least* be compatibility between the data types used in the exchange of knowledge. There must be a consistent view of the attributes and behavior of any data object shared among the integrated applications. While a single type model across all the languages in which interoperable programs will be written greatly simplifies their integration, it is too much of a restriction for most practical purposes. For example, some languages (*e.g.*, FORTRAN 77, Common Lisp) do not support abstract data types and therefore limit applications developers to a single type model.

3.4.1 Universal type model

An alternative is to use a "universal" data-type model. In this approach, knowledge exchange is via a universal representation. The most popular approach is to use an ASCII format, with applications translating data in and out of text file formats. Many programming languages support customized formatting of such files. An alternative medium of communication is to use Interprocess Communication (IPC) facilities, such as UNIX "pipes". IPC supports byte stream exchange between two applications that agree on the type model so that translation between the shared data model and the byte stream can be achieved.

3.4.2 Basic type model

A single, universal type model is usually based around the basic data types to be supported at the level of interoperability. Traditionally, these are types such as strings of text, real numbers, and integers. Extensions to support aggregate types, such as arrays, are straightforward.

Interoperability supporting basic types has been fundamental to the success of RPC protocols used in client-server systems today. While most RPC packages generate C language code for the interface, some provide features for supporting translation of interface specifications defined in other programming languages. There also exist systems for supporting RPCs in a mixed-language environment. Each supported language must provide standard translation routines that form the API for data interoperability. Some mixed language systems are generic while others are designed specifically for exchange between applications in a predetermined set of languages.

The basic type model does not support complex abstract data types such as "trees" and "hash tables", thereby complicating interoperability where more sophisticated programming languages such as Ada and C++ are to be used.

3.4.3 Standardized submodels

Another approach to type compatibility is to use a system-independent database manager to support operations on objects shared by the integrated applications. Objects that are not part of the shared submodel can be implemented and manipulated independently of the shared database. The shared sub-model will usually support simple data types and constructors for them, and the interoperating programs use a foreign query language to access stored data. Emerging object-oriented databases support rich submodels.

3.5 Interpreted languages

A common alternative to achieving software interoperability by exchanging knowledge between two executing processes is to instead exchange an application (and supporting data) in an interpretable format that supports subsequent processing on multiple platforms, each of which must provide an interpreter. Popular programming languages for interpretation-based applications are written in Lisp, Java, Tcl, Python and Perl. While this execution model cannot support concurrent interoperability or multiple programming languages, and is dependent on the availability of interpretation engines, it is very useful for simple applications (such as "applets" distributed on the World Wide Web) and for developing prototype applications. As the capabilities and availability of platforms supporting such languages exist, one expects an "interpreted" model of interoperability will be sufficient for many applications such as document exchange and those applications that do not require substantial computational support (for which interpreted systems were never intended).

4 SOFTWARE TOOLS FOR INTEROPERABILITY

This section presents a wide variety of different tools for building interoperable systems. This section also provides a brief overview of some emerging "middleware" technologies.

4.1 Component Object Model (COM)

This object-oriented model has been developed by Microsoft to facilitate interoperability. It is effective across different programming languages and operating system platforms. It supports the "distributed object computing" model in that the interoperable software is encapsulated with object-oriented interfaces.

Interoperable components must comply with a predefined "binary" data access interface (here "binary" means the memory image of the object is independent of software development tools and programming languages), beneath which there is no constraint on the application implementation.

Object interfaces are implemented as collections of tables of pointers to object member functions. Each interface has its own table and identification name (prefixed with an "I"). Access to the object has to be done through pointers in order to support a binary standard. Objects are written in C or C++. While individual interfaces are statically typed, objects are not, so programmers must verify object support for an operation at runtime.

The execution model at the interface is based on RPCs between the integrated components. A query facility allows components to obtain data dynamically from other components. Interface definition beyond the "base" standard is achieved with an IDL. COM is a language-neutral technology for packaging class libraries, written in various object-oriented or procedural languages, for distribution in binary form, either as part of the operating system, a tool, or an application.

While inheritance is a characteristic feature of object-oriented design, Microsoft has chosen not to support inheritance in COM for complexity reasons, especially in loosely-coupled and evolving distributed systems. Changes to a parent or child object in an implementation hierarchy can force changes in other components. As an alternative, "containment" and "aggregation" are used. The former technique involves a "client" component wrapping itself around the services interface of another object, encapsulating it from other client components. Aggregation modifies this arrangement by exposing the encapsulated component to other clients as if it were part of the outer object. While implementation inheritance is not supported, COM does support inheritance of component interfaces. This decision not to support the former has led to great debate when comparing COM with the CORBA approach to distributed object computing. Several articles comparing COM with SOM (IBM's implementation of CORBA) are available on the World Wide Web.

Microsoft's OLE (Object Linking and Embedding) 2.0 is an implementation within the framework of COM that is designed to work with Digital's ObjectBroker technology (an implementation of CORBA). Unlike other interoperability systems, OLE does not support integration of software across a network, although the groundwork is there for provision of this facility in the future. However, by integration with ObjectBroker, client applications on Microsoft PC platforms can currently interoperate with server programs running on UNIX and VMS systems.

OLE enables applications to create "compound documents" that contain information from a number of different sources. For example, it is possible to embed a spreadsheet object within a word-processing document. The embedded object retains its properties and is not in any way altered during this "cut and paste" operation. If the object needs to be edited, the Windows™ operating system will invoke the associated (originating) application which then loads the object.

4.2 Common object request broker architecture (CORBA)

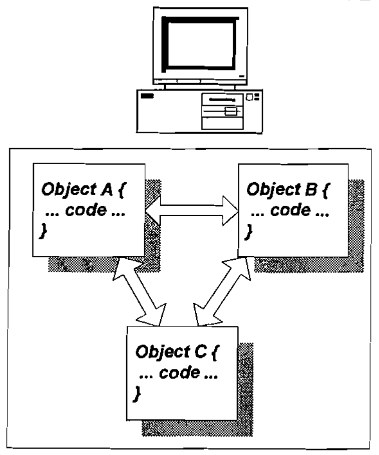
CORBA, defined by the Object Management Group (OMG), attempts to support interoperability in the rapidly evolving heterogeneous hardware and software markets. Unlike the proprietary COM standard, it is an open development, intended to support interoperability between any applications regardless of their implementation languages and operating platforms. The first version, which defined the IDL and API, was released in 1991. The second release (1994) defined support for interoperability where multiple vendors provide different object request brokers (ORBs).

An ORB is the middleware technology that establishes a communication channel between a client and server application. The service invoked can be on the local machine or across a network. The ORB is not responsible for managing any objects – it only handles the message passing among them. Object representations and implementation are left to the developers of the client and server systems.

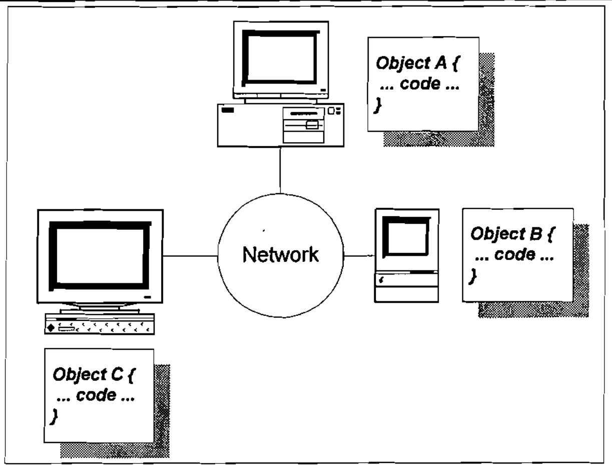
The client need only communicate with the ORB (using a single API), without concern for where the target server is located, how it has been implemented, or what system it is running on. The output from the server is returned to the client through the ORB. An IDL (based on C++) is used to simplify the protocol for interoperability between the client and server programs. Legacy applications can be integrated with new ones by "wrapping" the communication interface around the old software using the IDL. There are bindings to support the IDL with the C, C++, Smalltalk, and COBOL languages, and some vendors provide additional bindings for other languages.

The communication model supports "synchronous" and "asynchronous" requests. The former makes applications easier to write, but it blocks the calling program until the server responds through the ORB. Asynchronous communication allows the client to poll the ORB for feedback from the server, allowing it to perform other computations in the meantime. Service requests are either static or dynamic. Static requests are straightforward C code procedure calls, while dynamic support is required where some of the service parameters are unknown at the time of communication.

CORBA is part of the migration in application development away from the notion of applications being separate entities on single machines to the "network model" of applications built with distributed objects. Figure 4-1 shows the distinction between an object-oriented application running entirely on a single machine compared with a distributed model of the same application.



Non-distributed, Object-oriented Application: all objects run on the same machine



Distributed Object Computing: an application comprises objects running on networked machines

Figure 4-1: Distributed object computing allows application developers to logically link together remote objects to compose a logically-unified network-based program. Distribution of objects allows code and data resources to be located on network platforms better suited to their functional purpose (e.g., a numerical computation object might be more effective if it executed on a powerful supercomputer).

Distributed Object Computing (DOC) promotes the assembly of applications from common and compatible object types across heterogeneous platforms. CORBA is a standard (without a reference implementation) attempting to define the interaction behavior among such objects. While the standard attempts to be comprehensive (the OMG is a multinational consortium of 300 members including Digital, IBM, Sun and HP), many features, such as transaction security, are left to developers implementing CORBA compliant systems. Implementations of server applications often involve significant extensions to the CORBA API. Furthermore, the API does not deal with transactions, concurrency control, and data replication issues; and says little about persistent object storage. Three prominent implementations of CORBA are discussed below:

4.2.1 System Object Model (SOM)

While most object-oriented systems separate class definitions from instances of those classes, in SOM object classes are defined as metaclass object instances. They are distinguished from ordinary object instances by their comprising a table of instance methods to which the latter respond. Objects may be written in C or C++ (vendors are currently developing bindings for Smalltalk and COBOL). The CORBA IDL is extended to allow additional information about object implementations to be defined.

4.2.2 ObjectBroker

Digital's ObjectBroker was the first implementation of CORBA. Running on 20 different platforms, it can also be integrated with OLE (Digital is working closely with Microsoft), allowing PC applications to connect with CORBA servers. To improve security, there is support for DEC's Generic Security Services, making ObjectBroker the first secure ORB in the industry. Additional enhancements to interfaces and server management, and provision of drag and drop support for OLE are available.

4.2.3 Orbix

Orbix (from IONA Technologies) is a full and complete implementation of CORBA. First released in 1993, it has become the leading CORBA compliant ORB and is in widespread use in the telecommunications, engineering and government sectors. It provides C++, Ada95 and Java language bindings and is supported on more than 10 UNIX platforms, Windows, OS/2, Mac, VMS, QNX, and embedded real-time systems.

IONA has made numerous extensions to CORBA. An implementation repository services requests if there is no active server. Dynamic requests for services can make use of a "stream-based" API that is simpler than the standard CORBA interface. Programmers can also create local "cached" versions of remote objects, called proxies, to improve application performance. Programmers can use filters to control message passing to provide more complex object behavior, to integrate thread packages, and to assist in debugging tasks. For a combination of the COM and CORBA approaches, integration of Orbix with OLE 2.0 is possible.

4.3 Distributed computing environment (DCE)

The OSF (Open Software Foundation) DCE [OSF 1992] is a comprehensive set of services supporting the development, use and maintenance of distributed applications. The services are provided independently of operating systems and network interfaces.

A layered model, illustrated in Figure 4-2, works bottom-up from the operating system layer to the application layer. The environment, however, appears to the application as a single system rather than several disparate services. This integrated architecture encapsulates the physical complexity of computer networks and simplifies the design of interoperable applications. Services include secure RPC, distributed directories, time, threads, security, and data-sharing. A key knowledge-sharing component is a Distributed File System that can interoperate with the Network File System (NFS) from Sun Microsystems as well as other distributed file systems.

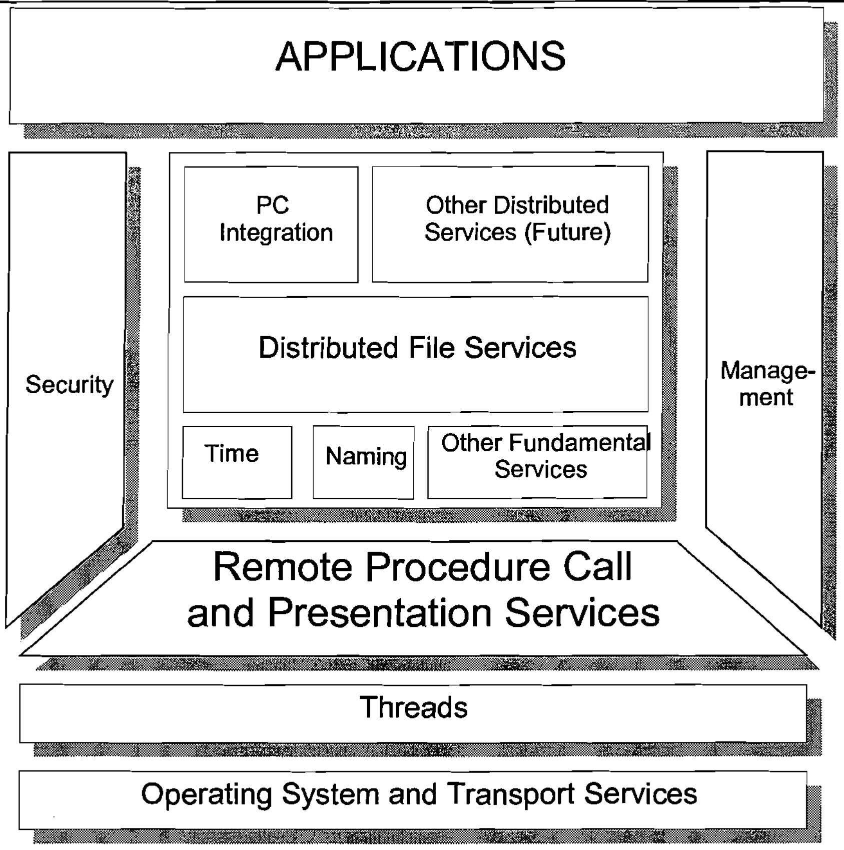

APPLICATIONS
Security
PC Integration
Other Distributed Services (Future)
Distributed File Services
Time
Naming
Other Fundamental Services
Manage-ment
Remote Procedure Call and Presentation Services
Threads
Operating System and Transport Services

Figure 4-2: The OSF DCE model users a layering of services to hide application developers from low-level issues. The integration of services thus appears to the user as an environment where services can be integrated. For example, the RPC service can be linked with the Security service.

The "fundamental" distributed services are: remote procedure call, directory service, time service, security service, and threads service. These services provide tools for programmers to create end-user services for distributed applications. By comparison, the "data-sharing" services (distributed file system and diskless workstation support) require no programming and provide users with capabilities built upon the fundamental services.

The RPC service includes an interface definition language and can be integrated with the threads, directory and security services. The distributed directory service provides a single naming model across heterogeneous file systems. It allows users to identify resources using a logical name that remains constant even if resource characteristics (such as their network addresses) change. The time service is a software-based facility that synchronizes system clocks across a network to support activity scheduling and the determination of event sequencing and duration. Accurate timing is also required in applications that use time-stamped data. The threads service enables programmers to create and control multiple application threads in a single process.

4.4 Knowledge Interchange Format (KIF)

Several approaches to interoperability have used the abstraction of message passing to model "software agents" [Genesereth 1994]. An agent is a computer program that shares knowledge with other agent programs through an intermediary called a "facilitator" (that is thus functionally equivalent to an ORB). Unlike other middleware technologies that support multiple languages across the execution intermediary, agent systems use a single Agent Communication Language (ACL) to achieve interoperability.

KIF, developed by the Interlingua Working Group of the DARPA Knowledge Standards Effort [Neches 1991], is a prominent specification for defining the content of messages within the ACL structure. This declarative language is a logically comprehensive, prefix version of first-order predicate calculus. It provides for the representation of meta-knowledge and supports a wide variety of expressions supporting constraints, rules, disjunctions, etc. with a vocabulary comprising variables, operators, functions and relations. KIF is designed for interchange of knowledge among disparate programs.

KIF is not intended as a primary language for interaction with the user. Programs interact with their users through whatever languages are appropriate to the applications. KIF is also unsuitable for representation of knowledge within a computer or within closely related sets of programs (though it can be used for this purpose). Consequently, programs convert knowledge bases from their own internal formats into KIF simply for the purposes of sharing the knowledge.

5 INTEROPERABILITY SYSTEMS AND ARCHITECTURES

This section presents several examples of systems and architectures that support software interoperability. These examples are described without explicit regard for any underlying software tools used to effect the implemented application integration, and they describe both systems specifically designed for a particular domain as well as more generic ones.

5.1 The AAITT system

The Advanced Artificial Intelligence Technology Testbed (AAITT), developed by the Air Force's Rome Laboratory [GE 1991], is a laboratory testbed for the design, analysis, integration, evaluation and exercising of large-scale, complex, software systems composed of both knowledge-based and conventional components.

In the military domain, software decision aids for intelligent command and control (C2) are usually developed to solve a relatively narrow problem within the overall C2 decision-making problem. Software aids typically use data stored in a local format, problem solving methodologies that reason over the data, and support interactive interfaces with the user. Difficulties arise in trying to integrate such a tool in a typical application environment with several databases and multiple decision aids. AAITT is a distributed environment designed to facilitate the integration and testing of multiple decision aids. By tailoring the interface of software aids with the testbed, unrelated software components can be integrated without the need for extensive re-engineering. This is particularly important where such aids are "black boxes" purchased from vendors or downloaded from bulletin boards (where no source code is available – only the component's binary image).

System modules comprise individual software components that are encapsulated with a communication "wrapper" that enables connection of the module to a network backplane. Modules must support the AAITT modeling and control protocols that are based on input and output "ports" for each module. Ports have associated code bodies that process system messages and can specify their own message formats as part of an Application Protocol. The Application Protocol is used to send messages between modules or between a module and the testbed manager module. Modules are controlled through a State Transition Protocol that supports the states: {START, CIM LOADED, CIM CONNECTED, LOADED, INITIALIZED, RUNNING, PAUSED}. The user embedding a module in the AAITT must specify programmatically how a module should react to state transition messages.

An AAITT application is then built by connecting multiple modules to this network. The modules are configured by the user to solve an overall problem. This software controlled architecture ("soft architecture") allows for easy reconfiguration, through a graphical interface, of the modules and their input/output connectivity, thus minimizing the recoding of module interfaces. For example, it is relatively easy to reconfigure the system from a "blackboard" integration model to a "data-flow" one. The application architect must assign each module to a particular network host. The components of the AAITT are shown in Figure 5-1.

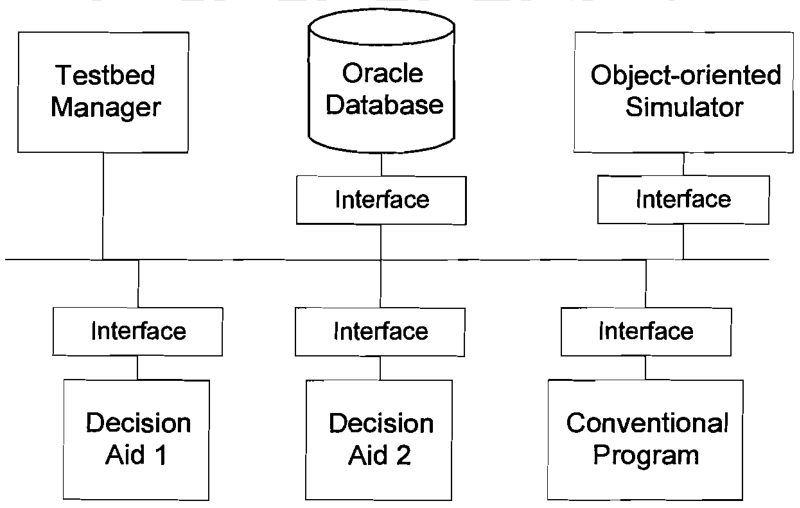


Figure 5-1: An AAITT application comprises various software modules connected to a network backplane. These modules act together to solve an overall problem and thus constitute a single application. The testbed manager allows the user to reconfigure module interaction very easily through a graphical interface.

The testbed manager is implemented as a workstation platform called the Modeling, Control and Monitoring (MCM) Workstation. Two other major sub-systems are the Distributed Processing Substrate (DPS) and the Module Framework. The DPS supports a heterogeneous interconnection of systems (e.g., databases, simulators, expert systems, conventional software) running on VAX, Sun and Symbolics hardware platforms. It is responsible for translating data representation between the various systems and programming languages used (C/C++, Common Lisp, and Ada). For implementing the DPS, AAITT developers selected the Cronus distributed computing environment [BBN 1989]. In addition, ABE (A Better Environment) [Erman 1990] is used to provide a module-oriented programming capability over the object-oriented model of Cronus;

this allows for restructuring of the application architecture at a higher level. The Module Framework facilitates the embedding of new software components in the AAITT. A Component Interface Manager (CIM) is a software wrapper for interfacing the component with the testbed. A library of generic CIMs is available, as well as a semi-automated generator for building and managing CIMs. CIMs support both synchronous and asynchronous communication. Software components connect with a CIM using a CIM-to-Component Communication (CCC) interface based on UNIX sockets, files, Cronus or other IPC mechanisms such as shared-memory. The MCM can query the status of various CIM elements through the CIM Query Protocol.

The MCM Workstation is a central console that provides the user with tools to configure and analyze applications. Performance metrics can be gathered through "breakpoint" and "logging" taps that are specified in the input/output interfaces of the application modules. Configuration can be performed using a graphical user interface. Modeling functions are used to define module interaction, and control functions allow for the loading, execution and resetting of individual components.

Specific problem scenarios are input to an application from an Oracle database that is also used for storing the results of an application run. Heuristics are then used to analyze the quality and behavior of the application following which alternative solution strategies can be investigated. The database module connects to the AAITT through an SQL interface.

Generic simulation is provided through the object-oriented simulation language, ERIC [Hilton 1990], that is based on the Common Lisp Object System (CLOS). ERIC allows for simulation halting, modification and resumption without the need for application recompilation.

AAITT has been demonstrated using three different software components: a tactical database, TAC-DB, developed by the Knowledge Systems Corporation in 1989; an ERIC land-air combat simulation called LACE; and two decision aids: the Air Force Mission Planning System (AMPS) developed by the MITRE Corporation, and portions of the Route Planner Development Workstation (RPDW) developed by the Jet Propulsion Laboratory. Initial demonstrations realized a 10:1 reduction in software integration costs when 9 independent contractor- and government-developed components were integrated in just 25 days using AAITT. Figure 5-2 shows the overall system architecture with the demonstration components included.

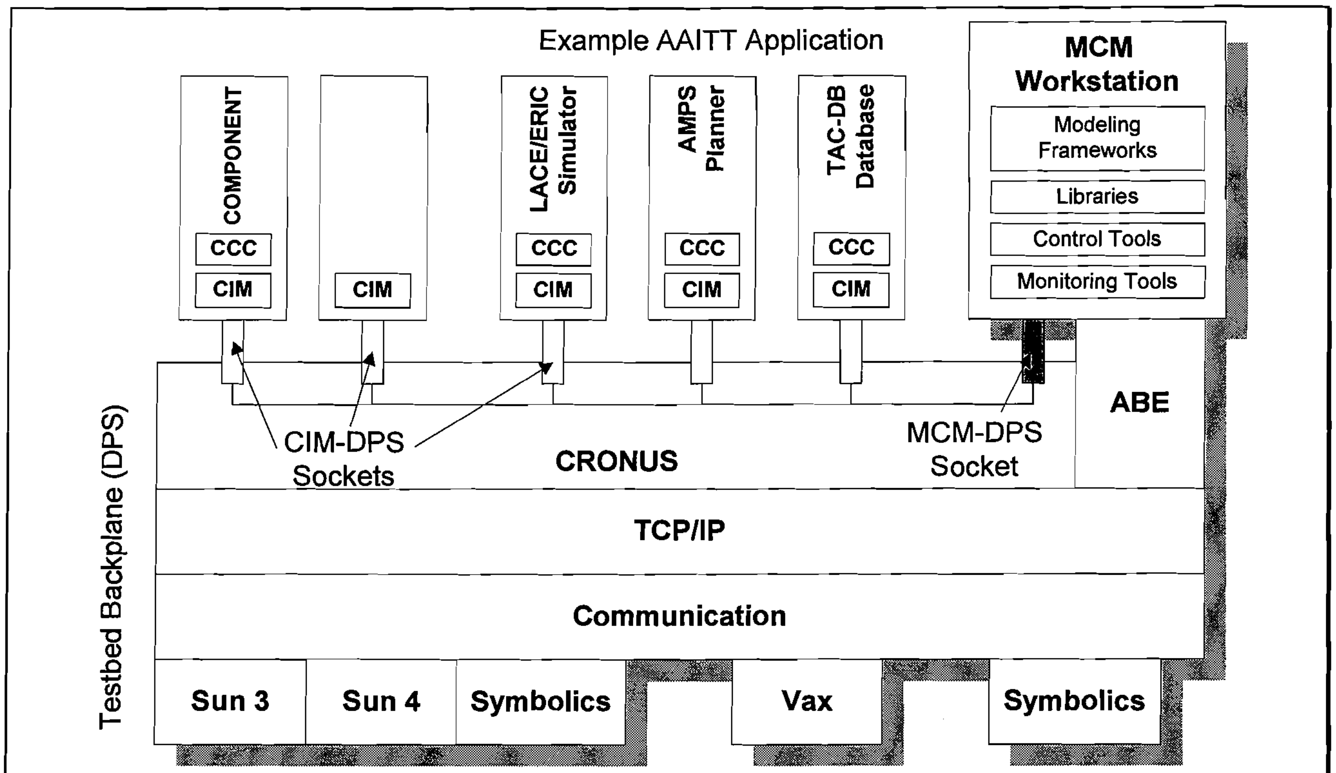


Figure 5-2: The AAITT architecture including the connection of application components to the network substrate. The MCM Workstation allows the user to reconfigure and refine the behavior of the connected components in the testbed environment. This allows applications to be tested and developed for full readiness in heterogeneous battlefield systems.

5.2 OpenAccess

OpenAccess [ATI], originally called EPRIWorks, is a software framework that enables applications to access data from a variety of data sources in a distributed environment. Since ATI developed the technology in collaboration with EPRI, it is based on the functional requirements of the diverse applications in the electric power utilities, specifically domains of maintenance, troubleshooting and performance. While the initial focus was on plant information systems, the framework has expanded to cover broad scale applications throughout a utility organization. OpenAccess is currently in use at more than a dozen utilities and has been licensed by major system integrators and OEMs.

In a power plant, process information sources include the Distributed Control System (DCS), process archives, data acquisition systems, and inspections. Additional data is obtained from applications for performance monitoring, vibration equipment, thermography and work order systems. DCS systems are specific to each vendor and no two provide the same data access method. Applications for monitoring and diagnosis use proprietary database formats. Client applications in current use were developed for specific computer platforms and based on particular data access protocols. Different computer platforms require operations and engineering staff to move from one display to another to perform analyses or view plant status information. Consequently, many operators transfer information among applications using a paper and pencil.

Such dependencies on data sources, formats and access methods, as well as particular hardware systems, inhibit the integration of new software into existing plant data systems as well as the porting of utility software applications to different plants and next generation computer environments. Furthermore, very few applications can access all the available data for a plant because of the difficulty in obtaining it from such a wide variety of sources.

Designed to connect islands of plant automation systems, OpenAccess provides a single access mechanism to eliminate the development and maintenance of specialized protocols and drivers for these sources. It is ideally suited for data warehousing applications that must deal with multifarious data formats that have to be consolidated for analytical purposes. Client applications are typically desktop PC programs, and servers can support more than one data source.

All data points are modeled based on physical pieces of equipment or logic decisions. The model then becomes part of the enterprise representation of business information. Associations with specific pieces of information are then correlated without the need to centralize the data (which is impractical if not impossible). The OpenAccess approach thus allows information to reside in different systems.

Its framework, depicted in Figure 5-3, provides services for modeling utility information, uniform data access in a multi-vendor environment, and automated data transfer across the network. The open nature of the solution dictated the adoption of X/Open and ISO standards.

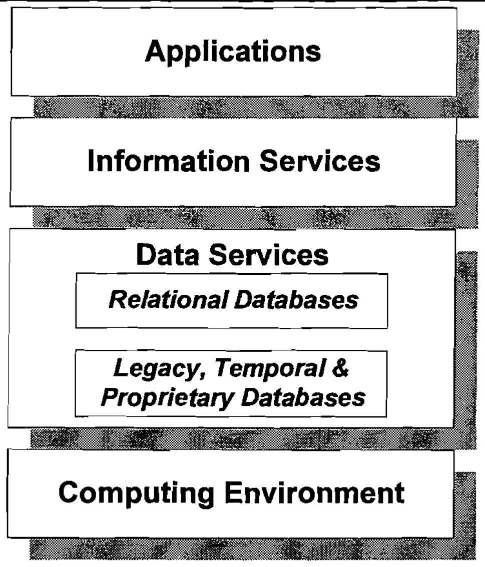


Figure 5-3: The OpenAccess framework collates distributed and varied data sources into a logically consistent model through various data and information management services. This allows application developers to rely on a single data access interface to the entire business information model.

ISO defines the Remote Database Access (RDA) protocol for client applications to access data from any source independent of a proprietary solution. OpenRDA is a client-server protocol compliant with this standard, and provides uniform data access where each source is viewed as an SQL-compliant relational database. For data access, client applications use X/Open CLI or Microsoft ODBC APIs. Network communication uses WINSOCK TCP/IP and/or

5.9 CIM-BIOSYS

Due to a lack of standards, manufacturing control systems (MCS) are typically highly fragmented and heterogeneous in terms of hardware, software, communications and data management. Proprietary systems are in widespread use, and connectivity among software components requires filters for transferring shared knowledge. Within the Computer Integrated Manufacturing (CIM) industry, there is a strong demand for interoperability standards.

A common approach to this problem has been to integrate applications through a shared database for which each application must support "import" and "export" schema. Disadvantages include data replication and the repeated translation and reinterpretation of information can cause data integrity problems.

Singh et al [Singh et al 1994] propose an integration architecture that uncouples MCS functions from their information repositories. While they recognize that data integration is the first step toward interoperability, functional interaction management enables the information to be treated independently from the functional capabilities realized by software applications.

The integration module of their system, called the CIM-BIOSYS (Building Integrated Open Systems), is an abstract layer above a SQL interface to a RDBM system. The BIOSYS comprises a set of tools supporting functional interaction between MCS components. Functions are decomposed into information, interaction, and communication services. Above the BIOSYS layer is a Functional Interaction Management Module (FIMM) that is SQL-compliant and provides the interoperability execution model for applications. Services within the FIMM are provided by way of ANSI C code functions. The execution model uses data-driven input/output tables, where information models are assigned to MCS functions as either input and/or output relationships. An interactive user interface allows MCS functions to be added or deleted through a configuration utility. The system uses dedicated window-based displays. UNIX "pipes" are used to connect FIMM services in a distributed environment. The system is being modified to support the EXPRESS language for definitions of the interoperability schema.

5.10 STARS

Boeing, as a prime contractor on the U.S. Advanced Research Projects Agency (ARPA) Software Technology for Adaptable, Reliable Systems (STARS) program [ARPA 1993], has developed and integrated a demonstration software engineering environment, SEE [Boeing 1993], that supports a process-driven, domain-specific software development environment. The STARS system separates development into two life-cycle views: domain engineering and application engineering. Multiple applications are supported by a single, ongoing domain engineering effort that develops appropriate reusable software assets based on a shared "domain model". Development of new applications assumes the existence of a reuse library for the domain. A graphical interface model is used for selection of appropriate components. Application developers make engineering decisions based on customer requirements for a specific project, following which the system identifies the applicable reusable components, retrieves them from the library, and adapts them to a specific application. The domain engineering effort is responsible for ensuring the quality of the software components. Boeing has implemented its library in an object-oriented mechanism called ROAMS. Figure 5-11 shows the logical relationship between the domain and application engineering efforts with regard to the development of applications that can interoperate easily.

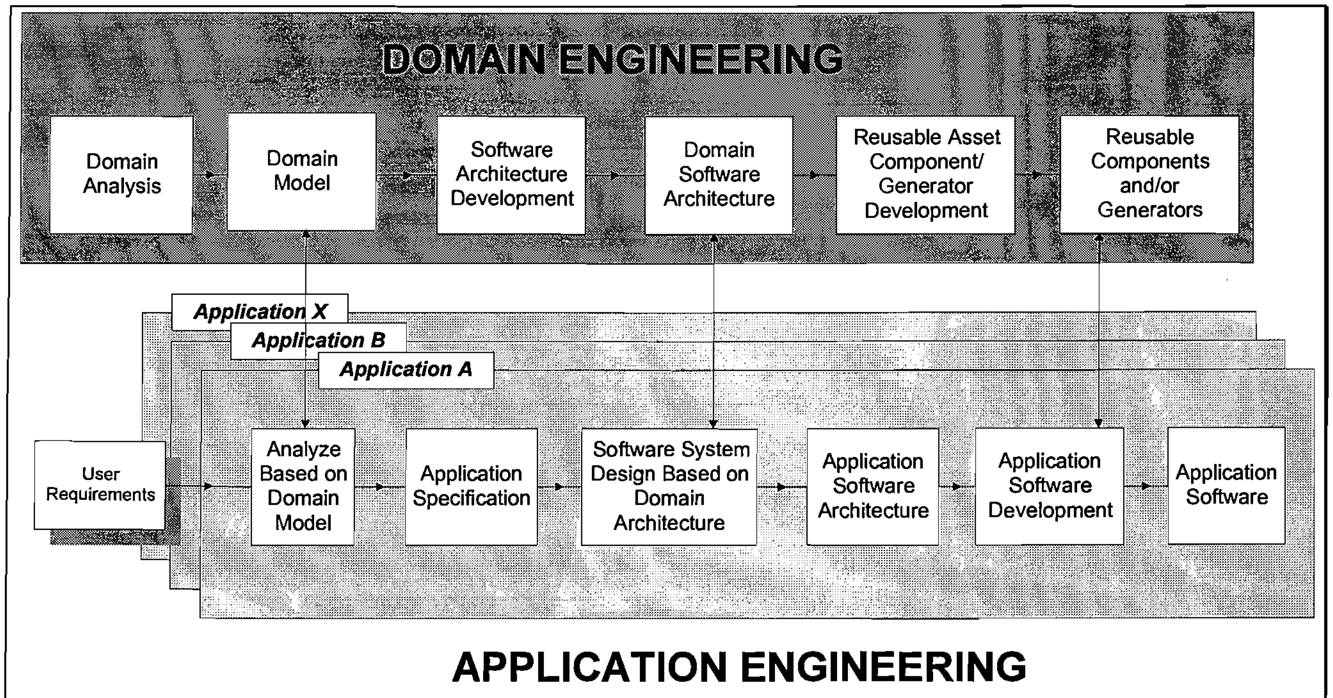

DOMAIN ENGINEERING
Domain Analysis
Domain Model
Software Architecture Development
Domain Software Architecture
Reusable Asset Component/ Generator Development
Reusable Components and/or Generators
Application X
Application B
Application A
User Requirements
Analyze Based on Domain Model
Application Specification
Software System Design Based on Domain Architecture
Application Software Architecture
Application Software Development
Application Software
APPLICATION ENGINEERING

Figure 5-12: The STARS system uses a domain engineering effort to identify reusable code components that application developers can extract from libraries and piece together to build applications that interoperate easily.

5.11 Summary Table

This section summarizes the main characteristics of the above approaches to software interoperability in Table 5-2. Column descriptions are as follows:

- Column 1: Working Demo Implementation (yes/no): An implementation of the methodology has been demonstrated at least as a prototype demonstration.
- Column 2: Shared Database (yes/no): a shared database centralizes the management and storage of data shared by the integrated applications. The column therefore indicates whether the methodologies are centralized or distributed in their management of the model(s) that underlie the software integration mechanism. For example, "Software Agents" can reasonably maintain very different data models, while an integrated framework such as KANON uses a single KB/DB with a query engine and semantic network processor.
- Column 3: Synchronous Communication (yes/no): Tightly-coupled systems inherently require a synchronization of the communication between integrated applications. Loosely coupled systems allow but do not require synchronous communication.
- Column 4: Domain Specific (yes/no): A domain specific methodology has been designed to integrated applications with a limited scope.
- Column 5: Level of Abstraction (data-type/specification/semantic/variable): This column indicates the level of abstraction at which applications share information.
- Column 6: Software Component Model (yes/no): A model based on software components achieves interoperability by controlling the interconnection of various components adapted for use in the integration environment.

Table 5-2: Summary of a variety of different methodologies and systems aimed at software interoperability.

Methodology	Working Demo	Shared Database	Synchronous communication	Domain specific	Level of Abstraction	Software component model
AAITT	Y	N	N	Y	data-type	Y
OpenAccess	Y	Y	N	Y	data-type	N
IRTMM	Y	Y	Y	Y	data-type	Y
Circle Integration	N	N	N	N	variable	N
Software Agents	Y	N	N	N	semantic	N
Loosely-coupled	Y	Y	N	N	data-type	N
KANON	Y	Y	Y	N	semantic	N
POLYLITH	Y	N	N	N	data-type	Y
UTM-0	Y	N	N	N	specification	N
CIM-BIOSYS	Y	Y	Y	N	data-type	N
STARS	Y	N	N	Y	data-type	Y
IFC	N	N	N	Y	semantic	N
COM	Y	N	Y	N	specification	Y
CORBA	Y	N	Y	N	specification	Y
DCE	Y	N	Y	N	data-type	Y
KIF	Y	N	N	N	semantic	N

6 THE FUTURE AND SOFTWARE INTEROPERABILITY

When computer networks were first established, the computational machine was still the workstation itself – the network was merely a communication medium with which workstations could access remote resources. The more recent adoption of client-server technology has moved the computational function of applications out onto the network, normally to a remote "server" machine, although the local "client" machine still performs interface functions and some data processing. The clustering of servers has changed the computing model to a "networked" one in which an application logically spans more than one physical device. Networking makes the interface boundary between programs less distinct and demands even greater accommodation of heterogeneous computing platforms. The networked computing model is driven even further by the rapidly growing "wireless computing" market, and wireless solutions increase the demand for networked infrastructures. Wireless telecommunications spending is expected to reach $60 billion by 1999 based on a compound annual growth rate of 30% [Dell'Acqua 1996]. Furthermore, the growing number of virtual enterprises and the demand for "groupware" products is stretching the interoperability horizon even further. This section speculates about how current advances in networking, application design, and ontologies are affecting the future of software interoperability.

6.1 Network Protocols

Numerous low-level network protocols exist for managing communication among networked computers. Common examples are the Internet Protocol (IP), Xerox Networking Systems (XNS), and IBM's Systems Network Architecture (SNA) and NetBIOS (for PC networks). These protocols are the foundation on which higher level protocols like SMTP (for electronic mail) and FTP (for remote file transfer) are built. The wide variety of protocols available inhibits software interoperability because applications that do not support a particular protocol cannot communicate with other programs which do. This leads to clumsy and esoteric knowledge translation mechanisms at the network level. However, the popularity of the Internet is creating enormous momentum behind IP, and present and future network architectures will simply encapsulate other protocols within IP packets thereby simplifying the design of interoperable systems.

IP is about 20 years old and its current version (version 4, i.e., IPv4) has severe limitations for today's Internet environment. One problem is that its 32-bit address space no longer supports the number of users wishing to get "on-line". The Internet Engineering Task Force (IETF) has responded by working on the next generation IP: version 6 (IPv6). This version will support 128-bit addresses. This large address space should support interconnecting more communication devices than will ever be needed. Field devices such as pumps and control switches will then be able to have their own network addresses, so the networked computing model expands beyond just interconnected workstations and mainframes to one in which software runs across a wide variety of hardware platforms.

Another feature of future protocols like IPv6 is the support for "flow specification" parameters. These will be used to prioritize information communicated among applications, an effect that will support more sophisticated execution models for software interoperability. For example, faster network processing of "real-time" or "control" packets will improve the performance capabilities of distributed systems and will encourage interoperability architectures to accommodate the consequent quality services emerging at the communication level.

Currently, higher level protocols running on top of IP (e.g., TCP and UDP) support only unicast (point to point) communication. An application wishing to integrate with multiple remote applications must send the same unicast messages to each one. With a "multicast" protocol, an application can communicate with many destinations by making a single call on the communication service. Multicast protocols both simplify the design of interoperable systems and greatly improve network performance (especially for media protocols like Ethernet that support multicast packets). Real-time applications such as "video conferencing" will greatly benefit from a multicast protocol. IP Multicast is a protocol that supports multicast communication in IP networks. This protocol can already be used in systems (such as UNIX) that support network "sockets". Another significant protocol under development is Real-time Transport Protocol (RTP). This provides real-time service on top of UDP/IP and chiefly is targeted at distributed systems managing interactive audio, video, and simulation.

6.2 Client-Server systems

Current developments in network architecture and computer design are leading to client-server systems based on a 3-layer approach that supports "thin" clients and "fat" servers (i.e., the data processing is performed at the server). Historically, client machines were "dumb" terminals that connected to a centralized mainframe computer in a 2-layer system. Today's 3-layer design, as shown in Figure 6-1, allows intermediate server machines to balance the processing load on the main servers.

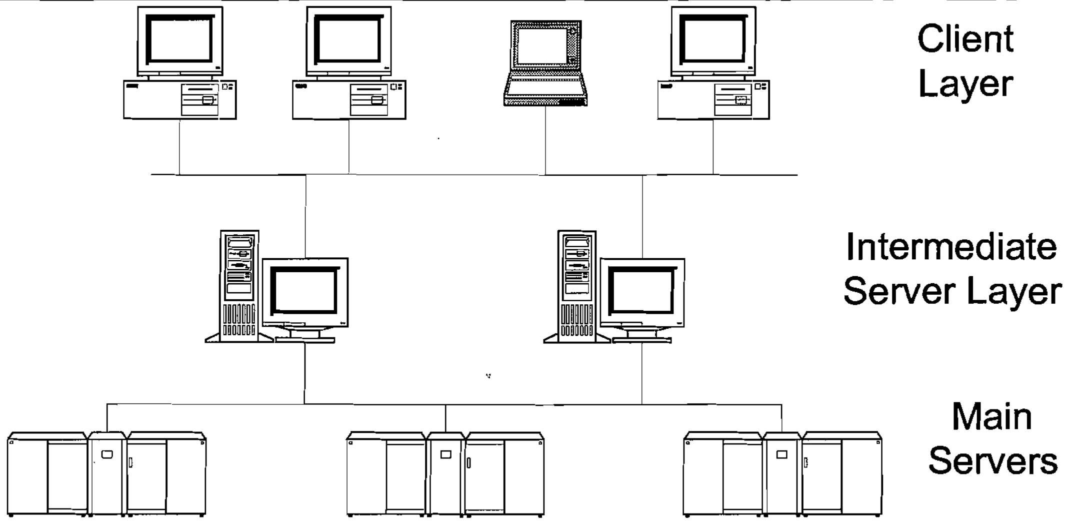


Figure 6-1: Modern client-server architectures are moving from a 2-tier (client-database server) model to a 3-tier approach that allows intermediate servers (often on desktop platforms) to support load-balancing across the main, mission critical servers. The intermediate level can also provide less important application services.

In market terms, the significance of this approach is that less reliable server platforms (like Windows NT from Microsoft) can be used at the middle level while the mission critical, legacy systems run on mainframes and UNIX servers at the bottom level. This architecture is designed to exploit the computing power of emerging symmetric multi-processor (SMP) server systems, while also supporting simpler client machines. Enormous efforts are currently underway by companies such as Oracle, IBM, and Sun Microsystems to produce a very simple client device (colloquially referred to as a Network Computer, or NC) that will run distributed applications in the 3-tier environment.

These developments in client-server architecture have two significant impacts on software interoperability. First, the execution model of applications must be "partitioned" for better alignment with the underlying hardware resources. In most cases, it will no longer be necessary for client machines to cache data and software as they will take advantage of powerful, remote processing facilities. Second, interoperable software modules can be tuned to specific server platforms without the complication of having to integrate with logic on the wide variety of desktop systems. For those applications that are to run across multiple servers, middleware technology is being developed to simplify the interoperability issues.

6.3 Application Design

Several issues affect the future of interoperable application design.

6.3.1 Software Components

Software engineering techniques are embracing a "software component" model as an alternative to the development of large applications and systems. Software components, whether object-oriented or not, solve a number of application development problems [Grimes 1995]. Uncoupling applications in terms of time allows later modifications to interoperate with earlier code. Further geographic uncoupling allows loosely-connected interoperability where applications can be integrated with no more than an understanding of their interfaces. Another motivation for component-based software is the scaling up to large systems. Traditional, structured programming techniques become unwieldy when systems scale to several hundred thousand lines of code. Components with standardized and published interfaces allow their functionality to be uncoupled from their interfaces. Software vendors can ship specific components that can be integrated easily into existing applications and systems. To simplify the interface specifications, IDLs are becoming more prominent. Interoperability technologies like CORBA, DCE and COM will support the interconnection of components across networks. Like any change in paradigm, the move to distributed, component-based applications will take time.

6.3.2 Code reuse

Many organizations are implementing software reuse programs. Such programs will have to be effective if component-based applications are to be engineered. A survey [Frakes 1995] of software engineers, managers, and educators investigated beliefs and practices in reusing code and other life-cycle objects.

The survey found that reuse of software assets is varied, based largely on the development environment. For example, development on UNIX platforms had greater support for and belief in reusable code and tools. No reasons were given as to why some assets are more accepted for reuse than others, though perception of the relevant functionality is a factor. Reuse levels differ across industries. For example, telecommunications companies reported the highest level of reuse while aerospace companies had the lowest, possibly because the former type of industries are at the leading edge of computing technology and so are more informed of latest software engineering tools and practices.

The choice of programming language appears to have no effect on aspects of the development process. In fact, the marketing of languages such as Ada and C++, usually thought to promote reuse, showed no significant correlation with reuse levels. Another finding was that higher level languages were no more reused than lower level ones like assembly and C.

Many people believe that software engineers prefer to build their own code rather than reuse someone else's – which is referred to as the "Not Invented Here" (NIH) syndrome. However, the survey found that 72% of developers report that they prefer to reuse code rather than develop it from scratch. To this extent, however, it was found that CASE tools and component repositories/libraries currently have little effect in the engineering of new applications. Conclusions for this finding include the possibility that such tools are either considered ineffective or are not being used properly.

Perceived economic feasibility does appear to influence reusability of code, as does corporate education on reuse, although at present very few companies have reuse training programs. The lack of education and training is given as a factor in why software engineering experience has no effect on reuse. The study of the software process, a growing area of research, shows that a defined process does affect reuse levels. Thus, gains in process maturity can translate into gains in software reuse. Developers also appear generally to be uninhibited by legal problems surrounding component reuse, especially since legal issues regarding contracting, ownership and liability for reusable components are still unresolved; in addition, most reuse presently takes place within companies where legal issues are less of a concern.

6.3.3 Legacy software

"Reverse software engineering" comprises tasks that are aimed at understanding and modifying legacy software systems, principally the identification of software components and their relationships to one another. As it is unreasonable to expect a system to be developed properly the first time, and as future technological developments require system upgrades, so reverse engineering is a new field of research (the first conference was in May, 1993) playing a more important role in integrating old and new software applications. A large application of reverse engineering tasks is currently underway in the upgrading of software systems used by the U.S. Department of Defense [Aiken et al 1994]; considered are over a billion lines of code defining thousands of systems running at more than 1700 data centers.

One method of migrating legacy software is to encapsulate it in "wrapper" code that then allows it to be used in a new execution environment. Such an approach preserves original functionalities, is cheap (old hardware need not be replaced), and no deep analysis of the code is required thus producing a short migration cycle. The disadvantage of interface wrappers is they yield no performance gain and provide no flexibility (the legacy system has to be used as a whole). Reverse engineering thus attempts reusable component recovery. The principle is that functional components of systems are recognized, recovered, adapted and reused in new systems. Recovered components are smaller and thus more readily distributed across networks.

A major challenge for cross-functional integration in reverse engineering is to identify what data elements are related to the integration. For example, "calculate pay" must access policy information from "personnel" and apply a "pay" function. A complete understanding of data sharing requirements demands that reverse engineering analysis determine both how interface elements are generated and also identify any non-interface data elements that are synonymous among modules and are therefore sharable. This task is hard because many legacy systems were designed for obsolete hardware platforms and do not support data and process models necessary for data standardization. Complicating this process is the common misconception that data standardization is achieved merely through identification of associated data elements and consistent naming in each system. Correct use of information from data elements requires that the business rules, policies, and functional dependencies be identified and represented in a data model for each system. Model integration must then be achieved before standardization is possible. Ning et al [Ning et al 1994] describe automated tools developed by Andersen Consulting for understanding legacy code.

6.3.4 Middleware

Middleware may become one of the key network industries for the rest of this decade [King 1992]. Middleware technology [Bernstein 1996] provides an enabling layer of software between the network and an application, allowing programmers to focus on business problems rather than communication. Middleware provides network services to the programmer, just as an operating system provides local resources such as disk, memory, and other peripherals. It offers a way for software vendors to solve heterogeneity and distribution problems by using standard protocols that enable applications to interoperate. Middleware services provide these standard programming interfaces and protocols. Legacy applications can be encapsulated with middleware services to provide access to remote functions, or modern, graphic user interface tools. Some middleware services allow for replacement of internal components within a legacy program, for example application-specific database functions can be replaced with more generic middleware database services.

Many major middleware products (e.g., OLE, CORBA, DCE) are based on the RPC model while others such as DECmessageQ and Pipes Platform use a message-based communication (often called message-oriented middleware, or MOM). The effect on the interoperable execution model was described in Section 3.1. Other issues are the degree of platform independence offered, the degree of encapsulation, the number of communication modes and the number of supporting utilities and services such as security, transaction, and time services.

Despite their advantages, middleware services are not a complete solution, especially if they are based on proprietary APIs and protocols (as is the case with many relational DBMSs.) Some vendors offer middleware services on popular platforms, thus limiting the customer's ability to integrate applications in heterogeneous systems. Furthermore, adoption of middleware solutions is inhibited by the number of services, a factor that greatly complicates application development, especially considering that the application programmer must still make hard design decisions such as how to partition functionality between the various application modules.

To simplify the use of middleware services, vendors also provide *frameworks*. These are software environments developed for specialized domains that simplify the APIs of the underlying middleware services they abstract. Frameworks can also be built for situations where middleware services cannot meet the requirements of new applications.

Examples of popular frameworks currently available are Microsoft Office, Lotus Notes (for office systems), Digital's Powerframe (for CAD), and IBM's NetView (for system management).

The future growth of the middleware market will probably be very aggressive, and major companies are already delivering products [Eckerson 1995]. While DCE is very popular with many IS managers, it is highly complex (having more than 400 APIs) and has no robust secondary market for DCE-based tools. It does not support asynchronous communication, though its RPC model can mimic asynchrony. The restructuring of the OSF may further weaken it. Many developers see a better solution with an object-oriented based middleware such as CORBA or OLE. However, object-based middleware is hampered by the lack of widely available object technology – many companies still operate procedural systems rather than object-oriented ones. Also OLE is currently limited to Microsoft platforms, though Microsoft's cooperation with Digital enables OLE connection with CORBA-compliant systems (as OLE has no ORB) – such a development raises the concern of the need for building "middleware for middleware". When Microsoft ships the next version of Windows NT (codenamed Cairo), all OLE-compliant applications will automatically become distributed on this platform and reliant on Microsoft middleware, that could lead to OLE becoming the de facto industry standard. To protect applications for changes in underlying middleware technology, developers can provide in-house wrapper APIs that can be layered over third-party middleware. Other developers [Kador 1996] are using interfaces like the World Wide Web to connect to servers, thus bypassing the complexities and costs of middleware solutions.

6.4 Ontologies

Software components, widely supported network protocols, and middleware all help to interconnect software modules in heterogeneous environments. However, such technologies do not solve interoperability except in the simplest cases (such as document sharing). Engineering industries are currently developing sophisticated "shared models" to provide at least data type interoperability among the more advanced applications, for example STEP. For applications that require more sophisticated (semantic level) interoperability, research in agent-based systems and interlingua such as KIF is ongoing.

STEP [ISO 10303] is a multinational effort aimed at producing a methodology for exchanging product and process models among users in industries such as aerospace, aeronautics, process, mechanical, electrical and automotive, where it has strong support. Work on STEP is also underway in architecture and construction. A STEP industry standard is defined by one or more application protocols (APs). An AP defines the scope and information requirements of a specific industry need and provides a map from them to an information model defined in the EXPRESS language. The AP also specifies software application conformance requirements for the standard. STEP follows an incremental methodology and STEP models support information sharing and hence interoperability. However, it is based on a static exchange of information as it was designed for data integration and not data interoperation; also, it does not support object behavior and the formalization of design knowledge that are required for improved business work processes. An overview of STEP (including motivation for extending it to support object behavior) and other standards (such as EDI and DXF) in the AEC industries is given in [Arnold 1996]. Other object-based models such as EDM [Eastman 1993] and MDS [Sause 1992] have also been proposed.

The Industry Alliance for Interoperability has issued an initial specification for semantic definition of a set of several hundred "foundation classes" for the Architecture Engineering Construction (AEC) industry. The Industry Foundation Class (IFC) specification describes a set of entities and their attributes. The initial release acknowledges the difficulty of enabling "interoperability" among software applications in a large and highly fragmented industry. (The AEC industry in the US has over 1,000,000 contractors, none of which has even a 5% industry share.) The initial specifications provide:

- Standard definitions for the attributes of entities that comprise an AEC project model;
- Structure and relationships among entities, from the perspective of AEC disciplines;
- Standard formats for data sharing, including static file data exchange and dynamic data exchange via CORBA.

Based on industry needs and the STEP standard, the IFC model describes four fundamental categories of entities:

- Products: physical elements in a building or facility, such as spaces, wells, doors, windows, equipment;
- Processes: the steps that are required to design, construct and manage a design, construction or operations project;
- Resources: the labor, material and equipment that are used to design, construct or operate a facility. Resources are used, but they do not become part of the facility itself.
- Controls: constraints on use of products, processes or resources.

The IFC-defined entity descriptions include some very specific entity types, such as IfcInsulation. Thus, at a simple level, IFC specifies some semantics. However, the IFC convention does not yet explicitly represent the design function of entities or many of their engineering behaviors, such as those that would be used to compute beam deflection or pump performance.

The IFC defines a number of industry-specific, i.e., semantic, datatypes. For example, the IfcActor object as one of an employee, person, department or organization; the IfcApplicance is one of a telephone, facsimile, copier, computer or printer.

The IFC standard is one example of emerging effort to enable semantic interoperability and use existing datatype and physical interoperability standards, e.g., CORBA. By integrating standards such as STEP or IFC's and CORBA, it may become possible for virtual enterprises to share manufacturing information. Hardwick et al [Hardwick et al 1996] describe a system (illustrated in Figure 6-2) that uses a World Wide Web interface to allow client applications to retrieve enterprise data modeled using STEP through application and database servers.

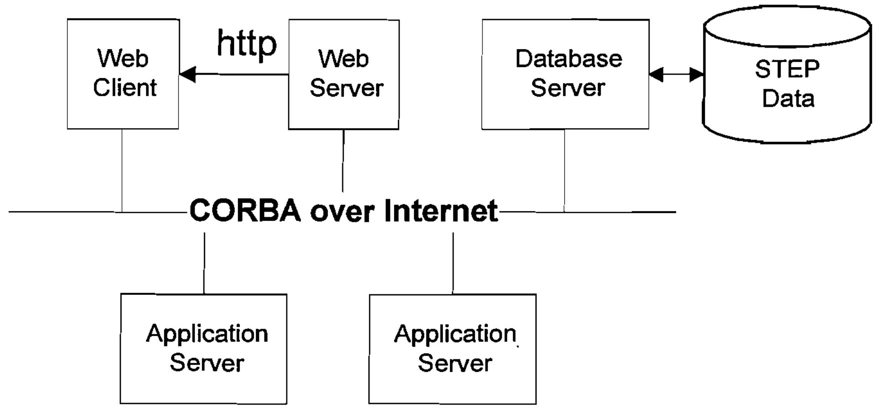


Figure 6-2: Virtual enterprises can use the STEP and CORBA standards to share product data through databases integrated via the World Wide Web.

Such an approach allows manufacturers to use different tools to process each other's data. While such a proposal does not address software interoperability per se but rather how applications can share data, Hardwick's findings include issues relevant to application integration. The authors point out that the communication medium between integrated applications must be cost-effective, portable and flexible; it must also have high performance (as STEP models can be large – they cite a case example of a simple model that required 2 megabytes of storage) and should not deliver unnecessary information (otherwise concurrent model access by other applications is inhibited). They note that barriers hindering effective inter-application communication include insufficient security controls, loss of control over projects (especially when communicating across corporate boundaries), lack of data and semantic interoperability among the applications.

An ontology for engineering analysis [Brooke et al 1995] is required if more direct application interconnection is to be possible. The typically unstructured development of analytical software complicates the integration of applications and the information they manipulate. In engineering analysis, the large number of physical and chemical phenomena typically require their own separate analysis application. The challenge of integrating such programs arises not just from the number of analyses required, but also their variety, particularly in the information processed. However, many such applications can be found to share common elements. The authors describe an approach to improve the integration of analytical tools and information by exploiting such commonality.

Engineering products are analyzed using models that predict behavior of physical phenomena, such as "thermal analysis". Traditional software design tends to implement analyses for individual classes of components or at most, in subsystems. The applications frequently adopt specific models and solution techniques – in engineering, for example, the finite element method is popular for stress and thermal analysis. Limits of component and subsystem models include the difficulty of applying them at different stages in the product life cycle or applying them at varying levels of detail.

6.5 Network-application Interoperability

Figure 6-3 shows that "software interoperability" is a broad term that covers three general approaches to interoperability. The first approach considers the level of abstraction at which applications or databases will share information. In other words, the focus is on the content and structure of the information exchanged or shared, not on the design of the applications or their behavior. The second approach attempts to integrate software components that are distributed across a network so that the integration of those components defines a logical application. Distributed object computing is the main thrust in this area at present with aggressive development behind standards such as OLE, CORBA and DCE. A third view of software interoperability has the objective of integrating distributed applications without regard for whether or not they comprise distributed objects or components – the level of abstraction at which data will be exchanged is not the principal focus of the research, perhaps because of a widely adopted standard that governs the level of abstraction at which knowledge among domain applications will be shared. "Interaction" interoperability combines a universal data-type interaction protocol with the evolving "network computer" model described earlier in this section.

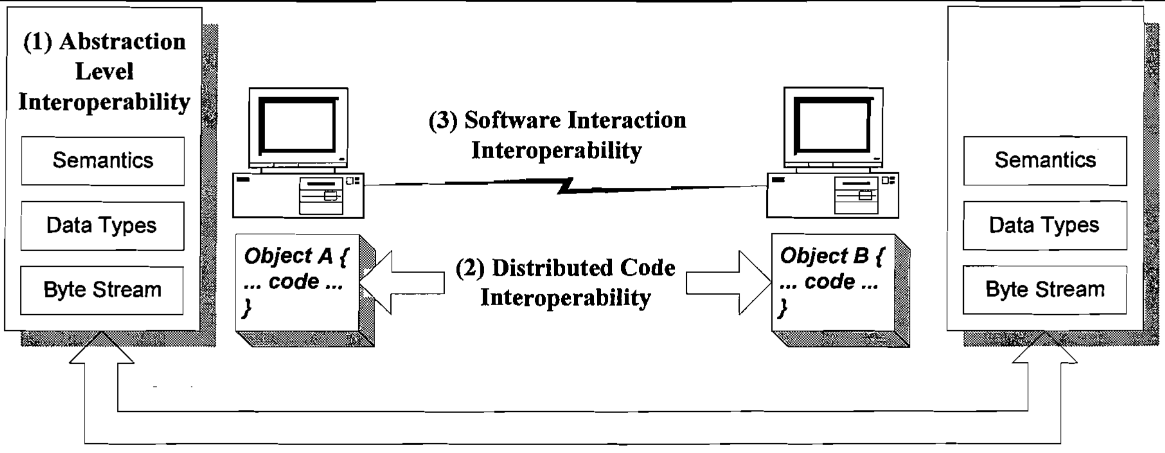

(1) Abstraction Level Interoperability
Semantics
Data Types
Byte Stream
(3) Software Interaction Interoperability
Object A { ... code ... }
(2) Distributed Code Interoperability
Object B { ... code ... }
Semantics
Data Types
Byte Stream

Figure 6-3: Software interoperability can be considered from three different viewpoints. The most common approach is to investigate the level of abstraction at which databases and/or applications can be integrated (view 1). A more recent development is to extend the object-oriented (or software component) paradigm so that objects can be distributed across network machines and still constitute a logical application (view 2). This is at a fine level of granularity and deals with the technical issues of integration software. By contrast, a third view is to consider the integration of applications, regardless of whether they are distributed or even object-oriented. Such an approach assumes an agreed abstraction level at which knowledge is shared or communicated (view 3). It focuses on the integration of network-based applications and assumes the functional and behavioral requirements of the system are met within the applications themselves.

The strength of a “network computer” model is the sharing and integration of distributed resources such as CPUs and memory, particularly by using a client-server protocol. There is already a large body of research in distributed and parallel computing for such applications in numerical analysis and large-scale simulations. However, these applications typically spread the computational load across the network within the scope of a single application. Future integration frameworks will provide software interoperability among multiple applications in a network environment where the information exchanged across machines is not a subset of the computational load for a single application but rather a sub-model in the applications’ domain.

In many engineering industries, the emerging STEP standard enables creation of suites of application protocols that use defined STEP datatypes as part of their software integration. For engineering industries, a new interaction interoperability among network-applications will therefore focus on developing an integration framework based on STEP. As STEP focuses on specific industries, for example with AP227 for plant spatial configuration, interaction interoperability will also have specific implementations for different industries such as the process industries. In engineering, STEP has developed many definitions for knowledge exchange, and an ISO standard should be available by the end of the decade. Adopting STEP as the standard for information exchange among distributed engineering applications may or may not be sufficient for data-type integration of plant applications in an individual industry. However, representation of the semantics of industrial component behavior (e.g., “pump curves” or thermodynamic models) is outside of current STEP standard efforts. Thus, attempting to share semantics will be difficult, particularly given the failure of advanced knowledge representation approaches like Expert Systems to reach broad acceptance in industry practice.

Due to the investment in legacy software systems, application developers in the plant industries are not extracting shared components from their applications and building traditional client-server systems. Instead, legacy software is being network-enabled through communication interface "wrappers". The process industries, for example, already have numerous design applications created to perform such specialized tasks as pipeline analysis and construction planning. It is therefore most practical to propose integration solutions that use legacy solutions when possible, rather than creating large numbers of new fine-grained applications using distributed object computing. For software interaction interoperability to work effectively, STEP models will need to represent enough form and function content so that individual applications can compute behaviors as necessary for particular purposes.

Existing and emerging applications for process facility design and maintenance comprise sufficient behavioral knowledge that exchanging STEP models in an integration framework should provide the necessary interoperability to enable the use of such resources in a tightly-coupled environment consistent with the notion of a "network computer". Figure 6-4 illustrates how a STEP model for a "centrifugal pump" component can be used to exchange form and functional information between an application for "pipe routing" and one for "instrumentation system design" without either application requiring complex behavioral views of the pump itself.

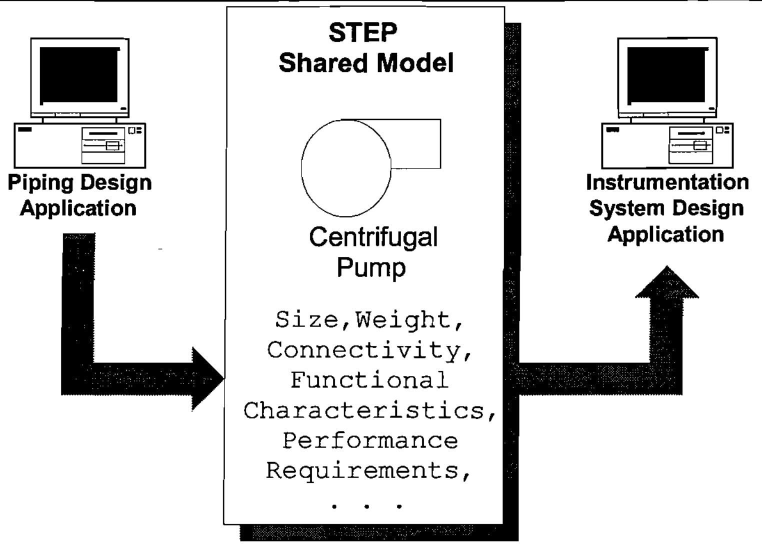

STEP
Shared Model
Piping Design Application
Centrifugal Pump
Size,Weight, Connectivity, Functional Characteristics, Performance Requirements, . . .
Instrumentation System Design Application

Figure 6-4: In some engineering industries, such as the process industries, the STEP standard allows applications to share "form" and "functional" knowledge for components of a common product (facility). Software interaction interoperability will exchange STEP data among legacy applications that support design, operations and maintenance of plant facilities.

In software interaction interoperability, analytical applications provide the behavioral information to add to the description of component form and function modeled in STEP. An example of direct integration of network-applications is illustrated in Figure 6-5. Such an integration framework would allow applications to share STEP models directly and so operate in a synchronized client-server manner. Figure 6-5 shows a hypothetical example of software interaction interoperability for a simple plant subsystem design.

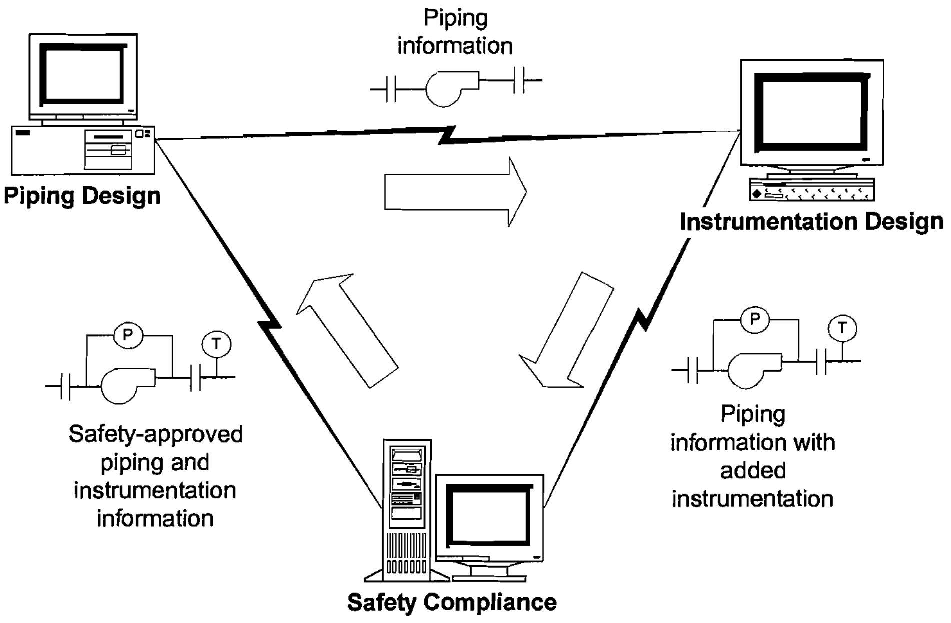

Piping information
Piping Design
Instrumentation Design
P
T
Safety-approved piping and instrumentation information
Piping information with added instrumentation
Safety Compliance

Figure 6-5: A software integration framework for network-applications will allow applications for engineering system design and management to exchange information directly using a client-server model. In this example, an application for pipe design passes information to a second application that adds instrumentation detailing and then passes that back to the piping design application via a third program that checks for safety compliance in the model.

The principle by which application behavior is integrated with the form and function of shared STEP models is more formally illustrated in Figure 6-6 using IDEF notation. The example shows how the piping application in Figure 6-5 would use its internal behavior to add form to an input functional specification. The integration framework would then coordinate this application output with the instrumentation application. The framework thus supports general integration of application behavior in the iterative development of a design model in a distributed environment to augment the form and functional content of STEP.

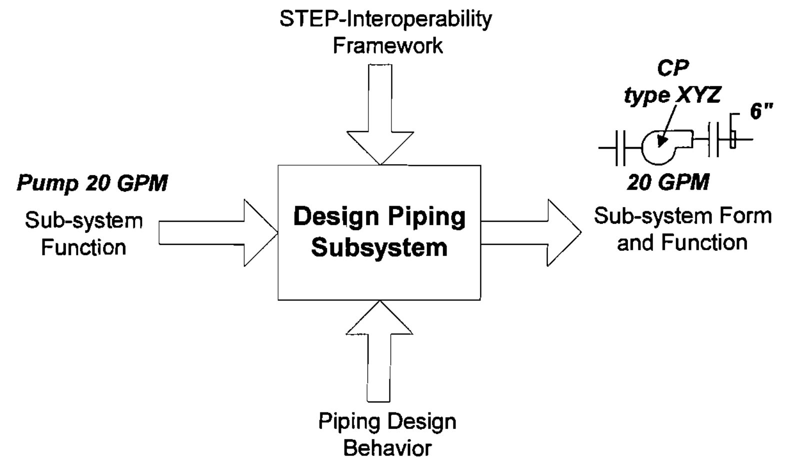


Figure 6-6: A STEP-interoperability framework will integrate application behavior among network-applications in specific engineering domains. The above IDEF diagram shows how application behavior is combined with the functional representation from a STEP model to produce form-function descriptions that can be provided to other applications during the iterative design process.

The integration framework will be based on a communication protocol for linking both legacy and new applications. The need for a formal communication protocol is critical in any distributed system. For example, the Hypertext Transfer Protocol (HTTP) is the driving force behind the World Wide Web (a virtual network based on a shared document model specified by HTML – the Hypertext Markup Language). While HTML is crudely analogous to the STEP specification, there is no equivalent to HTTP for managing software integration in the plant engineering industry. An integration protocol for the process industries must not only manage the exchange of STEP-compliant information but should also reflect current (and preferably improved future) work processes in the design and management of plant facilities.

7 CONCLUSIONS

The distribution of applications across modern networked environments has increased the complexity of attaining software interoperability. Tight coupling of software components on largely homogeneous machines is no longer sufficient as an interaction model. As more sophisticated network protocols become available and as software logic is embedded in a range of electronic devices, the boundary of an application and its interaction with other programs becomes less discernible. Integration across disparate platforms necessitates that distributed programs support at least a shared data-type model and preferably agree upon semantic level issues, especially since middleware technologies simply provide an encapsulated means of communication. The role of emerging ontologies in this regard is a key factor in the future of interoperable applications. However, for those environments in which there are widely-adopted standards and established legacy applications, it may be possible to formulate integration frameworks that can provide interoperability among distributed applications where information exchange is at a traditional "type" level and the domain behavior is captured in the applications themselves, thus obviating the need for explicit exchange of semantic knowledge.

8 GLOSSARY

AAITT: The Advanced Artificial Intelligence Technology Testbed. A testbed developed by the USAF Rome Laboratory for the design, analysis, integration, evaluation and exercising of large-scale, complex software systems.

AP: An Application Protocol defined within the STEP (ISO 10303) standard that specifies the scope and information requirements of a specific engineering industry need (e.g., process plant layout) and provides a map from such requirements to an information model defined in the EXPRESS language (also part of ISO 10303).

API: Applications program interface: the specification of the way one program can send input to or receive output from another. Most systems today provide APIs to support systems interfaces in a networked environment, at least at the level of simple data types.

Asynchronous/synchronous: Synchronous communication assumes a well-defined and regularly timed exchange of information between the two applications (RPCs, for example). An asynchronous model allows communication without the need for a coordinated communication channel. For example, with an "off-line" queuing facility, one application can pass information to another that need not be running at the time.

Client-Server: A model of distributed computing where an application (the client) invokes a software service provided by a remote application (the server). The two applications are perceived as part of a single application.

CORBA: Common Object Request Broker Architecture. An object-oriented multi-platform software interoperability standard being developed by the OMG (Object Management Group).

EXPRESS: An object modeling language defined as part of STEP (ISO1303).

Front-end/back-end: The "front end" of an application handles input and pre-processing; the "back end" performs analytical processing or handles data management requests on behalf of the front end.

IDL: Interface Definition Language. A high-level language that allows software developers to define the interfaces for object-oriented software components.

Interoperability: The ability for multiple software components to interact regardless of their implementation programming language or hardware platform.

IP: The Internet Protocol – a network communications specification used in common protocols such as TCP/IP and UDP/IP.

IRTMM: The Intelligent Real-Time Maintenance Management developed at Stanford University for application in process plants.

KIF: The Knowledge Interchange Format. A declarative language based on predicate calculus for the interchange of knowledge among disparate programs.

Middleware: A range of products/services that shield applications and their developers from a multitude of network communication protocols and database APIs.

Object: The basic element in the object-oriented programming (OOP) paradigm. An object comprises both data and code (procedures; usually called **methods**) that defines a behavioral interface to the encapsulated data. Properties of objects can usually be inherited by other related objects.

Object Request Broker (ORB): A logical entity responsible for establishing and managing communication between a client application and a remote object in a distributed environment.

OLE: Object Linking and Embedding. A proprietary interoperability technology developed by Microsoft Corporation to enable the interaction of embedded data objects across application boundaries.

Ontology: A shared vocabulary; an explicit specification using a formal and declarative representation for a particular topic. An ontology defines a shared understanding of a view of a domain, often describing entities, attributes, and relationships. An ontology will typically include classes, relations, facts, formal definitions, and informal descriptions (to aid people in understanding it). An effective ontology supports computational processes as well as a shared vocabulary between software and people.

OpenAccess: A software framework developed by the Electric Power Research Institute (EPRI) that enables applications to access data from various sources; it was formerly called EPRIWorks.

Process: The active component (as opposed to its binary image) of a program managed by an operating system.

Remote Procedure Call (RPC): An extension of the paradigm of a "local" procedure call that enables a client application to invoke a procedure in a remote (server) application as if the remote procedure were resident on the local machine. RPCs distribute application execution.

STEP: The Standard for the Exchange of Product data. Formerly known as ISO 10303, it is an emerging international standard for neutral data exchange in a wide variety of engineering disciplines.

Tight/loose coupling: Tightly coupled applications rely on each other for input and output, and they usually communicate synchronously. Loosely-coupled programs do not depend on each other for input or output and often communicate asynchronously.

9 REFERENCES

[Aiken et al 1994]: P. Aiken, A. Muntz, R. Richards; "DOD Legacy Systems Reverse Engineering Requirements," *Communications of the ACM*, Vol 37 No 5, May 1994, pp 26-41.

[Arnold 1996]: J. A. Arnold, P.M. Teicholz; "A Knowledge-Based Information Model for Components in the Process Industry," *Third Congress on Computing in Civil Engineering*, ASCE, Anaheim, California, June 17-19, 1996.

[ARPA 1993]: "Software Technology for Adaptable, Reliable Systems (STARS), STARS Conceptual Framework for Reuse Processes (CFRP)" Volumes I,II, Unisys STARS Technical Report STARS-VC-A018/001/00, Advanced Research Projects Agency (ARPA) STARS Technology Center, 801 N. Randolph St. Suite 400, Arlington VA 22203, October 1993.

[ATI]: Automation Technology, Inc., "OpenAccess: Enabling technology for universal information and data access from distributed heterogeneous systems," Draft.

[BBN 1989]: BBN Systems and Technologies Corp., *Cronus Advanced Development Model*, RADC-TR-89-151 Vol. 1, September 1989.

[Bernstein 1996]: P.A. Bernstein; "Middleware – A Model for Distributed System Services," *Communications of the ACM*, Vol 39 No 2, February 1996, pp 86-98.

[Boeing 1993]: "SEE Integration to Support Metaprogramming," Boeing STARS Technical Report CDRL 05104, Advanced Research Projects Agency (ARPA) STARS Technology Center, 801 N. Randolph St. Suite 400, Arlington VA 22203, June 1993.

[Brooke et al 1995]: D. M. Brooke, A. de Pennington, M.S. Bloor; "An Ontology for Engineering Analysis," *Engineering with Computers*, 1995, Vol 11 No 1, pp 46-45.

[Cutkosky 1993]: M.R. Cutkosky et al; "PACT: An Experiment in Integrating Concurrent Engineering Systems," *Computer*, Vol 26 No 1 1993, pp 28-37.

[Dell'Acqua 1996]: A. A. Dell'Acqua, J.F. Mazzaferro; "Wireless Communications," *Telecommunications*, March 1996, pp S1-S14.

[Eastman 1993]: C. M. Eastman, S. C. Chase, H.H. Assal; "System Architecture for Computer Integration of Design and Construction Knowledge," *Automation in Construction*, 2 (1993) pp 95-107.

[Eckerson 1995]: W. Eckerson; "Searching for the Middle Ground," *Business Communications Review*, September 1995, pp 46-50.

[EPRI 1996]: Electric Power Research Institute; "Integrated Knowledge Framework (IKF) for Coal-Fired Power Plants," EPRI Technical Report TR-106211-V1/2/3, EPRI Dist. Ctr. Pleasant Hill, CA, March 1996.

[Erman 1990]: L. Erman, J. Davidson, J. Lark, F. Hayes-Roth, *ABE: Final Technical Report*, TTR-ISE-90-104, May 1990.

[Feldman 1979]: S.I.Feldman, "Make -- a program for maintaining computer programs", *Software Practice and Experience*, Vol 9 No 4, 1979, pp 255-266.

[Fischer 1993]: M. Fischer, J.C. Kunz; "The Circle: Architecture for Integrating Software," *Journal of Computing in Civil Engineering,* ASCE, 9(2), pp 122-133.

[Frakes 1995]: W.B. Frakes, C. J. Fox; "Sixteen Questions about Software Reuse," *Communications of the ACM*, Vol 38 No 6, June 1995, pp 75-87.

[GE 1991]: General Electric/Advanced Technology Laboratories, *Advanced AI Technology Testbed*, F30602-90-C-0079, 1991.

[Genesereth 1994]: M.R. Genesereth, S. P. Ketchpel; "Software Agents," CIFE Working Paper #32, April 1994, CIFE, Stanford University.

[Gimes 1995]: J. Grimes, M. Potel; "Software is Headed Towards Object-Oriented Components," *Computer*, August 1995, pp 24-25.

[Hardwick et al 1996]: M. Hardwick, D. L. Spooner, T. Rando, K.C. Morris; "Sharing Manufacturing Information in Virtual Enterprises," *Communications of the ACM*, February 1996, Vol 39 No 2, pp 46-54.

[Hilton 1990]: M. Hilton, J. Grimshaw, *ERIC Manual*, RADC-TR-90-84, April 1990.

[ISO 10303]: International Organization for Standardization, ISO 10303 Industrial Automation Systems and Integration – *Product data representation and exchange – Part 1 Overview and fundamental principles*, ISO TC184/SC4/PMAG.

[Kador 1996]: J. Kador; "The Ultimate Middleware," *BYTE*, April 1996, pp 79-83.

[Karagiannis 1995]: D. Karagiannis, L. Marinos; "Integrating Engineering Applications via Loosely Coupled Techniques: A Knowledge-Based Approach," *Expert Systems With Applications*, Vol 8 No 2, pp 303-319, 1995.

[Khedro et al 1993]: T. Khedro, M. R. Genesereth, P.M. Teicholz; "Agent-Based Framework for Integrated Facility Engineering," Engineering With Computers, Vol 9 pp 94-107, 1993.

[King 1992]: S.S. King; "Middleware!," *Data Communications*, March 1992, pp 58-67.

[Kunz et al 1995]: J. C. Kunz, Y. Jin, R.E. Levitt, S-D Lin, P.M. Teicholz; "The Intelligent Real-Time Maintenance Management (IRTMM) System: Support for Integrated Value-Based Maintenance Planning," *IEEE Expert*, August 1996, pp 35-44

[Nadis 1996]: S. Nadis; *Computation Cracks 'Semantic Barriers' Between Databases*, Science, Vol 272 No 7, June 1996, page 1419.

[Neches 1991]: R. Neches, R. Fikes, et al.; "Enabling Technology for Knowledge Sharing", *AI Magazine* 12, 3 (1991), pp 36-56.

[Ning et al 1994]: J.Q. Ning, A. Engberts, W. Kozaczynski; "Automated Support for Legacy Code Understanding," *Communications of the ACM*, Vol 37 No 5, May 1994, pp 50-57.

[OSF 1992]: *Distributed Computing Environment – An Overview*, January 1992, OSF-DCE-PD-1090-4, Open Software Foundation/North America, 11 Cambridge Center, Cambridge MA 02142.

[Pan 1991]: J.Y. Pan, J.M. Tenenbaum; "An Intelligent Agent Framework for Enterprise Information," *IEEE Transactions on Systems, Man, and Cybernetics*, Vol 21 No 6, November 1991, pp 1391-1408.

[Purtilo 1994]: J.M. Purtilo; "The POLYLITH Software Bus," *ACM Transactions of Programming Languages and Systems,* Vol 16 No 1 pp 151-174, January 1994.

[Sause 1992]: R. Sause, K. Martini, G.H. Powell; "Object-oriented Approaches for Integrated Engineering Design Systems," *Journal of Computing in Civil Engineering*, Vol 6 No 3, Jul 1992, pp 248-265.

[Singh et al 1994]: V. Singh, F. Welz, R. H. Weston; "Functional Interaction Management: a requirement for software interoperability," *Proceedings of the Institution of Mechanical Engineers*, Part B Vol 208, 1994.

[Wegner 1996]: P. Wegner; "Interoperability," *ACM Computing Surveys*, Vol 28 No 1, March 1996, pp 285-287.

[Wileden et al 1991]: J.C. Wileden, A. L. Wolf, W.R. Rosenblatt, P. L. Tarr; "Specification-Level Interoperability," *Communications of the ACM*, Vol 34 No 5 pp 72-87, May 1991.