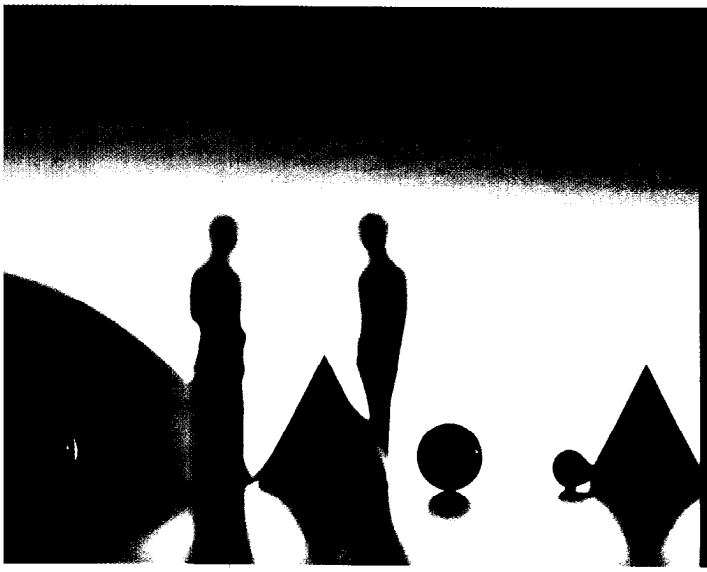

Object Technology

Hesham El-Rewini
University of Nebraska, Omaha

Scott Hamilton
Computer

Heralded by many as a promising solution to software complexity, object technology is coming into its own. This roundtable explores some recent trends, including distributed objects, OODBs, parallel computing, testing, and theoretical foundations.

Software development is an intrinsically difficult and time-consuming process. As software systems become larger and ever more complex, developers have been searching for mechanisms to control that complexity. The main goals have always been to lower the cost and improve productivity, reuse, and maintainability. A number of programming paradigms have in turn been heralded as the solution to the complexity problem. The procedure-oriented paradigm and structured programming were both popular for a while, and their fundamentals may still be valid. But today, all indicators point to object orientation as a more promising solution.

Although object orientation goes back to the 1960s when Simula was introduced, the object-oriented paradigm only started to gain momentum and popularity during the past few years. Many see it as an opportunity to rethink programming from an entirely fresh perspective. Unlike the procedural-oriented paradigm, where procedures are the fundamental software building blocks, OO software consists of interacting objects. Using objects for decomposition is thought to be more natural than using data or functions.

The term *object oriented* has been used to define a wide range of systems. Experts still disagree on which concepts and principles a system must embody to be considered object oriented. This introduction will not discuss the details of the different object-oriented concepts and classifications. Instead, some of the basics in object orientation are reviewed for those who may not be familiar with this paradigm.

We can perhaps distinguish three levels at which object-oriented concepts are applied. *Encapsulation* is the heart of the object-oriented paradigm. The idea is to hide the details of an object's internal data and operations. Objects communicate with one another through a well-defined interface. Thus, objects can be modified without affecting the rest of the system. At the first level, systems do not provide means to group objects with the same structure and behavior. One-level systems, wherein objects are derived from other objects, are called *classless* or *prototypical* systems. Code reuse is achieved through delegation, which means that objects del-

To learn more

To help readers learn more about object technology, we have put together a list of pointers to various Web sites that focus on object orientation: http://cs.unomaha.edu/object-orientation.

0018-9162/95/$4.00 © 1995 IEEE

egate the responsibility for executing operations to other objects. Microsoft's OLE/COM exemplifies this type of object technology.

At the second level, the mechanisms of inheritance and polymorphism are added. Abstraction can be specified via classes, where a class is a template for defining the behavior of a particular type of object. Classes can be organized in a hierarchical inheritance structure. A subclass inherits the operations and attributes of the parent class. Inherited operations can be altered by *overriding*, where the code is completely replaced, or by *extension*, when more functionality is added to the code. Objects are created by class instantiation, which lets programmers simply reuse an existing class having behavior similar to that required for a particular function. This is the idea behind class libraries. A class library is a collection of classes that programmers use to implement desired functionality. Class libraries provide some flexibility, but of course the programmer has to learn hundreds of classes and their relationships. Microsoft's MFC (Microsoft Foundation Classes) and Rogue Wave's Tools.h++ are examples of class libraries.

At the third level, frameworks represent a level of abstraction beyond that of class libraries. An OO framework is a reusable design that programmers can use, extend, or customize for specific computing solutions. A framework is not a class library; it is expressed as a class hierarchy plus a specification of the way its objects can interact. A framework can be thought of as a generic application that embodies a structure that is reusable in many specific applications. For example, MacApp is a framework for writing Macintosh applications. Since all Macintosh applications share the same structure (a main event loop that processes events), MacApp implements this common structure and lets software developers write the domain-specific routines used by the framework. Thus, frameworks allow the reuse of both design and code, but the developers must first identify the appropriate framework for a given application and write the domain-specific code. Taligent's CommonPoint, IBM's VisualAge, and ParcPlace-Digitalk's VisualWorks are other examples of frameworks.

As object-oriented technology is gaining more and more acceptance, groups such as OMG (Object Management Group) and ODMG-93 (Object Data Management Group) have made progress toward the standardization of object orientation. The goal is to provide a common architectural setting for object-oriented applications based on widely available interface specifications. Effective adoption of these standards enables portability of customer software across different products, ensures interoperability among systems, and encourages the creation of interchangeable, reusable software components. This in turn will reduce the complexity and lower the costs of software and improve productivity. For example, the Common Object Request Broker Architecture (CORBA) is a standard by OMG for distributed object-oriented client-server computing. CORBA provides the mechanisms by which objects transparently make requests and receive responses, as defined by OMG's ORB. The ORB provides interoperability between applications on different machines in heterogeneous distributed environments. Several commercial products that adhere to these standards are available, such as Hewlett-Packard's Distributed Smalltalk, Digital's ObjectBroker, IONA's Orbix, and IBM's SOMobjects.

In this roundtable. We present an assortment of short articles from both industry and academia on a number of interesting topics in object orientation. Yen-Ping Shan, Ralph Earle, and Skip McGaughey review several approaches for exploiting object technology in client-server environments using distributed objects on the server. A second article, by Sumi Helal and Ravi Badrachalam, examines Microsoft's COM standard and compares COM with CORBA.

Andrew Chien and Andrew Grimshaw each contribute a short article on the growing need for object orientation in the world of parallel computing. Chien speculates that large parallel software communicating via ORBs will be developed for servers and low-end APIs on symmetric multiprocessors. Grimshaw discusses the increasing use of C++ in the high-performance community, outlines the debate over how best to encapsulate parallelism in objects, and expounds on the need in the high-performance computing community to adapt and conform to existing standards in order to leverage commercial software.

Object-oriented databases are a growing field, as suggested by the four short articles presented here. Byung Lee outlines the current state of object-oriented systems and standards. Andrew Wade looks at the future of OODBs and identifies some short- and long-term directions. Dave Morse draws a parallel between the network database model and present-day OODBs and discusses his company's attempt to blend the relational and network models while realizing the benefits of object-oriented databases. Finally, Ahmed Elmagarmid and Evaggelia Pitoura discuss database system integration and the effect of applying object technology to the design and implementation of multidatabase systems.

Robert Binder contributes an article about the changes in OO testing and how its practice has improved during the last three years. He also reviews some of the unique problems and challenges facing testing in the object-oriented arena.

Finally, Peter Wegner addresses the confusion that exists in the theoretical foundation of object-based programming. He introduces the interactive paradigm for modeling objects and extends Turing machines by having them support external inputs during computation. This is a very interesting article that we would expect to generate much discussion. ▮

Acknowledgment

We'd like to thank the many authors who submitted short articles for this roundtable.

Hesham El-Rewini *is an associate professor at the University of Nebraska at Omaha; e-mail rewini@csalpha.unomaha.edu.*

Scott Hamilton *is articles editor for* Computer*; e-mail s.hamilton@computer.org.*

ROUNDING OUT THE PICTURE: OBJECTS ACROSS THE CLIENT-SERVER SPECTRUM

Yen-Ping Shan, Ralph Earle, and Skip McGaughey
IBM Software Solutions

Until now, object technology has proven beneficial only to the client side of the client-server picture. We believe there is no intrinsic or conceptual reason for this and in the near future, object technology will extend to cover the entire client-server configuration. The benefits of objects that apply to client implementations will apply equally well to the server and to distributed computing.

This has two implications. First, objects on the server side will become as common and useful as objects already are on the client side. Second, as server objects become a reality, a server must have a mechanism for interacting with clients. Distributed object technology becomes the natural choice to "glue" the client and server together.

Object-oriented technology first became popular on the client side because it does such a good job of addressing the complexity of GUI programming. The benefits of object technology that make this possible include encapsulation, reusability, and a close mapping between the problem and solution domains. (In this case, the real-world elements of the user interface map to self-contained software objects.)

On the server side, multiuser requirements and the greatly magnified scale of servers naturally lend themselves to complexity, which creates a need for object technology's benefits. The same reasoning applies to communication between client and server: With objects proliferating across the network, managing complexity and change becomes even more critical. Again, encapsulation, reusability, and problem/solution mapping come to the rescue.

In addition, when object behavior is available on both the client and the server side, it makes sense to use distributed objects as a communications mechanism that takes full advantage of the available behavior. This yields high productivity and broad tool support.

Establishing objects. In establishing objects on the server, three major approaches are currently being followed.

Objectifying the transaction processing monitors. When OO programming languages are introduced into the TP monitors, transaction programs that used to be written in 3GLs such as Cobol can be written in OO languages. Each invocation of a transaction still maintains the atomicity, consistency, isolation, and durability (ACID) properties essential to the integrity of business computing.

Objectifying the data servers. Data servers execute logic within the context of a database. This logic generally takes the form of the stored procedures and triggers supported by many relational databases. Developers can objectify their data servers if they can program their stored procedures as objects.

Objectifying the application servers. An application server is a nontransactional server similar to a database server, but with the logic in front of the database rather than in stored procedures. Unlike the two servers described earlier, which usually run for a short duration, an application server can stay running for a long time. Programmers can use an OO language to implement their application servers.

Glue. Creating and propagating server objects is the first step, but server objects must be connected to the client objects, and the glue is the distributed object technology. Depending on the nature of the objects on each side, there are three major ways to glue the clients to the servers.

Common buffer. The client and server agree privately on a buffer format, and the communication middleware ships the buffers back and forth between them. Although the abstraction level is low, this is the most heterogeneous approach. For instance, by using common buffers, a Visual C++ client can interact with a mainframe Cobol server program. The languages or vendor products used to implement the client and the server are not restricted. (In fact, with common buffers, neither client nor server even needs to include objects.)

Common middleware. The client and server are connected by a common middleware service such as an ORB (for example, SOMobjects or HP Distributed Smalltalk). Here the abstraction level is higher, which enables more tool support and higher productivity. Data conversion and marshaling are done by the middleware, and interaction across languages is possible. However, there is less heterogeneity than with the common buffer approach. For example, to interact with a legacy server program when source code is not available, it's more appropriate to use the common buffer approach than to wrapper the legacy code with the required object constructs.

Common language. This approach takes advantage of both sides being implemented in the same language. Example systems are IBM Distributed Smalltalk and Forte. The goal is to provide high local/remote transparency that masks the existence of underlying middleware. Programs can be created in a single address space and then partitioned into pieces to run on different machines. Because the client and server follow the same protocols, additional services are available (for example, distributed garbage collection) that are infeasible with common buffers or common middleware. The common language approach achieves the highest levels of abstraction and productivity, at the expense of some heterogeneity.

AS OBJECT TECHNOLOGY INCREASINGLY becomes a factor on servers and networks, we can expect to see a variety of approaches, with different benefits and restrictions. It's not likely, or even desirable, that any of the approaches mentioned above could so dominate that others become unnecessary. Besides pushing the boundaries of each individual approach, technology providers will have to ensure that the approaches coexist and are complementary. Only then can application developers enjoy the full benefits of an object-oriented client-server environment. ▮

***Yen-Ping Shan** is lead architect for Distributed and Host Smalltalk, **Ralph Earle** is senior editor for IBM VisualAge products, and **Skip McGaughey** is market manager for IBM's object-oriented application development tools; e-mail ypshan@vnet.ibm.com.*

COM VERSUS CORBA: WILL MICROSOFT COME OUT ON TOP?

Abdelsalam Helal, ***Purdue University***
Ravi Badrachalam, ***University of Texas at Arlington***

Microsoft defines the Component Object Model (COM) as an object-based programming model designed to promote software interoperability. The objective is to let two or more applications or software components easily cooperate with one another even if they were written by different vendors at different times, in different programming languages, or if they run on different platforms with different operating systems. In this brief article, we discuss Microsoft's COM and show how it addresses the issues of distributed object computing.

What is component software? A component is a reusable piece of software in binary form that can be plugged into other components from any other source. For example, a component might be a thesaurus tool sold by one vendor that can be plugged into different vendors' word processing packages. Software components must adhere to a binary external interface standard, but their implementation is completely unconstrained.

COM interfaces. COM applications interact with each other and with the system through collections of functions called "interfaces." A COM interface is a contract between components to provide a certain level of service or functionality. COM interfaces provide the following benefits:

- Adding new functionality to an object will not require recompilation for existing clients.
- The binary standard allows COM to transparently make an RPC call to an object in another process or on another machine.
- Any programming language that can create structures of pointers and explicitly or implicitly call functions through pointers can create and use COM objects. Such languages include C, C++, Pascal, Ada, and Smalltalk.

COM defines one special interface, IUnknown, to implement some essential functionality. This is the base interface that all objects must support. QueryInterface is a method on this interface that allows clients to dynamically find out at runtime whether an interface is supported by a COM object. The developer can create a description of the COM object's interface methods using COM's Interface Definition Language (IDL). (The COM IDL is based on simple extensions to the IDL used in OSF DCE.)

Identifying objects and interfaces with GUIDs. Future distributed-object systems might have millions of interfaces and software components that need to be uniquely identified. The probability of collision between human-readable names is nearly 100 percent in a complex system. To avoid this problem, COM uses globally unique identifiers (GUIDs) that are 128-bit integers and are virtually guaranteed to be unique in the world across space and time. GUIDs are the same as UUIDs (universally unique identifiers) as defined by OSF DCE.

Aggregation in place of inheritance. Microsoft believes that implementation inheritance creates serious problems in a loosely coupled, decentralized, evolving object system. While agreeing that implementation inheritance is a very useful technology and tight coupling is not a problem when the implementation hierarchy is under the control of a group of programmers, Microsoft argues that the contract between objects in an implementation hierarchy is implicit and ambiguous. When the parent or child component changes its implementation, the behavior of related components may become undefined.

In place of implementation inheritance, COM provides two mechanisms for code reuse called containment/delegation and aggregation. In the first mechanism, one object (the outer object) simply becomes the client of another object by internally using the second object (inner object) as a provider of services. Clients of the outer object never see the inner object, which is completely hidden (encapsulated).

The second mechanism, aggregation, is a special case of containment/delegation. An aggregated object is essentially a composite object in which the outer object exposes the inner object's interface directly to clients as if it were part of the outer object.

Local/remote transparency. COM lets clients transparently communicate with objects regardless of where they are located. A client accesses all objects through interface pointers. A pointer must be in-process, and any call to an interface always reaches some piece of in-process code first. If the object is in-process, the call reaches it directly. If the object is out-of-process, the call first reaches a proxy object provided by COM that generates the remote procedure call to the other process or machine.

COM library. While the core of the Component Object Model is the specification for how objects and their clients interact through the binary standard of interfaces, COM itself involves some system-level code and hence some implementation of its own. The COM library is a system component that provides the mechanisms of COM. (Microsoft has implemented the COM library in COMPOBJ.DLL on Windows and OLE32.DLL on Windows NT and Windows 95.)

How does COM differ from OMG's CORBA? A major difference between CORBA and COM is the way they implement their interfaces. COM specifies a series of interfaces that components must implement to interact with other component objects. All these interfaces must derive from the base interface IUnknown. CORBA does not specify a single base class, and vendors get to implement their own.

CORBA is a specification without a reference implementation. This is due largely to the conflicting interests of vendors, and OMG was forced to make CORBA deliberately vague, omitting certain details. Its lack of imple-

mentation detail is both its greatest strength and limitation. While providing the flexibility for more creative vendor implementations, it also opens up issues such as inconsistent administration, incompatible ORBs, and nonportable servers. Microsoft's COM, on the other hand, is very specific in its implementation details, but this tight control can lead to nonoptimal solutions.

Another subject of major controversy between COM and CORBA is implementation inheritance. Implementation inheritance is the classic inheritance in the object-oriented sense, utilizing class hierarchies. Interface inheritance refers to the ability to reuse interfaces between objects without imposing class hierarchies. It follows the object-oriented concept of encapsulation. Microsoft believes that implementation inheritance is improper when applied to interprocess object models, so COM supports interface inheritance but not implementation inheritance.

IBM, on the other hand, claims that a truly object-oriented system should support this feature and has implemented it in SOM, one of the first successful implementations of CORBA (commercially available since 1991). IBM says that SOM is object-oriented, while COM is object-based. This debate has merits on either side, and whether object-oriented programming can be done effectively without implementation inheritance is something that individual developers should decide.

Another point of comparison is the level and variety of implementations based on these two standards. While CORBA has many implementations that support object interactions across networks, Microsoft's OLE (an implementation of COM) does not yet support cross-platform object interaction.

Forecast. Microsoft is a member of OMG but has chosen not to support CORBA and has instead defined its own standard. None of the other members of OMG are as focused on desktop software as Microsoft is, and with Microsoft's predominance, COM is expected to become an industry standard in the desktop market. Further, Digital has decided to port COM to its OpenVMS and OSF/1 platforms and is working on a bridge that will make COM and CORBA interoperable. Finally, Microsoft is pushing its Windows NT operating system as a replacement for Unix on the server side. Given Microsoft's strength and popularity, this may very well happen within the next few years. If this becomes a reality, with major hardware vendors porting and supporting Windows NT as a server-quality operating system, distributed COM is bound to play a prominent role in the distributed interoperable objects arena. ■

Abdelsalam (Sumi) Helal *is a visiting assistant professor at the Department of Computer Sciences at Purdue University; e-mail helal@cs.purdue.edu.*

Ravi Badrachalam *is with the University of Texas at Arlington; e-mail ravi@cse.uta.edu.*

Parallelism and Objects

SCALABLE PARALLEL SYSTEMS ?=? SCALABLE OBJECT REQUEST BROKERS

Andrew A. Chien, ***University of Illinois***

Object-based computing provides numerous advantages for parallel and distributed computing. Notably, objects offer the benefits of object-oriented programming: modularity, encapsulation, and reuse, which have been widely demonstrated for uniprocessor systems. In parallel and distributed systems, these features can be used to cleanly manage separate address spaces, locality and distribution, concurrency, and resources. However, scalable parallel systems have not yet been widely used for parallel object applications. Below, I briefly discuss some of the opportunities for parallel objects and scalable parallel systems.

What do scalable parallel systems offer scalable ORBs? The increasing acceptance of a distributed object model based on emerging standards such as the Common Object Request Broker Architecture (CORBA), IBM's Distributed System Object Model (DSOM), and a variety of others presents an unprecedented opportunity for parallel system vendors. If enterprise computing follows the evolution of the desktop platform, distributed information management and distributed desktop applications are likely to become the dominant applications for parallel computers. Such applications will be based on encapsulated objects developed for a model such as Microsoft's OLE, Novell's AppWare, or the standards mentioned above. For vendors of scalable parallel hardware, such developments are an exciting opportunity; ORBs represent an open API for distributed and parallel applications and have the potential to provide the wealth of application software needed to sustain the market for high-end scalable systems.

The increasing acceptance of a distributed object model presents an unprecedented opportunity for parallel system vendors.

The synergy is critical. The wealth of electronic information becoming available as businesses automate both the collection and acquisition of information as well as integrate their disparate information resources can easily swamp the computing resources available in small-scale server systems. Possible alternatives include clustering workstations or going with scalable parallel systems (MPPs). And in fact, scalable hardware systems on the market today offer several key advantages: highly tuned communication software, single system image, and hardware design for reliability and maintainability. These advantages support the building of high-performance ORBs that will provide performance scale-up with greater convenience than is possible with a distributed cluster.

What do scalable parallel object systems need to support scalable ORBs? The federal government's High Performance Computing and Communications

(HPCC) initiative clearly provides the critical hardware and software basis for scalable ORBs. However, because the HPCC efforts have focused on scientific grand challenges (only recently on national challenges), the software developed for parallel systems has generally supported single jobs, scientific applications, and APIs well-suited to those applications. Scalable ORBs can leverage scalable hardware and high-bandwidth communication software, but they have several additional requirements: high-performance multithreading, high-performance distributed resource management, and compatibility with desktop application software.

To date, scalable parallel systems have provided only modest support for multithreading, as the absolute highest performance with scientific applications can often be attained with a single thread. In server type applications, efficient multithreading is an essential tool for program decomposition, performance, and software complexity management. In addition, most scalable parallel systems provide little or no assistance for distributed resource management; this again is largely a consequence of the focus on scientific applications with predictable resource requirements. Fortunately, the increasing use of irregular, adaptive algorithms for parallel scientific computing is broadening these requirements. Distributed object applications typically have rapidly varying dynamic resource requirements that must be managed efficiently to achieve scalable performance. Finally, compatibility with desktop software is a critical issue if third-party vendors are to be persuaded to port and run their distributed objects on a scalable server. For scientific applications, the focus has been on source compatibility, but for commercial application software, where the source may not be available, binary and operating system environment compatibility are critical issues. If scalable parallel systems are to serve this market, they must address these concerns.

Unlike the Fortran community, the parallel C++ community has not reached a consensus on how to best introduce parallelism into the language.

A closing perspective. Parallel system use is exploding in the commercial world, and we will soon have a much larger parallel software base. However, the bulk of the parallel systems being installed are low-end symmetric multiprocessors, not multimillion-dollar scalable parallel computers. So one might reasonably expect that much of the parallel software will be developed for server and low-end APIs. To provide true scalability for commercial applications, it must be possible to scale applications built to a single API over a wide range of performance. How then will scalable systems provide scalability in both the software and the hardware? One possible path is through distributed object request brokers. ■

Andrew A. Chien *is an associate professor in the Department of Computer Science and the National Center for Supercomputing Applications at the University of Illinois; e-mail achien@cs.uiuc.edu.*

PARALLEL PROCESSING AND OT

Andrew Grimshaw, ***University of Virginia***

Parallel programming has experienced a long and difficult maturation process. The reasons are many, but one critical problem is the difficulty in programming the newly developed architectures. Porting and tuning an application to a new architecture can take as long as the interval between new architectures. In this environment, applications developers face a daunting challenge, especially with increasingly large and complex applications. Developers must identify parallelism in an application, translate that parallelism into code, manage communication and synchronization manually, and, at the same time, keep abreast of architectural change.

To address these problems, many researchers have turned to object-oriented programming, considered a powerful software engineering tool for sequential software. Attributes such as encapsulation, polymorphism, fault-containment, and software reuse have made complex sequential software more tractable, and researchers want to capture those benefits for the parallel computing community.

The majority of object-oriented parallel computing projects focus on C++ because of its widespread use and the availability of C++ compilers on most parallel platforms. Unlike the Fortran community, the parallel C++ community has not reached a consensus on how to best introduce parallelism into the language. One reason is that the base language is so flexible that constraining parallelism to arrays, as in high-performance Fortran, makes no sense when so many different data structures and types are possible. Hence, there are task (control), data, and combined task and data parallel projects.

A deeper split within the community exists as to how to provide parallelism. There are two primary schools of thought. The first, the libraries group, argues for building highly optimized, extensible class libraries that encapsulate parallelism. Users could use these class libraries without knowing anything about parallelism or about what goes on inside the class library. The heart of the library group's argument is that C++ already provides a powerful mechanism for language extension via classes, inheritance, and templates. Additional extensions would only clutter the language. Furthermore, with no consensus on language features, compiler vendors are unlikely to support any language extensions, and users will not want to risk embracing the "wrong" feature.

The second school of thought, the extensions group, argues that the best way to achieve parallelism is via language extensions. The heart of the argument is that parallel composition is as important a concept as sequential composition. With concurrency a part of the language, compiler technology can more readily be brought to bear on optimization.

The future: Object orientation and HPC. Independent of the libraries and language extensions debate,

object-oriented computing has a future in high-performance computing. The HPC community is at a crossroad. In the past, the community mainly employed expensive supercomputers and often developed its software in-house because the commercial marketplace would not. Today, however, cost-effective PCs and workstations have closed the performance gap, and a booming commercial software business targets these desktop machines. At the same time, many traditional HPC users are downsizing and no longer have the resources to develop everything in-house. Therefore, the HPC community has tremendous incentive to leverage the existing commercial software base.

The OO paradigm can impact the new environment in three ways: leveraging the huge investment in commercial software, integrating parallel components, and making parallel computing relevant to the computing community at large. Leveraging commercial software is crucial. The scientific/parallel software market is minuscule compared to the desktop market, where billions of dollars are being spent.The scientific computing community does not have the resources to duplicate that effort. It must adapt, conform to existing standards, and exploit the desktop market, a market that is moving increasingly toward object-based and object-oriented interoperability standards.

Object-based techniques also relate to interoperability. The HPC community has no interoperability standards. Thus, it is difficult to use parallel components developed by different research groups in a single application. There is also desire to construct multidisciplinary simulations—for example, coupling ocean and atmosphere models in a global climate model. The individual components of these simulations are often stand-alone parallel codes. While the file system can be used as an interface and data transport mechanism, more effective techniques are needed. Object technology can be used as an interface description mechanism, a data transport and coercion mechanism, and, using wrappers, as a mechanism to extend the life of legacy components. Unfortunately, existing IDLs do not address many of the special interface issues relevant to a parallel program.

Finally, if the HPC community is to impact the desktop market, it must conform to existing and evolving standards. If HPC components conformed to standard interface descriptions, parallel components could be used transparently by commercial applications developers. That would make parallel computing relevant to a broader user base and encourage vendors to develop parallel hardware for the commercial market. The easiest way to do this near-term is to encapsulate parallelism within objects, making the parallel component a particularly fast version of an existing sequential code.

Fortunately, within the distributed systems community there are standardization efforts to define object architectures and IDLs. The HPC community should embrace these efforts, lobbying for changes needed to best support its purposes. Only then can high-performance parallel computing begin to make the transition into the mainstream. ■

Andrew Grimshaw *is an associate professor of computer science at the University of Virginia; e-mail grimshaw@virginia.edu.*

Object Databases

OBJECT-ORIENTED DATABASES: SYSTEMS AND STANDARDS

Byung S. Lee, ***University of St. Thomas***

Since the mid-eighties when Gemstone was introduced as the first object-oriented database management system (ODBMS), a dozen other commercial ODBMSs have joined the fierce competition in the market. Although we call them all ODBMSs, they differ in their system concepts and data management standards. Below, I discuss the past evolution and future prospects of those ODBMSs.

Evolution of systems. There have been three approaches to building an ODBMS: extending an object-oriented programming language (OOPL), extending a relational DBMS, and starting from the ground up. The first approach realizes an ODBMS by adding to an OOPL persistent storage for multiple concurrent accesses with transaction support. This extension has the advantage of reusing the type system of a programming language as a data model and thus achieves a seamless integration between programming language and database manipulation language. This approach has become the most popular in the commercial world so far and is represented by commercial ODBMSs such as ObjectStore, Versant, Objectivity, and O2.

In the second, extended relational approach, an ODBMS is built by enhancing an existing relational DBMS with object-oriented features such as classes and inheritances, methods and encapsulations, and complex objects. Exemplary systems are the research prototypes Postgres95 and Starburst, which were incorporated into the commercial products Illustra and DB2/6000 V2, respectively.

The third approach is revolutionary in the sense that an ODBMS is built from the ground up, as represented by UniSQL and OpenODB. Research prototypes like Orion and Zeitgeist belong to this category as well. These systems provide their own proprietary data models and data manipulation languages.

Lately, a new paradigm of ODBMS, called object-relational DBMS (ORDBMS), has begun to draw increasing attention. The objective of an ORDBMS is to support both relational and object-oriented database applications. Systems in the category of ORDBMSs at the time of this writing are extended relational DBMSs such as Illustra and DB2/6000 and ground-up ODBMSs such as OpenODB and UniSQL.

Evolution of standards. In September 1991, the Object-Oriented Database Task Group of the ANSI Database Systems Study Group published its final report for establishing "a framework for future standards activities in the object information management area." Subsequently, two ongoing efforts have begun—one by the Object Data Management Group (ODMG) and the other by the American National Standards Institute (ANSI) SQL3 (also called Object SQL) committee. In October 1993, ODMG published its first version of standard ODMG-93, which defines the data

model, query language, and language bindings with which all commercial ODBMS systems are advised to comply. The ANSI SQL3 committee has not yet finished developing a full-fledged standard of SQL3. Their target publication years are 1997 for some major components and 1998 for the other components. Both standards are still evolving and in some sense competing with each other. This lack of a solid, unified standard is a barrier to be overcome as soon as possible.

Future of systems. There is no doubt that ODBMSs will gain increasing market share. In particular, ORDBMSs will gain more popularity because of their dual support for relational and object-oriented data management capabilities. Virtually all commercial ODBMSs will move in that direction. One interesting question in this regard is who will arrive there first. Right now, Illustra, UniSQL, OpenODB, and DB2/6000 V2 are considered the first ORDBMSs. Other ODBMSs will follow the same track. On the other hand, conventional relational DBMSs, including Informix, Ingres, Oracle, and Sybase, may gradually be turning into ORDBMSs. However, their current object-oriented extension is limited to providing SQL3 wrappers and storing complex data as binary large objects (BLOBs) and thus entails poor query processing and optimization. Nonetheless, some users may be willing to wait for relational DBMS vendors to release full-fledged OO extensions.

ODBMSs will not replace relational DBMSs in conventional database markets such as inventory management, airline reservations, finance, and investment management. Rather, the use of ODBMSs will be restricted to complex applications such as design engineering and network management. Geographic information systems and electronic book technology are emerging as new areas of complex applications. Ironically, hierarchical DBMSs, such as IMS, will still dominate in terms of deployed data volume for at least another decade.

Future of standards. The ODMG standard will soon emerge as the strongest standard in the commercial ODBMS community and will influence current commercial ODBMSs. A few commercial products in compliance with ODMG-93 are soon to be released, and a revised version of the ODMG standard, ODMG-95, is on its way. ANSI SQL3 covers more comprehensive features such as rules and triggers in addition to object-oriented features, but its current progress is slow. SQL3 will be the primary target standard for extending conventional relational DBMSs such as Informix, Oracle, and Sybase with OO features. Both ODMG-95 and ANSI SQL3 will add impetus to the ORDBMS trend.

We may see the recurrent phenomenon of vendors running ahead of ANSI committees, which was evident with the Internet protocol—for example, TCP/IP against ANSI X.25. Eventually, however, it is likely (and desirable) that the two streams of standards, ODMG and SQL3, will influence each other or even merge. ■

Byung S. Lee *is an assistant professor of graduate programs in software at the University of St. Thomas in St.Paul, Minnesota; e-mail bslee@stthomas.edu.*

WHERE ARE OBJECT DATABASES HEADED?

Andrew E. Wade, ***Objectivity Inc.***

By definition, all ODBMSs support storing and sharing of objects, with an interface to the object that allows combining and invoking operations, thus raising four important issues for deploying mission-critical applications: integrity, scalability, reliability, and flexibility.

To illustrate the first, consider ODBMS support for caching. Unlike server-centered DBMSs, most ODBMSs actually move referenced objects into the application's address space, with both an up- and a downside. The upside is performance. Operations across address spaces take milliseconds, while operations within an address space take tenths of microseconds on the same machine, so access is 10,000 times faster. Complex applications often achieve 10-100 times better performance from such caching and relationship management. On the downside, however, an application with a bug can damage a cached object and corrupt the database. Thus, a key issue for the ODBMS is what it must do to prevent such corruption. Most systems today (either all or some of the time) give the application a direct pointer to the objects. Eventually, such pointers become invalid; they are always invalid after a commit, because commit means semantically that the object can move to other users. If the programmer uses the pointer at the wrong time, it will crash or, worse, dereference into the middle of some other object and corrupt the database.

A better approach uses one level of indirection to insulate the user and guarantee integrity. To the user, this looks exactly the same, but underneath the ODBMS transparently adds one extra pointer dereference. Then, wherever the object moves, the ODBMS can automatically fix the pointer (recache the object if necessary), so that execution continues normally. With this approach, there is no corruption from bad references, and the cost of an extra pointer dereference is unnoticeable in most applications.

An indirect approach can also enable the ODBMS to swap objects, which is a key for scalability. A prototype that works well with 10 objects should not collapse when put into production with 10 million objects. Indirection allows an intelligent cache management strategy that swaps out old, unused objects, making room for new objects, so performance can scale linearly with the number of objects processed to unlimited database size.

Scalability in the number of users is also an increasingly important issue. Systems built around client-side-only functionality tend to bog down quickly with only a handful of users. Systems built around server-side-only functionality can do very well for simple, short transactions—for example, those typical of traditional DBMSs and OLTP systems—but suffer from a server bottleneck. All requests must pass through the same server queue, so the more users you add, the longer they wait at that queue. ODBMSs are migrating to a distributed client-server architecture, which spreads functionality across clients and servers to avoid such a bottleneck. The server can move clusters of objects simultaneously and supports multithreading and simultaneous communication links to each client, so there is no waiting unless object conflicts arise, in which case the usual locking keeps everything

safe. The client-side object manager supports indirection for integrity and intelligent cache management for scalability. A single logical view over all the distributed databases allows transparent and dynamic server reconfiguration, so even as server capacity is reached, administrators can simply add new servers and spread objects across them. The result is scalability in clients (due to direct communication links) and also in servers (due to distributed ability to add servers and spread objects across new servers).

Another emerging mission-critical need is fault tolerance. It's great to be able to access objects transparently in New York or Tokyo, but what happens when the satellite link goes down? Support for redundant replicates of system structures is needed to allow users to continue work, even though they can't access new objects at the other site until the satellite link is restored. Additionally, using object replicates allows continued object access, with update correlation performed according to user-chosen rules when the link is reestablished.

Longer term directions? Early ODBMS users were sophisticated technologists who had no fear of diving into the intricacies of C++ and ODBMSs, but users of the future will demand easier interfaces and more productivity tools. In line with the move to open systems, they'll demand that tools be open too, rather than tied to single vendors. ODBMSs have begun to move in this direction. Some now support SQL, leveraging existing programs and user training, as well as Microsoft's Open Database Connectivity (ODBC), which allows off-the-shelf plug-and-play with most of the standard 4GLs, GUI tools, report generators, form generators, and so forth. Some third-party object tool vendors have begun to integrate their products with the leading ODBMSs, and the ODMG standard will accelerate this trend. (ODMG is planning future support for dynamic object type access and better tool support.)

One area in which ODBMSs have been weak is security. Broader use will demand more sophisticated security models to support objects, object clusters, and composite objects distributed across networks of computers. The trend is toward wider integration among computers. Instead of separate personnel, finance, manufacturing, and order-entry databases, companies today want all these systems interrelated. The incoming order will automatically communicate to the manufacturing database to schedule the products, the personnel affected, and the finances. The ability to handle complex cross-database structures such as composite objects and to work across multiple servers, multiple databases, multiple schemas, and multiple interfaces will drive development of the ODBMS, which is architecturally the ideal basis for such systems. As consumer demands grow and technology evolves, applications will support more complex information, more relationships and dependencies, and more complex operations. All this means that as time goes on, more applications will move from the domain of traditional databases to that of ODBMSs. ▮

Andrew Wade *is vice president of Objectivity Inc.; e-mail drew@objy.com.*

OODBSs AND THE MARRIAGE OF NETWORK AND RELATIONAL DATABASE MODELS

Dave Morse, ***Raima Corporation***

Twenty years ago, the new gospel of databases hailed the superiority of the emerging relational database model over the older network (Codasyl) model. Relational database management systems (RDBMSs) allowed "database creation on the fly," or nearly instant changes in a database's schema to quickly accommodate changes in a corporation's environment. RDBMSs made it easier to create and modify records by storing them on separate tables and linking them through indexes. In contrast, records in the network model were joined through direct and much less mutable relationships.

Network model rigidity was identified as a major problem that RDBMSs solved. Still, the network database never died. While relational databases today garner the most attention and have fueled the rise of numerous Silicon Valley start-ups, some 90 percent of the world's data exists in network and hierarchical model databases, according to the Gartner Group. IMS, IBM's network model database system, is still the largest revenue contributor to IBM's Software Solutions Division.

In fact, the network model seems to be experiencing a renaissance as a new technology called the object-oriented database (OODB) gains attention. Some of the network model's key distinguishing virtues seem uncannily embodied in the emerging OODBs. A case in point is the pointer-based method for establishing relationships between data in both the network and OODB models. First, such direct joins result in faster access time and typically better performance. Second, pointer-based OODBs support navigational relationship mechanisms by creating predefined paths and unique IDs for each object. Third, pointer-based systems support complex (for example, recursive) and unconventional data relationships more easily than relational databases do.

However, OODBs are more than just a return to the network model. They provide significant advantages by being intuitive to use and by hiding nonessential information from the user. The desire to model complex real-world relationships through complex database designs has brought the relational model into question. Object-oriented partisans claim relational databases are ill-suited for storing and manipulating today's complex data. Such complex structures don't map well onto the tables that form the infrastructure of relational databases; neither do the images, and the audio, video, and other multimedia data that are expected to become common in corporate repositories. Complexity in a relational database typically requires creation of more and more indexes, and proliferation of indexes can quickly cause a program to slow unacceptably.

In contrast, OODB systems start out with the assumption that data structures are to be viewed as objects that encapsulate not only data but also information about the behavior of this data. The growth of this new type of database has spawned an ever-growing set of classifications for OODBs. Some are OODBs with a highly graphical interface and little underlying similarity to the common relational database. However, a new category has emerged under the rubric "object-relational database" and appears to be gaining acceptance among the many database programmers who don't

want to give up all the benefits of RDBMSs or who prefer to migrate to OODBs more slowly. Most of the products in this new category reflect relational database vendors' efforts to retain some of the structure of relational databases while offering OODB functions. These object/relational databases range from pure OODB management systems that map objects to relational database tables, to relational wrapper libraries that make an RDB table appear as an object.

The course followed by software engineers at Raima illustrates how one vendor has chosen to marry relational and network database models to realize the benefits of object-oriented database development. This blending of models was accomplished at the level of the Data Definition Language (DDL) through an extension of the ANSI SQL standard, which in our Velocis database server includes a CREATE JOIN statement. This creates a predefined join between multiple tables. Application developers can then follow this pointer-based relationship in accessing persistent objects (stored data). Thus, a variety of objects can be stored in a network model database structure and be accessed without indexes. Yet indexes can also be programmed into the same databases in the instances when the relational model is more appropriate. These indexes are highly useful for random lookups from tables that have many records.

It is possible to create an object-oriented wrapper or set of class libraries that encapsulate storage and database navigation into C++ class definitions and to place it on top of a relational database structure. Yet because they sit on top of relational database structures, they can fall down when it comes to delivering speed and low consumption of computing resources, because queries still must navigate through multiple indexes that place their own demands on memory and storage. Ideally, developers could place object-oriented wrappers on top of a pointer-based structure, delivering the speed and leanness that are often missing when manipulating complex data under the relational model.

As a result of the growing complexity and sheer volume of corporate data, the demand for object-oriented databases with incorporated network model features is certain to grow. Yet it is clear that the corporate mainstream still views OODB warily. Proponents of relational and network model databases, after all, can point to two decades of refining and building safeguards into their systems, while OODB products are quite new. Moving object-oriented systems into the corporate database mainstream will take a concerted educational effort, which will include identifying the network model precepts underlying many OODBs.

It will also require adoption by OODB vendors of those standards that built managers' trust in relational databases. Among the most important are Jim Gray's ACID principles, stating that a database transaction must be atomic, consistent, isolated, and durable. Some object-oriented databases don't yet support them and hence appear more likely to lose or corrupt data. This attribute is critical in gaining support for OODB technology in applications involving on-line transaction processing, in which the nature of the transactions leaves no room for error. ■

Dave Morse *is director of marketing for Raima Corp.; e-mail dmorse@raima.com.*

USING OBJECT TECHNOLOGY FOR DATABASE SYSTEM INTEGRATION

Ahmed Elmagarmid and Evaggelia Pitoura
Purdue University

Today, worldwide high-speed networks connect numerous information systems and provide access to a wide variety of data resources. Since these systems were not developed with future integration in mind, there is an increasing need for technology to support seamless access and integration of the provided services and resources. Integrated systems that include database systems—that is, multidatabase systems—pose special requirements, and many researchers have suggested using object-oriented techniques to facilitate building multidatabase systems. (See Table 1 for a list of research projects on object-oriented multidatabases.)

The object-oriented paradigm has influenced multidatabase system design and implementation in three dimensions: system architecture, schema architecture, and transaction management.

System architecture. Applying object-technology to system architectures has resulted in distributed object-based architectures. In this scenario, component system resources are modeled as objects and their services as methods, which constitute the object interfaces. To interact with the heterogeneous system, a client issues requests in a common object-oriented language. Distributed object managers (DOMs) translate these requests in terms of the available services, direct them to the appropriate systems, and respond in the common language. Modeling distributed resources as objects supports heterogeneity because the messages sent to a component depend only on its interface, not on its internal implementation. This approach also respects system autonomy; component systems can operate independently and transparently, provided their interfaces remain unchanged.

Schema architecture. Objects have also influenced schema architecture. In a multidatabase, each component system's data model is translated into a common or canonical data model (CDM) for all participating systems. Then, information in each component database is integrated into a global, unified schema. Different types of multidatabase systems are created by different levels of integration. CDM objects can be of a different granularity than the distributed objects. At one extreme, the whole database may be modeled as a single, distributed complex object.

Object models are semantically rich. They provide a variety of type and abstraction mechanisms and relations for expressing the complex interschema relationships and potential conflicts among data at each component system. Methods enable arbitrary combinations of information stored in component databases and integration of nontraditional databases through behavioral mapping. Finally, the metaclass mechanism—that is, the mechnism for defining classes—adds flexibility by allowing arbitrary refinements to the model itself.

Transaction management. Object technology has influenced heterogeneous transaction management in various ways. One application uses object-oriented techniques to

Table 1. Multidatabase projects. (Complete systems support network communication and operating system facilities in addition to database services.)

System	Type	Integrated systems
Pegasus	Complete data management system	Information systems of various data models
ViewSystem	Tool (environment) that supports integration	Information bases
CIS/OIS	Integration tool	File systems, information retrieval systems, databanks
OMS	Framework	Engineering information systems
DOMS	Complete system	Database systems, hypermedia applications, conventional applications, etc.
UniSQL/M	Multidatabase system	SQL-based relational databases and the UniSQL/X database system
Carnot	Complete system	Database systems, knowledge-base systems, and process models
InterBase	Complete system	Database systems and Unix utilities
FIB	Multidatabase system	Database systems

implement the extended transaction models and workflow systems needed to express the complex structure of multidatabase tasks. It models transactions as objects and their interactions as the methods of these objects. Flat transactions correspond to simple objects and extended transactions to complex objects.

Multidatabase systems with an object-oriented common data model can use semantic information to produce more efficient transaction management. Since each database object comes with a specific set of methods, semantic serializability can be employed as the correctness criterion, type-specific operations can be used, and compensating actions can be defined per method.

Finally, there is a conceptual similarity in transaction management research in multidatabase and object-oriented database systems. Both deal with layered databases, where each object (component database) is a single database responsible for its own consistency and transactions span more than one database.

THE ABOVE DIMENSIONS ARE ORTHOGONAL in the sense that systems may support object-orientation on one dimension but not necessarily on others. For example, a database system with a relational CDM can participate in a distributed object architecture by being considered a single distributed object, and database systems with object-oriented CDMs can participate in nonobject-based system architectures. Moreover, sytems that do not support objects at the other two dimensions can use object-oriented techniques in implementing their transaction management schemes. However, a fully object-oriented multidatabase should support the same object model at all dimensions to avoid design and implementation problems. Thus, the minimum set of features the model must support needs to be defined.

Because requirements differ for each dimension, the resulting data models emphasize different features. At the system architecture level, most proposed models are programming-based and focus on such issues as efficient implementation of remote procedure calls and naming schemas. At the schema architecture level, most proposed models are database-oriented, support persistency and database functionality, and have extended view-definition facilities. Issues include efficient query processing, classification and systematic treatment of conflicts, and interschema relationships and automation of integration. Finally, at the transaction management level, most proposed models support active objects appropriate for modeling transactions and their interactions. Issues include customized transaction processing, efficient implementations, and increased concurrency. ▮

Ahmed K. Elmagarmid *is a professor of computer science at Purdue University; e-mail ake@cs.purdue.edu.*

Evaggelia Pitoura *received a PhD degree in computer science at Purdue in 1995.*

Software Quality and Object Orientation

TRENDS IN TESTING OBJECT-ORIENTED SOFTWARE

Robert V. Binder, ***RBSC Corp.***

The general view of testing in object-oriented development has undergone a curious and rapid change in the last three years. Testing has risen from obscure curiosity to pat cliché, but for the most part remains poorly understood and insufficiently practiced. Before 1992, researchers, methodologists, and programming gurus had presented or published only a few testing strategies. Practitioners tested, but usually not very much or very well: They muddled through with techniques adapted from conventional testing or just punted and hoped for the best. A few object-oriented bigots asserted that testing was altogether unnecessary and antithetical to the "paradigm" ("we don't test, we iterate"). But just like nature, the universe of software concepts abhors

a vacuum. In the past three years, the rate of publication about object-oriented testing increased dramatically. The rate has dropped this year, possibly because the easy problems have been solved and reported.

This flurry of activity has changed attitudes. Three years ago, when testing was discussed, one was often admonished to test "thoroughly." However, there was no operational definition for this catch phrase and precious little guidance offered about how to accomplish thorough testing. Today, there are many technical approaches and process strategies, and a few useful test tools. Most recent books on object-oriented development provide at least a cursory discussion of testing. Those published a few years ago were either silent about it or assured us, in a sentence or two, that the magic of objectification would greatly ease and simplify testing, if not completely obviate its need.

This change is encouraging, since "thorough" testing of object-oriented systems presents nontrivial challenges. There are at least three unique problems. First, objects typically exhibit sequentially dependent behavior. An object's response is determined by the sequence of messages it has previously accepted. Thus, objects can be specified, implemented, and tested as state machines. Testing class methods in isolation (as one would do with a conventional routine) is not sufficient or effective. Most systematic approaches to object-oriented testing are state-based. There are two main flavors of state-based testing. Sequential state-based testing seeks fault-revealing message sequences. States define correct and incorrect behavior. This approach is typically specification-based. The domain approach to state-based testing posits states from values that may be assumed by instance variables of the class under test. The interaction of these states and messages is used to derive test cases.

The good news about the sequential flavor of state-based testing is that it has been studied and applied for 40 years to circuits, hardware components, and telecommunication protocols. Much of this knowledge can be transferred to object-oriented software testing. The bad news is that state-based testing can be difficult. State spaces can be very large, meaning that automation and state-folding are required. Addition of built-in capabilities to set and report state in the implementation under test is a practical necessity.

Second, object-oriented systems are inherently less testable than conventional systems. Although object-oriented languages remove some chronic fault sources (weak encapsulation, global data hazards, type mismatches, and so on), they present new problems in understandability owing to delocalization and fragmentation of plans. A static set of variables determines the state space of programs written in conventional languages. The runtime behavior of conventional code is intellectually tractable when it is well structured. In contrast, inheritance and polymorphism in object-oriented languages often result in paths and states that are not obvious from source code or its specification. The frequently made suggestion that encapsulation eases this difficulty is whistling past the graveyard. Test case setup can be very difficult. Objects are typically composed of other objects, so setting the state of an object under test may require setting the state of all the objects nested within. Design for testability and routine use of built-in test can mitigate these problems.

Third, as a practical matter, iterative and incremental development coupled with hurry-up management often results in class libraries that lack functional orthogonality and have no documentation save for source code. This is not a new problem, though it is worsened by features of object-oriented languages that tend toward strong coupling among components. This leads to increased risk of faults, reduced testability, and low reusability. These effects are compounded as libraries get larger and older.

These issues conflict with two fundamental tenets of testing: There must be (1) a specification to test against, and (2) some automatic measure of the extent to which the system under test has been exercised ("test coverage").

Maintaining currency between a specification and an implementation has always been a practical problem for functional (black-box) testing. With the rapid, iterative development practices often used for object-oriented development, this synchronization is more difficult. Thus, the software process typically associated with object-oriented development can be a significant obstacle to repeatable, nonsubjective testing.

Although test coverage metrics are imperfect quality indicators, they provide a rough measure of the extent to which a test suite has exercised an implementation. Automatic identification and instrumentation of control paths in statically composed systems is routinely performed by commercial off-the-shelf test tools. However, the number of paths in dynamically composed object-oriented systems is typically much larger, if not practically infinite. It is not yet clear which computable coverage metrics will provide an unambiguous measure of test coverage or a useful indication of field reliability.

Where time-to-market is a dominant business concern, testing is sometimes perceived as a bottleneck, especially by those who see object-oriented technology as a silver bullet. However, it is not testing per se, but the ineffective integration of testing with the software process that produces a bottleneck. The need to test will not wither away in object-oriented systems. Organizations that must produce highly reliable software systems must still test. The present-day tolerance for showstoppers in desktop applications will be short-lived in that burgeoning market. In the face of intense competitive pressure, a comprehensive and rational strategy to achieve high testability will be a strategic advantage—not a bottleneck. Since technical factors loom large in testing object-oriented systems, an effective strategy must address both technical and procedural concerns.

The state of the art and the practice of testing object-oriented software have improved significantly in recent years. For those of us working on object-oriented testing, it is an exciting time. Effective, repeatable, highly automated testing will increase reusability and help to realize the promise of object technology. There are several primary problems to be solved: definition of widely accepted patterns for built-in test, practical process strategies for high testability, and development of high-leverage test automation. In the near term, those working on a test process for object-oriented development can look forward to advances in test automation but will need to live with the limitations of an emerging engineering discipline. ■

Robert V. Binder *is president of RBSC Corp.; e-mail rbinder@mcs.com.*

Theoretical Foundations

INTERACTIVE FOUNDATIONS OF OBJECT-BASED PROGRAMMING

Peter Wegner, *Brown University*

Though object-based programming has become a dominant practical technology, its conceptual framework and formal theoretical foundation are as confused as ever. This is reflected both in arguments over languages for first courses in computing[1] and in the proliferation of nonformal software design and life-cycle methodologies. The confusion is due in part to the fact that the observable behavior of objects cannot be modeled by algorithms, Turing machines, or traditional mathematics. The irreducibility of object behavior to that of algorithms has radical consequences for both the theory and practice of computing. In particular, Turing machines lose their status as the most powerful computing mechanism, since objects and interactive software systems have observably richer behavior.[2]

Programming paradigms have evolved from machine language in the 1950s to the procedure-oriented paradigm in the 1960s, structured programming in the 1970s, and the object-based paradigm in the 1980s. In the 1990s, methods for structuring *collections* of objects are being developed, including frameworks, design patterns, and protocols.

The transition from machine to procedure-oriented programming involves a quantitative change in the granularity of actions while retaining an algorithmic (action-based) programming model. The transition from procedure-oriented to object-based programming is a more radical qualitative change from programs as algorithms that transform data to programs as systems of persistent, interacting objects. The contract of an algorithm with its clients is like a sales contract that guarantees a value for every input, while the contract of an object with its clients is like a marriage contract that constrains behavior over time for all possible contingencies of interaction ("for richer for poorer . . . till death do us part").[3]

Objects differ fundamentally from procedures in their semantics, composition mechanisms, and structuring mechanisms. Whereas the composition of two procedures to yield a composite procedure can be simply modeled by function composition, the composition of two objects cannot be directly modeled as an object because interobject structuring by message protocols or broadcasting is entirely different from internal object structuring. Object-based behavior, composition, and structure are not expressible by (reducible to) procedure-oriented modeling primitives.[2]

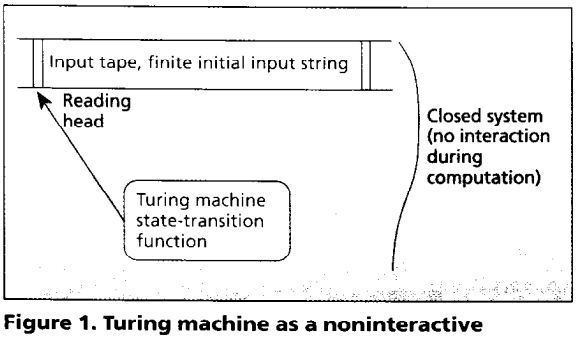

Figure 1. Turing machine as a noninteractive computing mechanism.

Procedures are mathematically modeled by functions that transform inputs to outputs. The computable functions are a very robust class of transformations that express the behavior of Turing machines, the lambda calculus, and algorithms of any programming language. Church and Turing conjectured in the 1930s that this robust notion of computing corresponds to the intuitive notion of what is computable. While it is natural for mathematicians like Church and Turing to hypothesize that the intuitive notion of computing corresponds to the entirely mathematical notion of computable functions, this mathematical notion of computing fails to express the temporal behavior of objects and distributed systems. Since object and distributed-system behavior must be modeled by any adequate intuitive notion of computing, the Church-Turing model is too restrictive to capture the intuitive notion of computing.

Functions compute their output from an initially specified input without any external interaction, while objects permit interaction during computation. Turing machines likewise compute output from an initial input tape and cannot interact with an external environment during computation. Object-based systems may be distinguished from procedure-oriented systems by their ability to interact while they compute. Objects are open interactive systems, while functions and Turing machines are closed algorithmic systems (see Figure 1).

Turing machines can be extended to be interactive by adding input actions supporting external inputs during computation. This simple extension transforms Turing machines from closed to open systems, extending their expressive richness to that of objects. We call Turing machines with input actions *interaction machines*. Interaction machines better approximate the behavior of actual computers than Turing machines: They can model the passage of time during the course of a computation, while Turing machines cannot. Interaction machines can model the behavior of airline reservation and banking systems that interact with clients in real time, and of embedded, reactive, and hard real-time systems. They are a more accurate model of actual computers than Turing machines because actual computers persist in time and can interact with the external environment during the course of a computation.

Interaction machines generally have output as well as input actions: Input actions are sufficient for behavior richer than that of Turing machines, while output actions allow two-way interaction with the environment. Input and output actions can be viewed as "logical" sensors and effectors that have a logical effect on data in their environment even when they have no physical effect. Interaction machines can directly realize embedded and reactive systems of software engineering and embodied systems of artificial intelligence. Conversely, embedded,

reactive, and embodied systems are interactive and not in general expressible by Turing machines.

Formally, interaction machines are more powerful than Turing machines because they capture the behavior of Turing machines with oracles, which Turing himself showed to be more powerful than Turing machines.[4] Moreover, interaction machines cannot be modeled by any sound, complete first-order logic.[3] Godel's incompleteness result for integers holds also for interaction machines. These formal irreducibility results for interaction machines complement the informally demonstrable fact that interaction machines are richer than Turing machines in modeling interaction and the passage of time.

The incompleteness of interaction machines implies that object-oriented, software engineering, and distributed systems do not have a "complete" formal specification. Complete specification of system behavior is not merely hard but impossible for object-based and distributed systems. However, formal specifications can still play an important role because interfaces, views, and modes of use of interactive systems can be formally specified. Though a complete elephant cannot be specified, we can completely specify some of its parts and forms of behavior (its trunk or its mode of eating peanuts). The change of emphasis from complete to partial specification is evident in software engineering models such as use-case analysis.[5]

The use of partial interface descriptions in specifying interactive systems is illustrated by airline reservation systems, which have multiple distributed interfaces of the following kinds:

- Travel agents—making reservations on behalf of clients,
- Passengers—making direct reservations,
- Airline desk employees—making inquiries on behalf of clients,
- Flight attendants—aiding passengers during the flight itself,
- Accountants—auditing and checking financial transactions, and
- Systems builders—developing and modifying the system.

Each of these modes of use may have multiple instances. The system as a whole has many distributed interfaces, each with an easily specifiable mode of normal use that may break down under abnormal conditions like strikes or system failure. Airline reservation system requirements may be specified by the set of all interfaces (modes of use) it should support. A system satisfies the requirements if it supports these modes of use even though the complete behavior of the system for all possible modes of use is unspecifiable.

Though object-oriented systems cannot be completely formalized, their modes of use, views, and interfaces can sometimes, though not always, be formalized. We call such partial descriptions *harnesses*, since they serve both to constrain system behavior (like the harness of a horse) and to harness system behavior for useful purposes. Object-oriented analysis and design must replace the goal of complete formalization by the more modest goal of ensuring that systems have harnesses corresponding to desired forms of useful behavior. Requirements of object-oriented systems may be given as a collection of harnesses that determine tractable constraints on an inherently intractable interactive system (see Figure 2).

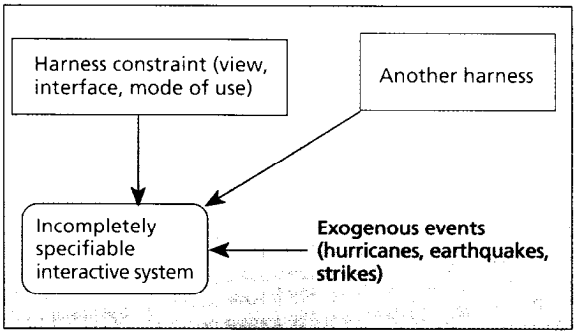

Figure 2. Harness constraints for interaction machines.

Harness constraints are useful in describing theoretical models as well as practical software engineering systems. Turing machines can be thought of as interaction machines whose tape acts as a harness on the interactive state-transition mechanism. Finite automata have the same state-transition mechanism as Turing machines but a more restrictive harness. The Chomsky hierarchy of finite, pushdown, linear-bounded, and Turing machines all have the same state-transition mechanism but progressively more permissive harnesses that allow more expressive subsets of the computable functions to be computed. However, if the tape is viewed as a harness, then Turing machine tapes are clearly not the most permissive possible harnesses, since they impose the strong restriction of no external interaction during the computation.

The idea of allowing harness constraints to be weaker than those of Turing machines is quite natural in this context. Turing machines impose a system-defined harness constraint that excludes important forms of legitimate computational behavior in the interests of mathematical tractability. They are an unnecessarily strong abstraction of the behavior of actual computers, while interaction machines provide a weaker abstraction that allows a larger class of computational phenomena to be modeled (see Figure 3).

Turing machines shut out the external world during computation, while interaction machines model processes

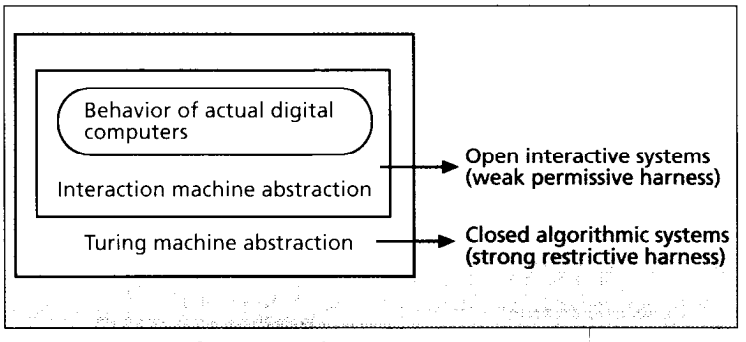

Figure 3. Layers of increasingly restrictive abstraction.

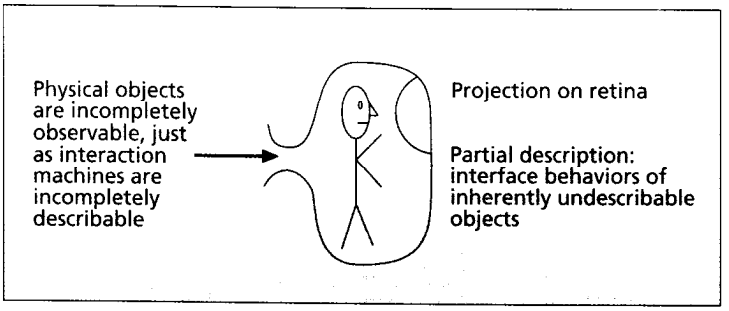

Figure 4. Plato's Cave as a paradigm for describing incomplete behavior.

of observation and express the intuitive notion of empiricism. Turing machines are mathematical models, while interaction machines are empirical models that can express behavior of an empirical world, like models of physics and the natural sciences. Interaction machines precisely define the term "empirical computer science": The irreducibility of interaction to Turing machines reflects the irreducibility of empirical computer science (software engineering, distributed systems) to theoretical computer science. Turing machines lose their status as the most powerful computing mechanism and become instead the most powerful model that can be completely formalized.

Modeling by partial description of interface behaviors is normal in the physical sciences. The incompleteness of physical models is forcefully described by Plato in his parable of the cave, which indicates that humans are like dwellers in a cave who can observe only the shadows of reality on the walls of their cave but not the actual objects in the outside world (see Figure 4).

Plato's pessimistic picture of empirical observation caused him to deny the validity of physical models and was largely responsible for the eclipse of empiricism for 2,000 years. Modern empirical science is based on the realization that partial descriptions (shadows) are sufficient for controlling, predicting, and understanding the objects that shadows represent. The representation of physical phenomena by differential equations allows us to control, predict, and even understand the phenomena represented without requiring a more complete description of the phenomena. Similarly, computing systems can be specified and controlled by interfaces that describe their desired behavior without completely taking into account all possible behavior.

Turing machine models of computers correspond to Platonic ideals in focusing on mathematical tractability at the expense of modeling accuracy. To realize logical completeness, they sacrifice the ability to model external interaction and real time. The extension from Turing to interaction machines, and of procedure-oriented to object-based programming, is the computational analog of the liberation of the natural sciences from the Platonic worldview, which led to the development of empirical science.

We can distinguish three paradigms of programming:

- *Declarative paradigm*—functional and logic programming,
- *Imperative paradigm*—procedure-oriented programming, and
- *Interactive paradigm*—object-based and distributed programming.

Until now the debate has been mainly concerned with declarative versus imperative paradigms. However, declarative and imperative paradigms are similar when compared with the interactive paradigm. Declarative and imperative paradigms have the same expressive power, whereas the interactive paradigm has greater expressive power. The interactive paradigm can model software engineering applications, agents in AI, and distributed systems, while declarative and imperative paradigms are too weak to model such applications. Object-based programming is not simply an alternative programming style: It is a fundamental extension with greater expressive power, different conceptual foundations, and new modeling techniques. The transition from algorithmic to interactive models is occurring in subareas of computer science like software engineering, computer graphics, artificial intelligence, and distributed systems. Theoretical as well as practical models of computing must adapt to reflect the fact that our intuitive notion of computing is becoming less algorithmic and increasingly interactive and object-based. ■

References

1. H. Abelson et al., "The First Course Conundrum," *Comm. ACM*, Vol. 38, No. 6., June 1995, p. 116.
2. P. Wegner, "Interaction as a Basis for Empirical Computer Science," *Computing Surveys*, Vol. 27, No. 1, Mar. 1995, pp. 45-48.
3. P. Wegner, "Tutorial Notes: Models and Paradigms of Interaction," Tech. Report CS95-21, Computer Science Dept., Brown University, Providence, R.I., Sept. 1995.
4. A. Turing, "Systems of Logic Based on Ordinals," *Proc. London Math Soc.*, 1939.
5. I. Jacobson, *Object-Oriented Software Engineering*, Addison-Wesley/ACM Press, New York, 1991.

Peter Wegner *is a professor of computer science at Brown University; e-mail pw@cs.brown.edu.*