The Synergy Between Object-Oriented Programming and Open Systems Interconnection

R. Greg Lavender
Dennis Kafura

TR 91-31

Department of Computer Science
Virginia Polytechnic Institute and State University
Blacksburg, Virginia 24061

November 22, 1991

The Synergy Between Object-Oriented Programming and Open Systems Interconnection

R. GREG LAVENDER

MCC
3500 W. Balcones Center Dr.
Austin, TX 78759-6509
Tel: 512.338.3252
Fax: 512.338.3600
lavender@mcc.com

DENNIS G. KAFURA

Dept. of Computer Science
562 McBryde Hall, Virginia Tech
Blacksbug, VA 24061-0106
Tel: 703.231.5568
Fax: 703.231.6075
kafura@cs.vt.edu

Abstract

The software engineering practice of building distributed object-oriented applications can be significantly improved by exploiting an observed synergy between object-oriented programming (OOP) and Open Systems Interconnection (OSI). The synergy arises because there are corresponding and complementary elements in each of OOP and OSI. These elements are detailed and the synergy resulting from their integration is explained. The architecture of a prototype implementation, the goal of Project Synergy, is described. The environment created by Project Synergy supports application development using abstract classes defined in an implementation-independent manner, implemented in possibly different programming languages, and executed in a distributed environment on possibly heterogeneous processor architectures.

Area of Research (both authors): object-oriented programming, distributed computing, communication protocols, concurrency, persistence, actor model.

1. Introduction

A powerful synergy exists between the object-oriented programming model and the layered communications model for Open Systems Interconnection. The purpose of this paper is to explain how this synergy can be used to simultaneously:

- empower the object-oriented paradigm by infusing it with the distributed programming capabilities inherent in open systems communication, and
- harness the power of an open system by organizing its services within an object-oriented language framework.

In the context of object-oriented software engineering practice, OSI can be considered both an existing technology and a new technology. Work on OSI standardization and implementation is over a decade old; considerable experimentation and implementation has been done at the *lower layers* (transport and below). In this sense, OSI is an existing technology. However, only

recently have full protocol stack implementations been readily available. Moreover, few applications offering advanced services (other than mail, directory, remote terminal, and file transfer services) have been defined and implemented. In this sense, OSI is a new technology — one ready to be fully exploited. In the words of Marshall Rose [Rose 90], "Work began on OSI in 1978....the market is still waiting for OSI's promise." The position taken in this paper is that the promise of OSI can best, and perhaps only, be realized by merging object-oriented programming concepts with the *upper layers* defined by OSI. The upper layer structure (abstract syntax notation, application service elements, presentation and session services) offers a rich set of communication services. The potential result is an environment within which sound engineering of distributed object-oriented applications can be undertaken.

The integration of object-oriented structures and OSI services also performs an important, two-way technology transfer function. Technology transfer usually means the transfer, within a single domain, of knowledge, tools, or techniques from laboratory research to practical application. However, in this paper, the term technology transfer is also used to mean the transfer of knowledge, tools, and techniques between different domain; in this case, between the programming languages domain and the communications domain. The term technology transfer is appropriate in either case because the desired goal is the same — improving software engineering practice. The technology transfer process associated with the integration of OOP and OSI is two-way because the software engineering practice of both domains are improved. This symmetric improvement is another reflection of the synergy between OOP and OSI.

2. Background

The synergy between object-oriented programming and open systems interconnection arises from complementary elements in each. In this section, the corresponding elements are identified and their complementary nature is explained. Figure 1 depicts the corresponding elements. The elements from object-oriented programming are *persistence*, *communication*, and *concurrency*; while the elements from OSI are *layered services*, *structured communication*, and *remote operations*. The integration of these elements results in a computational (application development) paradigm based on persistent, communicating, active objects. The diagram in Figure 1 suggests that when corresponding two-dimensional entities are properly related the resulting three-dimensional entity possesses qualities inherently different, in a positive sense, than the original entities. The remainder of this paper provides justification that such a dimensional change does occur when the elements depicted in Figure 1 are combined appropriately.

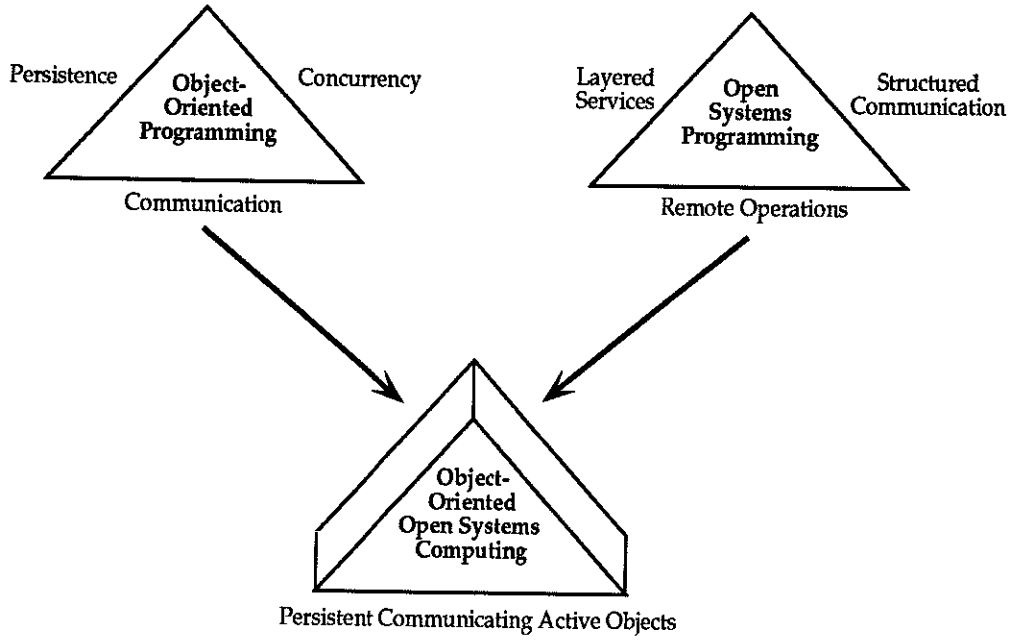

Figure 1. Elements of the Synergy

The three elements of object-oriented programming contributing to the synergy are not accidental, they arise naturally from the requirements of contemporary and envisioned software systems. Wegner [Wegner 90] uses the term "mega-programming" to refer to systems which exhibit distribution, concurrency, persistence, and heterogeneity. The National Collaboratory is cited as an example of a system imposing these requirements. Similar requirements are given by Zdonik and Maier [Zdonik 90] for systems developed by data intensive programming in-the-large. Such systems, exemplified by CAD tools, are large and complex in both function and data. Concurrent access by independent, distributed users to long-lived entities is typical for such applications. Finally, Hewitt describes the needs of an Open Information System (OIS) — a system which is open-ended, incremental and evolutionary [Hewitt 84]. An OIS is exemplified by an enterprise-wide information system of the future [Hewitt 90]. A prototype OIS [deJong 91] relies on concurrent, distributed objects. The three elements from OSI are necessary for the exchange of structured data between distributed computing agents. Layered services are necessary in order to maintain a separation of concerns. Structured communication channels are necessary for interactions in which the abstraction of a simple byte stream channel is insufficient. Remote operations enforce a common semantics on the interactions between agents.

The combination of the elements from the object-oriented programming domain and those from the open systems domain offer the potential for building software systems with require-

ments congruent to those just enumerated. The characteristics of these elements are elaborated on in the following sections.

2.1 Elements from Object-Oriented Programming

From a programming language perspective, the object-oriented paradigm may be viewed as a union of the following paradigms [Wegner 90]:

- a paradigm of program structure,
- a paradigm of state structure, and
- a paradigm of computation.

Only the later two paradigms are relevant to this paper. Objects, the data they encapsulate, the mechanisms for controlling and synchronizing access to the data, and the life-time of an object are all elements of an object-oriented paradigm of state structure. The ability of objects to execute independently and to interact via communications are the essential elements of an object-oriented paradigm of computation.The essential elements derived from this composite view of the object-oriented paradigm are: persistence, communication, and concurrency.

Persistence

An object is persistent if it outlives its execution environment. The only part of an object that must minimally persist is the encapsulated state, represented by the set of typed instance variables. However, since objects interact by communicating it is also advantageous to make the state of the object interface persistent. For example, if objects communicate via message passing, then the interface message queue should also be persistent. If an object communicates with a remote object using a communication protocol, then the state of the protocol machine should be part of the persistent state. Furthermore, if an object is a concurrent object requiring synchronization control, then the state of the synchronization mechanism should be considered part of the persistent state. Thus, an object which is potentially persistent, which communicates, and which is concurrent, can be viewed as having at least three state components: an *encapsulated* state, a *communication* state, and a *synchronization* state.

From the perspective of a method, only the encapsulated state is part of the "real" object since the encapsulated state and a run-time stack frame provide the total environment required by a method. Both the communication state and the synchronization state are "meta" states in the sense that they exist as part of the meta environment of the object. Thus, any approach to persistence in which objects possess a communication state and a synchronization state must consider the meta environment of the object.

Communication

Computation in an object-oriented system is realized by inter-communication amongst objects. The interface of each object defines a protocol for client interaction with the object. By separating the communication interface of an object from the method implementations, a uniform view of object interaction is offered regardless of whether the object is local or remote to a client attempting to communicate. For example, remote objects may be represented locally by an interface which specifies their interaction protocol. To a local object, interaction with a remote object requires no additional effort and therefore leads to conceptual economy when programming a system of interacting objects, some of which may be remote.

Concurrency

The central notion in concurrent object models is the interaction of independent objects [Hoare 88], [Milner 89], [Agha 86]. Synchronization is necessary when concurrent access to a shared state must be serialized in order to ensure a consistent state. New language mechanisms have been proposed for synchronizing concurrent interaction of objects. The reason new synchronization mechanisms are required is that the inheritance mechanism in most object-oriented languages interferes with the synchronization mechanism [America 87], [Briot 90], [Kafura 89], [Matsuoka 90], [Tomlinson 89]. Recent promising steps towards understanding the inheritance of concurrent object behavior have been suggested using the formalisms of CCS as a starting point [Lavender 90], [Nierstrasz 90]. Since the object interface defines the observable behavior of an object, synchronization mechanisms for concurrent objects focus on controlling the visibility of methods in the interface. The synchronization mechanism "hides" a method if its execution will result in an inconsistent state.

2.1 Elements from Open Systems Interconnection

In the current section the following essential elements of open systems programming are presented: layered services, structured communication, and remote operations.

Layered Services

A common technique used in modeling a complex system is to hierarchically partition the services of the system into successively dependent layers. A layered partitioning of services provides a functional separation of concerns by defining a service at a layer N in terms of the services available at layer $N\text{-}1$. The seven-layer ISO Reference Model [ISO 7498] is commonly partitioned into upper and lower layers. The upper layers consist of all services above the transport service; the lower layers include transport through physical. From a conceptual point of view, the transport layer offers a logical partitioning point because the upper layers,

to a large degree, can be insulated from the particular type of transport service offered. The purpose of the transport service is to provide either a reliable unstructured byte-stream service or an unreliable datagram service. A specific transport protocol in conjunction with an appropriate network protocol is able to offer one of these services.

Remote Operations

An open systems application is a computational process in which a relatively significant portion of the computation is concerned with establishing and managing associations with other *application processes* residing on different nodes in a network. Internally, an application process is a collection of *application entities*, each representing some communication aspect of the application process. Remote communication is realized for each application entity through the use of one or more *application service elements* (ASEs). ASEs provide common abstractions needed by most application entities and a higher-level interface to the services offered by the presentation layer. Two common ASEs are the Association Control Service Element (ACSE) and the Remote Operations Service Element (ROSE) [ISO 9072-1]. The ACSE provides a service for binding and unbinding an association between two application entities. The ROSE provides the general request/reply invocation services required to implement various remote interaction semantics, including traditional RPC semantics. Structuring the ASEs as objects creates an application environment with simpler, more abstract services and one which is safer as the control of arguments and the proper use of defaults can be insured by the ASE designer.

Structured Communications

Next to basic connectivity issues, the most important aspect of distributed programming is the ability to impose the structure of application data onto the otherwise unstructured bit stream offered by the underlying communication service. Peer application entities request operations and transmit application data values as arguments and results. The interacting entities may reside on machines whose hardware architectures differ in their representation of common values, such as integers. Such heterogeneity necessitates techniques enabling the consistent invocation of remote operations and the correct interpretation of data values.

A common approach to structured communication is to provide a remote procedure call (RPC) protocol [Birrell 84] and a machine independent data representation language. Various RPC protocols and data representation languages exists; for example, Sun Microsystems provides a popular datagram based RPC [Sun 87] used in conjunction with the External Data Representation (XDR) [Sun 88] language for specifying C language data types.

Abstract Syntax Notation One (ASN.1) [ISO 8824] is defined as an ISO standard data representation language which is more powerful than XDR. ASN.1 is used in conjunction with the Remote Operations Service Element to provide a remote operations service for OSI-based applications.

There are two points of synergy between the object-oriented paradigm and structured communications. First, ASN.1 and the ROSE offer sufficient mechanisms for supporting remote communication among cooperating objects. In particular, ASN.1 has the representational power to express the strongly-typed method signatures found in a typical class definition. Second, the encapsulation properties of objects allow the translation mechanics implied by XDR or ASN.1 to be concealed from the user of the object.

3. Capturing and Exploiting the Synergy

Recently, we initiated the Synergy Project to develop a prototype system synthesizing object-oriented programming and the OSI upper layer protocols. As justification of the synergy between OOP and OSI, the major improvements brought to OSI and object-oriented programming by their integration, and their relation to RPC, are discussed first in this section. Subsequently, the *communication infrastructure* and the *application infrastructure* of Project Synergy are detailed.

3.1 What Does OSI Gain?

First, the structuring facilities of object-oriented programming can be used to hide the complexity of the OSI protocols from the application developer. By applying an object-oriented approach to protocol implementation, the richness of the OSI communication services become available through a clean and simple interface. Behind this interface can be hidden all the forbidding detail which currently complicates application development. Second, object-oriented programming can be used to control the complexity of the protocol implementation. We have observed, for instance, that the OSI protocol stack neatly divides into two primary inheritance hierarchies (described below). Greater structuring permits easier experimentation and creates opportunities for performance improvement (e.g., by multi-threading). Third, several experimental systems (e.g., [Dixon 89], [Leddy 89], [Campbell 87]) have demonstrated that inheritance is a useful mechanism for disseminating and specializing the layered services (in our case, OSI communication services) provided by an underlying system.

3.2 What Does OOP Gain?

First, objects become naturally distributed. The object-oriented paradigm assumes the dimensions of a distributed development paradigm. Second, OSI-based communication provides interaction among objects executing on heterogeneous processor architectures, employing fundamentally different data representations and implemented in different languages. Third, the tight coupling between OSI and object-oriented programming expands the notion of persistence to include persistent communication state as well as the state of an object's encapsulated application data. The key point is that substantial improvements cannot be achieved in isolation; only by integrating the object-oriented paradigm with OSI can they be effectively realized.

3.3 Relation to RPC

Remote procedure call mechanisms have two principal aims:

- preserving familiar programming structure within a distributed computing environment; and,
- providing transparent interoperability among heterogeneous architectures.

The advantage of OOP/OSI over RPC appears in three ways. First, both OOP/OSI and RPC preserve familiar programming structures: objects and procedures, respectively. However, an object is a more robust programming structure — that is, after all, the major contribution of object-oriented programming. Second, asynchronous communication semantics, often expressed in a message-passing (datagram) metaphor, are a more natural basis for remote operations than the synchronous procedure call semantics offered by RPC. RPC suggests only one (albeit, a useful one) model of interaction — a client/server model. Object-based communication, while permitting client/server interaction, also permits asynchronous peer-to-peer communication. Third, both RPC and OOP/OSI are supported by mechanisms for achieving transparent interoperability. In fact, there is little fundamental difference between the techniques each uses to achieve this goal.

The principal difference is that OSI is an international standard unifying all users; contending, incompatible RPC protocols create only isolated islands of users. Recent efforts to bridge different RPC protocols (heterogeneous RPC [Bershad 87]) is, we believe, fundamentally wrong. The international networking community has converged on the OSI standards. While debate may continue to rage over specific technical merits of OSI, there is no productive, long-term alternative to its use.

From an object-oriented perspective, application entities are objects which communicate with remote objects using a remote operations mechanism. From this perspective, the boundary surrounding an application entity object can be exploited to encapsulate:

- *complexity*: the existing interfaces to application services entail lengthy argument lists containing system structures which the user must retain and supply with later uses of the service. These system structures can be better represented as objects.
- *distribution*: an application process may be composed of both locally communicating objects and remotely communicating objects. The application developer need make no distinction between remote and local objects.
- *protocol*: each pair of peer application entities uses a separate, but not necessarily unique, protocol. The protocol used in interacting with a remote object may be completely hidden from the application developer.

3.4 Communication Infrastructure

In order for an application to exploit the combination of components depicted previously in Figure 1, a communications infrastructure must exist. The communications infrastructure used in Project Synergy is based on an object-oriented re-engineering of the ISO Development Environment (ISODE) [Rose91], a widely used C language implementation of the OSI upper layers. The C++ programming language is used for obvious reasons. The primary advantage of applying an object-oriented rationalization to the ISODE is that classes and class inheritance (both single and multiple) facilitate the following:

- reduced complexity of the service access point interfaces and internal protocol machine structure at each layer,
- encapsulation and physical concatenation of the state of upper layer protocol machines,
- higher performance through inter-layer function inlining, and
- enhanced support for concurrent operations and asynchronous communication.

At this stage in Project Synergy, the claim of a reduction in complexity is based on qualitative evidence obtained by comparison of the amount of information that the programmer must provide and maintain when requesting service from a service access point. Intuitively, an object-oriented rationalization offers a simpler paradigm of service interaction for the application developer.

An *N-service access point*, or N-sap, provides the access point to a set of functions representing the service at layer *N*. An *N-protocol machine*, or N-pm, provides the mapping between N-saps. Figure 2 depicts the set of N-saps and N-pms that arise when two peer application entities communicate. The similarity between the properties of N-saps/N-pms and the properties

of an object are striking. Furthermore, the hierarchical structure of the N-saps and N-pms depicted in Figure 2 strongly suggests a compositional approach to structuring the layers of the ISO Reference Model based on inheritance of function. A typical compositional approach is to view the protocol stack as a monolithic, *horizontally composed* structure in which N-pms contain the machinery required to multiplex associations among N-saps. An alternative is to view the protocol stack as a *vertically composed* structure of independent instances of related sets of N-saps and N-pms. An obvious conclusion is that each N-sap and N-pm can be represented by a pair of classes implementing the *N*-service. Furthermore, class inheritance can be used to compose each *N*-layer in a manner enforcing the strict encapsulation required by each service layer.

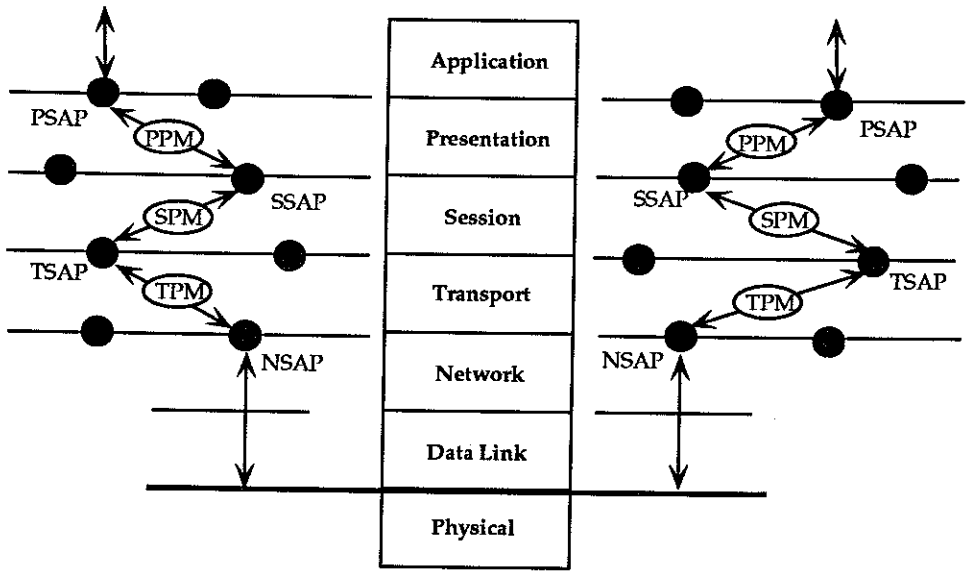

Figure 2. Service Access Point and Protocol Machine Structure

Figure 3 illustrates the inheritance hierarchy derived from a vertical partitioning of the upper layer N-saps and N-pms depicted in Figure 2. The inheritance hierarchy is an inversion of the upper layers. The inversion occurs because the inheritance relation is a reuse relation rather than a specialization relation. The concept of a *virtual* N-sap class is introduced as a light-weight interface between protocol machines in the hierarchy. The idea is that a subclass of a virtual N-sap, such as a protocol machine, is a trusted service user and therefore need not be subject to the usual access point checking mechanisms. The top-most class in the hierarchy represents a virtual Tsap (transport service access point) which is the logical partitioning point of the upper and lower layers. The bottom classes represent the service access points of presentation service and are meant to be extended further by protocol machine and service

access point classes representing application service elements, such as the ACSE and the ROSE.

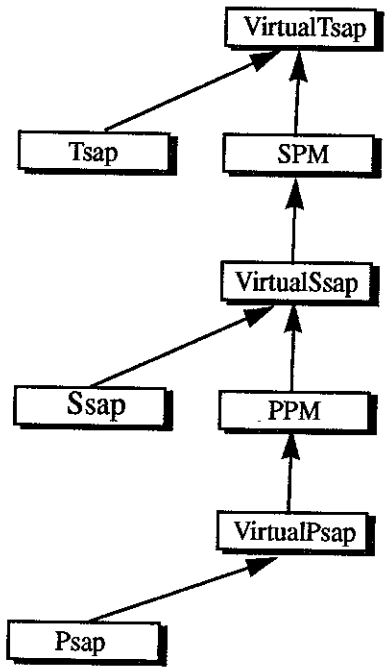

Figure 3. Upper Layer Inheritance Hierarchy

The advantage of a class inheritance structure of the upper layer protocols is that the entire state of an association, represented by a hierarchy of N-saps and N-pms, can be encapsulated and collapsed into a single composite run-time object. Encapsulation implies locality of state. Locality of state in conjunction with a synchronization protocol for establishing a communication synchronization point facilitates the controlled interruption of communication. Controlled interruption of communication enables techniques for making remote communication persist beyond the execution life-time of application entities.

The class inheritance mechanism of C++ provides access control which ensures that the encapsulation of the protocol layers is maintained. Up-calls and down-calls through the protocol layers are accomplished by super/sub-class method invocations within the composite object produced by the compiler. Heavy use of inline functions makes many of these "inter-layer" method invocations less expensive than standard function calls, resulting in better performance without compromising conceptual structure. In addition, the vertical partitioning of the protocol layers in this manner facilitates concurrent method invocation among separate instances of the protocol stack since the state of the protocol machines of an instantiated protocol stack is protected. Further implementation details of implementing the upper layer protocols using object-oriented techniques is the subject of a forthcoming paper [Lavender 91].

3.5 Application Infrastructure

The communication infrastructure just described has been the primary focus of Project Synergy since its inception. In this section, the preliminary concept for the applications infrastructure is described.

The developer of a distributed application, the *user*, will have available a universe of existing classes (types) which have been previously implemented and made available for reuse by *providers*. These classes will be defined in a programming language-independent fashion. The class definition notation we use is Abstract Syntax Notation One (ASN.1), although any other similar type definition language with equivalent expressive power could, in principle, be used. We chose ASN.1 primarily because it is part of the OSI standards and there exist freely available tools for processing ASN.1 specifications [Rose 91].

Retrieving classes for reuse from the universe of predefined classes is an interesting question, but one which is beyond the current scope of the Synergy Project. One could foresee, however, employing the Synergy environment to build a distributed system aiding in the identification and retrieval of needed classes. The retrieval system itself illustrates the kind of application we imagine Synergy would support — systems requiring access to persistent objects (the long-lasting ASN.1 descriptions of available classes) in a distributed and heterogeneous environment (classes may be offered by other systems/organizations without prior agreement) and which may be implemented in various languages (the classes offered by a system/organization will be implemented in the developing organization's language of choice which may differ from the user's language of choice).

The provider of a class fully implements the class in the provider's language of choice. A converter examines this implementation and produces the class specification expressed in ASN.1. This ASN.1 specification is made available (ideally through a distributed data-base) to potential users. The ASN.1 specification is input to a responder generator. This generator produces "boilerplate" methods needed to interface with the OSI-based run-time environment on the provider's side. The responder methods and the methods developed by the provider are bound together by the compiler/linker to produce an entry in a library. Once installed in the library the class is available for use. This is all that the supplier need be aware of.

A user examines the available class specifications and selects those which are appropriate for use. While the user may examine the specification in the ASN.1 syntax, in Figure 4 we show that a converter is used to generate a class definition in the user's language of choice. Selected class definitions are added to the application code under development and are also input to an initiator generator. This generator produces a boilerplate class whose methods interface to the

ISO-based run-time environment on the user's side. The full application and the output from the initiator generator are bound together by the compiler/linker..

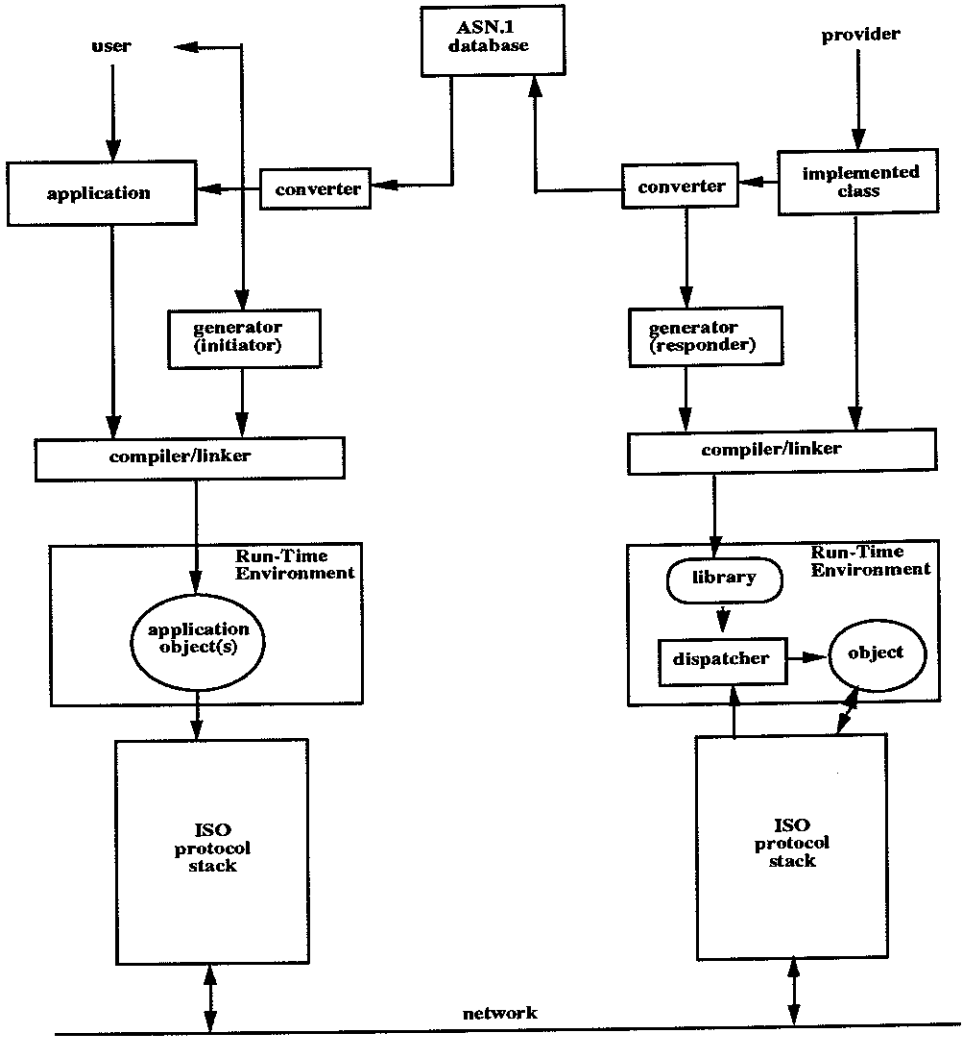

Figure 4. Application Infrastructure

We now consider the events which occur when the application instantiates an object of a class provided by another node in the network. The application instantiates a local object in the boilerplate class produced by the initiator generator. The constructor of this class uses the association control service element to establish an association with a dispatcher on the host supplying the implementation of the desired class. Using this association, the user's object requests the dispatcher to instantiate an object of the class, execute the object's constructor and bind the association to that object. The constructor of the user's object completes when the remote object has been completely created. Thereafter, invoking a method of the user's local object results transparently in the invocation of the remote object's methods using the

OSI remote operations facilities. Destructing the user's local object results in a termination protocol being followed to destruct the remote object and close the association

4. Related Research

A brief review of research work deemed relevant to the work described in this paper is presented. The intent is to distinguish the work previously described as sufficiently different from other recent work rather than provide an exhaustive survey of object-oriented distributed systems.

Distributed Operating Systems

Several experimental distributed operating systems have been implemented in which the objective is fast network-based interprocess communication and support for lightweight threads. Two deserving mention with regard to this paper are the Stanford V system [Cheriton 88] and the Amoeba system [Mullender 89].

V is interesting because it incorporates a fast and efficient transport protocol and heterogeneity via protocol-based system services. The VMTP (Versatile Message Transaction Protocol) [Cheriton 89] is a request-reply transport protocol supporting remote procedure call semantics. A lightweight application process communicates with other processes distributed among a cluster of workstations and servers using an RPC mechanism based on VMTP. An interesting departure V makes from typical distributed operating systems is the definition of system services in terms of protocols. By defining services in terms of protocols, a degree of heterogeneity is introduced. Rather than the usual operating system view of heterogeneity as "the same kernel on many platforms," different kernels may be used, exploiting particular hardware, but distributed kernel services interact through standard protocols.

Amoeba [Mullender 89] was designed as a homogeneous system and steps were taken to exploit specific hardware features. Unlike the V system, Amoeba follows the "same kernel on many platforms" approach to heterogeneity; gateways relying on traditional interconnection techniques, such as X.25, interconnect remote Amoeba kernels [Renesse 87]. At the application level, Amoeba does offer an object-oriented paradigm to programming distributed applications. However, Amoeba does not offer the same degree of heterogeneity provided by the V system or by standard communication protocols.

Distributed Object Systems

Distributed object systems are commonly implemented on top of an existing network-based or distributed operating system. For example, Emerald [Black 87] and Argus [Liskov 87] are object-based or abstract data type systems, while ESP [Leddy 89], SOS [Shapiro 89], and Arjuna [Dixon 89] are object-oriented systems incorporating inheritance. All are run-time systems implemented over traditional operating systems, such as Unix.

ESP (Extensible Systems Platform) is a C++ run-time system supporting distributed C++ objects. The important contribution of ESP is the notion of uniform communication. Local and remote method invocations, whether synchronous or asynchronous, are handled uniformly by a single invocation mechanism. The advantage to this approach is that a high degree of conceptual economy is realized on the part of the application developer. The major disadvantage to inter-object communication is that only basic data types may be transmitted as part of messages. There is no high-level support for value transmission of user-defined types. That is, there is primitive support for independent data representation beyond big-endian little-endian transformations.

SOS, like ESP, is based on the distribution and inter-communication of C++ objects. A primary feature of SOS is the support for object persistence. SOS introduces language extensions to C++ which facilitate the semi-automatic collection of the persistent state of an object. The most interesting aspect of SOS is the notion that persistence and migration are similar. SOS includes the concept of a "proxy" which supports the migration of an object from an object storage server to the executable store of a machine. The developers of SOS made extensions to the base C++ language definition to support persistence. However, their approach does not fully consider persistence as a part of the meta behavior of an object in conjunction with communication and concurrency.

The focus of the Arjuna system is to provide fault tolerant distributed computing using a transaction model. Like SOS, Arjuna supports persistent objects. Unlike the semi-automatic persistence mechanisms in SOS, Arjuna provides an almost fully-manual persistence mechanism. There is no notion of persistence as a meta behavior of an object. The implementor of an object must specify completely the structure of the object's data components so that the migration mechanism can properly collect the object's state. The mechanisms of Arjuna can not be considered a serious approach to persistent objects. A worthwhile note is that Arjuna uses the class inheritance mechanisms of C++ to provide both the persistence and transaction mechanisms.

Several current industrial research and development efforts have also given useful guidance to the Synergy project. Similar architectures and goals are used in both DSG's Distributed Software Engineering Toolset (DSET) product and NCR's Cooperation system. Project Synergy differs from DSET because Synergy integrates OOP into the OSI framework more extensively. DSET, which is also based on the ISODE, does not go to the extent of applying an object-oriented rationalization to the protocol layers. NCR's Cooperation and Synergy differ in that Synergy places greater emphasis on the restructuring of the OSI protocol layers as a foundation for open systems programming. The overall spirit of the Synergy project was derived from many efforts focused on an open computing environment and the use of standards. Such efforts include the OSF/1 system and the X windows system. In the communication domain, Sun's recently announced transport independent communications mechanism and the OSI RPC mechanisms are examples of current work which is attempting to instill greater flexibility and independence in the interaction among objects.

The Object Request Broker (ORB) defined by OMG [OMG 91] offers an application infrastructure similar to that proposed by Project Synergy. Superficial differences are that Project Synergy is based on the OSI protocols and ASN.1 while ORB is based on IDL and RPC. The correspondence of the IDL grammar and C++ seems to suggest a better fit than ASN.1. An obvious difference between our work and OMG is that we do not restrict objects to only client-server interaction. It is unclear at this time whether or not this is really a significant difference. The goals of the application structure of our project need to be further reviewed with respect to the recent accomplishments of OMG.

Acknowledgments

Chris Tomlinson of MCC provided valuable feedback on the initial idea of using inheritance to structure the upper protocol layers. Phil Cannata of Bellcore/MCC provided the environment and support which allowed the further development of these ideas.

References

[Agha 86] Gul Agha, *ACTORS: A Model of Concurrent Computation in Distributed Systems*, MIT Press, 1986.

[America 87] Pierre America. "Inheritance and subtyping in a parallel object-oriented language," *Proceedings of the 1987 European Conference on Object-Oriented Programming (ECOOP'87)*, June 1987, pp. 232–242.

[Bershad 87] Brian N. Bershad, Dennis T. Ching, Edward D. Lazowska, Jan Sanislo, and Michael Schwartz. "A remote procedure call facility for interconnecting heterogeneous computer systems," *IEEE Transactions on Software Engineering*, SE–13(8), August 1987, pp. 880–894.

[Birrell 84] Andrew D. Birrell and Bruce J. Nelson. "Implementing remote procedure calls," *ACM Transactions on Computer Systems*, 2(1), February 1984, pp.39–59.

[Black 87] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter. "Distribution and abstract types in Emerald," *IEEE Transactions on Software Engineering*, SE-13(1), January 1987, pp. 65–76.

[Briot 90] Jean-Pierre Briot and Akinori Yonezawa. "Inheritance and synchronization in object-oriented concurrent programming," in *ABCL: An Object-Oriented Concurrent System*, (ed. A. Yonezawa), MIT Press, 1990.

[Campbell 86] Roy Campbell, Gary Johnson and Vincent Russo, "Choices: Class Hierarchical Open Interface for Custom Embedded Systems," *Operating Systems Review*, 21(3), July 1987, pp. 9–17.

[Cheriton 88] David R. Cheriton. "The V distributed system," *Communications of the ACM*, 31(3), March 1988, pp. 314–333.

[Cheritton 89] David R. Cheriton and Carey L. Williamson. "VMTP as the transport layer for high-performance distributed systems," *IEEE Communications Magazine*, June 1989, pp. 37–44.

[deJong 91] Peter deJong, "A Framework for the Development of Distributed Organizations," unpublished paper, 1991.

[Dixon 89] G. N. Dixon, G D. Parrington, S K. Shrivastava, and S. M. Wheater. "The treatment of persistent objects in arjuna," *Proceedings of the 1989 European Conference on Object-Oriented Programming (ECOOP'89)*, July 1989, pp. 169–189.

[Hewitt 84] Carl Hewitt and Peter de Jong. "Open systems," in *On Conceptual Modeling*, (ed. Michael L. Brodie), Springer-Verlag, 1984, pp. 147–164.

[Hewitt 90] Carl Hewitt. "Towards open information systems semantics," unpublished paper, 1990.

[Hoare 88] C. A. R. Hoare. *Communicating Sequential Processes*, Prentice-Hall, 1985.

[ISO 7498] International Standards Organization. *Information Processing—Open Systems Interconnection—Basic Reference Model*, International Standard 7498.

[ISO 8824] International Standards Organization. *Information Processing— Open Systems Interconnection—Specification of Abstract Syntax Notation One (ASN.1)*, International Standard 8824, 1987.

[ISO 9072-1] International Standards Organization. *Information Processing—Text Communication—Remote Operations part 1: Model, Notation and Service Definition*, International Standard 9072–1, 1988.

[Kafura 89] Dennis G. Kafura and Keung Hae Lee. "Inheritance in actor based concurrent object-oriented languages," *Proceedings of the 1989 European Conference on Object-Oriented Programming (ECOOP'89)*, July 1989, pp 131–45.

[Lavender 90] Greg Lavender and Dennis Kafura. "Specifying and inheriting concurrent behavior in an actor-based object-oriented language," Technical Report TR 90–56, Department of Computer Science, Viriginia Tech, 1990.

[Lavender 91] Greg Lavender, Dennis Kafura, and Chris Tomlinson. "Implementing communication protocols using object-oriented techniques," in preparation, 1991.

[Leddy 89] Bill Leddy and Kim Smith. "The Design of the Experimental Systems Kernel," *Proceedings of the Conference on Hypercube and Concurrent Computer Applications*,Monterey, CA, 1989.

[Liskov 87] Barbara Liskov, Dorothy Curtis, Paul Johnson, and Robert Scheifler. "Implementation of argus," *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, November 1987, pp. 111–122.

[Matsuoka 90] Satoshi Matsuoka, Ken Wakita, and Akinori Yonezawa. "Analysis of inheritance anomaly in concurrent object-oriented languages," extended abstract presented at the ECOOP/OOPSLA'90 Workshop on Object-based Concurrency, October 1990.

[Milner 89] Robin Milner. *Communication and Concurrency*, Prentice-Hall, 1989.

[Mullender 89] Sape J. Mullender, Guido van Rossum, Andrew S. Tanenbaum, and Robbert van Renesse. "Amoeba: a distributed operating system for the 1990s," *Computer*, May 1990, pp. 44–53.

[Nierstrasz 90] Oscar Nierstrasz and Michael Papathomas. "Towards a type theory for active objects," in *Object Management*, pp. 295—304, (ed. D. Tsichritzis), Centre Universitaire D'Informatique, Universite De Geneva, 1990.

[OMG 91] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, OMG Document Number 91.8.1, Draft 26 August 1991.

[Renesse 87] Robbert van Renesse, Andrew S. Tanenbaum, Hans van Staveren, and Jane Hall. "Connecting RPC-based distributed systems using wide-area networks," *Proceedings Seventh International Conference on Distributed Computing Systems*, IEEE, 1987, pp. 28–34.

[Rose 90] Marshall T. Rose. *The Open Book: A Practical Perspective on OSI*, Prentice-Hall, 1990.

[Rose 91] Marshall T. Rose, Julian P. Onions, and Colin J. Robbins. *The ISO Development Environment User's Manual—Version 7.0*, Vols. 1-5, X-Tel Services Ltd, Nottingham, July 1991.

[Shapiro 89] Marc Shapiro, Philippe Gautron, and Laurence Mosseri. "Persistence and migration for C++ objects," *Proceedings of the 1989 European Conference on Object-Oriented Programming (ECOOP'89)*, July 1989, pp.191–204.

[Sun 87] Sun Microsystems. "RPC: remote procedure call protocol specification version 2," *Request for Comments 1057*, SRI Network Information Center, June 1988.

[Sun 88] Sun Microsystems. "XDR: external data representation standard," *Request for Comments 1014*," SRI Network Information Center, June 1987.

[Tomlinson 89] Chris Tomlinson and Vineet Singh. "Inheritance and synchronization with enabled-sets," *ACM OOPSLA'89 Conference Proceedings*, October 1989, pp.103–112.

[Wegner 90] Peter Wegner. "Concepts and paradigms of object-oriented programming," *OOPS Messenger*, 1(1), August 1990, pp. 7–87.

[Zdonik 90] Stanley B. Zdonik and David Maier, "Fundamentals of Object-Oriented Databases," in *Readings in Object-Oriented Database Systems*, (eds. S.B. Zdonik and D. Maier), IEEE Press, 1990, pp. 1–32.