Technical Report
CMU/SEI-96-TR-017
ESC-TR-96-017

Transitioning a Model-Based Software Engineering Architectural Style to Ada 95

Anthony B. Gargaro
A. Spencer Peterson

August 1996

Technical Report
CMU/SEI-96-TR-017
ESC-TR-96-017
August 1996

Transitioning a Model-Based Software Engineering Architectural Style to Ada 95

Anthony B. Gargaro

Computer Sciences Corporation

A. Spencer Peterson

Product Line Systems Program, Domain Analysis Team

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This report was prepared for the

SEI Joint Program Office
HQ ESC/AXS
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER

(signature on file)

Thomas R. Miller, Lt Col, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number F19628-95-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 52.227-7013.

This document is available through Research Access, Inc., 800 Vinial Street, Pittsburgh, PA 15212.
Phone: 1-800-685-6510. FAX: (412) 321-2994. RAI also maintains a World Wide Web home page. The URL is http://www.rai.com

Copies of this document are available through the National Technical Information Service (NTIS). For information on ordering, please contact NTIS directly: National Technical Information Service, U.S. Department of Commerce, Springfield, VA 22161. Phone: (703) 487-4600.

This document is also available through the Defense Technical Information Center (DTIC). DTIC provides access to and transfer of scientific and technical information for DoD personnel, DoD contractors and potential contractors, and other U.S. Government agency personnel and their contractors. To obtain a copy, please contact DTIC directly: Defense Technical Information Center, Attn: FDRA, Cameron Station, Alexandria, VA 22304-6145. Phone: (703) 274-7633.

Table of Contents

List of Figures

Transitioning a Model-Based Software Engineering Architectural Style to Ada 95

Abstract: This report describes the transition of an existing Model-Based Software Engineering architectural style to Ada 95. The report presents an overview of a software architecture for developing product families of domain-specific applications comprising reusable components, explains recognized deficiencies in the existing Ada mapping to this software architecture, and proposes solutions for correcting these deficiencies using a mapping to Ada 95. The report concludes with observations gained during the transition exercise and recommendations for future activities aimed towards deploying and enhancing the proposed mapping.

1 Introduction

Model-Based Software Engineering (MBSE) is a disciplined engineering approach that relies on constructing models of software applications within a *product family* to achieve the benefits of reuse, shorter time to market, and higher quality. The models provide the necessary information to support, economically and effectively, future changes to a software product family. MBSE relies upon proven *domain analysis* and *engineering* techniques to specify different models of a domain such that recurring patterns used in the software design of related applications can be discovered.

Patterns of function, structure, and coordination capture fundamental abstractions about the product family. When such patterns are incorporated in a software architecture, designers can more readily develop applications that are upwardly compatible, and better manage software complexity and changes. Costs can be reduced since applications are not repeatedly developed from scratch and prior products are more easily reused. Unfortunately, current design techniques do not adequately address how to specify and use these patterns, nor do they adequately specify an infrastructure for composing such patterns from reusable abstractions.

The Object Connection Architecture (OCA) is a generalized form of a group of related architectures that have been developed and refined by various projects within the SEI for several years. The OCA structural abstractions (as described in [Peterson 94]) have been mapped to Ada 83 [ISO 87], and this mapping has been used successfully as a part of MBSE to implement an application prototype from the Army movement control domain [Cohen 92]. However, the lack of suitable abstractions necessary to achieve effective software reuse [Cohen 90] compromises the utility of the resulting architecture for implementing a wider range of applications and has motivated a transition to Ada 95 [ISO 95].

This report presents an overview of the OCA and the current Ada 83 mapping, a summary of issues raised by this mapping, proposals for how such issues may be addressed in a transition of the mapping to Ada 95 [ISO 95], and a partial Ada 95 mapping that supports distributed execution environments. In addition, recent work in software architecture specifications relating to the mapping is cited during the discussion. The principal objective of the mapping to Ada 95 is to exploit the potential of the OCA as a leveraging technology for establishing a framework within which reusable software components may be developed and reused to facilitate creating product families. In addition, this framework is specified in sufficient detail to facilitate the progressive transition of less formal analysis and design artifacts to an executable prototype of a distributed systems architecture.

2 Model-Based Software Engineering

MBSE promotes a disciplined approach towards achieving product line engineering of software intensive systems.[1] Figure 2-1 illustrates a conceptual overview of the principal activities and products of MBSE; the aggregation of domain analysis, design, and implementation is often referred to as *domain engineering*, as denoted by the shaded area.[2] As shown in this figure, a generic design results from completing the domain design activity.

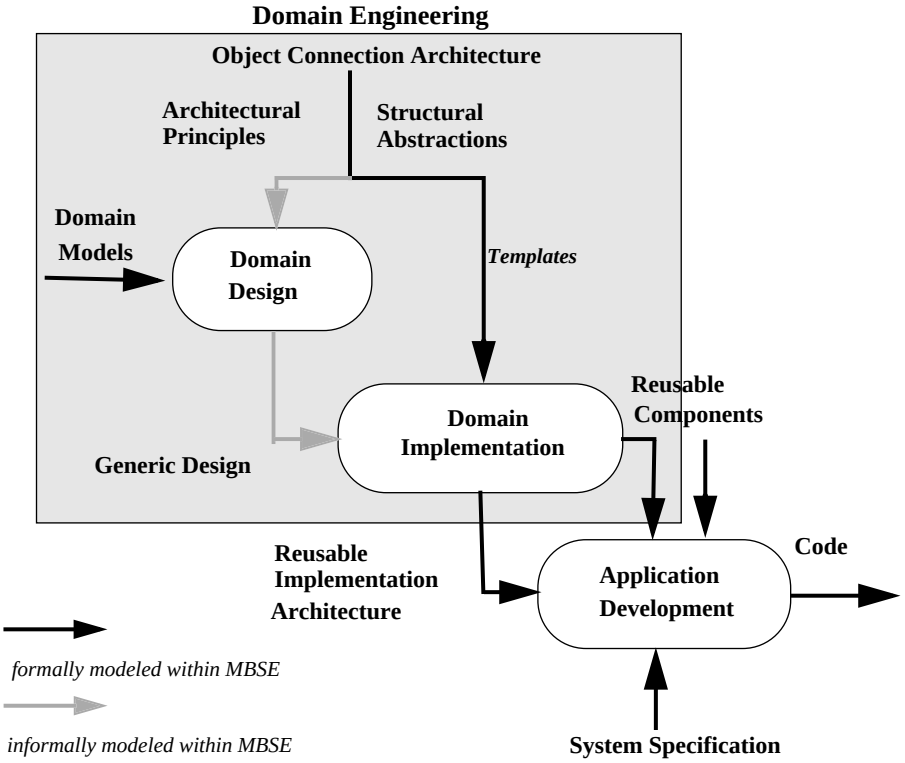


Figure 2-1: MBSE Conceptual Process

2.1 MBSE Overview

In Figure 2-1, the architectural principles and generic design are informal specifications to the extent that, unlike the domain models, no formal models are currently provided (as denoted by the dashed lines). For example, a typical architectural principle might be the layering strategy of software components that must be reflected in the generic design as a means to

---

1. Two draft SEI technical reports, “Implementing Model Based Software Engineering In Your Organization: An Approach to Domain Engineering “ and “Software Engineering in a Product Family: A Model-Based Software Engineering Approach to Reuse” , authored by James Withey in 1994 and 1995 respectively, provide a more detailed description of MBSE.

2. Domain analysis, although not shown as a process in Figure 2-1, produces the domain models shown as the input to the domain design process.

increase software reuse [Zweben 95]. A generic design results from completing the domain design activity. It is commonly represented as a system behavioral model that identifies the abstractions and interactions for applications of the product family obtained from information in the domain models. The generic design is the input to the domain implementation activity so that more specific information may be incorporated within the controlling framework of the structural abstractions, with corresponding templates, to transform the generic design into a *reusable implementation architecture*. Consequently, the mapping of the OCA structural abstractions to programming language templates adds formality to the architectural principles used to specify the generic design, allowing the resulting reusable implementation architecture to be used for application generation. Ideally, the reusable implementation architecture provides an executable prototype that evolves and matures throughout the lifetime of the product family. Once available, different applications may be developed from reusable components as indicated by the application development activity. The system specification identifies application-specific requirements.

The mapping of the OCA structural abstractions into code templates facilitates developing the generic design into a reusable implementation architecture that may be executable. Moreover, depending upon the available abstractions, the reusable implementation architecture provides an infrastructure for both creating reusable components and integrating existing components. Thus, the domain implementation activity focuses upon adapting and refining the OCA mapping according to the specifications contained in the generic design; for example, the generic design may include requirements for a particular software quality attribute [Bass 94] (in addition to the reusability attribute). The templates specify the data types, features, operations, and performance that are inherited in the reusable software architecture. In addition, the templates avoid overspecification by not dictating particular environmental capabilities such as the capacity of processing resources that are required to execute applications in the product family.

2.2 OCA Legacy

The legacy for the OCA derives from previous SEI architecture development experiences. The first development of an OCA-like architecture was performed by the Ada Simulator Validation Project (ASVP) in 1986 and 1987. [Lee 88] described an abstraction-oriented view of flight simulations, with *object managers* providing abstractions for real-world aircraft parts working together as complex systems under the control of an *executive*. The systems communicated through *connection* packages, which were responsible for maintaining linkages between *import* and *export* data for each system. The structural abstractions described in the report are the foundation of the OCA.

Two projects originated from the work performed by the ASVP, the Software Architecture Engineering (SAE) and the Real-Time Simulator Projects. Each was funded by different sponsoring organizations and focused on enhancing the ASVP architecture in various ways. The SAE work culminated in the development of the Object Connection Update (OCU) architecture, which was used in several application areas, including a missile radar seeker

system, a mine sweeping trainer/simulator, and a joint service electronic warfare simulation environment. [Rissman 90] provides a good description of many aspects of the OCU model. The Real-Time Simulation Project work culminated in an approach called *Structural Modeling*, which has been used in multiple flight simulators built under the direction of the Air Force Aeronautical Systems Command (ASC), including those for the B-2, ATF, and C-17. Further information on Structural Modeling is provided in [Abowd 93]. Figure 2-2 shows the chronological development of the OCA.[3]

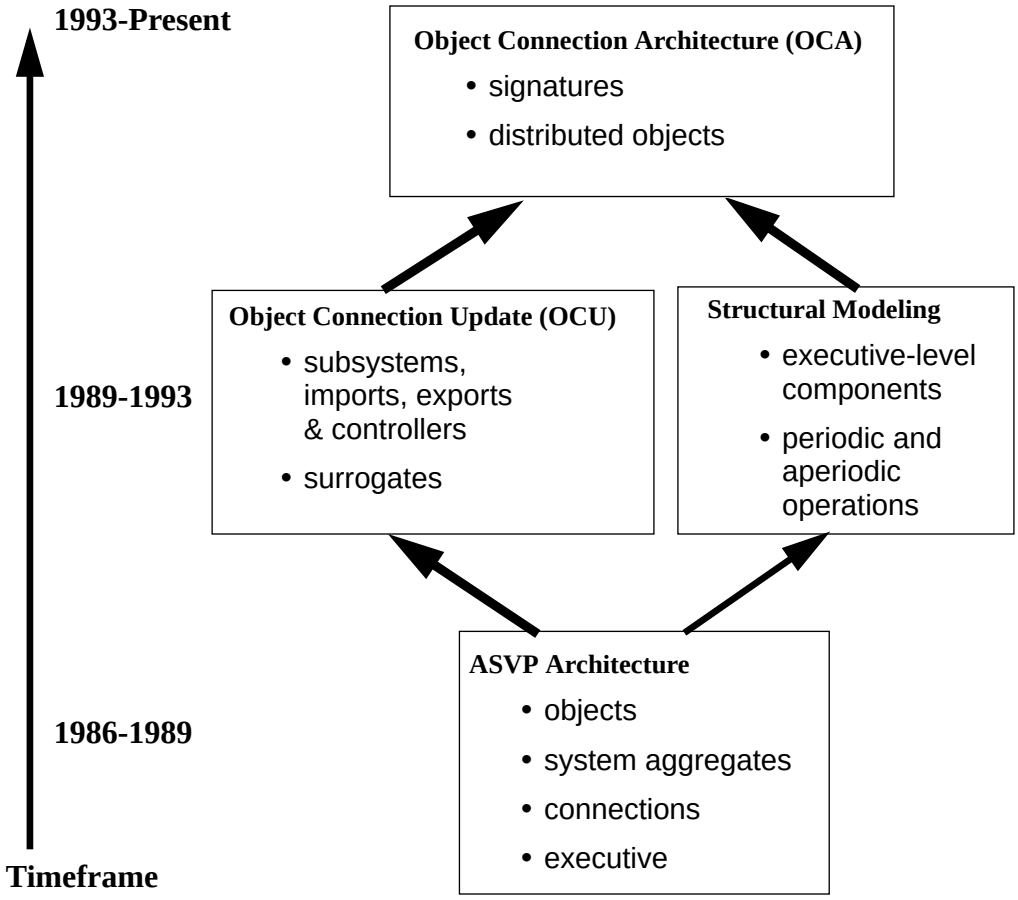


Figure 2-2: OCA Legacy

The Application of Software Models (ASM) Project investigated both of these architectures and found them to be lacking sufficient generality when attempting to build the system prototype previously mentioned. One of the major shortcomings was the inability of either architecture to specify operations over a wide range of data elements or with differing computational semantics in an efficient manner. This is due, in part, to the real-time nature of

3. It should be noted that the OCA and Structural Modeling activities are still ongoing as of the publication of this report.

the domains in which the previous architectures operated versus the more interactive nature of domains such as that of movement control. The development of the OCA sought to keep those features of its predecessors that supported a pattern-base architectural style while increasing its general utility. The notion of *signatures* is one of the means used to achieve that result. The signatures concept is explained briefly in Section 4.1 of this report.[4] It accounts for a large amount of the decoupling of service description from callability needed to make distribution of objects possible.

4. See [Peterson 94], pages 16 - 18 for a more complete discussion of signatures.

3 Architectural Overview

Reusable implementation architectures are derived from the OCA templates and a generic design. They provide

- a framework for logically partitioning applications
- a model for coordinating and connecting different components comprising applications
- an infrastructure for composing applications from existing or newly developed reusable components

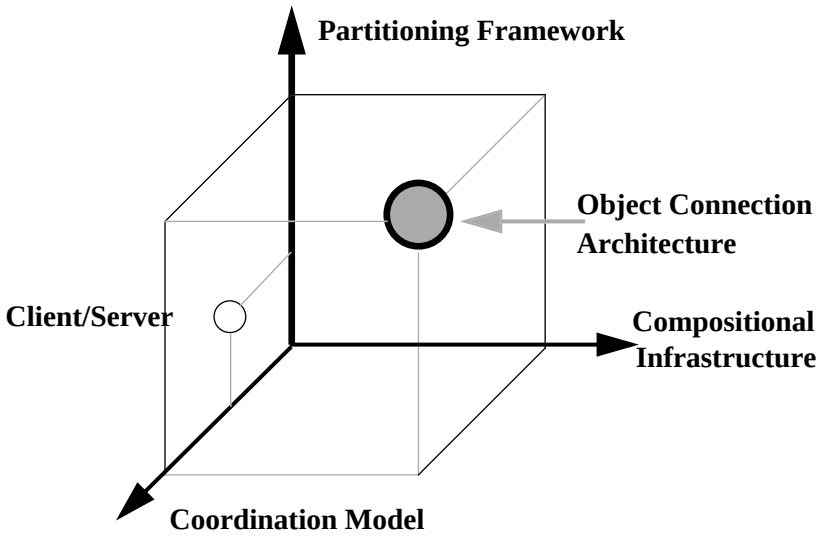


Figure 3-1: OCA Conceptual View

Figure 3-1 illustrates a conceptual view of the OCA as the unification of three guiding principles that underlie sound architectural styles: a *partitioning framework*, a *compositional infrastructure*, and a *coordination model*. Each principle provides a set of rules as denoted below.

1. The *partitioning framework* specifies the rules for designating and aggregating structural abstractions. The partitioning rules aid in defining where domain information and/or functional entities fit in an overall framework.
2. The *compositional infrastructure* specifies the rules for integrating objects into the structural abstractions. Objects are any software entities that provide the low-level services required by the structual abstractions.
3. The *coordination model* specifies the rules for connecting structural abstractions and for establishing communications mechanism between them, as needed.

The locus of unification shifts depending upon the generic design in order to achieve a reusable implementation architecture. For example, the locus of a typical client/server architecture is shown in Figure 3-1 as defined by a partitioning framework (designation of client and server processes) and a coordination model (the sequence of interactions between clients and servers). Templates for implementing the patterns conforming to these rules are included as a mapping to a programming language.[1] Since these templates take a canonical form, they become the design patterns for applications. It is in this context that the OCA is discussed in subsequent sections.

3.1 OCA Principles

The underlying architectural principles of the OCA are driven by the requirement to maximize object reuse across a product family. A reusable implementation architecture is a blueprint for developing applications of the product family.

The notion of a reusable object and its associated operations (services) is fundamental to understanding the OCA. Objects represent common capabilities or properties of the product family with well-defined service interfaces, and encapsulated state and behavior. Such objects provide a higher, more understandable, and more systematic level of abstraction than traditional data and procedural abstractions. This reflects the emerging trend to represent an application domain as a continuum of object abstractions.

This notion is widely used in developing reusable software components and is intuitive from the name Object Connection Architecture. The OCA depends upon reusable objects that are consistent with the information available from the domain models.[2] Such objects may be created independently of the OCA or they may already exist in legacy applications from the domain.

More important, but less accepted, is the notion of connected objects. This notion derives from an exposition of a software architecture [Shaw 93] that emphasizes rigorously specifying the partitioning and interconnectivity properties to be enforced upon a system's functional components. The distinction between a functional entity, a *component*, and a connectivity entity, a *connector*, is offered as a progressive step towards enhancing the compositional and partitioning properties of a software architecture [Shaw 94a]. A functional component represents the locus of computation and state; whereas a connector represents the locus of relations among components. It is this approach to partitioning and connectivity together with a clearly defined coordination model [Carriero 92] that are the important architectural principles of the OCA.

---

1. The term OCA mapping is used throughout this report to connote the transformation of structural abstractions to the corresponding templates (or structural code units) of a programming language.

2. It should be noted that while the currently defined OCA is described in the context of the three models of the SEI *Feature-Oriented Domain Analysis* (FODA) methodology [Kang 90] for identifying reusable objects, the models and methodology are incidental to this discussion of principles.

3.2 Object Reuse

Object reuse requires incrementally composing reusable objects from existing objects to represent new or enhanced capabilities and features. It requires that all object interactions are conducted through well-defined interfaces corresponding to a particular compositional strategy. Objects must be free of uncontrolled interactions with other objects to safeguard the reuse of each object. For example, if two objects are mutually dependent upon common state information, a change to this state information by either object may potentially compromise the services of the other object; the absence of such deleterious interactions has been termed *compositional orthogonality* in an early effort to specify rules for writing reusable Ada components [Gargaro 87]. Moreover, the service interface provided by an object to a client must be insulated from any subtle expectations of how the services are implemented. If this insulation is defective, different implementations for the same service may unintentionally invalidate the services provided by client objects. For example, one implementation of a service may block execution of the client, while another implementation may guarantee never to block the execution of the client. Thus, it is important that any expectations of service quality be specified explicitly. Typically, this information is identified by the features specifications included in a domain model [Cohen 92] and must be reflected in both the generic design and templates for the service interface.

3.3 Subsystem Reuse

The capability to construct application-specific software from domain reusable objects requires that objects be connected through a controlled and systematic approach. For example, combining the capabilities of two objects to compose an aggregate capability may require that the result produced by the services of one object be used by the services of the other in a prescribed order. Thus, at a minimum, there is a need to introduce a client object that requests the services of the two objects in the prescribed order to provide the aggregate capability as a service. It is through the actions of this client that the two objects cooperate (are connected) without compromising their mutual independence; moreover, it is unnecessary for either object to be aware of the actions of the client or of each other. When this cooperation involves many objects and must be coordinated or synchronized in some disciplined manner, these actions are more complicated (for example, when the services of different objects are not executed serially and reside in a loosely coupled execution environment). Thus, such clients may use connector objects to satisfy the role described by [Shaw 94a]. In this role, connector objects may be viewed as abstract data types that provide a protocol for calling the client's constituent objects' services.

3.4 Application Reuse

The capability to partition an application into autonomous collections of software components amplifies the granularity of software reuse. To achieve this capability, the architectural principles and partitioning framework must guide allocating, configuring, and executing the application within the constraints imposed by the execution environment. For a large-scale

application, comprising many partitions, the application may span multiple execution environments; furthermore, the execution environments may comprise heterogeneous systems.

Ultimately, effective application reuse requires defining three architectural tiers of reuse. The tiers facilitate using the structural abstractions to define consistent patterns of composition and control within applications of a domain rather than to implement low-level objects of the domain. These tiers are discussed in the next chapter in terms of object composition, connection, and configuration.

4 Structural Abstractions and Reuse

The preceding chapter presented the capabilities and principles that serve as the basis for reuse in the OCA. Three principal structural abstractions provide the patterns for reuse that are consistent with this basis.[1] These structural abstractions are

- object managers
- controllers
- executives

Objects represent the fundamental service providers within a domain. One or more objects become a reusable component when encapsulated by an *object manager*. It is through the interface to the object manager that the services of these objects are made available to potential clients in a manner that is consistent with a specific feature or function identified by the domain models. The object manager maintains a current state to control and coordinate the actions required to use the services of the different objects. In this way, objects remain independent of one another and the object manager.

Controllers represent the role of connection objects within the domain. Through the specification of a *controller*, one or more object managers are aggregated to form a *subsystem*. The controller provides the interface to the subsystem and the necessary interconnection and data exchange among the object managers. Thus, a controller is a client of its constituent object managers so that the subsystem is implemented through the combined services of the object managers. In this way, each object manager is independent of its peers, and subsystems cooperate only through the data that need to be exchanged to execute the object managers' services. Whereas object managers must be called through their interface to perform a service, subsystems may execute concurrently with calls to their controller interface, thereby providing a more dynamic structural abstraction. For example, a controller may include default execution in the absence of an active call; the periodic reclamation of resources is a good example of such default behavior.

Executives represent the role of a management object for the application. In contrast to a controller and an object manager, an *executive* provides a domain specific service. This service is consistent with the executive's role as a structural abstraction that encapsulates subsystem activities of an application within a particular execution environment.

The three principal abstractions — object managers, controllers, and executives — facilitate the three levels of reuse. The object manager supports the *object composition tier* (for object-level reuse); the controller supports the *connection tier* (for subsystem-level reuse); and the executive supports the *configuration tier* (for application-level reuse). Object reuse is achieved from using the object class libraries that are typically available in an application domain. The

1. It should be noted that the OCA defines a fourth structural abstraction termed a *surrogate*. This abstraction is a variation of a controller; however, in this report the distinction between controllers and surrogates is unnecessary.

resulting object managers are connected to build reusable subsystems using controllers. Finally, reusable subsystems are configured to build applications using an executive.

The levels are illustrated in Figure 4-1.

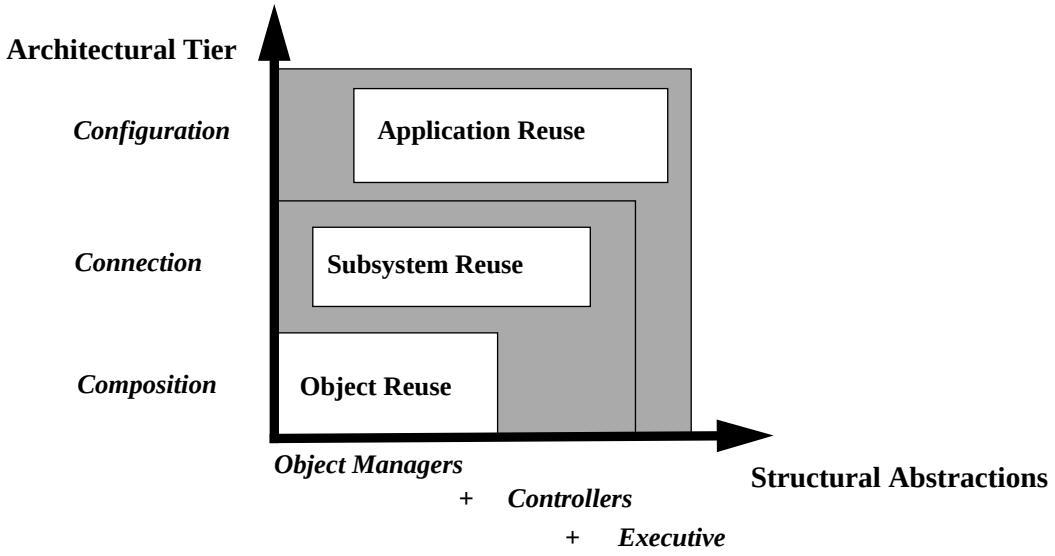


Figure 4-1: OCA Levels of Reuse

The structural abstractions are completed by specifying two support abstractions: the *import* and *export* abstractions. The roles of these abstractions in the OCA are implicit from their names; they support the disciplined exchange of data among subsystems. Specifically, an import provides the input data from other subsystems to a controller; in contrast, an export provides the output data from a controller required by other subsystems. Through the import and export abstractions, the objects and object managers included in different subsystems remain opaque to each other. Figure 4-2 shows the relationship of the tiers of reuse and the structural abstractions. In this figure the italicized font denotes that at the named tier, the structural abstraction is defined to provide a reusable object. The normal font denotes that the structural abstraction is used at that tier. For example, object-level reuse is achieved by defining an object manager that uses objects that exist in the application domain.[2]

Complementing the compositional abstractions are meta-abstractions called *signatures*. There are three different kinds of signatures corresponding to each level of reuse: (1) application signatures, (2) subsystem signatures, and (3) object signatures. Each kind of signature fulfills a similar role by describing the features available at each level of reuse so that the corresponding abstractions are not compromised by the client nor the service provider. For example, the application signature may be used to identify the various subsystems that comprise the application together with the features that describe the different configurations of these subsystems. In this way, an executive may determine what capabilities may be offered by the application simply by examining the signatures information and determining

2. For completeness, objects are included in the figure as structural abstractions.

what subsystems are available. The executive has no knowledge of how a subsystem is implemented, while a subsystem implements a feature in a way consistent with the applicable signature information.

	LEVEL OF REUSE		
Structural Abstractions	**Object**	**Subsystem**	**Application**
Object	**Composition**		
Manager	***Composition***	**Connection**	
Controller		***Connection***	**Configuration**
Export/Import		***Connection***	**Configuration**
Executive			***Configuration***

Figure 4-2: Relationship of Abstractions to Levels of Reuse

Thus, signatures provide contracts for component reuse and correspond to the outputs of the domain models; for example, they contain information about the commonality and variability implemented in an application. Signatures do not presuppose a particular design methodology; they allow the systematic specification of an application within a product family.

No restrictions are enforced on the execution of the OCA, although event and timeline synchronization at the configuration tier is required by the product family[3] from which the OCA derives. Thus, coordination of subsystems in the configuration tier may be specified in a domain dependent manner. For example, subsystems may be allocated, invoked, scheduled for execution, and terminated either synchronously or asynchronously. Consequently, subsystems may be thought of as comprising an *asynchronous ensemble* [Carriero 92], where the problems of coordination are best understood as orthogonal to the application processing.

The important architectural consideration is that there is a coherent coordination model associated with the execution of the structural abstractions encapsulating the reusable software components of an application. This coherency promotes potentially increased flexibility in the adaptation and configuration of an application to different execution environments.

3. The original coordination model was developed for the Air Vehicle Structural Model (ASVM) [Abowd 93] product family.

The executive at the configuration tier is limited to coordinating subsystem execution; whereas, at the connection tier, the controllers must coordinate both communication and execution of the object managers. It is this level of coordination that allows each object manager to be executed independently of its peers. Through such independence, the potential to support execution environments that gain leverage from contemporary *distributed object* technology [OMG 93] becomes achievable. Each controller and its associated export and imports provide the necessary connection to facilitate communication between subsystems similar to the interfaces connecting distributed objects. This, in turn, leads to more fault-tolerant applications when the executive provides capabilities to replace or replicate subsystems. To an extent, this approach is consistent with the earlier work of the SEI Durra Project [Barbacci 93] that allows an application to be described as a set of components, a set of alternative configurations for dynamically connecting components, and a set of runtime conditional configurations. Unfortunately, abstractions to express the configuration of applications comparable to those available in Durra have not yet been introduced into the OCA.

In contrast to the configuration and connection tiers, coordination at the composition tier is determined by the object manager; no architectural constraints are placed upon how coordination is achieved among its constituent objects. Coordination depends only upon the implementation of the objects (in the class libraries) of the application domain. For example, objects implemented in Ada may be more adaptable to real-time coordination models [Gargaro 89, Fernandez 93] than objects implemented in other languages.

4.1 Architectural Style Mapping

In the original mapping, each structural abstraction is mapped to an Ada program unit (template). Except for the executive, which is the Ada main program, each abstraction comprises a package specification and, where appropriate, a package body declaration.

Reusable objects are specified as object manager packages. Each object manager package name appends the suffix "_Manager" to the identity of the object. The specification declares the visible subprogram (services) and exceptions associated with the object. The types of the formal parameters of these subprograms are declared either in the corresponding object signatures package or software engineering unit packages[4] on which each object manager semantically depends. In the package body, the object state data and subprogram bodies are declared, together with any local ancillary subprograms. The package body may depend semantically on other program units, providing these units do not encapsulate other objects.

Each object signatures package name appends the suffix "_Signatures" to the identity of the object. The specification declares the features associated with the object as one or more enumeration types. The specified enumeration literals are used to select and control the services provided by the object. There is no body for an object signatures package.

4. Software engineering packages declare data types common to a domain.

Controllers are specified as subsystem controller packages. Each controller package name appends the suffix "_Controller" to the identity of the subsystem. The package specification declares the visible subprogram (services) associated with the subsystem. The types of the formal parameters of these subprograms are declared in the corresponding subsystem signatures package or the application signatures package on which the controller semantically depends. In the package body, the subsystem state data and subprogram bodies are declared, together with any initialization required by the subsystem. The package body semantically depends upon the object managers that implement the subsystem, the import packages of all subsystems with which it cooperates, and the corresponding subsystem export package.

Each subsystem signatures package name appends the suffix "_Signatures" to the identity of the subsystem. The specification declares the features associated with the subsystem as one or more enumeration types. The specified enumeration literals are used to select the objects and services necessary to support the subsystem services. The package depends semantically on the corresponding object manager and object signatures packages. In addition, the specification declares any object signatures package data that must be *re-exported,* as defined in [Bardin 88], and all the different subsystem states. There is no body for a subsystem signatures package.

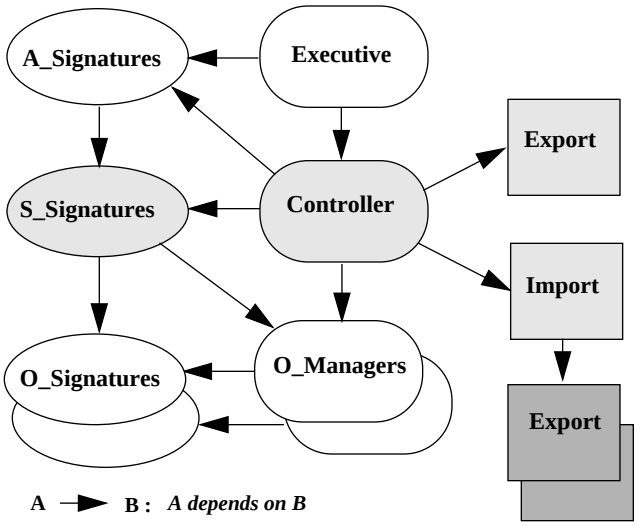


Figure 4-3: Subsystem Template Dependencies

The executive provides the initial thread of control in its role as the Ada main program; it depends upon the application signatures package and the different controller packages for the subsystems comprising the application.

Figure 4-3 shows the principal semantic dependencies of the original Ada mapping for a subsystem template. The shading denotes subsystem-specific abstractions. The various shapes convey the intrinsic differences among the roles of the structural abstractions. Namely, signatures are passive and do not include any executable code; executives, controllers, and

object managers are active and may include one or more threads of control; exports and imports are passive but may include executable code.

4.2 Ada Mapping Issues

The original Ada mapping is a conservative mapping that uses only the most straightforward Ada constructs; the mapping omits the use of both generics and tasking. This has avoided many of the semantic fringes of the language where compiler implementations have been either unreliable or have varied, thereby protecting the portability and reliability of the mapping. However, this emphasis on portability and reliability has resulted in a mapping that does not fully exploit the architectural principles of the OCA. Both the composition of components and the coordinating interactions among components reflect a static and tightly coupled orientation. This approach is consistent with the original requirements of the Ada language [Whitaker 93]. However, for developing reusable software a more dynamic and loosely coupled object orientation requires support beyond that specified in the previous Ada standard. Such support is discussed in [Atkinson 91].

The static nature of the mapping is manifested in the semantic dependencies that must be explicitly specified among the different kinds of packages and the corresponding encodings necessary to locate and control their services. The encodings, due in part to the earlier limitations of Ada, enforce a static, inflexible structure on the architecture. This leads to a less abstract and more code-intensive mapping, making applications more difficult to comprehend and maintain, since packages must be distinguished using the special encodings. Although it is desirable for applications to adhere to a structural pattern or template that can be checked by the mapping semantics, the mapping should not be so restrictive as to require naming explicitly every possible package that provides data or some service.

For example, consider the import package. Each import package associated with a subsystem must name the export packages associated with each subsystem on which it depends for data. The import package then explicitly maps this data to a function call in order to access the data. This requires that the function recognize each export package that may provide this data via an input parameter identifying a particular export package. Thus, the dependency has been introduced into the implementation in the form of a parameter value and the corresponding code that references the value. Moreover, each possible export package that is recognized must be handled separately.

The tight coupling imposed by the mapping is evident from the specification of the executive as the single main Ada program. Unless the mapping relies upon a distributed implementation of the Ada task model or extra-linguistic mechanisms that support multiple cooperating main programs, a more loosely coupled mapping is not possible. Both of these options introduce undesirable ramifications that compromise the portability of the OCA. In addition, adapting the Ada task model to distribute an Ada program typically imposes restrictions upon the application and execution environment in order to avoid the overhead penalties of uncontrolled remote accesses.

The role of a controller package as a connection entity is significantly reduced by the imposed tight coupling. The strong encapsulation of its constituent service objects promotes locating loosely coupled objects across a distributed execution environment with the appropriate connection semantics supported by the controller. However, the current mapping is restricted to using the conventional assignment statement and subprogram call features of Ada to specify connection semantics.

An observation in evaluating the mapping with respect to generic units is appropriate. Generic units have been offered traditionally as a feature of Ada that promotes implementing object-based reusable abstractions. However, the current mapping does not exploit the use of generic units. This omission may be explained partially by the fact that traditionally generic units have contributed primarily to the composition tier in providing reusable objects. Nonetheless, the use of generic units at the connection tier may lead to more reusable controllers by using the *semi-abstract* method of *bequeathment* [Cohen 90].

The use of the semi-abstract method, which is well-suited to the bottom-up compositional approach, suggests that object managers be specified as generic units. Each controller is then constructed from instantiations of object managers with each controller possibly being specified as a generic unit. Since a controller may be viewed as an abstraction that tends to exhibit the properties of strong coupling and weak cohesion, bequeathment allows for improved reuse. This suggestion is supported by the following eloquently stated conclusion [Cohen 90] regarding the semi-abstract method: “This cooperative interaction is the basis for the intermediate levels (classes) that constitute the proper level of abstraction for domain-specific reusable software.”[5]

Finally, a comparison of the current OCA mapping with the structural model from which it is derived shows that the mapping complies closely with the overall objectives of the structural model. For example, the broad objective of the structural model is “to take a problem domain of great complexity and scale and to abstract and scale it to a coarse enough level to make it manageable, modifiable, and able to be communicated to a diverse user and development community” [Abowd 93]. The current mapping clearly facilitates such an objective. In addition, it provides a specific instance of the definition proposed for a structural model: “A structural model is a reusable collection of classes of differing levels of abstraction providing the basis from which the flight simulator software is derived” [Abowd 93]. The one significant departure from the structural model, as mentioned previously, is that the configuration tier does not impose upon an application the specificity of the Timeline Synchronizer coordination model. Consequently, the executive provides a less time-dependent abstraction for coordinating the subsystems of the application. While this is suited to a wider range of applications, it would be unsuited to deadline driven real-time systems such as the ASVM.

5. In the mapping to Ada 95, tagged types have been used in preference to generics to achieve this capability.

5 Transitioning Mapping to Ada 95

The previous chapter evaluated the current Ada mapping and identified compromises to the OCA architectural principles primarily because of language limitations in the original Ada standard. In contrast, this chapter examines Ada mapping issues that should be considered in subsequent mappings of the OCA with respect to architectural quality, components and connectors, distributed objects, and shared objects.

5.1 Architectural Quality

Recent work has suggested that a principal motivation for developing software architectures is to promote uniformly the *extra-functional qualities*[1] across all applications in the domain for which the architectures are designed [Bass 94]. Eight extra-functional qualities are defined; reusability is included as one of the eight qualities. Six of the other qualities are required by a domain-specific architecture since they typically complement or enhance software reuse. These qualities are

1. ease of creation
2. stability
3. modifiability
4. integrability
5. portability
6. reliability

Only the performance quality is omitted since it is often considered application-specific rather than domain-specific. Thus, it is appropriate to examine a mapping with respect to the *unit operations* through which the extra-functional qualities may be influenced. Six unit operations are identified as having an influence upon achieving the extra-functional qualities; they are

1. abstraction
2. compression
3. replication
4. resource sharing
5. separation
6. uniform composition

Each of these operations is described in the following paragraphs.

---

1. The term *extra-functional* is used in preference to the original term, non-functional, in accord with a suggestion from Mary Shaw.

A mapping should clearly delineate the way in which each unit operation is supported (or not supported). In this way, it should be apparent to what degree the extra-functional qualities are achievable. For example, a mapping that provides support for only one unit operation is unlikely to satisfy many (if any) of the extra-functional qualities. It is beyond the scope of this report to discuss how each unit operation may influence the extra-functional qualities.

A mapping that supports *abstraction* hides implementation details. Consequently, there should be no dependencies upon a particular compiler or execution environment other than what is explicitly permitted by the Ada standard. For example, a mapping that depends upon the use of calls to a UNIX-like socket capability to support communication within a distributed program fails to hide implementation details. A preferred approach would be to eliminate the dependency by specifying remote subprogram calls to perform distributed communication that is portable among execution environments. When the mapping cannot be accomplished within the bounds of the Ada standard, then interfaces to other open systems standards through the OCA surrogate analog of the OCA controller may be used to hide implementation details. Moreover such surrogates may exploit the Ada bindings to the standards, although this is not always feasible. The use of the existing OCA mapping to Ada has exposed several implementation dependencies that were addressed through specialized interfaces when using the OCA to implement a domain-specific application [see Peterson 94: Appendix F]. However, these dependencies are not inherent in the OCA mapping.

A mapping that supports *compression* may improve implementation execution time and performance efficiency. It was mentioned previously that performance is an application specific quality rather than a domain-specific quality. Consequently, the value of compression is more appropriately exploited once an architecture has been modeled and prototyped for the application. Moreover, compression conflicts with unit operations that are necessary to achieve other qualities enabling software reuse. Inevitably, compression leads towards a tightly coupled mapping; such a mapping is contrary to one of the OCA principles. Thus, compression support in an OCA mapping is not considered desirable.

A mapping that supports *replication* promotes fault-tolerant implementations. An important architectural principle implicit in the OCA is the flexibility to replicate controllers and object managers under the regimen of the coordination model. While the contribution of replication to software reuse is problematical, a requirement for replication is motivated by the increasing trend in many domains to support nonstop applications. In addition, high-performance architectures rely on replication for massively parallel execution. Unlike more conventional techniques for improving performance efficiency, replication cannot always be retrofitted into a software architecture. Thus, replication should not be precluded explicitly by the mapping. In the existing mapping, the opportunity to replicate at the OCA level is handicapped by the compilation rules of the original Ada standard.

A mapping that supports *resource sharing* is essential for the OCA implementation. Implicit in the OCA is a requirement for sharing resources and the necessary synchronization to ensure logically consistent execution. Resource sharing is subsumed by the coordination model at the configuration and connection tiers. Since a surrogate is typical of a shared resource, the mapping of the coordination model may specify facilities to guarantee that multiple interactions for surrogate services are not susceptible to race conditions that lead to logically inconsistent execution. For example, a surrogate that encapsulates a single thread-of-control non-Ada application must synchronize execution of the application's components when called concurrently. Typically, this encapsulation is required for integrating legacy components and commercial-off-the-shelf (COTS) software products, such as database and graphical user interface systems, into an application using a pattern-based wrapper integration approach.[2] Frequently, past experience has indicated that using COTS products in a multi-threaded environment results in the unwanted blocking of concurrent threads; allowing such products to be isolated as a surrogate ensures that such blocking does not disrupt time-critical threads.

A mapping that supports *separation* emphasizes the distinction among objects through rigorously defined interfaces. Only through separation is it practical to develop complex applications for distributed object execution and to organize development of large-scale applications. Fundamental to the OCA is the classification and separation of objects at different tiers of reuse. Ideally, a mapping should promote systematic separation in the context of object classification where the commonality and differences among objects are clearly delineated within a type class hierarchy. In addition, exploiting opportunities to refine separation using a hierarchy of generic compilation units facilitates the subsequent control, management, and reuse of applications.

A mapping that supports *uniform composition* gains leverage from the composability properties of objects to achieve object reuse. The uniform composition of objects is an OCA principle that must be reflected in the mapping. Similar to separation, uniform composition is ideally facilitated in the context of object classification, whereby new or enhanced capability may be obtained from systematically refining a type-class hierarchy. In addition, the mapping of the configuration and connection tiers should exploit opportunities for dynamically composing capabilities.

Finally, a mapping should promote the use of *encapsulation* to "layer" abstractions. "Layering" new abstractions upon existing abstractions intuitively increases software reuse. Recently a study has been conducted that indicates substantial improvements in both cost and quality when such "layering" is supported in a disciplined manner [Zweben 95].

---

2. For more information on this topic, see the paper by Diane Mularz, "Pattern-based Integration Architectures," in the proceedings of the Pattern Languages for Programs (PLoP'94) Conference, to be published by Addison Wesley.

5.2 Components and Connectors

The ability to characterize entities within a software architecture in terms of whether they are components or connectors is a popular distinction between an architectural description language and a programming language [Shaw 94b]. The first-class status of connectors in an architectural description language is emphasized by including requirements to define semantics for connectors and their compositions; to generalize interconnectivity rules to address asymmetry, locality, and multiplicity; and to establish type classes for system configuration and organization.

Programming languages are poorly equipped to describe connector semantics because of their traditional focus upon implementing functionality rather than controlling and mediating the interactions of independently composed objects. For example, in the current mapping, the import/export paradigm is insufficient to express a variety of possible interconnections. However, until there is community consensus on a common architectural description language, architectural mappings should attempt to emulate connectors using programming language constructs. This is particularly important in the OCA, which depends upon the composition and interconnection of objects and patterns for implementing applications within a specific domain.

The OCA facilitates developing patterns of composition and interconnection through its three tiers of reuse. Compositional patterns may be stipulated by the mapping and checked by the compilation tool set. In contrast, interconnectivity patterns are more difficult to stipulate and check, since they involve roles and relationships rather than algorithms and data structures. This difficulty may be partly ameliorated by viewing connectors as objects of an abstract data type whose operations are governed by a set of rules (or protocol) and restrictions that guides their use (for example, by specifying a particular pattern of operation calls and the corresponding objects that may be connected through these calls). Unfortunately, it is unlikely that a mapping can enforce such rules for calling patterns with the same guarantees that are available for parameter type checking of a conventional call.

Emulating a connector requires that its abstract behavior be understood independently of how it is used in the mapping. For example, this would allow the behavior of a controller to be understood regardless of the object managers that it interconnects. In addition, a controller should allow the permitted patterns of composition and interconnection to vary within the boundaries of the type classes of the connection objects that are used. However, such direct emulation of controllers as pure connection objects is beyond the scope of the revised mapping.

While a formal definition of connectors and appropriate descriptive notations for them are subjects of ongoing research [Allen 94], exploiting the notion of connector-like templates by the mapping may achieve a more disciplined coordination model.

5.3 Distributed Objects

Recently, the capability to employ a more object-based or object-oriented approach for interconnecting strongly typed objects in a distributed execution environment has emerged. Objects that claim to exemplify this capability have been termed *distributed objects* [Gargaro 95] in Ada 95 and *network objects* [Birrell 94] in Modula-3. The premise underlying such objects is that their operations may be called when the caller (client) is located remotely from the callee (server). Moreover, there should be no semantic difference (other than performance) between the results of a remote call that connects objects in different address spaces and a local call that connects the same objects in a single address space.

Because a strictly object-based or object-oriented approach prohibits clients from accessing server state directly, the necessity to call an operation promotes type-safe disciplined communication within a distributed execution environment. This requires support for inter-address space communication, light-weight threads (tasks), object type-classes with single inheritance, user-defined storage reclamation, and object marshaling to and unmarshaling from data streams. Figure 5-1 illustrates the common notion of the distributed object paradigm. A client uses (or calls) an object through operations made available by the interface of the type class for the object. Because references to the object are to a potentially different address space from that of the client, calls to its operations are considered to be dynamically bound remote calls to the server address space where the object is located. A server declares objects of the type class and provides implementations for each of the operations inherited by the different object types from the type-class interface; thus, there is an implicit dependency of each operation on the object and where it is declared.

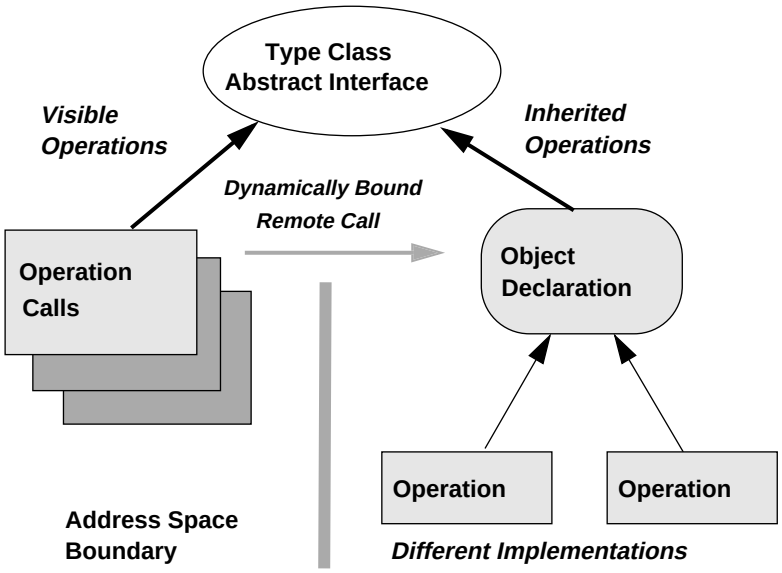


Figure 5-1: Distributed Object Paradigm

In Ada 95, a distributed object is an object whose services, implemented as primitive subprograms, can interconnect clients and servers that are executing in different address spaces and on different computers. The type of a distributed object must be declared as a *tagged type* in an invariant state package together with its primitive subprograms. A *remote access type* designating the corresponding class-wide type, declared in a package to which a remote *categorization* pragma applies, allows the primitive subprograms to be remotely *dispatched* (called) by dereferencing values of this access type. Both packages are restricted to avoid uncontrolled remote references and to prevent passing semantically inconsistent parameter values as subprogram arguments; for example, a local access value may not be passed as an argument. The object is created by the server and a remote access value designating the object is made available to clients. Typically, remote access values are passed to clients through an *object request broker* facility that may be application specific. A client simply dereferences the remote access value as the *controlling operand* of a dispatching primitive subprogram to achieve a remote call. Unlike the statically bound conventional remote procedure call paradigm, calls to distributed objects are dynamically bound; moreover, if there are no output or result parameters, the subprograms may be called asynchronously.

Distributed objects capitalize on the object-oriented support provided by tagged types. Objects of types within the derivative class of a tagged type may use the normal extension and override features to harmonize with the location where the objects are declared. This offers more flexibility in a distributed application than a nondistributed application because of the potential variations in execution environments. For example, a service provided by an object may depend upon the local resources available to the partition in which the object is to be elaborated. In such instances, the inherited subprogram for this service must be overridden by a subprogram that is commensurate with the available resources, and an object of the corresponding derived type elaborated.

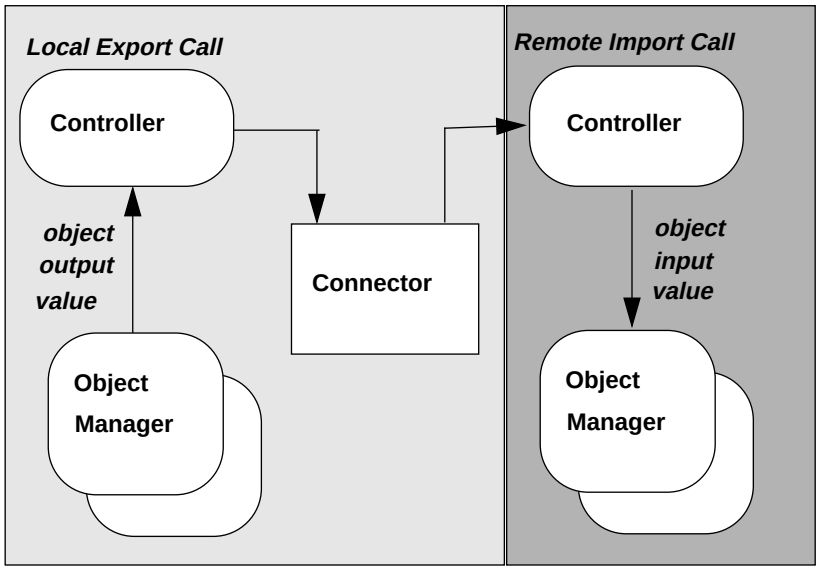


Figure 5-2: Remote Connector Template

Distributed objects may be used as remote connectors to achieve a more disciplined coordination model. For example, one such use is to map the import and export abstractions to distributed objects as depicted in Figure 5-2. In the figure, two subsystems (controller and dependent object managers) reside in different address spaces (denoted by the shaded rectangles). The exporting controller creates a locally accessible connector and makes it available to importing controllers through a remotely called subprogram. Thus, an output value from some object manager service may be exported to the connector using a local export call (i.e., a normal subprogram call). Conversely, an input value to some object manager service may be imported from the connector using a remote import call (i.e., a remote subprogram call).

An advantage of using distributed objects in the OCA mapping is that it permits the same kind of analysis and reasoning to be applied in constructing an application regardless of whether the application is ultimately configured in a distributed or nondistributed execution environment. Also, the use of distributed objects in a mapping reflects the potential loose coupling of subsystems.

5.4 Shared Objects

Typically a shared object is accessible across a distributed execution environment without regard to its locality. This convenience of access is moderated by restrictions on the type of the objects and where the objects may be created. The motivation for shared objects is to support commonly used paradigms for data exchange in distributed execution environments where the cost of remote communication must be avoided. This implies that the address space of the shared object is reachable from all references without relying upon some underlying (and perhaps hidden) communications protocol. Ideally, a shared object is accessed with the same efficiency as a locally declared object.

There is no architectural principle that requires the use of shared objects in a mapping. In fact, the permissive access promoted by such objects would seem to contravene the precepts of a disciplined coordination model. However, it is recognized that many contemporary distributed applications (e.g., AVSM) rely upon multiprogramming when configured for tightly coupled execution environments. In such environments, synchronized access to shared objects is an efficient and safe approach for exchanging data among components that execute independently. For example, in a straightforward adaptation of the current OCA mapping to Ada 95 there is the opportunity to use shared objects for the import and export abstractions. The respective templates would be specified as *shared passive packages* and assigned to a *passive partition.*[3]

3. The intent of a passive partition is to represent a shared address space across one or more computers on which the application is configured. A shared passive package enforces compilation time restrictions that preclude a passive partition from having any runtime system dependencies or separate thread-of-control.

A mapping for subsystems to exchange data through a shared object must accommodate the potentially independent execution of subsystems on different computers. This requires that the coordination model incorporate some scheme for accessing shared objects. In Ada 95 this may be accomplished by declaring the object to be of a *protected type*; execution of each subprogram of the type is synchronized to serialize access to the shared object. Thus, a subprogram call to import or export a value is blocked until any in-progress call is completed, perhaps requiring some form of spin lock when calls originate from different computers.

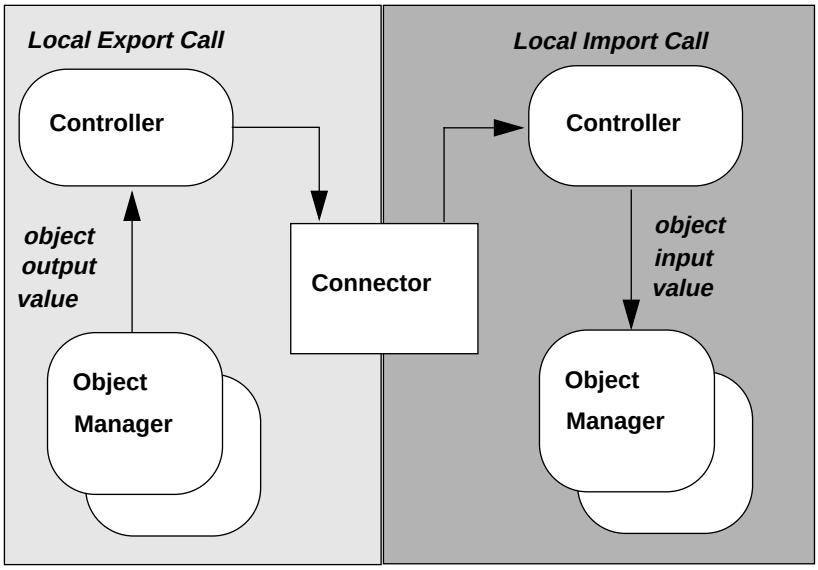


Figure 5-3: Local Connector Template

Figure 5-3 depicts a local connector template as a shared object. The abstraction is similar to the remote connector template except that the connector is shown as being locally accessible to both the exporting and importing subsystem controllers. In contrast to a remote connector, the input value to the object manager service is imported using a local import call.

6 Overview of Mapping to Ada 95

The mapping of the structural abstractions to Ada 95 templates may be materialized using an abstraction-based, an object-based, or a distributed-object style mapping. In the revised mapping, a combination approach is followed. This allows a systematic transition of the existing mapping to a revised mapping using Ada 95 while implementing the recommendations presented earlier in the report. Important criteria of the revised mapping are to maintain fidelity to the structural abstractions and to avoid changes unless there is evidence of a significant benefit to software reuse.

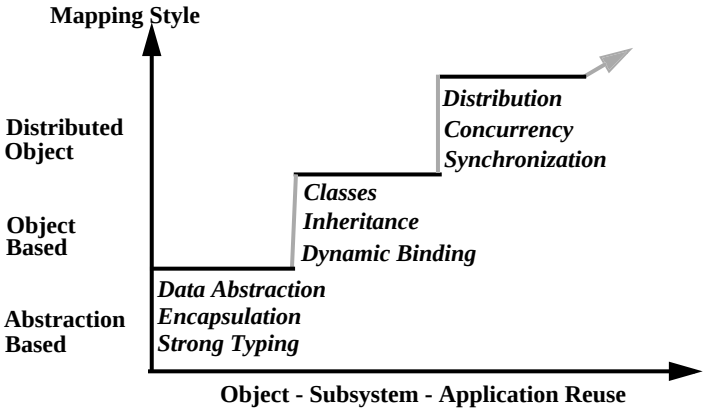


Figure 6-1: Mapping Transition Approach

Figure 6-1 illustrates the different features of the mapping styles that contribute to improving software reuse resulting from the transitional approach. For the purposes of this figure, the original mapping is characterized as an abstraction-based mapping. The object-based features are used to support an enhanced compositional framework, whereas the distributed-object paradigm is used to promote a straightforward partitioning framework and a consistent coordination model. In this figure, it is seen that the transition achieves improvements in object, subsystem, and application levels of reuse.

Object reuse is improved by the introduction and use of type classes, inheritance, and dynamic binding into the composition tier mapping. Each object manager template belongs to a type class that provides an abstract interface specification of the services that may be used at the connection tier. Furthermore, implementations of an object manager may vary by dynamically binding to different objects based upon the features that are supported by the interface.

Subsystem reuse is improved by the introduction and use of distribution, concurrency, and synchronization into the connection tier mapping. Each controller template belongs to a type class that provides an abstract interface specification of the subsystem services that may be used at the configuration tier. Each interface specification is defined to allow the corresponding service to be executed remotely; thus, the unit enclosing a controller and its dependent units are used to form a partition, the Ada 95 unit of distribution. In addition,

connector templates are introduced as generic distributed objects to replace the export/import templates. These templates may be used within a subsystem whenever object managers in different subsystems need to exchange data. Concurrency and synchronization controls to such data maintain the logical consistency of the data. Finally, implementations of a controller may vary by dynamically binding to different object managers based upon the features that are supported by the subsystem interface.

Application reuse is improved by the introduction and use of distribution, concurrency, and synchronization into the configuration tier mapping. The executive provides an interface to the connection tier that allows controllers and connectors to become available for use as distributed objects. Depending upon the product line, the executive configures the subsystems for application execution by controlling the values of the features that are to be supported. These features determine the implementations of the different controllers that are called to provide the services of the corresponding subsystem comprising an application. The ability to distribute subsystems as partitions combined with the capability to support concurrency and synchronization of subsystem services provides a dynamic execution environment that is adaptable to a variety of distributed configurations without changing the executive, controller, connector, or object manager templates.

Unlike the original mapping, the revised mapping adopts a less definitive specification style. This is partially due to the need to address more general-purpose application domains. For example, in the existing mapping, the services associated with objects are characterized in terms of three operations (*Construct, Destruct,* and *Fetc*h); whereas in the revised mapping the services associated with objects may be characterized in terms of many different unspecified operations unless the objects have a specific architectural role, such as connector objects. Similarly, the revised mapping allows more flexibility in naming and structuring the templates comprising an application, while remaining consistent with the architectural style of the product line. This flexibility promotes a more systematic approach to adapting abstract object interfaces to domain-specific attributes and features. Moreover, it emphasizes that the organization of and patterns provided by the templates are essential contributions to supporting a reusable implementation architecture. Figure 6-2 summarizes the Ada 95 features used in the revised mapping to achieve this transitional approach.

For each structural abstraction (listed vertically),[1] the supporting Ada 95 features (listed horizontally) are identified by a numeral denoting the type of support rendered. The features are subdivided to show their association with the three OCA axes of Figure 3-1. In many instances, an abstraction is supported by a combination of features; however, only the principal feature support is included in this figure. A brief explanation of the support denoted by each numeral is as follows:

1. The Export/Import abstractions have been recast into Connectors as discussed in Chapter 5.

.

	Compositional Infrastructure		Partitioning Framework		Coordination Model	
Structural Abstractions	**Class Wide Types**	**Abstract Subprograms**	**Child Units**	**Partition Model**	**Protected Types**	**Remote Subprogram Calls**
Objects	**1**					
Managers	**2**	**5**	**8**			
Controllers	**3**	**6**	**8**	**9**		**12**
Connectors	**4**	**7**	**8**	**10**	**11**	**12**
Executive			**8**			**13**

Figure 6-2: Ada 95 Features Used in Revised Mapping

1. Class-wide types (or more accurately, type-class hierarchies) facilitate encapsulating legacy software as reusable objects at the compositional tier to provide different implementations of common application functionality.
2. Class-wide types facilitate the construction of reusable object managers that are independent of any external state and the implementation of their constituent objects.
3. Class-wide types facilitate the construction of reusable controllers that may select implementations of their constituent object managers corresponding to the variations (or different features) of common application functionality.
4. Class-wide types facilitate the construction of connector type classes to implement export and import templates.
5. Abstract subprograms specify common interfaces to object manager type-classes.
6. Abstract subprograms specify common interfaces to controller type classes.
7. Abstract subprograms specify the interface to the connector type class.
8. Child units encapsulate the declaration of each object manager, controller, and connector type. In addition, they organize and structure the templates.

9. Partitions facilitate the distributed execution of subsystems. They comprise the library units of a subsystem, one of which must declare a controller type object.

10. Partitions facilitate the location of local and remote connector objects for the distributed execution of an application.

11. Protected types provide a default coordination scheme for local and remote connector objects.

12. Remote calls facilitate the use of dynamically bound calls from the executive to controllers and from controllers to connectors in a distributed environment.

13. Remote calls facilitate statically bound calls from controllers to the executive in a distributed environment. Such calls are typically used to configure and reconfigure an application.

It is not possible within the scope of this report to present the complete templates for each tier of reuse. The following sections outline informally the mapping approach for each tier. In Appendices A - C, the corresponding templates for each tier are presented. These templates specify the compositional infrastructure, partitioning framework, and coordination model, and provide a basis for understanding the critical revisions to the original mapping.

6.1 Composition Tier

The composition tier of the OCA mapping supports object-level reuse through the templates shown in Figure 6-3. Common object manager functionality is specified in terms of the services that an object manager provides to the connection tier. The services form an abstract interface of the type class defining the object manager. Any object of a type derived from this type class inherits this abstract interface.

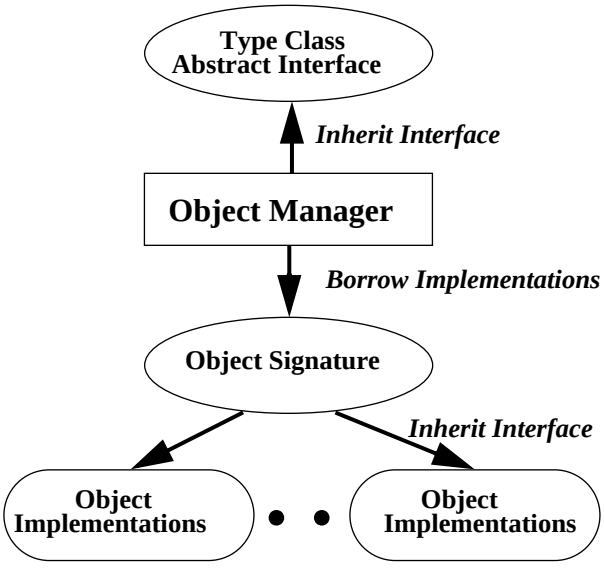


Figure 6-3: Mapping of the Composition Tier

Each service of an object manager may have one or more implementations that represent the variability of the different objects that are required to implement the functionality provided by the service. These different objects, in turn, represent common functionality previously defined in the domain. The differences in granularity of reuse between objects and object managers depend upon the particular models used to analyze the domain. For the purposes of the mapping, it is assumed that the objects exist in some programmatic form that allows them to be included as a part of the object manager. For example, if an object exists in the form of an Ada package, then the functionality is readily included; whereas if the object exists in some form of a class construct of C++, then the object must be placed within an Ada wrapper.

The object implementations template provides the means for including objects into object managers through an abstract interface that is inherited from a signatures type class declared in the object signatures template.When an object is to be used by an object manager, an abstract interface must be included in the object signatures template to represent the object's services. This interface must be specified in a way that is consistent with the features that are available to describe domain variability for object reuse.Thus, it is through the object signatures that the object manager may *borrow* the implementations of the objects necessary to provide the functionality inherited from the template for the type-class abstract interface.

Borrowing an implementation through the use of the object signatures template is the key aspect of the composition tier mapping that supports object reuse. For each object that may be included in an object manager, there must be a function that maps the object's functionality into a set of one or more features that correspond to well-defined variability among the different object implementations. In this way, an object manager may include objects that provide the required functionality defined by its abstract interface.

For example, an object manager that is designed to write data to a communications network may use different objects depending upon the quality of service required. The features for quality of service are defined in the signatures package together with the corresponding abstract interface for the objects. The object manager borrows the implementation that provides the required service by creating an object through the mapping function and the features corresponding to the object implementation.

6.2 Connection Tier

The connection tier of the OCA mapping supports subsystem-level reuse through the templates shown in Figure 6-4. Common subsystem functionality is specified in terms of the services that a subsystem provides to the configuration tier. The services form an abstract interface for the type classes defining controller and connectors. Any controller or connector object of a type derived from these type classes inherits the corresponding abstract interface.

.

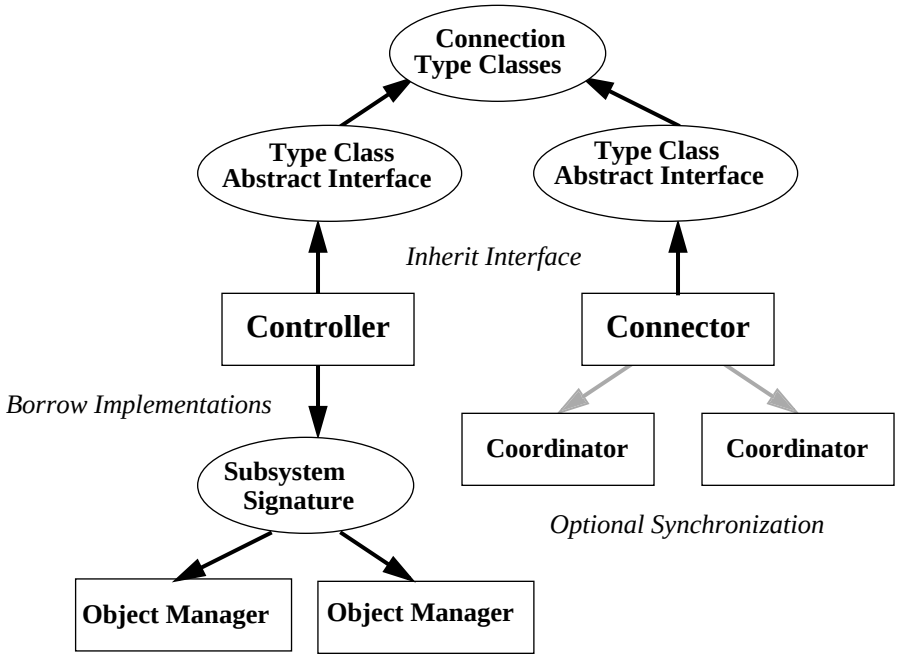


Figure 6-4: Mapping of the Connection Tier

The template for connection type-classes allows both controller and connector objects to be referenced at the configuration tier without requiring explicit knowledge of their types. This is essential since the executive is responsible for accepting the different controllers and connectors for use as distributed objects irrespective of their type-class abstract interface.

The connection type classes do not define an abstract interface since objects of this type are never called to perform a subsystem service. All subsystem services are performed using the type-class interface inherited by the controller template. Only when a subsystem service is to be called does it become necessary to recognize the type class for the interface.

The abstract interfaces for controllers are specified in a manner that allows for distributed processing, such that calls to the corresponding implementation of their services are remote subprogram calls. In this way, a controller object may execute in a different address space from where the call was made. Thus, it is possible for each subsystem to execute in separate address spaces. For example, if the application is configured on a network of computers, each subsystem may execute on different computers.

Similar to the composition tier mapping, controllers borrow implementations of object managers through the subsystem signatures template. However, in this case the signatures template does not provide an inherited interface as it simply depends on any object manager needed by the subsystems. When a controller wishes to call the services of an object

manager, the features controlling the variability of the object manager's implementation are used to borrow the corresponding implementation using a mapping function in the signatures template. Thus, the controllers are not required to depend upon the individual object manager templates but only upon the subsystem signatures template.

The abstract interfaces for connector templates are defined to allow for the exchange of data among subsystems. Connectors may be specified to support different exchange protocols and data types identified by the domain models. Only the simple connector templates described in Sections 5.3. and 5.4 are included in the revised mapping. Each connector template may optionally depend upon a coordinator template when the exchange of data is to be insulated from concurrent access. Coordinator templates may be used to embed any required synchronization protocol into a connector implementation.

6.3 Configuration Tier

The configuration tier of the OCA mapping supports application-level reuse through the templates shown in Figure 6-5. Application-level functionality is achieved by calling the services of the controllers for the corresponding reusable subsystems comprising the application. Similar to the other two tiers, the implementations of the subsystem controller are borrowed through the use of the application signatures template.

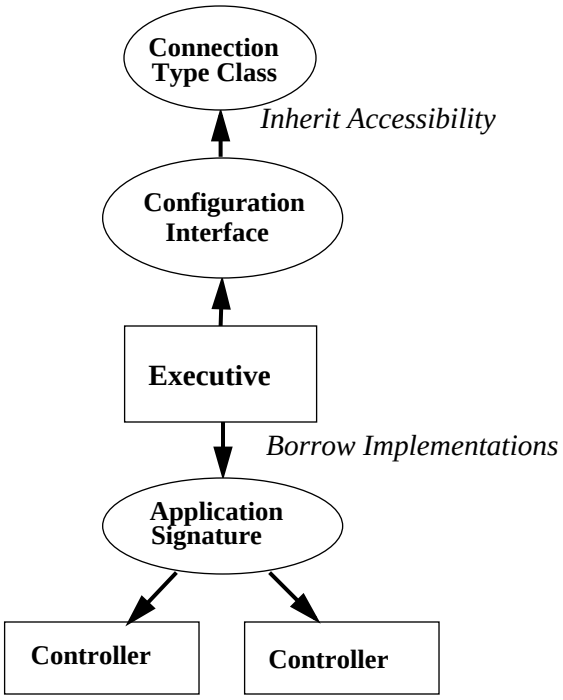


Figure 6-5: Mapping of the Configuration Tier

The application signatures package identifies the subsystems and connectors for the application. However, unlike the other two tiers, borrowing a controller implementation provides a more responsive computing environment than dynamically binding to the implementation's object managers directly. In the case of the configuration tier, requirements for software reuse must be accompanied with requirements for application reliability and fault tolerance.

The executive template specifies a configuration interface that depends upon the connection type class from the connection tier. This configuration interface is called by the subsystem controllers to advertise their availability and non-availability to the executive. In addition, this interface is used to notify the executive that a subsystem has created a connector that may be accessed by other subsystems. Since the connection type class provides the type class through which the controllers and connectors are accessed, the configuration interface is said to inherit accessibility from the connection type class.

Because the configuration interface is called from subsystem controllers that may be located in different address spaces (or partitions) from the executive, the services specified in the interface are callable remotely. The executive is responsible for managing these calls to establish and maintain the application's configuration. For example, unless the necessary subsystems have called the executive informing it of their availability, the executive may defer commencing execution of the application. Similarly, if a subsystem becomes unavailable, the executive may suspend or change the mode of an application to reflect this condition.

Whenever a subsystem requires data from another subsystem, it must have access to a connector for the corresponding data type. A subsystem gains access to a connector using the configuration tier interface. When requesting access to a connector, a subsystem may specify that the connector be from the same subsystem as a previously accessed connector so that all data are obtained from the same subsystem. In contrast, a subsystem may request access to more than one connector for the same data type and import data values from different subsystems in the event that different subsystems have made available connectors for the same data type.

The sequencing and control of calls for subsystem services by the executive is not specified in the revised mapping since this is regarded as application dependent.

7 Conclusions and Future Work

This report has identified several issues in the existing OCA mapping to Ada. These issues limit the adaptation of the OCA to different domains and execution environments in its role as a generic design model with a domain engineering approach. The two most prominent limitations are the static dependencies imposed by the mapping and the lack of support for distributed execution of systems. Since both of these limitations may be ameliorated by using the object-oriented and partitioning features in Ada 95, a revised mapping to Ada 95 has been presented that pays special attention to exploiting these features.

7.1 Lesson Learned

In addition to the principal objective of transitioning the OCA mapping to Ada 95, the exercise provided some experience and insight into the challenge of using a programming language that supports type-safe object-oriented features for large-scale distributed systems. When the mapping was started, there was little documented experience with any similarly capable language available to guide the work. Thus, the approach was deliberately conservative. Only well-understood features were used and complex feature interactions were avoided. In particular, the use of genericity combined with object-oriented programming was restricted to those parts of the mapping that were not included in the current Ada 83 mapping (i.e., coordinators).

The experience seems to support the expectation by many that Ada 95 will become the preferred programming language for developing and reengineering large-scale applications comprising heterogeneous systems. The ability to develop wrappers for legacy components and allow for them to be accessed across a network of computers appears straightforward for application writers, regardless of whether or not a reusable implementation architecture is specified. Moreover, the retention of a consistent unified semantic model across an application may offer a degree of control and integrity that is appealing to those applications considering the use of other alternatives such as using a mapping of CORBA/IDL to Ada 95 [OMG 95].

One particular issue that is central to the in-progress investigation is the integration of distributed objects within a unified coordination model for the OCA. Although not included in the mapping, different schemes for coordinating access to data that are enforced locally at the object level are possible. A variety of synchronization features are available in Ada to achieve such coordination. However, such schemes currently lack a formality (or paradigm) that allows improved reasoning about and understanding of a unified inter-tier coordination model. For example, in the more involved interactions among distributed objects, a local synchronization imposed by one object may conflict with that of another object, leading to possible compromises to the coordination among the architectural tiers of the application. Moreover, since the principal motivation for the OCA is to promote reusable components, it is desirable

that coordination contracts be clearly expressed in the specification of all templates, rather than embedded in the implementation of the template for a distributed object. Ideally, such contracts should be specifiable in a form so that they may be applied (or inherited) throughout the OCA at each tier of reuse.

It is unclear from the in-progress exercise whether or not the above mapping techniques can be enhanced to allow for wider application in the OCA. The use of distributed objects may become excessively complex if applied as a mapping extension to represent object manager templates. Nevertheless, in the highly distributed execution environments of the future, the need to extend the composition tier mapping for distributed execution seems likely. The initial results of using distributed objects to enhance the organizational and communication patterns of the OCA are promising. In particular, the capability to specify a framework for integrating reusable objects into subsystems that may be situated across a network of computers removes a deficiency of the original mapping.

The revised mapping does not mandate that applications conform to specific partitioning patterns. Rather the mapping manifests the pattern properties possessed by partitions so that an application may choose a pattern that attains the desired extra-functional qualities in the context of the OCA conceptual model. For example, one instance of an application may choose to map a single subsystem to a partition, while another instance may choose to map multiple subsystems to a partition. The difference in choice depends upon the execution environment. This tends towards a more dynamic view of linguistic support for distributed systems than what may be achievable from an Ada 95 implementation. For example, it suggests the need for a new linguistic construct similar to a partition type [Gargaro 90a, Gargaro 90b] and for configuration facilities such as those specified in Durra [Barbacci 93]. Nevertheless, this view remains an important guideline for implementing more reusable applications and for perhaps influencing the development of Ada 95 tool suites for distributed systems.

In [Shaw 94b], six classes of properties characterizing an ideal architectural description language are postulated; these are

1. composition
2. abstraction
3. reusability
4. configuration
5. heterogeneity
6. analysis

Results from this exercise indicate that, for the first four properties, Ada 95 provides a reasonable substitute until ideal architectural description languages emerge. Only for the analysis property, requiring that “it should be possible to perform rich and varied analyses of architectural descriptions,” is there reason to believe that Ada 95 is seriously deficient. This

suggests that some of the problems with at least one existing language [Shaw 94b] have been remedied. However, a more complete and thorough study is required to substantiate this more optimistic opinion.

In summary, the revised mapping maps subsystems to partitions at the connection tier where controllers import/export data using connectors to exchange data among subsystems. Controllers, and optionally connectors, are managed as distributed objects through the executive at the configuration tier. At the composition tier, class-wide types are used to provide an abstract interface through which object managers encapsulate and make available the services of reusable objects according to domain-specific implementation criteria that are specified in the corresponding Signatures.

7.2 Future Work

Once the mapping has been validated through use, there are several areas where future work may be continued. A near-term objective is to prepare an analysis of what actual benefits accrued from the revised mapping together with a comprehensive description of an application developed using the OCA templates.

In addition, there are several more long-term proposals that need to be considered. For example, the OCA currently lacks a formalization that unifies the various intuitive concepts and views presented in this paper. A valuable contribution would be to examine the relationship among the architectural principles, the tiers of reuse, and the required structural abstractions (not necessarily those presented in this paper). A better understanding of this relationship may lead to eliminating some of the informality and conceptual overlap that often occurs in describing the OCA independently of specific OCA templates. Other examples would be the investigation of enhancing the OCA to support Durra-like configuration and reconfiguration capabilities to meet the increasing needs for fault-tolerant safety-critical applications, and the development of application generators using the OCA templates.

Appendix A Composition Tier Mapping

The mapping of the architectural abstractions to Ada 95 templates may be materialized using an abstraction-based mapping, an object-based mapping, or a combination of both. In the following partial mapping, a hybrid approach is proposed. This allows a systematic transition of the existing mapping to Ada 95 and implements the recommendations presented in the body of this report. The specification of the revised mapping has been balanced to encourage subsequent refinements and includes sufficient detail to allow evaluation of both the OCA and Ada 95 as tools that will contribute to developing reusable software architectures. It is likely that many of the rich abstraction, encapsulation, generic, and object capabilities of Ada 95 have yet to be fully exploited in the proposed templates.

The composition tier provides a disciplined method for using domain objects in a manner consistent with the models that describe the commonality and variability of objects comprising the application domain. The method specifies how to develop new, reusable domain-specific objects and how to reuse existing domain objects. Thus, the method may be viewed as supporting either a top-down or bottom-up paradigm for developing reusable objects.

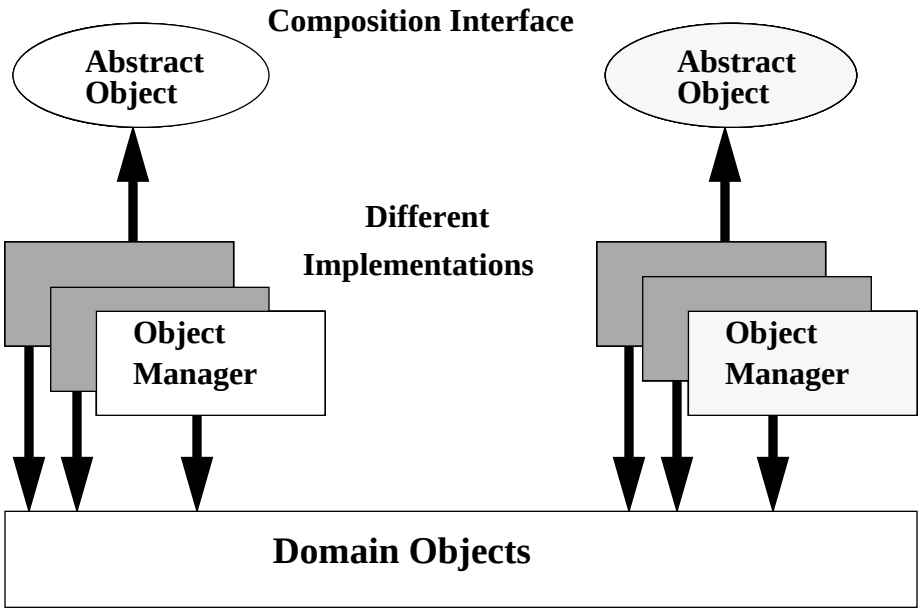
The composition tier comprises object managers. Subsystems, specified in the connection tier, use the services offered by one or more of the object managers via their controllers to perform application-specific actions. For each class of object manager types, a set of abstract services is specified. Through the implementation of these services, object managers are bound to the appropriate domain objects. In this way, object managers become components, where the services they provide reflect the attributes and features of the application domain by using well-understood commonality and variability as leverage to increase the reuse of domain objects.[1]

Each implementation of an object manager type service uses one or more objects together with the necessary abstractions to control and sequence their execution. The synthesis of these objects provides the functionality required by the corresponding abstract services. The objects are not necessarily domain specific and may have one or more different implementations. A necessary requirement is that the objects' services do not compromise the infrastructure of the mapping. Thus, objects may be implemented in different languages providing that their services conform to the language-defined interfacing features of the mapping.

Figure A-1 illustrates an informal overview of the composition tier mapping. In this figure, the composition interface is represented by one or more abstract object types. Different object managers (as denoted by the various shadings) depend upon each type to provide specific implementations using existing objects that are available in the domain.

---

1. Typically, this commonality and variability is formalized and captured in the models resulting from a domain analysis.



A -> B : A *depends on* B

Figure A-1: Composition Tier Dependencies

A.1 Composition Interface Mapping

The composition interface consists of one or more abstract object specifications. An abstract object specification is provided by a type-class from which different object manager types may be derived. The abstract services are declared as primitive subprograms for the corresponding tagged type (i.e., the type of the controlling operand of each primitive subprogram) in a package that follows the structure of the Composition_Interface package. This allows different implementations of the services to be specified by declaring a derived type in the type-class. The form of the Composition_Interface packages is as follows:[2]

```
with ...
package Composition_Interface is
    type Abstract_Object is abstract tagged private;
    type Abstract_Object_Access is access all Abstract_Object'Class;
    procedure Object_Service
        (Control : Abstract_Object; Param : ...; ...) is abstract;
    ...
    Composition_Interface_Error : exception;
```

2. In the following templates, italicized names are used to denote that the named construct is an exemplar of a construct with different names permitted by the mapping; e.g., *Object_Service* denotes one of possibly many differently named subprograms.

```
    ...
private
    type Abstract_Object is abstract tagged null record;
end Composition_Interface;
```

In the above specification there may be one or more abstract subprograms declared to represent the services supplied by the types derived from the type Abstract_Object. Similarly, each primitive subprogram may have one or more parameters, in addition to the mandatory controlling parameter named Control.

A.2 Object Manager Mapping

"*The intent of the Object Manager is to provide a standard mechanism for invoking the operations needed to provide the services for the physical/logical object that is to become a part of a system*." [Peterson 94, page 43]

The abstract subprograms specify the services available through object managers that are of types in the type-class hierarchy derived from Abstract_Object. These services may be called from the connection tier providing that an object of the type is accessible to a controller. Since objects of type Abstract_Object cannot be declared, any object made accessible to the controller must be declared of a type derived from Abstract_Object in a template that corresponds to the package Object_Manager. Furthermore, each abstract subprogram must be overridden. The Object_Manager package is declared as a child package of Composition_Interface as follows:

```
with Composition_Interface.Signatures;
package Composition_Interface.Object_Manager is
    type Object_Manager is new Abstract_Object with private;
    type Object_Manager_Access is access all Object_Manager;
private
    type Object_Manager is new Abstract_Object with record
            Object_A : Signatures.Signature_Access := Signatures.Map(...);
            Object_B : Signatures.Signature_Access := Signatures.Map(...);
            ...
    end record;
    procedure Object_Service
        (Control : Object_Manager; Param : ...);
        . ..
end Composition_Interface.Object_Manager;
```

The specification of Object_Manager depends upon a child package, Signatures; this package is explained in a subsequent paragraph. For the purposes of introducing the implementation approach for object managers, it is sufficient to understand that Signatures declares reusable object types that may be constructed to implement the abstract subprograms. In the private part, the object manager type is extended with components of the reusable object types.

Consequently, each declaration of an object manager creates the necessary reusable objects whose subprograms are to be called by the implementation of its abstract subprograms. Within the body of each primitive subprogram of object manager there are calls to the appropriate objects from the Signatures type-class hierarchy.

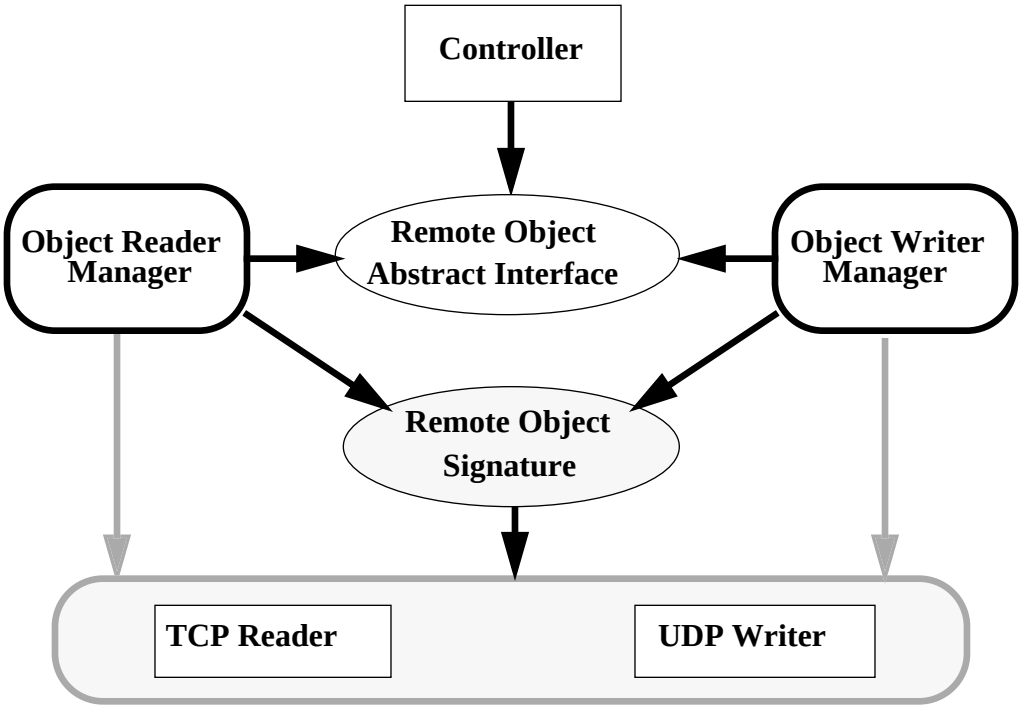
The mapping specifies that object managers are declared in child packages of the corresponding Composition_Interface package. The parent-child composition provides an intuitive continuity between the abstract subprograms and their actual implementations. In addition, there are two reasons for requiring this composition. The first is to ensure that any additional mapping information in the private part of Composition_Interface is visible to the object manager packages. The second is to allow the bodies of object manager packages to depend upon any additional private child packages of Composition_Interface. This allows for subsequent extensions to the partial mapping.

The implementation of the primitive subprograms are provided in the body of Object_Manager as follows:

```
package body Composition_Interface.Object_Manager is
    procedure Object_Service
        (Control : Object_Manager; Param : ...; ...) is
        ...
    begin
        ...
        Signatures.Service (Control.Object_A, ...);
        ...
        Signatures.Service (Control.Object_B, ...);
        ...
    end Object_Service;
    ...
end Composition_Interface.Object_Manager;
```

Depending upon the processing to be performed, Object_Service may use one or more objects from different type-class hierarchies associated with the component extensions. What is left unspecified in the mapping is the actual code that sequences the calls to the various subprograms of these objects; this is beyond the scope of the current mapping. The mapping does not preclude Object_Service from calling its constituent services concurrently; in such instances, Object_Service would include any necessary concurrency control and synchronization.

A consequence of this mapping is that, at the connection tier, a controller may select object implementations from the different object managers simply by referencing an object of a class-wide type that provides the desired implementation. In the course of its execution, a controller may use a combination of different object implementations. For example, as illustrated in Figure A-2, the abstract interface includes a type-class for a remote object type together with read and write services.



A -> B : A *depends on* B
(dashed line denotes implicit dependence)

Figure A-2: Composing an Abstract Interface

From this type, derivate types for object managers may be declared to perform the actual read and write services based upon the features selected through the information in the remote object signature. The features allow the use of different communication protocols to implement the read and write subprograms for the remote object type using reader and writer objects that are assumed to exist as domain objects. Thus, as implied by the figure, a remote object may be read using the Transport Control Protocol (TCP) reader and written using the User Datagram Protocol (UDP) writer depending upon the quality-of-service feature that was specified for reading and writing in the respective object managers. In addition, the subsystem controller need only depend upon the abstract interface of the remote object type and not the specific object managers; this will be explained in Appendix B. Section A.5 describes the composition of an object that corresponds to this figure.

A.3 Signatures Mapping

"For *objects, Signatures consist of information about the details of the functionality, in particular the processing options, provided by the object's operations. Object signatures contain abstract names for services (and their underlying algorithms) provided by the objects, and hide the mapping of the abstract service (and the selected name) to the implementation of the service."* [Peterson 94, page 17]

The Signatures mapping specifies how reusable objects of the domain are encapsulated (or viewed) for use by object managers. In addition, the mapping identifies the features through which different implementations of the objects are made available to object managers using the map subprogram. The specification for a conforming object manager signature is specified as a child package of Composition_Interface. The parent-child composition denotes that the Signatures package applies to all object managers of this type and has the following form.

```
package Composition_Interface.Signatures is
    type Feature_Type is ...;
     ... -- other features
    type Signature_Type is abstract tagged limited private;
    type Signature_Access is access all Signature_Type'Class;
    procedure Service
        (Control : access Signature_Type; ...) is abstract;
     ... -- other services
    function Map
        (Options : Feature_Type; ...) return Signature_Access;
     ... -- other signatures
    Feature_Set_Not_Implemented : exception;
private
    type Signature_Type is abstract tagged limited null record;
    ...
end Composition_Interface.Signatures;
```

In contrast to object managers, there is only a single Signatures package for each Composition_Interface package. For each reusable object of the domain, an abstract type, Signature_Type, is declared together with the corresponding services that are to be called by the object managers. These services are declared as abstract subprograms. Only one abstract type and one subprogram is shown; however, typically there will be many abstract types, each having many subprograms. An exception is defined for valid implementation feature sets for which there is no implementation available.

Associated with the abstract types are the features used to select alternative implementation schemes. The way in which features are specified is not defined in the mapping; the only requirement is that for those combinations of features that determine an implementation of a reusable object, a Map subprogram must be declared to return an object through which the

corresponding subprograms can be called. Thus, unless a Map subprogram is declared with the desired Feature_Types as formal parameters, there is no authorized way in which the corresponding implementation alternative can be called by the object managers. The template for the Signatures body comprises the implementation of the Map subprograms as follows:

```
with Composition_Interface.Implementations;
package body Composition_Interface.Signatures is
    function Map
        (Options : Feature_Type; ...) return Signature_Access is
    begin
        return Implementations.Map(Options, ...);
    exception
        when others =>
            raise Feature_Set_Not_Implemented;
    end Map;
     ...
end Composition_Interface.Signatures;
```

The above template depends upon a private child package, Implementations, as described in the next section. It is through the mapping of Implementations that the fundamental reusable assets of the domain are made accessible to the object managers by calling the Map subprogram. This subprogram in turn calls the corresponding Map subprogram provided by Implementations; if no corresponding subprogram exists then the feature combination has no implementation and an exception is raised.

A.4 Implementations Mapping

The implementation objects accessible to an object manager are mapped through a private child to the Composition_Interface package. Consequently, as a private child, the implementation objects cannot be accessed outside the Composition_Interface and its child unit hierarchy. Each implementation object is mapped as a value of a nonabstract derivative type of a signatures type. The derivative type, Implemented_Type, is extended to include components of those features that are used to select the implementations of the abstract subprograms for the signatures type.

Each of the abstract subprograms declared for the signatures type is overridden with a subprogram of the corresponding implemented type. In addition, a Map subprogram is declared for each combination of features that may be used to select this implemented type; the subprogram profiles must correspond directly to their signatures types counterparts.

```
with Composition_Interface.Signatures;
private package Composition_Interface.Implementations is
    type Implemented_Type is new Signatures.Signatures_Type with private;
    procedure Service(Control : access Implemented_Type; ...);
    ...
    function Map(Options : Signatures.Feature_Type; ...) return Signatures_Access;
```

```
        ... other implemented types
private
    type Implemented_Type is new Signatures.Signatures_Type with
        record
            Feature : Signatures.Feature_Type;
            ...
        end record;
end Composition_Interface.Implementations;

with Domain_Objects, ...;
package body Composition_Interface.Implementations is
    procedure Service
        (Control : access Implemented_Type; ...) is
    begin
        ... use services provided by Domain_Objects
    end Service;
    ...
    function Map
        (Options : Signatures.Feature_Type; ...) return Signature_Access is
    begin
        ...
        return new Implemented_Type'(Signature_Type with Options, ...);
    end Map;
end Composition_Interface.Implementations;
```

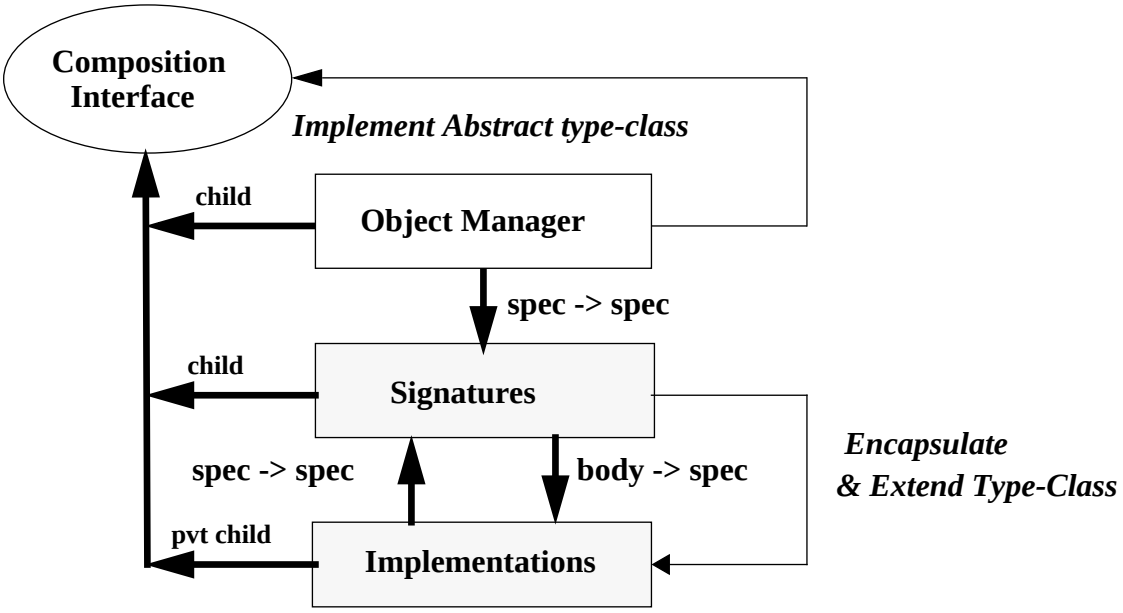


Figure A-3: Composition Tier Conceptual Mapping

Figure A-3 illustrates the various templates and their interdependencies that comprise the composition tier mapping. While not explicitly included in the mapping, additional templates may be specified to compose object managers from other object managers when there is a significant overlap in their implementations.

A.5 Composing an Abstract Object

The various constructs presented in the previous subsections may be applied to develop the example composition illustrated in Figure A-2. To provide additional context, it is assumed that the purpose of the controller is to provide a conversion between objects that are represented by the Abstract Syntax Notation One (ASN.1) protocol and objects that are represented by the External Data Representation (XDR) protocol [Meyers 93]. The example object manager and its signatures are shown below.

```
with System;
package Remote_Object_Interface is
    type Remote_Object is abstract tagged private;
    procedure Read
        (Obj : Remote_Object; Adr : out System.Address; Len : out Positive) is abstract;
    procedure Write
        (Obj : Remote_Object; Adr : System.Address; Len : Positive) is abstract;
    Read_Not_Supported,
    Write_Not_Supported : exception;
private
    type Remote_Object is abstract tagged null record;
end Remote_Object_Interface;
```

The abstract object type Remote_Object is declared in Remote_Object_Interface, one of the packages constituting the composition tier. Two subprograms, Read and Write, are declared through which all object managers derived from this type may be manipulated by controllers.

```
package Remote_Object_Interface.Signatures is
    type Protocol_Type is (ASN, XDR);
    type Transfer_Type is (TCP, UDP);
    type Endpoint_Type is abstract tagged limited private;
    type Endpoint_Type_Access is access all Endpoint_Type'Class;
    function Map
        (Protocol : Protocol_Type; Transfer : Transfer_Type) return Endpoint_Type_Access;
    procedure Read
        (Obj : access Endpoint_Type; Adr : out System.Address; Len : out Positive) is abstract;
    procedure Write
        (Obj : access Endpoint_Type; Adr : System.Address; Len : Positive) is abstract;
    Feature_Set_Not_Implemented : exception;
private
    type Endpoint_Type is abstract tagged limited null record;
end Remote_Object_Interface.Signatures;
```

The child signatures package of the Remote_Object_Interface package describes the features associated with the remote object. In this example, the domain includes two data representation protocols, ASN and XDR, and two transport layer protocols, TCP and UDP. Each remote object is associated with an endpoint to which the different protocols may be assigned. The Map subprogram is declared to obtain a reference to an endpoint message reader/writer with the required protocols.

```
with Remote_Object_Interface.Signatures;
package Remote_Object_Interface.Object_Reader is
    type Reader is new Remote_Object with private;
private
    type Reader is new Remote_Object with record
        Rdr : Signatures.Endpoint_Type_Access
            := Signatures.Map(Signatures.ASN, Signatures.TCP);
    end record;
    procedure Read (Obj : Reader; Adr : out System.Address; Len : out Positive);
    procedure Write (Obj : Reader; Adr : System.Address; Len : Positive);
end Remote_Object_Interface.Object_Reader;
--
with Remote_Object_Interface.Signatures;
package Remote_Object_Interface.Object_Writer is
    type Writer is new Remote_Object with private;
private
    type Writer is new Remote_Object with record
        Wtr : Signatures.Endpoint_Type_Access
            := Signatures.Map(Signatures.XDR, Signatures.UDP);
    end record;
    procedure Read (Obj : Writer; Adr : out System.Address; Len : out Positive);
    procedure Write (Obj : Writer; Adr : System.Address; Len : Positive);
end Remote_Object_Interface.Object_Writer;
```

Two child packages of Remote_Object_Interface declare derivations of Remote_Object as object manager types for reading and writing data. The object manager types, Reader and Writer, are extended with an endpoint type component, show below. The default initialization of the object manager types call the Map subprogram from the Signatures package to create a message reader or writer for the endpoint depending upon the requested features. In this way, object managers may be declared for reading or writing remote objects with different protocols.

```
package body Remote_Object_Interface.Object_Reader is

    procedure Read (Obj : Reader; Adr : out System.Address; Len : out Positive) is
    begin
        Signatures.Read(Obj.Rdr, Adr, Len);
    end Read;
```

```
        procedure Write (Obj : Reader; Adr : System.Address; Len : Positive) is
        begin
            raise Write_Not_Supported;
        end Write;
    end Remote_Object_Interface.Object_Reader;
    --
    package body Remote_Object_Interface.Object_Writer is
        procedure Write (Obj : Writer; Adr : System.Address; Len : Positive) is
        begin
            Signatures.Write(Obj.Wtr, Adr, Len);
        end Write;
        procedure Read (Obj : Writer; Adr : out System.Address; Len : out Positive) is
        begin
            raise Read_Not_Supported;
         end Read;
    end Remote_Object_Interface.Object_Writer;
```

The implementations of Read and Write for the object managers call the corresponding subprograms using their assigned message reader and message writer from the Signatures package. In this example, the message reader may not write, and similarly, the message writer may not read; an attempt to do so raises an exception.

```
    with Remote_Object_Interface.Implementations;
    package body Remote_Object_Interface.Signatures is
        function Map
            (Protocol : Protocol_Type; Transfer : Transfer_Type) return Endpoint_Type_Access is
        begin
            return Implementations.Map(Protocol, Transfer);
        exception
            when others =>
                raise Feature_Set_Not_Implemented;
        end Map;
    end Remote_Object_Interface.Signatures;
```

The implementation of the Map subprogram in the Signatures package body calls the corresponding subprogram from the Implementations package. If there is no implementation available for the requested protocols, an exception is raised.

```
    with Remote_Object_Interface.Signatures;
    use Remote_Object_Interface.Signatures;
    private package Remote_Object_Interface.Implementations is
        type TCP_Type is new Endpoint_Type with private;
        type UDP_Type is new Endpoint_Type with private;
        function Map
            (Protocol : Protocol_Type; Transfer : Transfer_Type) return Endpoint_Type_Access;
    private
```

```
        type TCP_Type is new Endpoint_Type with record
            Protocol : Protocol_Type;
            Transfer : Transfer_Type;
        end record;
        type UDP_Type is new Endpoint_Type with record
            Protocol : Protocol_Type;
            Transfer : Transfer_Type;
        end record;
        procedure Read
            (Obj : access TCP_Type; Adr : out System.Address; Len : out Positive);
        procedure Write
            (Obj : access TCP_Type; Adr : System.Address; Len : Positive);
        procedure Read
             (Obj : access UDP_Type; Adr : out System.Address; Len : out Positive);
        procedure Write
            (Obj : access UDP_Type; Adr : System.Address; Len : Positive);
    end Remote_Object_Interface.Implementations;
```

The implementation of endpoint type is extended to include the protocol and transfer features for the message reader/writer object managers in a private child of Remote_Object_Interface. For each combination of supported features, a specific type is declared with corresponding overriding subprograms. In this example, it is known that readers will use TCP/ASN and writers will use UDP/XDR for this application.

```
    with TCP, UDP, ASN, XDR; -- domain objects
    package body Remote_Object_Interface.Implementations is
        function Map
            (Protocol : Protocol_Type; Transfer : Transfer_Type) return Endpoint_Type_Access is
        begin
            if Protocol = ANS and Transfer = TCP then
                return new TCP_Type'(Endpoint_Type with ASN, TCP);
            elsif Protocol = XDR and Transfer = UDP then
                return new UDP_Type'(Endpoint_Type with UDP, XDR);
            else
                raise Feature_Set_Not_Implemented;
            end if;
        end Map;
        procedure Read
            (Obj : access TCP_Type; Adr : out System.Address; Len : out Positive) is ...
        procedure Write
            (Obj : access TCP_Type; Adr : System.Address; Len : Positive) is ...
        procedure Read
            (Obj : access UDP_Type; Adr : out System.Address; Len : out Positive) is ...
        procedure Write
            (Obj : access UDP_Type; Adr : System.Address; Len : Positive) is ...
    end Remote_Object_Interface.Implementations;
```

The body of the package depends upon the objects from the application domain to implement the supported features: namely, TCP, UDP, ASN, and XDR. The Map subprogram creates and returns a reference to a message reader/writer object. In this example, the components in the endpoint type extension are initialized but are not used. Each overriding endpoint subprogram is assumed to be implemented using the facilities provided by the domain objects. In this way, the corresponding subprograms in the composition interface dispatch to the appropriate implementation using the message reader/writer object to which they have been mapped.

Appendix B Connection Tier

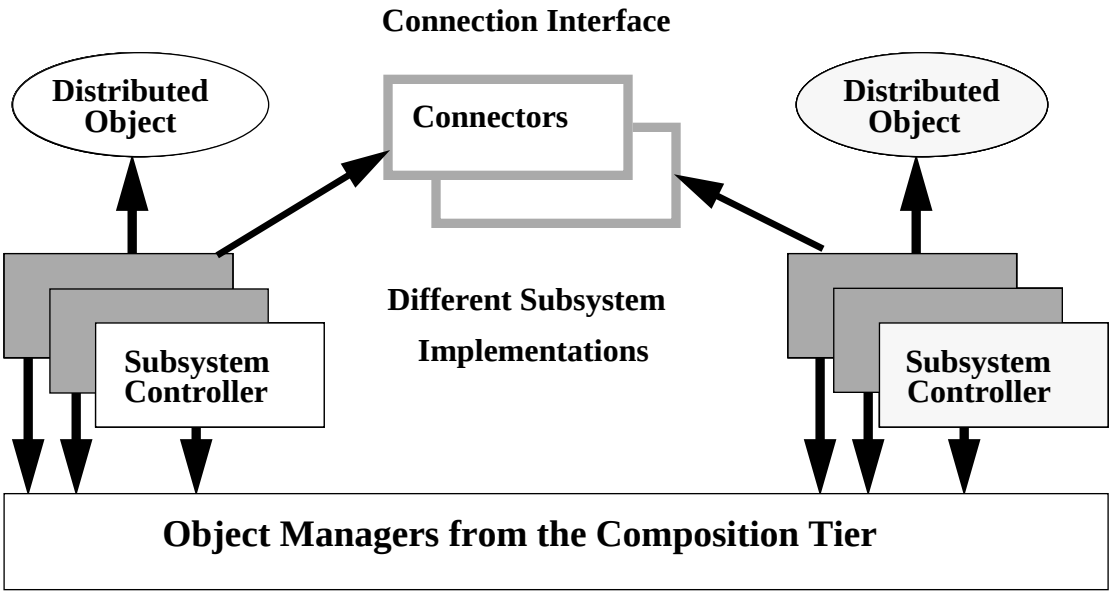
The principal requirement motivating the connection tier is to provide a disciplined method for combining the services of object manager components into reusable abstractions called controllers. Controllers allow a domain application to be modeled as a group of subsystems that may be called through their corresponding interfaces. In a manner similar to that presented in the composition tier, the method allows these interfaces to have multiple implementations reflecting both the functional commonality and variability derived from the domain models.

The connection tier comprises templates for controllers, connectors, and coordinators. A fundamental property of cooperating subsystems is that there is no direct interaction among their respective controllers. All data are exchanged among subsystems through a connector abstraction whose interface includes the essential Export and Import services. The connector template provides support for connecting subsystems using either *remote* or *shared* connectors. This is explained in Section B.4. An additional abstraction, termed a coordinator, is provided by the revised mapping to facilitate coordination within a subsystem. This is explained in Section B.5.

It is at this tier of reuse that the notion of distributed objects, as described in the body of this report, is introduced to allow an application to be partitioned across a network of processing resources. Each subsystem interface is inherited from a controller type-class that may be used to declare controllers as distributed objects. The abstract interface associated with the controller type-class provides a subsystem interface for which multiple implementations may exist. Each controller type is declared within a library package to encapsulate the implementation. This library package, together with its semantic dependencies, forms the subsystem.

Figure B-1 informally illustrates the conceptual framework of the connection tier. In this illustration, the subsystem interfaces are shown as distributed objects declared with different controller implementations (denoted by the shading of the rectangle). Each of these implementations is constructed from one or more object managers from the composition tier. Connectors appearing between the different subsystems illustrate their role as conduits for disciplined data exchange between subsystems; the use of broken lines conveys the dynamism inherent in their use.

Consistent with the properties of a distributed object, the services specified in the abstract interface of a controller type-class may be called from an address space (or processing resource) that is different from where the call originates. Hence, a controller object provides a flexible approach to situating subsystems on a network of processing resources. The mapping for subsystems requires that their constituent object managers be situated in the same address space (or processing resource) because object managers referenced by a controller must be named explicitly as semantic dependencies and their services called using normal (local) subprogram calls.



A -> B : A *depends on* B

Figure B-1: Connection Tier Dependencies

B.1 Connection Interface Mapping

The mapping for the connection interface comprises the type-classes for subsystem controllers and connectors. The form of the Connection_Interface package is as follows:

```
package Connection_Interface is
    pragma Pure;
    type Connector_Id is range ...;
    type Subsystem_Id is range ...;
    type Subsystem_Connector
        (Connector : Connector_Id) is abstract tagged limited private;
    type Subsystem_Controller
        (Subsystem : Subsystem_Id) is abstract tagged limited private;
    Connection_Interface_Error : exception;
private
    type Subsystem_Connector
        (Connector : Connector_Id) is abstract tagged limited null record;
    type Subsystem_Controller
        (Subsystem : Subsystem_Id) is abstract tagged limited null record;
end Connection_Interface;
```

Through these type declarations, different subsystem controller and connector types are derived in child packages. The configuration tier depends upon this package to provide configuration services for controllers and connectors; this is explained in Appendix C.

B.2 Subsystem Controller Mapping

"*The Subsystem Controller provides a uniform interface to the underlying capabilities of the subsystem.*" [Peterson 94, page 45]

The abstract services provided by a type-class for a controller are declared as primitive subprograms of an abstract type derived from the Subsystem_Controller type. This type is declared in a remote types package together with a general access type designating a class-wide type. A consequence of this mapping is to allow all subsystem services to be dynamically bound using remote access values as controlling operands. The Subsystem package is declared as a child package of the Connection_Interface as follows:

```
package Connection_Interface.Subsystem is
    pragma Remote_Types;
    type Abstract_Controller is new abstract Subsystem_Controller with private;
    type Controller_Access is access all Abstract_Controller'Class;
    procedure Subsystem_Service
        (Control : access Abstract_Controller; Param : ...) is abstract;
    ...
private
    type Abstract_Controller is new abstract Subsystem_Controller with null record;
end Connection_Interface.Subsystem;
```

In the above specification, there may be one or more abstract subprograms declared to represent the services supplied by the types derived from the type Abstract_Controller. Similarly, each primitive subprogram may have one or more parameters, in addition to the mandatory controlling access parameter.

These services may be called from the configuration tier providing that a remote access value designating an object of the declared type is accessible to the executive. Since objects of type Abstract_Controller cannot be declared, any object made accessible through a remote access value must be declared of a type derived from Abstract_Controller in a package that corresponds to the package Controller. Furthermore, each inherited abstract subprogram must be overridden. The Controller package is declared as a child package of Subsystem as follows:

```
package Connection_Interface.Subsystem.Controller is
    pragma Elaborate_Body(Controller);
end Connection_Interface.Subsystem.Controller;

with Composition_Interface;
with Configuration_Interface.Executive;
with Connection_Interface.Subsystem.Signatures;
package body Connection_Interface.Subsystem.Controller is
    ... subsystem state
    type Controller is new Abstract_Controller with record
```

```
        Object_Manager : Composition_Interface.Abstract_Object_Access
            := Signatures.Map(...);
        ... other object managers
    end record;
    procedure Subsystem_Service (Control : access Controller; Param : ...);
        ...
    Controller_Instance : aliased Controller;
    Controller_Instance_Access : Controller_Access := Controller_Instance'Access;
    procedure Subsystem_Service (Control : access Controller; Param : ...) is
    begin
        Composition_Interface.Object_Service(Control.Object_Manager, Value, ...);
        -- calls to other object managers' services
    end Subsystem_Service;
    ...
begin
    -- Promote subsystem
end Connection_Interface.Subsystem.Controller;
```

Unlike the corresponding Object_Manager package of the composition tier, the Controller package specification is empty. This is a direct consequence of the desire to declare a controller type that supports the notion of a distributed object. The type and the distributed object, Controller_Instance, are encapsulated in the package body so that references to the object must be provided through some means other than mentioning the package as a semantic dependency of some other unit.

In the mapping, this reference is achieved through the services provided by the configuration tier. When the package elaborates, it calls the Promote_Subsystem service of the Configuration_Interface.Executive to establish that there is a remote access value available to reference the Controller_Instance. The exact details of promoting distributed objects are deferred until the discussion of the configuration tier.

The encapsulation of a single distributed object within a package provides a mapping of a controller to a subsystem. The subsystem state is maintained for each subsystem by the execution of the services provided by its controller. Neither the state nor the subprograms providing the services are accessible outside of the subsystem without explicitly requesting the remote access value that designates the controller, since the implementation cannot be called without the explicit action of the subsystem to make its controller object available. Thus, the package Controller and the library units on which it semantically depends comprise an instance of a subsystem that may be used to construct a partition that may be freely situated on any one processing resource. In this way, the package hierarchies that are composed from the Connection_Interface may be considered isomorphic to a partition; namely that each hierarchy is well formed so that its services may be called remotely and it may be replicated as a partition within a distributed application.

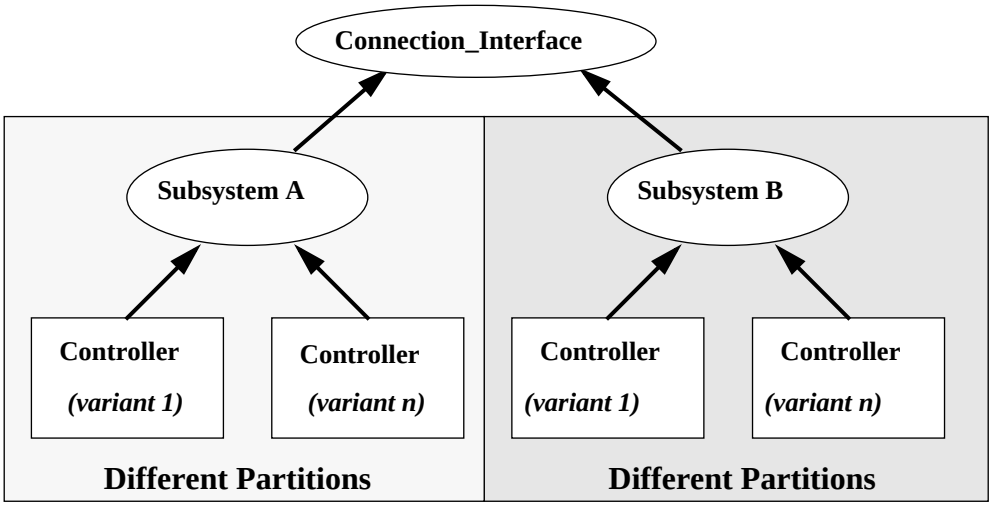


Figure B-2: Partition Mapping

Figure B-2 illustrates the package hierarchy and indicates that the Connection_Interface and subsystem packages may be safely referenced in any partition (since they are categorized as pure and remote types packages respectively). In this figure, each controller variation is identified as a partition (however, mapping more than one controller to a partition is not precluded); the shaded areas denote different processing resources.

Similar to the way in which an object manager type was extended in the composition tier, each controller type is extended with the object managers on which its subprograms depend.This is achieved through the use of a child package Signatures that is explained in the subsequent paragraph. Within the body of each primitive subprogram of controller there are calls to the appropriate object managers' subprograms that have been made available through the Signatures package and the Composition_Interface. Thus, it is unnecessary for the Controller package to depend explicitly on each object manager that it requires for its implementation; only the Composition_Interface and the Signatures packages are required.

Depending on the processing to be performed, Subsystem_Service may use one or more of the object managers included in the extended controller type. Again, what is left unspecified in the mapping is the actual code that sequences the calls to the various subprograms of the object managers.

B.3 Subsystem Signatures Mapping

"*Subsystem signatures allow the use of a subsystem's operations without explicit reference to (or knowledge of) the underlying objects comprising it, the specific features they provide, or even the names that have been chosen to represent the services at the object level*." [Peterson 94, page 17]

The Signatures mapping identifies the features through which object managers and their different implementations are made available to a subsystem controller. The specification of a conforming subsystem signature is declared as a child package of its corresponding Subsystem package; thus denoting that it applies to all Controller packages (instances of subsystems). The specification has the following form:

```
with Composition_Interface, ...
package Connection_Interface.Subsystem.Signatures is
    type Features_Type is ...
    ... -- other features
    function Map
        (Options : Feature_Type; ...) return Composition_Interface.Abstract_Object_Access;
    ... -- other signatures
    Feature_Set_Not_Implemented : exception;
end Connection_Interface.Subsystem.Signatures;
```

While the above template declares only a single feature type, typically the Signatures package will declare as many feature types as required to support composing a subsystem controller from reusable object managers. An object manager is selected by associating each object manager implementation with a set of features corresponding to the services that it offers through the Composition_Interface. In contrast to object manager signatures, only features that select different implementations of an object manager are declared since different object managers must be referenced through mentioning the appropriate Composition_Interface as a semantic dependency in the context clause.

The reason for this “weaker” feature orientation in the Signatures package, when compared to the Signatures package of the composition tier, is that the specifications of the Composition_Interfaces are developed explicitly to reflect abstract services based upon the features from the domain models. Thus, the subsystem controllers are presented with a carefully specified set of reusable object managers in contrast to the less reusable objects that may be presented to the object managers.

The sets of features that may be used to select instances of an object manager are used to specify the formal parameters of a Map subprogram; this subprogram returns an object through which the subprograms of the object manager may be called. When different implementations exist for an object manager, the implementation variations can be parameterized through an appropriate set of feature types. In this way, there is no reference nor knowledge required by the subsystem controllers about the composition of an object manager. Hence, unless a Map subprogram is declared with the desired feature types as formal parameters, there is no authorized way in which the corresponding implementation can be called by the subsystem controllers. The implementations of the Map subprograms are declared in the Signatures package body as follows:

```
with Composition_Interface.Object_Manager, ...
package body Connection_Interface.Subsystem.Signatures is
```

```
        function Map
            (Options : Feature_Type; ...) return Composition_Interface.Abstract_Object_Access is
        begin
            return new Composition_Interface.Object_Manager.Object_Manager;
        end Map;
        ...
    end Connection_Interface.Subsystem.Signatures;
```

The Signatures package body depends upon each object manager that is made available through the declared features. The Map implementation simply creates an object of the appropriate Object_Manager type; this allows controller subprograms to use different implementations of object managers using the Composition_Interface since the controller type includes in its extension the selected implementations.

B.4 Subsystem Connectors Mapping

"*The import structure defines the interface for data input from other subsystems. ... The export structure defines the data output interface for use by other subsystems."* [Peterson 94, page 16]

The abstract services provided by a type-class for a connector are declared as primitive subprograms of an abstract type derived from the Subsystem_Connector type; these are the Import and Export subprograms. Two type-classes are declared to support both *remote* and *shared* connectors. Through these two type-classes, subsystems may exchange data either using the distributed object or shared object templates depending upon how their corresponding partitions are to be configured to the available processing and storage resources.[3] The mapping does not prescribe the conditions under which these templates are used since the choice may depend upon environmental and operational factors that are beyond the information included in the domain models.

The connector mapping provides two generic child packages of Connection_Interface named Remote_Connection and Shared_Connection. Each package declares a type Abstract_Connector with two mandatory primitive subprograms Import and Export as follows:

```
generic
    type Data_Type is private;
package Connection_Interface.Remote_Connection is
    pragma Remote_Types;
    type Abstract_Connector is abstract new Subsystem_Connector with private;
    type Connector_Access is access all Abstract_Connector'Class;
    function Import
```

3. Typically, the configuration of subsystems to partitions will not be known when an application is developed. Thus, when this knowledge is unavailable, it is best to assume that the subsystems are to be configured in partitions that do not have access to a passive partition containing a shared object template.

```
        (Cxn : access Abstract_Connector) return Data_Type is abstract;
    procedure Export
        (Cxn : in out Abstract_Connector; Value : Data_Type) is abstract;
private
    type Abstract_Connector is abstract new Subsystem_Connector with null record;
end Connection_Interface.Remote_Connection;

generic
    type Data_Type is private;
package Connection_Interface.Shared_Connection is
    pragma Shared_Passive;
    type Abstract_Connector is abstract new Subsystem_Connector with null record;
    type Connector_Access is access all Abstract_Connector;
    function Import
        (Cxn : access Abstract_Connector) return Data_Type is abstract;
    procedure Export
        (Cxn : access Abstract_Connector; Value : Data_Type) is abstract;
end Connection_Interface.Shared_Connection;
```

The generic forms of the packages allow the declaration of the type-classes to support the exchange of different data types through the declaration of the generic formal type parameter, Data_Type. Each instantiation provides an access type that enables a subsystem to import data of that type.

Both forms include categorization pragmas. Similar to the use of the Remote_Types pragma in the *Subsystem* package, the presence of the pragma in the Remote_Connection package allows a general access type designating a class-wide type, Abstract_Connector'Class, to be declared so that the connector services may be dynamically bound using remote access values. Thus, connectors may be implemented as distributed objects. A subsystem may create from an instantiated package a connector object and make its corresponding remote access value available to other subsystems through the configuration tier so that objects of the data type may be imported independently of where the subsystems are situated.

In contrast, the presence of the pragma Shared_Passive in the Shared_Connection package allows a general access type designating Abstract_Connector to be declared so that connector services may be dynamically bound using access values that are globally accessible to more than one subsystem. Thus, connectors may be implemented as shared objects by using an access value to a nonabstract connector type and by making this value available to other subsystems through the configuration tier. Objects of the data type may be imported providing that the subsystems have access to the shared passive package where the access type is declared. The requirement for global accessibility is achieved by assigning instantiations of Shared_Connection packages to passive partitions. Typically, passive partitions provide globally accessible data throughout a distributed application. Consequently, shared connectors are similar to the import and export abstractions that are specified for non-distributed applications in the original mapping.

An important distinction between remote and shared connectors is that both Export and Import subprograms for a shared connector may be called through its corresponding access value, while for a remote connector only the Import subprogram may be called through its corresponding remote access value. This distinction reflects the fact that a remote connector is situated with the subsystem controller that creates the connector and that calls to Export are always local calls (refer to Figure 6-2). In contrast, shared connectors are not situated with the subsystem controller that creates them; this is a consequence of declaring shared connectors or allocating shared connectors from access types that are declared in a shared passive package. A shared connector is not included in a subsystem, whereas a remote connector is included in a subsystem in the form of a distributed object.

In the above templates, both connector types are abstract types, thereby precluding the declaration of objects of these types. The purpose of the types is to establish the remote and shared properties that are to be inherited in their respective type-class hierarchies and to provide a type-class that is available to client subsystems wishing to import data. Type hierarchies are declared in generic child packages and provide the implementations for the Export and Import subprograms by declaring nonabstract connector types. Through different connector implementations, subsystems may exchange data compatible with the requirements of a particular application. In addition, subsystems are not limited to obtaining data of a given type from a single subsystem. Data may be imported from any connector for that type that is accessible to a subsystem. Consequently, connectors may be viewed as first-class objects [Shaw 94a] since they support, in a sense, the software glue that is essential for achieving subsystem reuse.

Different implementations for connectors are instantiated from generic child packages of Remote_Connection and Shared_Connection. The mapping provides default packages; however, other packages may be declared providing that they conform to the same requirements as the default packages (namely, that data are exchanged using the same subprograms). In the first set of default packages to be presented, the data object to be exported/imported is encapsulated within a protected type to ensure that access to the data is synchronized. Thus, whether a connector is remote or shared, the same synchronization is performed. Subsequently, an alternate set of packages is described that allows more flexibility in specifying how access to the data is synchronized. For example, it may be inappropriate to enforce the same synchronization upon both a remote and shared connector since limitations on what may be supported for a shared connector may be unacceptable for a remote connector. The form of this default set of generic package specifications is as follows:

```
generic
package Connection_Interface.Remote_Connection.Default_Implementation is
    type Connector is new Abstract_Connector with private;
    function Import (Cxn : access Connector) return Data_Type;
    procedure Export (Cxn : in out Connector; Value : Data_Type);
private
    protected type Cache is
        procedure Export (Value : Data_Type);
```

```
            function Import return Data_Type;
        private
            Content : Data_Type;
        end Cache;
        type Connector is new Abstract_Connector with record
            Synchronize : Cache;
        end record;
end Connection_Interface.Remote_Connection.Default_Implementation;
--
generic
package Connection_Interface.Shared_Connection.Default_Implementation is
        pragma Shared_Passive;
        type Connector is new Abstract_Connector with private;
        type Connector_Access is access all Connector;
        function Import (Cxn : access Connector) return Data_Type;
        procedure Export (Cxn : access Connector; Value : Data_Type);
private
        protected type Cache is
            procedure Export (Value : Data_Type);
            function Import return Data_Type;
        private
            Content : Data_Type;
        end Cache;
        type Connector is new Abstract_Connector with record
            Synchronize : Cache;
        end record;
end Connection_Interface.Shared_Connection.Default_Implementation;
```

The above templates are declared as generic packages, although there are no formal parameters declared. There are two reasons for this: (1) generic packages may have only generic child packages; and (2) frequently, as will be shown later, it will be useful to include generic formal parameters to tailor the implementations for connectors. It should be noted that the generic formal parameter of the parent may be referenced within the child unit as if it were declared as a formal parameter of the child package.

Each package derives a nonabstract type extension from the connector type-class and encapsulates within a protected type component the data object that is associated with the connector. This protected type precludes exporting a value if there is an import in progress; however, values may be exported concurrently. This restricted course-grained synchronization follows from the language restrictions necessary to safeguard the preelaboration guarantee of a shared passive package. These restrictions disallow the use of more capable constructs such as entry queues. However, as a default implementation, the entryless protected type provides the minimum safety essential for a reusable connector type. In addition to the extended type, a general access type is declared in the child of Shared_Connection (which must be a shared passive package), designating the extended

type so that objects of the type may be dynamically allocated and references to them managed through the configuration tier.

The creation of a connector type-class hierarchy requires a two-step instantiation process. Initially the abstract type-class must be declared for the particular data type, and then a suitable implementation of the connector must be instantiated as a child of the first instantiation. For example, if an Integer data type (with the default implementation) is to be exchanged among subsystems using a shared connector template, it would be necessary to complete the following two instantiations:

```
with Connection_Interface.Shared_Connection;
package Local_Integer_Connection is new Shared_Connection(Integer);
pragma Shared_Passive (Local_Integer_Connection);

with Connection_Interface.Shared_Connection.Default_Implementation;
package Local_Integer_Connection.Default
    is new Local_Integer_Connection.Default_Implementation;
pragma Shared_Passive (Local_Integer_Connection.Default);
```

The first instantiation, Local_Integer_Connection, is to be used by the importing client subsystem, whereas the second instantiation is to be used by the exporting server subsystem. The bodies for the remote and shared packages are identical except for the difference resulting from the Export subprogram profile. The body for the default implementation of a shared connector is as follows:

```
package body Connection_Interface.Shared_Connection.Default_Implementation is
    protected body Cache is
        procedure Export (Value : Data_Type) is
        begin
            Value := Content;
        end Export;
        function Import return Data_Type is
        begin
            return Content;
        end Import;
    end Cache;
    function Import (Cxn : access Connector) return Data_Type is
    begin
        return Cxn.Synchronize.Import;
    end Import;
    procedure Export (Cxn : access Connector; Value : Data_Type) is
    begin
        Cxn.Synchronize.Export(Value);
    end Export;
end Connection_Interface.Shared_Connection.Default_Implementation;
```

B.5 Subsystem Coordinators Mapping

"*In the OCA, the coordination model is realized in rules and templates that determine how the building blocks interact with one another*." [Peterson 94, page 13]

The coordination services for subsystems are provided through the Initialize and Finalize subprograms of a limited controlled type, Coordinator, that is declared in the child package Coordination of Connection_Interface. Execution of these subprograms results in the specified coordination to be performed using the primitive subprograms of a class-wide access discriminant Agent associated with the type Coordinator. Both controllers and connectors may use coordinators to implement a coordination scheme that is appropriate to the services that are supported. In this way, a subsystem may coordinate concurrent calls from the configuration tier, and connectors may synchronize access to data that are to be exchanged among subsystems. This coordination is performed within a subsystem and not across subsystems; the latter form of coordination is provided through the configuration tier.

The mapping does not explicitly specify the manner in which a controller or connector uses a coordinator since this depends upon the kind of service or services supported. For example, in some instances there is no need for coordination. The assumption is that when coordination is required, the controller or connector type may be extended to include an Agent component to achieve coordination. Subsequently, the use of a coordinator in the implementation of a connector is shown as an example of how the previously specified default synchronization scheme may be replaced by a generic synchronization scheme. A coordinator type is derived from the predefined Ada controlled type in a child package Coordination as follows:

```
with Ada.Finalization; use Ada.Finalization;
package Connection_Interface.Coordination is
    type Agent is abstract tagged limited private;
    type Coordinator
        (Agent : access Agent'Class) is new Limited_Controlled with null record;
private
    procedure Engage (Opn : access Agent) is abstract;
    procedure Disengage (Opn : access Agent) is abstract;
    procedure Initialize (Cdr : in out Coordinator);
    procedure Finalize (Cdr : in out Coordinator);
end Connection_Interface.Coordination;
```

Because the coordinator type is a derivative of Limited_Controlled, there is a language guarantee that objects of the type are initialized (using Initialize) upon creation and are Finalized (using Finalize) upon reclamation. This guarantee is the key to achieving coordination; however, it introduces an irregularity in the mapping in that the coordinator type cannot be declared as an abstract type in the Connection_Interface package. In the above specification, the Agent discriminant provides two abstract coordination services, Engage and Disengage, that are called in the bodies of Initialize and Finalize. Whenever coordination is required by a controller or connector service, the respective type includes a suitable Agent

component whose coordination services are called by simply declaring a local coordinator in the service to be coordinated. The declaration of a local coordinator guarantees that the Engage coordination is performed during elaboration of the subprogram body and that the Disengage coordination is performed prior to leaving the scope of the local coordinator. A minimal implementation of the Coordination package body has the following form:

```
package body Connection_Interface.Coordination is
    procedure Initialize (Cdr : in out Coordinator) is
    begin
        Engage (Cdr.Agent);
    end Initialize;
    procedure Finalize (Cdr : in out Coordinator) is
    begin
        Disengage (Cdr.Agent);
    end Finalize;
end Connection_Interface.Coordination;
```

In the case of the connector Import and Export subprograms, a coordinator might be used as follows to implement a remote connector (a language restriction on the semantic dependencies of a shared passive package precludes its use to implement a shared connector). An example Remote_Connection package with coordination is given below:

```
with Connection_Interface.Coordination; use Connection_Interface.Coordination;
generic
    type Scheme is new Agent with private;
package Connection_Interface.Remote_Connection.Coordinated_Implementation is
    type Connector is new Abstract_Connector with private;
    function Import (Cxn : access Connector) return Data_Type;
    procedure Export (Cxn : in out Connector; Value : Data_Type);
private
    type Connector is new Abstract_Connector with record
        Content : Data_Type;
        Coordinate : aliased Scheme;
    end record;
end Connection_Interface.Remote_Connection.Coordinated_Implementation;


package body Connection_Interface.Remote_Connection.Coordinated_Implementation is

    function Import (Cxn : access Connector) return Data_Type is
        Synchronize : Coordinator (Cxn.Coordinate'Access);
     begin
        return Cxn.Content;
     end Import;
```

```
        procedure Export (Cxn : in out Connector; Value : Data_Type) is
            Synchronize : Coordinator (Cxn.Coordinate'Access);
        begin
            Cxn.Content := Value;
        end Export;
    end Connection_Interface.Remote_Connection.Coordinated_Implementation;
```

The above template provides a number of advantages compared with the earlier default implementation for a shared connector. The primary advantage is that it allows coordination schemes to be inherited and their implementations parameterized independently of the services that are to be coordinated. For example, in the case of a connector, the coordination is separated from the data type that is associated with the connector. In this way, the coordination scheme can be modified without requiring changes to the services. The template specifies a formal parameter Scheme as the coordination scheme to be employed. Since it is a type in the Agent type-class the abstract coordination subprograms, Engage and Disengage, may be implemented as required.

It is important to illustrate the specification of how a coordination scheme is implemented using this template, although this is not strictly an integral part of the mapping. In the following example, a coordination scheme is used that is similar to the synchronization of the default implementation specified for a shared connector.

```
package Connection_Interface.Coordination.Coordination_Scheme is
    type Agent_Wrapper is new Agent with private;
private
    protected type Synchronize is
        entry Block;
        procedure Unblock;
    private
        The_Barrier : Boolean := False;
    end Synchronize;
    type Agent_Wrapper is new Agent with record
        Scheme : Synchronize;
    end record;
    procedure Engage (Opn : access Agent_Wrapper);
    procedure Disengage (Opn : access Agent_Wrapper);
end Connection_Interface.Coordination.Coordination_Scheme;
```

The type Agent_Wrapper provides a convenient means to encapsulate a protected object that implements the synchronization using a simple block/unblock scheme. This type may now be used to instantiate the generic unit Coordinated_Implementation as follows:

```
with Connection_Interface.Remote_Connection;
package Remote_Integer_Connection is new Remote_Connection(Integer);
pragma Remote_Types;
```

```
with Connection_Interface.Coordination.Coordination_Scheme;
with Connection_Interface.Remote_Connection.Coordinated_Implementation;
package Remote_Integer_Connection.Coordinated
    is new Remote_Integer_Connection.Coordinated_Implementation
        (Scheme => Connection_Interface.Coordination.Coordination_Scheme.Agent_Wrapper);
pragma Remote_Types;
```

The above instantiations are similar to the ones previously presented for a shared connector. The only difference is in the second instantiation that creates a remote connector; it has been parameterized with a coordination scheme to provide a synchronization scheme that is equivalent to that embedded in the shared connector. This synchronization is achieved by providing an appropriate body for the package Coordination_Scheme as follows:

```
package body Connection_Interface.Coordination.Coordination_Scheme is
    protected body Synchronize is
    -- block/unblock implementations
    end Synchronize;
    procedure Engage (Opn : access Agent_Wrapper) is
    begin
        Opn.Scheme.Block;
    end Engage;
    procedure Disengage (Opn : access Agent_Wrapper) is
    begin
        Opn.Scheme.Unblock;
    end Disengage;
end Connection_Interface.Coordination.Coordination_Scheme;
```

An advantage of using this template is that the coordination scheme providing the synchronization is associated with the type of the object rather than the object services. Consequently, all types within a connector type-class hierarchy may have overridden subprograms that are synchronized consistently. When there are many services, such as is likely for controllers, associating synchronization with the type facilitates a potentially more uniform coordination model.

B.6 Summary of the Connection Tier Mapping

The preceding sections have presented the connection tier incrementally in terms of the controller, connector, and coordinator objects that are specified in the partial mapping. The informal illustration of the mapping in Figure B-1 may be refined to show more accurately the various templates specified in the mapping and their interdependencies.

In Figure B-3, the partial mapping dependencies are focused upon the role of the controller abstraction. The distinction between remote and shared connectors is hidden to simplify the figure.

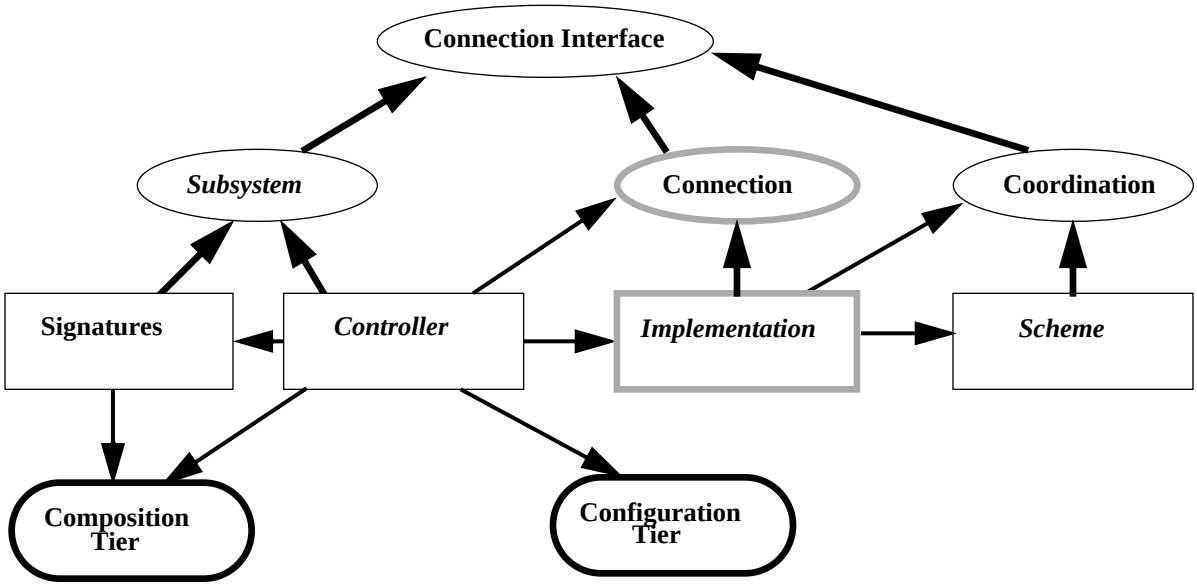


Figure B-3: Dependencies Between the Connection Tier Templates

The dashed lines indicate a generic template where the controller is dependent upon an instance of the template rather than the generic template itself. For example, in order for a controller to export data, an instance of the required Implementation for the appropriate type must be referenced. Whereas, in order for a controller to import data, only an instance of the required connector for the appropriate data type must be referenced. In this latter case, the importation of data should be unconcerned with where data are located or what specific implementation has been used for the connector.

Appendix C Configuration Tier

"*We also maintain that the major source of developing applications for heterogeneous machines is not implementing the basic data operations, which are hidden in the task's code, but rather in making use of available resources: loading and executing programs in the different processors, routing data, reconfiguring the application etc*." [Barbacci 93]

The principal requirement of the configuration tier is to provide a disciplined method for coordinating and sequencing the services of reusable subsystems to build domain specific applications. This requirement is supplemented by derived requirements to (1) support the configuration and reconfiguration of subsystem controllers and connectors, (2) improve both processing resource utilization and fault tolerance, and (3) provide start-up and termination of an application. Unlike the existing mapping, specific patterns for coordinating subsystems are not specified; instead, a more flexible approach is adopted that allows the mapping to exploit the full range of Ada 95 capabilities for coordinating and sequencing a multi-threaded (task) environment. It is beyond the scope of the revised mapping to specify the numerous options that are available for coordinating subsystems since such coordination is domain dependent.

The configuration tier mapping allows this coordination to be achieved in a way that is consistent with the connection tier mapping; namely that different subsystem implementations may be used without changing the executive or subsystem controllers. This consistency is achieved by providing a set of services to the connection tier through which the executive obtains the necessary information to accomplish the desired subsystem utilization. The following paragraphs outline the essential services provided by the configuration tier interface. Subsequent sections explain these services in more detail and describe the corresponding templates.

In order for the services of a subsystem to be called at the configuration tier, the controller associated with the subsystem must register and promote itself by communicating with the executive; this action is termed *promoting* a subsystem. Once a subsystem is promoted, the controller object (a remote access value designating the controller) may be used within the executive to call its corresponding services. A subsystem may choose to suspend or terminate its availability within an application, in which case it unregisters itself with the executive; this action is termed *demoting* a subsystem. Depending upon the implementation of the controller, a subsystem may promote and demote itself as required. The services of a demoted subsystem may not be called at the configuration tier since the associated partition may have terminated; alternatively, a subsystem may have detected some internal condition requiring that its services be temporarily suspended (e.g., while it performs storage reclamation).

Similarly, in order for the Import service of a connector to be called by different subsystems, the controller creating the connector must register the connector with the executive; this action is termed *publicizing* a connector. A complementary action termed *privatizing* a connector unregisters a publicized connector; in this way, a controller may limit the subsystems having access to the connector. When a subsystem requires data from another subsystem (i.e., it

needs to import data), it requests a connector object from the executive; this action is termed *realizing* a connector. Once a connector object has been realized it may be used to import data by the controller of the subsystem that is originating the action. There is no requirement that a subsystem be promoted in order to publicize or realize a connector. For example, it is permissible for an application to be developed where there is no need for a particular subsystem's services;[4] that is, the subsystems execute independently of the executive except to publicize and realize connectors.

Figure C-1 illustrates an example of interaction between configuration and connection tiers.

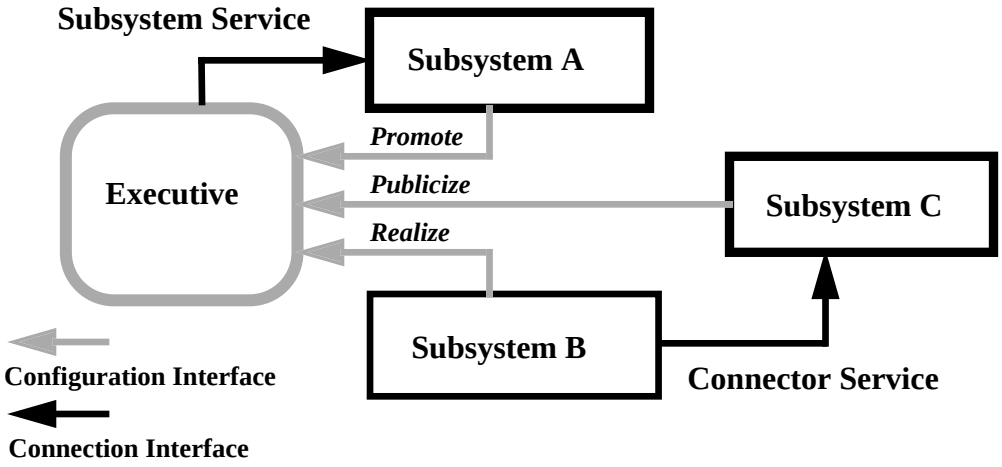


Figure C-1: Interaction Between Configuration & Connection Tiers

In this figure, three subsystems are shown calling the configuration interface to the executive. One subsystem promotes itself, one subsystem publicizes a connector, and one subsystem realizes a connector. As a consequence, the only allowable connection interface calls are the ones shown. The executive may call the subsystem services of the controller for the promoted Subsystem A and the connector service (Import) may be called by the controller of Subsystem B that realized the publicized connector by Subsystem C (it is assumed that the publicized and realized connectors are type compatible in this instance).

C.1 Configuration Tier Interface Mapping

The template for the configuration interface comprises remote class-wide access types designating objects in the subsystem controller and connector type-classes. Values of the declared types may therefore reference any distributed object declared of a type within the corresponding derived type-class hierarchy. In addition, for consistency of presentation,

4. Alternatively, a subsystem interface may have no services declared in its abstract specification.

System.Address is renamed for the purposes of designating shared objects; it is important to note that this declaration requires that package System be implemented as a Pure package so that it may have an invariant state. The form of the Configuration_Interface package is as follows:

```
with System; -- package System is pragma Pure
with Connection_Interface; use Connection_Interface;
package Configuration_Interface is
    pragma Remote_Types;
    type Promoted_Access is access all Subsystem_Controller'Class;
    type Publicized_Access is access all Subsystem_Connector'Class;
    subtype Shared_Access is System.Address;
    Configuration_Interface_Error : exception;
end Configuration_Interface;
```

Using these type declarations, an appropriate interface is specified through which subsystem controllers may call the executive.

C.2 Executive Mapping

"*The executive provides the operating environment for the subsystems within the application and, in most cases, is the arbitrator over conflicts between processes competing for time and access to shared resources*." [Peterson 94, page 19]

The services provided by the executive are declared in a remote call interface package as statically bound remote subprograms. The services support promoting and demoting subsystem controllers, and publicizing, privatizing, and realizing connectors. The following generic child package executive outlines a specification of this interface:

```
generic -- domain specific parameterization
package Configuration_Interface.Executive is
    pragma Remote_Call_Interface;
    type Partition_Id is range ...;
    function Register_Subsystem
        (Name : Subsystem_Name) return Subsystem_Id;
    procedure Promote_Subsystem
        (Controller : Promoted_Access; Subsystem : Subsystem_Id; Partition : Partition_Id);
    procedure Demote_Subsystem (Controller : Promoted_Access);
    -- Remote connector support
    function Register_Connector
        (Name : Connector_Name) return Connector_Id;
    procedure Publicize_Connector
        (Connector : Publicized_Access; Connection : Connector_Id; Partition : Partition_Id);
    function Realize_Connector
        (Name : Connector_Name; Inherit : Publicized_Access := null) return Publicized_Access;
    procedure Privatize_Connector (Connector : Publicized_Access);
```

```
        -- Local connector support
        procedure Publicize_Connector
            (Connector : Shared_Access; Connection : Connector_Id; Partition : Partition_Id);
        function Realize_Connector
            (Name : Connector_Name; Inherit : Shared_Access := 0) return Shared_Access;
        procedure Privatize_Connector (Connector : Shared_Access);
        ... -- other domain specific services
        Invalid_Connector,
        Invalid_Subsystem : exception;
    end Configuration_Interface.Executive;
```

The executive is declared as a generic unit to allow multiple instantiations. No formal parameterization is specified so that only identical copies of the executive may be executed. This permits configuration tier redundancy when required by fault-tolerant applications. Since a single instantiation of a remote call interface package may be assigned to only one partition, the implementation of subsystem controllers must name the executive(s) instantiations on which they semantically depend. As a consequence, calls to the services provided by the executive are statically bound; thus, unlike controllers, instances of executives are not declared as distributed objects.

The fact that the executive is not a distributed object is fundamental to the mapping. This allows an application to be started by initiating the executive once its partition has completed elaboration with the assurance that the statically bound remote calls for its services will not require intervention of an intermediary partition. In this way, the executive services provide a straightforward approach for bootstrapping a distributed application. Each subsystem controller is allowed by the mapping to promote itself and publicize connectors to which it exports data once subsystem elaboration has completed.

When a subsystem promotes itself, it must provide to the executive some form of identification. Unless this identification is present, there is no convenient means for the executive to determine from which subsystem the call originated. Thus, prior to a subsystem controller promoting itself, it must register with the executive by calling Register_Subsystem. The subsystem provides in this call the name of the subsystem (from the application Signatures package) and receives a Subsystem_Id that corresponds to the given subsystem name. In this way, different instances for the same subsystem may be managed by the executive. Each instance of the subsystem controller is declared with its associated discriminant providing the corresponding Subsystem_Id. In order to declare an instance of a controller, a valid Subsystem_Id value must be specified; otherwise the subsystem cannot be promoted.

The specification for the Promote_Subsystem subprogram includes a formal parameter of Subsystem_Id. Values of this type (when the attribute ‘Value is applied) correspond to enumeration literals denoting the names of subsystems. These enumeration literals are similar to the corresponding specification technique used in the original mapping and are declared in the application Signatures package. In this way, there is no forced dependency on the different subsystems; the executive depends only upon the abstract controller. If a

subsystem promotes itself and is not specified in the Signatures package, then ‘Value fails and an exception is raised. When there are multiple instances of the same subsystem, calls to Promote_Subsystem from each instance will provide the same identity; however, the value of the controller object will differ.

In order to allow the executive to dynamically determine the application configuration in terms of subsystems assigned to partitions, the specification for the Promote_Subsystem subprogram includes a formal parameter of type Partition_Id. Partition_Id is declared in the Executive package and facilitates the executive supporting application-specific requirements that may depend upon available processing resources. It is expected that values of Partition_Id will be consistent with the values returned by the Ada 95 ‘Partition_Id attribute. The following example illustrates how a subsystem controller may promote and register itself during its elaboration.

```
with Configuration_Interface.Executive; ...
package body Connection_Interface.Subsystem.Controller is
    type Controller is ...
    Controller_Instance : aliased Controller (Subsystem =>
            Executive.Register_Subsystem (“This_Subsystem”);
    Controller_Instance_Access : Controller_Access := Controller_Instance’Access;
    ...
begin
    Executive.Promote_Subsystem
        (Configuration_Interface.Promoted_Access (Controller_Instance_Access),
        Controller_Instance.Subsystem, -- identity of the subsystem in which Controller is declared
        Connection_Interface.Subsystem.Controller’Partition_Id);
end Connection_Interface.Subsystem.Controller;
```

Once a subsystem is elaborated, the subsystem services may be called by the executive using the value of the controller operand as a controlling operand to the appropriate abstract subprograms. It should be noted that each subsystem must be declared with a different name; the string “This_Subsystem” in the example is used only to illustrate a call to Register_Subsystem. Moreover, the technique through which the executive associates multiple instances of the same subsystem to the enumeration literal denoting the subsystem identity is not specified since this level of detail is beyond the scope of the mapping. However, the three parameters of the Promote_Subsystem call provide sufficient data for managing multiple instances of subsystem controllers.

Similar to registering subsystems, the executive includes a subprogram declaration to support registering connectors. A subsystem registers a connector by calling Register_Connector and providing the connector’s name; the returned Connector_Id may be used in subsequent calls to publicize this connector. The Connector_Id must be used as the discriminant value to declare a connector object that corresponds to the registered name.

Three overloaded services are declared in the Executive to support publicizing, privatizing, and realizing local and remote connectors. Publicize_Connector must be called whenever a subsystem controller wishes to permit exported data to be imported by other subsystem controllers. The parameters to the call include the access type (or System.Address) designating the connector, the connector identification (obtained from the discriminant of the designated object), and the Partition_Id of the partition in which the connector is located. Conversely, Privatize_Connector must be called when it is determined by the application that no further subsystem controllers may import data from a connector.

A connector is made accessible to import data through a call to the Realize_Connector service. The call includes the connector identification and an optional Inherit operand.The connector identification allows for the appropriate type and implementation of a connector to be specified, while the Inherit operand specifies a previously realized connector. This previously realized connector, when present, requires that the returned connector must have been publicized by a subsystem controller from the same partition (typically this will be the same subsystem controller). In this way, the executive may ensure that, when necessary, the selected imported data items originate from the same subsystem (or partition); this may be required if the data made available by different connectors are interrelated or depend upon a common state.

The mapping for connectors distinguishes between remote and shared connectors by using values of the remote access type Publicized_Access or Shared_Access respectively. The use of a remote access type has been explained previously as an abstraction to implement distributed objects. In contrast, the use of Shared_Access (i.e., System.Address) is a mechanism, rather than an abstraction, to allow subsystem controllers to publicize access values designating shared connectors, since it is unacceptable to use the access type that designates the connector.[5] As a consequence, normal type safety is forfeited by the use of System.Address. Values of the access type designating a local connector are converted to and from System.Address using an appropriate instantiation of System.Address_To_Access_Conversions.

C.3 Application Signatures Mapping

"*The Applications Signatures package is the top-level namespace for the application to be built.*" [Peterson 94, page 51]

The Signatures mapping identifies the subsystems and connectors included in an application. The specification of a conforming application Signatures package is declared as a child package of Configuration_Interface and has the following form:

5. This is because it is impractical to require that the executive semantically depend on all of the shared passive packages where shared connectors may be allocated.

```
package Configuration_Interface.Signatures is
    type Subsystem_Names is (...);
    type Connector_Names is (...);
end Configuration_Interface.Signatures;
```

The two enumeration types define enumeration literals corresponding to the names of the application-defined subsystems and connectors. The values of attribute ‘Pos applied to the enumeration literal corresponding to a subsystem or connector names yields the respective Subsystem_Id or Connector_Id.

The implementation of the executive depends upon the Signatures package. It is through the enumeration types that subsystem and connector names provided by the controller are resolved into values of Subsystem_Id or Connector_Id. Each controller and connector object must be associated with a Subsystem_Id or Connector_Id value. While this mapping provides flexibility, there is no safeguard to prevent a subsystem or connector object from being associated with an invalid value; that is, the name may be valid for some different object type. In this instance, it may be necessary to require that the services for the types provide some form of runtime detection that the discriminant is invalid. This particular issue needs further investigation.

References

[Abowd 93] Abowd, Gregory D.; Bass, Len; Howard, Larry; & Northrop, Linda. *Structural Modeling: An Application Framework and Development Process for Flight Simulators* (CMU/SEI-93-TR-14, ADA 271348). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1993.

[Allen 94] Allen, Robert & Garlan, David. *Formal Connectors* (CMU/SEI-CS-94-1150). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1994.

[Atkinson 91] Atkinson, Colin. *Object-Oriented Reuse, Concurrency and Distribution - An Ada-Based Approach.* New York, NY: ACM Press, 1991.

[Barbacci 93] Barbacci, M.R.; Weinstock, C.B.; Doubleday, D.L.; Gardner, M.J.; & Lichota, R.W. “Durra: a Structure Description Language for Developing Distributed Applications.” *Software Engineering Journal 8,2* (March 1993): 83-94.

[Bardin 88] Bardin, Bryce &Thompson, Christopher. “Composable Ada Software Components and the Re-Export Paradigm.” *ACM Ada Letters 8,1* (January 1988): 58-79.

[Bass 94] Bass, Len & Kazman, Rick. *Towards Deriving Software Architectures From Quality Attributes* (CMU/SEI-94-TR-10, ADA 283827). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1994.

[Birrell 94] Birrell, Andrew; Nelson, Greg; Owicki, Susan; & Wobber, Edward. *Network Objects* (SRC-115). Palo Alto, CA: Systems Research Center, Digital Equipment Corporation, 1994.

[Carriero 92] Carriero, Nicholas & Gelernter, David. “Coordination Languages and their Significance.” *Communications of the ACM 35*, 2 (February 1992): 97-107.

[Cohen 90] Cohen, Sholom G. *Ada 9X Project Report - Ada Support for Software Reuse* (CMU/SEI-90-SR-16). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1990.

[Cohen 92] Cohen, Sholom G.; Krut, Robert W.; Peterson, A. Spencer; & Stanley, Jay L. *Application of Feature-Oriented Domain Analysis to the Army Movement Control Domain* (CMU/SEI-91-TR-28, ADA 256590). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1992.

[Fernandez 93] Fernandez, Jose L. *A Taxonomy of Coordination Mechanisms Used in Real-Time Software Based on Domain Analysis* (CMU/SEI-93-TR-34, ADA 279014). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1993.

[Gargaro 87] Gargaro, Anthony & Pappas, Frank. "Reusability Issues and Ada." *IEEE Software* 4, 4 (July 1987): 43-51.

[Gargaro 89] Gargaro, Anthony & Arico, Frank. "Disciplined Reusable Ada Programming for Real-Time Applications," 443 - 455. *Proceedings of the Seventh Annual Conference on Ada Technology*. Atlantic City, NJ, March 13-16, 1989. Fort Monmouth, NJ: US Army CECOM, 1989.

[Gargaro 90a] Gargaro, Anthony; Goldsack, Stephen; Volz, Richard; & Wellings, Andy. "Supporting Reliable Distributed Systems in Ada 9X: An Initial Proposal," 292 - 321. *Distributed Ada: Development and Experiences, Proceedings of the Distributed Ada Symposium*. University of Southhamption, December 11 - 12, 1989. Cambridge, England: Cambridge University Press, 1990.

[Gargaro 90b] Gargaro, Anthony; Goldsack, Stephen; Volz, Richard; & Wellings, Andy. *A Proposal to Support Reliable Distributed Systems in Ada 9X* (Technical Report 90-10). College Station, Texas: Texas A&M University, 1990.

[Gargaro 95] Gargaro, Anthony. "Towards Distributed Objects for Real-Time Systems," 36 - 43. *IEEE Third Workshop on Parallel and Distributed Real-Time Systems*. Santa Barbara, CA, April 25, 1995. Los Alamitos, CA: IEEE Computer Society Press, 1995.

[ISO 87] International Organization for Standardization. *Reference Manual for the Ada Programming Language* (ISO/IEC 8652), 1987.

[ISO 95] International Organization for Standardization. *Programming Language Ada - Language and Standard Libraries* [ISO/IEC 8652:1995(E)], February 1995.

[Kang 90] Kang, Kyo C.; Cohen, Sholom G.; Hess, James A.; Novack, William E.; & Peterson, A. Spencer. *Feature-Oriented Domain Analysis (FODA) Feasibility Study* (CMU/SEI-90-TR-21, ADA 235785). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1990.

[Lee 88] Lee, Kenneth J.; Rissman, Michael S.; D'Ippolito, Richard S.; Plinta, Charles; & Van Scoy, Roger L. *An OOD Paradigm for Flight Simulators, 2nd Edition* (CMU/SEI-88-TR-30, ADA 204849). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon Unversity, 1988.

[Meyers 93] Meyers, B. Craig & Chastek, Gary J. *The Use of ASN.1 and XDR for Data Representation in Real-time Distributed Systems* (CMU/SEI-93-TR-10, ADA 273769), Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1993.

[OMG 93] Object Management Group. *Common Object Request Broker Architecture, Version 1.2*. Framingham, MA: Object Management Group, 1993.

[OMG 95] Object Management Group, et. al. *IDL=> Ada Language Mapping Specification, Version 1.1* (OMG Document 95-5-16). Framingham, MA: Object Management Group, 1995.

[Peterson 94] Peterson, A. Spencer & Stanley, Jay L. *Mapping a Domain Model and Architecture to a Generic Design* (CMU/SEI-94-TR-8, ADA 283747). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1994.

[Rissman 90] Rissman, Michael R.; D'Ippolito, Richard S.; Lee, Kenneth J.; & Stewart, Jeffrey J. *Definition of Engineering Requirements for AFECO - Lessons from Flight Simulators* (CMU/SEI-90-SR-25). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1990.

[Shaw 93] Shaw, Mary & Garlan, David. Ch. 1, “Introduction to Software Architecture,” 1-39. *Advances in Software Engineering and Knowledge Engineering*. World Scientific Publishing Company, 1993.

[Shaw94a] Shaw, Mary. *Procedure Calls Are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status* (CMU/SEI-94-TR-02, ADA 281026). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1994.

[Shaw 94b] Shaw, Mary. *Characteristics of Higher-Level Languages for Software Architectures* (CMU/SEI-94-TR-23). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1994.

[Whitaker 93] Whitaker, William. “Ada - The Project.” *History of Programming Languages Conference. ACM SIGPLAN Notices 28*, 2 (March 1993): 299-332.

[Zweben 95] Zweben, Stuart H.; Edwards, Stephen H.; Weide, Bruce W.; & Hollingsworth, Joseph E. “The Effects of Layering and Encapsulation on Software Development Cost and Quality.” *IEEE Transactions on Software Engineering 21, 3* (March 1995): 200-208.

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION
Unclassified

1b. RESTRICTIVE MARKINGS
None

2a. SECURITY CLASSIFICATION AUTHORITY
N/A

3. DISTRIBUTION/AVAILABILITY OF REPORT
Approved for Public Release
Distribution Unlimited

2b. DECLASSIFICATION/DOWNGRADING SCHEDULE
N/A

4. PERFORMING ORGANIZATION REPORT NUMBER(S)
CMU/SEI-96-TR-017

5. MONITORING ORGANIZATION REPORT NUMBER(S)
ESC-TR-96-017

6a. NAME OF PERFORMING ORGANIZATION
Software Engineering Institute

6b. OFFICE SYMBOL (if applicable)
SEI

7a. NAME OF MONITORING ORGANIZATION
SEI Joint Program Office

6c. ADDRESS (city, state, and zip code)
Carnegie Mellon University
Pittsburgh PA 15213

7b. ADDRESS (city, state, and zip code)
HQ ESC/AXS
5 Eglin Street
Hanscom AFB, MA 01731-2116

8a. NAME OFFUNDING/SPONSORING ORGANIZATION
SEI Joint Program Office

8b. OFFICE SYMBOL (if applicable)
ESC/AXS

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
F19628-95-C-0003

8c. ADDRESS (city, state, and zip code))
Carnegie Mellon University
Pittsburgh PA 15213

10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO	PROJECT NO.	TASK NO	WORK UNIT NO.
63756E	N/A	N/A	N/A

11. TITLE (Include Security Classification)
Transitioning a Model-Based Software Engineering Architectural Style to Ada 95

12. PERSONAL AUTHOR(S)
Anthony B. Gargaro, A. Spencer Peterson

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (year, month, day)	15. PAGE COUNT
Final	FROM TO	August 1996	80

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB. GR.

18. SUBJECT TERMS (continue on reverse of necessary and identify by block number)
Ada, domain engineering, model-based software engineering (MBSE), object connection architecture, software architecture

19. ABSTRACT (continue on reverse if necessary and identify by block number)

This report describes the transition of an existing model-based software engineering architectural style to Ada 95. The report presents an overview of a software architecture for developing product families of domain-specific applications comprising reusable components, explains recognized deficiencies in the existing Ada mapping to this software architecture, and proposes solutions for correcting these deficiencies using a mapping to Ada 95. The report concludes with observations gained during the transition exercises and recommendations for future activities aimed towards deploying and enhancing the proposed mapping.

(please turn over)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT
UNCLASSIFIED/UNLIMITED ■ SAME AS RPT □ DTIC USERS ■

21. ABSTRACT SECURITY CLASSIFICATION
Unclassified, Unlimited Distribution

22a. NAME OF RESPONSIBLE INDIVIDUAL
Thomas R. Miller, Lt Col, USAF

22b. TELEPHONE NUMBER (include area code)
(412) 268-7631

22c. OFFICE SYMBOL
ESC/AXS (SEI)

ABSTRACT — continued from page one, block 19