VLDB Journal, 2, 241-284 (1992) Stanley Zdonik, Editor

A Toolkit for the Incremental Implementation of Heterogeneous Database Management Systems

Pamela Drew, Roger King, and Dennis Heimbigner

Received December 4, 1990; revised version received September, 19, 1991; accepted May 20, 1992

Abstract. The integration of heterogeneous database environments is a difficult and complex task. The A la carte Framework addresses this complexity by providing a reusable and extensible architecture in which a set of heterogeneous database management systems can be integrated. The goal is to support incremental integration of existing database facilities into heterogeneous, interoperative, distributed systems. The Framework addresses the three main issues in heterogeneous systems integration. First, it identifies the problems in integrating heterogeneous systems. Second, it identifies the key interfaces and parameters required for autonomous systems to interoperate correctly. Third, it demonstrates an approach to integrating these interfaces in an extensible and incremental way. The A la carte Framework provides a set of reusable, integrating components which integrate the major functional domains, such as transaction management, that could or should be integrated in heterogeneous systems. It also provides a mechanism for capturing key characteristics of the components and constraints which describe how the components can be mixed and interchanged, thereby helping to reduce the complexity of the integration process. Using this framework, we have implemented an experimental, heterogeneous configuration as part of the object management work in the software engineering research consortium, Arcadia.

Key Words. Heterogeneous databases, extensible databases, open architectures, reconfigurable architectures, incremental integration, heterogeneous transaction management, database toolkits.

1. Introduction

A proliferation of database applications over the past two decades has created a large number and wide variety of database environments. Further, more and more

Pamela Drew, Ph.D., is Lecturer of Computer Science, University of Science and Technology, Hong Kong. Roger King, Ph.D., is Associate Professor, and Dennis Heimbigner, Ph.D., is Research Faculty Member, Department of Computer Science, University of Colorado, Boulder, CO. (Reprint requests to Dr. Drew, Hong Kong University of Science and Technology, Department of Computer Science, Clear Water Bay, Kowloon, Hong Kong, e-mail:pam@uxmail.ust.hk.)

applications require access across these disparate stores. Researchers have made significant progress in understanding some aspects of integrating heterogeneous persistent stores, particularly from a theoretical point of view. Yet, integration of heterogeneous systems in practice still proves difficult; actual implementations in both research and industrial environments are infrequent and typically provide limited functionality.

This article describes the A la carte Framework and an actual heterogeneous configuration we have built from it. The goal of the work is to support the incremental integration of existing database facilities into heterogeneous, interoperative distributed systems. The Framework addresses the three main issues in heterogeneous systems integration. First, it identifies what the problems are in integrating heterogeneous systems. Second, it identifies the key interfaces and parameters required for autonomous systems to interoperate correctly. And third, it demonstrates an approach to integrating these interfaces in an extensible and incremental way.

1.1 Requirements of a Heterogeneous Management System

There are several requirements that a heterogeneous management system (HMS) should satisfy. First, it should be able to accommodate systems which may or may not provide certain types of functionality required for them to operate properly in a concurrent heterogeneous environment. For example, a heterogeneous transaction management system may require some local concurrency mechanism with which to communicate in each autonomous system. Any autonomous system lacking this functionality would have to be extended with a concurrency control mechanism of its own to be able to communicate with the heterogeneous manager. This characteristic of heterogeneous environments requires integration processes that will help extend each autonomous system with the functionality that it may need.

Second, an HMS must be able to integrate systems that provide incompatible implementations for the functionality that they do provide. A lack of information about variations in the internal implementation of closed architecture systems can hinder this process and can make it difficult to design a heterogeneous manager which operates correctly, or optimally. This characteristic of heterogeneous environments requires open, extensible architectures in which integrating processes can truly manage the heterogeneity of disparate implementations.

And third, an HMS must be tailorable and flexible in order to accommodate requirements placed on it by end-user applications, and the autonomous configuration systems that it integrates. Requirements of end-user applications might include the type of functionality the user wants to integrate (e.g., schema representation) and the query stream characteristics of the global applications (e.g., whether operations are mostly updates or reads, the likelihood of conflict, and the required degree of sharability). The nature of each autonomous system also places a set of requirements on the HMS. For example, the types of internal algorithms (e.g., concurrency control) and the interface calls provided by each autonomous system will have an impact on what type of algorithms the HMS should use to integrate them.

1.2 A la carte and Its Contributions

The A la carte Framework addresses these issues by providing an extensible set of reusable components to integrate heterogeneous, persistent, object stores. There are three different types of functionality required to implement a heterogeneous configuration; the A la carte Framework provides components to serve each of these functions. Components of the first type are combined to create behavior such as heterogenous transaction management that is typically implemented as part of an HMS itself. Components of the second type specify integrating protocols that implement mappings between different implementations of the same function, such as concurrency control, in autonomous systems and are also typically implemented as part of an HMS itself. These integrating components help ensure correct interoperation between autonomous systems with disparate implementations. Components of the third type are used to extend or "bootstrap" an autonomous system with the functionality it needs to operate in a heterogeneous configuration if it does not provide the required behavior in its native state. In this case, A la carte components are combined with an existing system to, in effect, create a new autonomous system that has the capabilities required to operate in a heterogeneous architecture. The Framework also provides a mechanism for capturing constraints which describe how components can be mixed and interchanged with others, thereby helping to reduce the complexity of heterogeneous systems integration. Finally, each component can have different implementations, within certain constraints, so that some tailorable HMS behavior can be achieved.

The A la carte Framework could be used to integrate a spectrum of domains found in heterogeneous systems like schema representation, query processing, transaction management, and recovery. However, it is not always necessary to integrate all of the domains in a set of autonomous systems. Since diverse application domains can require different types of functionality from a heterogeneous manager, it is appropriate, and natural, that certain types of functionality do not appear in a particular configuration. For example, a recent workshop on heterogeneous systems revealed that some applications which manipulate seismographic data don't require heterogeneous schema integration, but instead only require access to heterogeneous information (Drew et al., 1990). Thus, only subsets of functionality in heterogeneous systems may need to be integrated at a particular time; the required subsets may also change as the requirements of the application change.

This incremental approach to integration is required in an experiment we have performed with the Arcadia consortium (Taylor et al., 1988). In this case, only the transaction management and recovery schemes of the configuration systems need to be integrated. Software engineering applications frequently access objects that are loosely structured and can be interpreted in relatively raw formats. Builders of these applications prefer to have direct concurrent access to the native form of the objects for their manipulation. The schema representations of each of the underlying stores do not need to be integrated, but the transaction management and recovery schemes do. Thus, the initial scope of the A la carte Framework is in the domains of transaction management and recovery.

It is important to understand that the primary goal of this work is to find a mechanism for describing the components that make up heterogeneous management systems so that their integration can be studied and better understood. Reusable components created in this framework will have different implementations which do ultimately execute some algorithm used in heterogeneous database integration, but the invention of these algorithms is not our goal. Also, though the Framework does provide a mechanism to capture rules or constraints on how these components can be combined, we do not propose to define all of the rules that may apply to them. Instead, we propose that we have identified some of the general definitive characteristics of these components about which constraints can be written. We only propose a few likely candidates for the constraints themselves based on our experience in the domain.

1.3 An Experiment

The A la carte investigation has been performed as part of a larger set of work on object management for software engineering environments being done by the Arcadia research consortium (Taylor et al., 1988). The Arcadia consortium, supported by DARPA, brings together researchers from several industrial research facilities and universities including the University of Colorado at Boulder, the University of California at Irvine, and the University of Massachusetts at Amherst to investigate next-generation software environments. Because the consortium is in the experimental stages of investigating prototypical integrated software engineering environments, the requirements for the appropriate object stores are still evolving and several object management stores are being used in different contexts for different purposes. Further, heterogeneous database systems are a natural characteristic of any realistic software environment. Thus, an extensible framework that can be easily reconfigured to integrate different object stores provides a convenient platform for the consortium to evolve its object management requirements. With this experiment, we have driven the construction of an HMS using the A la carte Framework with some "real-world" requirements.

This article is an introduction to the A la carte Framework, an overview of the experimental heterogeneous system we have constructed from it, and the prototype environment we envision will ultimately encompass the Framework. Section 2 gives a brief description of some of the relevant work related to A la carte. Section 3 then gives an overview of the A la carte approach to heterogeneous management system construction and the overall architecture of the Framework. The Framework is further detailed in Sections 4 and 5. Section 6 gives a set of requirements for the Arcadia heterogeneous configuration and a brief description of each of the autonomous systems within it. Section 7 describes an application of the Framework to the Arcadia object management systems and the heterogeneous configuration that has been constructed as a result. And finally, Section 8 provides some concluding remarks and initial recommendations for designers of future systems.

2. Related Work

There are several relatively disparate fields of research that are relevant to the A la carte project. This related work can be categorized by specific characteristics of the A la carte Framework.

2.1 Extensibility

There are many research projects which share the A la carte Framework's characteristic of extensibility. In the database community, approaches to incorporating extensibility into both data models and systems internals have been investigated. The so-called extensible data model work (Gray, 1978; Maier et al., 1986; Manola and Dayal, 1986; Schwarz et al., 1986; Stonebraker and Rowe, 1986; Andrews and Harris, 1987; Banerjee et al., 1987; Rowe and Stonebraker, 1987; Carey et al., 1988; Linneman et al., 1988) concentrates on providing schema representations in which the user can add new user-defined types and operations. A la carte is currently not focused on this domain of extensibility.

More closely related to our research are those systems which provide extensible or reconfigurable architectures like EXODUS (Carey et al., 1986*a*, 1986*b*, 1990;

Richardson and Carey, 1987), GENESIS (Batory et al., 1990), Starburst (Schwarz et al., 1986), POSTGRES (Stonebraker and Rowe, 1986, Rowe and Stonebraker, 1987); and the "Open Object-Oriented DBMS" project underway at Texas Instruments (Thatte, 1991*a*, 1991*b*). A la carte shares some of the goals of these projects, particularly with the Texas Instruments Open OODB project which is also now defining a meta-architecture for the construction of object systems. However, A la carte's primary focus is supporting tailorable heterogeneous database systems rather than autonomous database systems.

In general, there has not been a significant amount of work done in the database community to integrate heterogeneous systems from an "extensible architecture" point of view, though the requirement for this type of solution has been stated (Jones et al., 1985). There are two projects, besides A la carte, that have taken the first steps. The first project, called the Object Management System (OMS), provides an object framework within which heterogeneous systems can communicate (Heiler, 1989). This system, like A la carte, provides mappings between interfaces of heterogeneous configuration systems. However, A la carte takes this notion one step further to provide a taxonomy which will decompose configuration systems into the domains that an application needs to integrate.

The other project builds heterogeneous systems from components that are modeled as active objects (Buchmann, 1990). These architectures incorporate systems that may have very diverse characteristics. Like A la carte, this system "wraps" component systems lacking in certain required behavior with functionality that boot-straps the components into a common architecture. However, this work is focused on the language support to create these active wrappers, whereas A la carte's also concentrates on specifying a reusable framework which defines the functionality and information required of the wrapping components.

The operating systems community has also begun to investigate extensibility and tailorability for parallel and distributed computing as seen in PRESTO (Bershad et al., 1988*a*, 1988*b*) and Choices (Russo and Campbell, 1989). Each of these systems provide a set of extensible objects which are responsible for a certain type of operating system functionality, e.g., spin-locks and threads. These systems differ from A la carte in their domain of study and in some of the types of constructs that are used to capture the components of the domain.

2.2 Reusable Frameworks

In addition to the TI Open Object-Oriented DBMS project just referenced, there are several other works which, like the A la carte Framework, use some form of meta-information as a basis of analysis or integration. The most closely related work is the ACTA transaction management framework (Chrysanthis and Ramamritham, 1990, 1991) which provides a set of reusable model constructs for characterizing autonomous transaction facilities. Its constructs provide a mechanism for defining the interactions between transactions with special emphasis on support for modeling "non-traditional" transaction management algorithms like nested (Moss, 1981) or cooperative transactions (Skarra, 1989, 1990; Nodine et al., 1990). The A la carte Framework shares the goal of providing a method for modeling and understanding the interactions of transactions at a meta-level. One of the approaches that we have considered for representing the behavior of transactions in a heterogeneous architecture builds upon the ACTA constructs.

There are, however, fundamental differences between the interactions that A la carte must capture and those that ACTA models. For example, ACTA gives a formalism of "rewrite rules" which define a process for transforming two separate transaction management algorithms into one mechanism. The A la carte Framework, on the other hand, is concerned with showing how separate algorithms interact and what types of dependencies should be enforced between them to ensure correct behavior. Also, ACTA does not address communications or other implementation issues that are of primary importance to heterogeneous systems integration; A la carte ties each of these areas into one unifying framework.

There is a body of work in the software engineering community which also provides a meta-architecture for interoperating programs. Interoperability research in the programming languages domain has been characterized by two approaches: Representation Level Interoperability (RLI) and Specification Level Interoperability (SLI). RLI provides a way for programs to share simple data types despite implementation differences (Bershad et al., 1984; Jones et al., 1985; Gibbons, 1987; Liskov et al., 1988; Hayes et al., 1990; Maybee and Dykes, 1990), and provides a lower level approach to interoperability than does A la carte.

SLI, on the other hand, is much more closely related to the A la carte Framework. SLI consists of a set of abstract data types which provide a unified type model for programming languages and a mechanism for mapping these abstract data types into particular programming language type implementations. To a certain extent, the A la carte Framework shares this approach in specifying an architecture for interoperating heterogeneous database systems. Both systems define a unifying model which can be mapped into specific implementations in their respective domains. However, the A la carte Framework attempts to capture more of the semantics in the domain than the simple types that might be manipulated by the autonomous systems.

2.3 Heterogeneous Database Systems

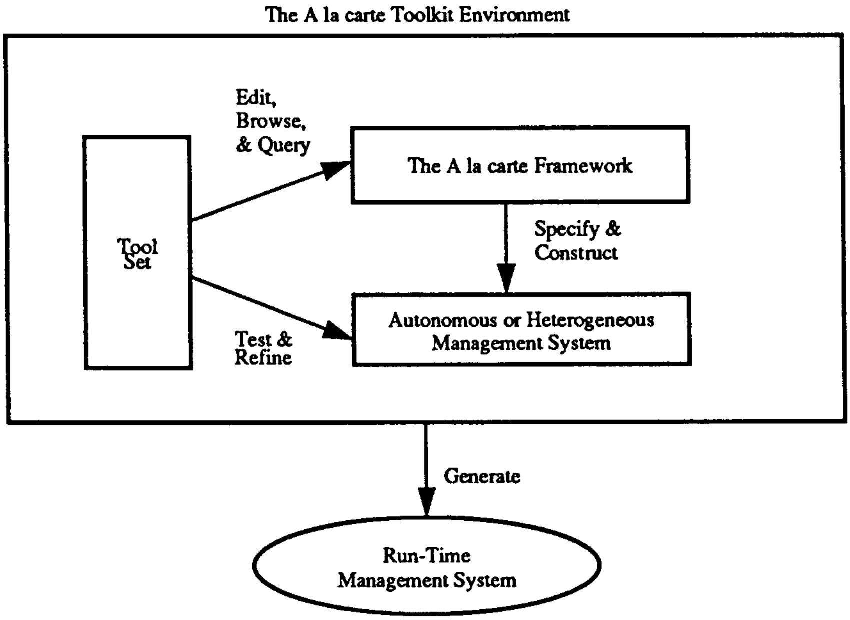
Finally, there has been a significant amount of work done on mechanisms to integrate heterogeneous databases, the domain of the A la carte Framework. This work can be roughly described by two major points of focus. First, there is a set of work which investigates integrating the schemas and query processing of systems with different native data models. To this end, there are several different approaches. One approach entails creating a global schema and query language which unites all of the autonomous system schemas and translates queries from the global scheme to each local language (Landers and Rosenberg, 1982; Siegel and Madnick, 1989; Templeton, 1989). Another approach is the federated database work in which various DBMSs maintain private schemas and use a message-passing and import/export mechanism to share information about schemas and transactions (Heimbigner and McLeod, 1985; Buneman et al., 1989; Litwin et al., 1989). A la carte takes an approach closer to the federated database approach by allowing global transactions access to local database schemas and query facilities directly.

The other point of focus is in the domain heterogeneous transaction management and recovery which is the current scope of the A la carte Framework. Within this domain, two approaches have been taken. In one approach, each autonomous DBMS need not be modified in order to accommodate the global transaction management system (Breitbart and Silberschatz, 1988, 1990; Du and Elmagarmid, 1989*a*, 1989*b*; Litwin and Tirri, 1989). Local transactions can execute against the local DBMSs without having any knowledge of the global transactions accessing the database. However, not until recently has any mechanism been proposed with reliable recovery in the event of a failure of the global transaction processing system (Breitbart and Silberschatz, 1988), and only then under the strict limitation that the global and local transactions access disjoint data sets. The other approach is to modify the component DBMSs to provide the information required by the global transaction manager (Elmagarmid and Leu, 1987; Pu, 1988; Du and Elmagarmid, 1989*a*). Limited support for transaction recovery can be provided since the local DBMSs can share some of the required information with the global transaction management system. Our goal of providing integrating protocols will help specify what other types of information should be exported out of autonomous systems so that a common ground can be found between these two approaches.

3. An Overview of A La Carte

The A la carte Framework is encompassed in the A la carte Toolkit, an environment which supports a user in the construction of a heterogeneous system. In this section,

Figure 1. A User's View of the A la carte Toolkit Environment

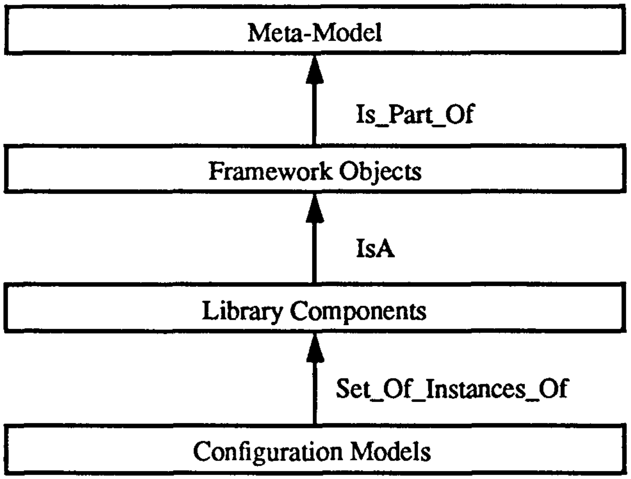

The A la carte Toolkit Environment
Tool Set
Edit, Browse, & Query
The A la carte Framework
Specify & Construct
Autonomous or Heterogeneous Management System
Test & Refine
Generate
Run-Time Management System

we give a high-level overview of the environment and how it is used to implement heterogeneous configurations. We describe the environment's architecture from the user's point of view, the internal management system which guides the user through the construction process, and an architectural view of the Framework itself. In Sections 4 and 5, the Framework is described in detail. For a more narrowly focused description of how a user interacts with this environment, we direct the reader to Drew et al. (1990).

3.1 The User's View

In order to help reduce the complexity of heterogeneous systems integration, the A la carte Toolkit Environment encompasses the A la carte Framework and partially automates the integration process, as shown in Figure 1. It is composed of four main components: the A la carte Framework, the A la carte Tool Set, the heterogeneous management system constructed by the user, and the run-time system that can be generated from the environment. The arrowed lines in Figure 1 represent user actions. The user can use the tools provided in the A la carte Tool Set to edit, browse, and query the A la carte Framework from which the heterogeneous architecture

Figure 2. Architecture of the A la carte Framework



is constructed. This environment is supported by an internal management system which performs user-specified operations on the components in the Framework and partially controls the construction of a heterogeneous configuration. The user can also create, if desired, a "run-time" version of an A la carte management system. In this fashion, although it is not a goal of this research to create production software, a user of A la carte can extract an executable system from the environment's internal management facilities to achieve better performance.

3.2 The Architecture of the A la carte Framework

There are four different views of heterogeneous database system behavior in the A la carte Framework: the Meta-Model, the Framework Objects, the Library, and the Configuration Models. Each provides a different level of abstraction of heterogeneous systems behavior and each is related to the next view in the architecture, as depicted in Figure 2.

The A la carte Meta-Model, depicted at the top of Figure 2, captures characteristics of heterogeneous transaction management systems that define how they interoperate. It provides a context in which the rest of the architecture's components can be placed. This model describes properties that are common to heterogeneous transaction management systems, as well as the dependencies that exist between them. For example, one attribute, termed the visibility of a transaction management system, defines how objects are shared during different phases of a transaction's life-time.

Each transaction management system follows some procedure for enforcing consistency and recovery criteria, such as serializability and atomicity, on transactions. The attributes defined in the meta-model provide a way of describing this criterion. By describing different management schemes in terms of these properties, we can model how they will interact in a heterogeneous architecture. Using these properties, we can also write rules or constraints on how heterogeneous transaction management systems can interact. These rules can be applied during the systems integration process.

The Framework Objects are the reusable components in the Framework that can be used in the integration of heterogeneous systems. A Framework Object's definition includes an interface specification and, possibly, a set of protocols which integrate disparate implementations of parameters that the Framework Objects need to share. The Framework Objects inherit the properties and constraints described in the Meta-Model that apply to them. Conversely, the Framework Object definitions themselves are part of the A la carte Meta-Model. In this fashion, a new Framework Object will have a context when it is first introduced. This context will help identify which components already exist in the Framework that can interoperate properly with the new component. It is important to note that these components represent a *functional decomposition* of a heterogeneous transaction management system; the sum of their interactions and the other properties in the meta-model define the consistency characteristics for the overall system.

The A la carte Library, implemented in an object-oriented class library, contains behavioral specializations of the Framework Objects. For example, a *Concurrency Control* Framework Object could have two Library Component implementations: one as a *Timestamping* Concurrency Control Manager and the other as a *Locking* Concurrency Control Manager. Again, the Library Components will inherit the meta-model information of its parent Framework Object.

Finally, the Configuration Models provide example configurations of classes from the Library which can be used as templates for HMS construction. The configurations are instantiations of a set of Library Components. As a user creates instantiations of the Library Components, the constraints which the component has inherited from the Meta-Model are enforced. A Configuration Model represents the communication lines required between a particular set of Framework Objects. It also represents an implemented heterogeneous architecture. The Configuration Models can be used to guide the user in creating new combinations of Framework Objects. They can also be refined to meet a particular environment's requirements. Of course, if none of the configurations are close to satisfying the current requirements, the user can always create a new HMS by combining an all together different set of Framework Objects into a configuration with some set of algorithms implementing their respective functionalities.

3.3 The Internal Architecture of The Toolkit

The A la carte toolkit provides an internal management system to help guide the user through the HMS construction and combination process. This internal manager is composed of two main components: the *Creation Manager* and the *Constraint Manager.* The Creation Manager is responsible for the instantiation of a set of Library Components selected by the user. The Constraint Manager is responsible for enforcing the constraints which apply to them. As the construction proceeds, the Creation Manager queries the user for the required input to instantiate each component, and consults the Constraint Manager for the legality of the specification.

There are three important points to note about the Constraint Manager and how the user interacts with it. First, it is not a goal of A la carte to define a new constraint language to be used for the analysis of heterogeneous systems. Instead, we have relied on existing language classes as a basis for experimentation. So far, the types of constraints that we have identified can be effectively captured by a declarative first-order logic programming language like Prolog (Sterling and Shapiro, 1986). Second, it is not a goal to identify all possible constraints that apply to heterogeneous systems integration; the current set seems to be a first good approximation based on our experience with the domain. And third, it is left to the user of A la carte to correctly specify any new constraints. A la carte cannot guarantee correct execution of any prototypical management system; the constraints are provided as a service to the user.

4. Meta-Model Properties

To understand the integration of heterogeneous transaction management systems, a meta-model of those properties which are basic to a system's interoperability is needed. These properties can be used to describe different consistency and recovery criteria, such as serializability and atomicity, that transaction management schemes enforce on transactions. These properties include a model of which configuration systems share control over which transactions,[1] a model of how the configuration systems manage change to the states of objects under their respective control, a model of how the configuration systems allow sharing of objects between transactions,[2] and a model of the configuration's operational components, or Framework Objects, and the interactions between them. It is the last model that links the A la carte Framework to actual systems implementation. There can, of course, be other properties which we have not identified. However, this set seems to be a good first approximation to some of the key characteristics of heterogeneous transaction management systems. For a detailed description of each of these properties, please see Drew (1991).

There are different ways that these properties can be used to analyze the interoperability of a heterogeneous configuration. One way is to compare the description of a potential heterogeneous architecture with the description of a "legal" architecture. Of course, with the advent of more advanced design environments, a number of consistency criteria ranging from strict serializability (Bernstein et al., 1987) to quasi-serializability (Alonso et al., 1987; Du and Elmagarmid, 1989*b*) to user-defined correctness criterion (Siegel and Madnick, 1989; Skarra, 1989; Nodine et al., 1990) have been proposed. Thus, there is not a single canonical model with which all other systems can be compared. Instead, the potential architecture can be compared with *one* of the "legal" architectures to see how it differs.

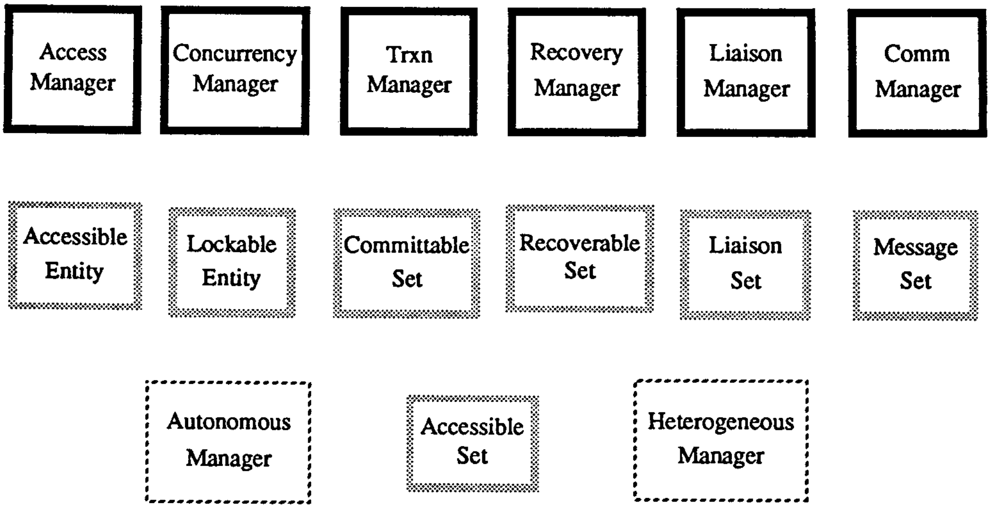
Another way to use the properties is as a basis for understanding illegal, or undesirable, states in a heterogeneous configuration. For example, management systems which share control over transactions, e.g., a heterogeneous transaction manager and an autonomous transaction manager, must be able to coordinate their behavior. Otherwise, inconsistencies, such as the inability to recover from failure and deadlock, are possible (Du and Elmagarmid, 1989*a*; Breitbart et al., 1990). By explicitly examining how control is shared in a heterogeneous architecture, we can identify when potential inconsistencies can arise.

Finally, the properties also can be used to identify dependencies that can exist between operational components in the configuration. Unlike autonomous transaction management systems, dependencies can be created between transactions under the control of different management systems in a heterogeneous architecture.

[1] In heterogeneous architectures, liaison transactions that are generated by a heterogeneous transaction manager to execute at an autonomous site can be under the control of at least two systems, the heterogeneous manager and the autonomous transaction manager.

[2] The ACTA transaction management model (Chrysanthis and Ramamritham, 1990, 1991) supports a similar concept for autonomous systems.

Figure 3. A la carte Framework Objects



For example, in a 2-phase commit protocol, there is a dependency between a global, heterogeneous transaction and its liaison transactions. In this case, the commit operation of the global transaction first sends a "prepare-to-commit" message to each of the liaison transactions; a positive response means that it will commit successfully when instructed to do so. If each of the liaison transactions responds affirmatively, then the global transaction will send a second "commit" message to each of the liaison transactions; this step will allow the global transaction to finish its commit operation as well. If for some reason any of the liaison transactions respond negatively, or not at all, the global transaction will not commit.

Illegal combinations of functionality (i.e., those that could lead to an undesirable configuration state) and dependencies between configuration components can be represented as constraints associated with the Framework Objects. These constraints are enforced during the system integration process in the Toolkit environment.

5. The Framework Objects

Our functional decomposition of a heterogenous transaction management system is modeled by the current set of A la carte Framework Objects depicted in Figure 3. There are three types of objects in this design. The first type, shown in the bold boxes, are the Functional Domain Managers that are responsible for the implementation of the operational components in a particular heterogeneous architecture (e.g., a Concurrency Control Manager[3]). The second type, shown in the dotted boxes, are the objects that the Functional Domain Managers manipulate and coordinate (e.g., a Lockable Entity). A brief definition of these objects is listed below.

- The Access Manager provides the access primitives {*Create, Delete, Read, Write*} and operates on *accessible_entities* that can be one of the types {*ByteStream, Record, Tuple, Object, ComplexObject, File.*}

- The Transaction Manager provides the primitives {*Begin, PrepareToCommit, Commit, Abort*} to manipulate the states of transactions and operates on *committable_sets* that can be one of the types {*Object, Page, Record, Segment, File, Database.*} In a heterogeneous transaction manager, these sets are heterogeneous and span several transaction managers.

- The Concurrency Control Manager provides the concurrent access primitives {*GrantReadAccess, GrantWriteAccess, ReleaseReadAccess, ReleaseWriteAccess*} and operates on *lockable_entities* that can be one of the types {*Object, Page, Record, Segment, File, Database.*}

- The Recovery Manager provides the primitives {*Undo, Redo, Restart*} to return a system to a consistent state and operates on *recoverable_sets* that can be one of the types {*Object, Page, Record, Segment, File, Database.*}

- The Liaison Manager provides the primitives {*BeginExecution, EndExecution, Connect, Disconnect, BeginLocalTrxn, CommitLocalTrxn, AbortLocalTrxn, LocalPrimitive1, LocalPrimitve2,...*} to manage liaison transactions created by a heterogeneous transaction manager to execute at a local site. Many of the primitives in this interface will directly reflect the interface of the local system. The Liaison Manager operates on *liaison_sets* that include local client code for autonomous systems.

- The Communications Manager provides the primitives {*BeginExecution, EndExecution, Connect, Disconnect, CallServer, Receive, Reply*} to coordinate communication between clients and servers in heterogeneous architectures. The Communications Manager operates on *message_sets* that can be one of the types {*Datagram, Streams.*}

[3] This object performs scheduling of access to objects. Its operations combined with the operations of the Transaction and Recovery Managers implement the system's overall consistency criteria for concurrent access to objects.

The third type of object in the Framework, shown in dashed boxes, are those which provide the "glue" to combine different Functional Domain Framework Objects together; these objects also provide the end-user interface of the resulting system. These objects are defined as follows:

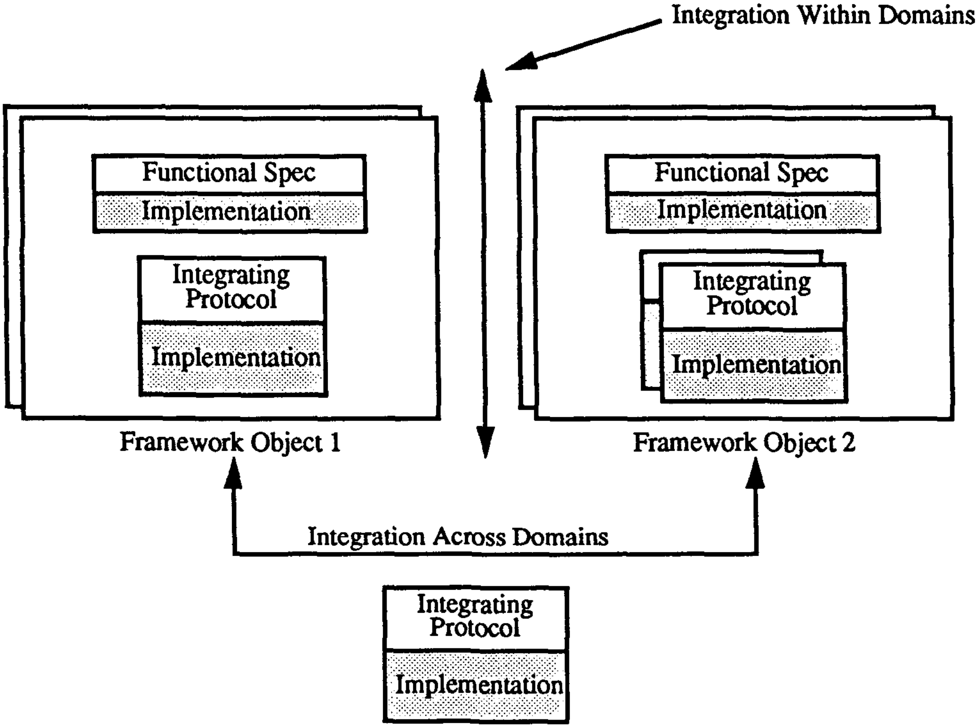
- The Autonomous Manager provides the primitives {*CreateDB, DeleteDB, OpenDB, CloseDB, BeginTrxn, CommitTrxn, AbortTrxn, CreateObject, DeleteObject, ReadObject, WriteObject*} for accessing an autonomous system. This manager object is used when an autonomous system needs to be combined with the functionality provided by an A la carte Framework Object to operate properly in a heterogeneous architecture. In this case, the autonomous system is treated as an implementation of the Functional Domains that it provides. The Autonomous Manager Object calls the autonomous system for its native functions and calls other A la carte Library Components for functionality which provides the required extensions. The Autonomous Manager operates on *accessible_sets* (e.g., databases).

- The Heterogeneous Manager provides the primitives {*BeginHetTrxn, CommitHetTrxn, AbortHetTrxn*} for accessing a heterogeneous management system and operates on *committable_sets*. This manager object is used to integrate Framework Objects that implement algorithms for managing heterogeneous versions of their Functional Domains.

5.1 The Integration of Framework Objects

As shown in Figure 4, the Framework Objects can be integrated in two dimensions. One dimension is within the same functional domain and across the systems within a particular architecture. For example, in this dimension of integration, the transaction managers of a set of systems could be integrated, but other functions of each system could remain autonomous. Rather than treating each autonomous system as a monolithic "black-box" and integrating all of its functions into a heterogeneous architecture, it is sometimes convenient to integrate only a subset of the services that a particular system provides. Further, as we shall see in an upcoming example, even if all of the functional domains of a set of systems are to be integrated, it may be necessary to integrate a set of systems in this modular way in order to achieve correct behavior.

Figure 4. Framework Object Structure and Dimensions of Integration

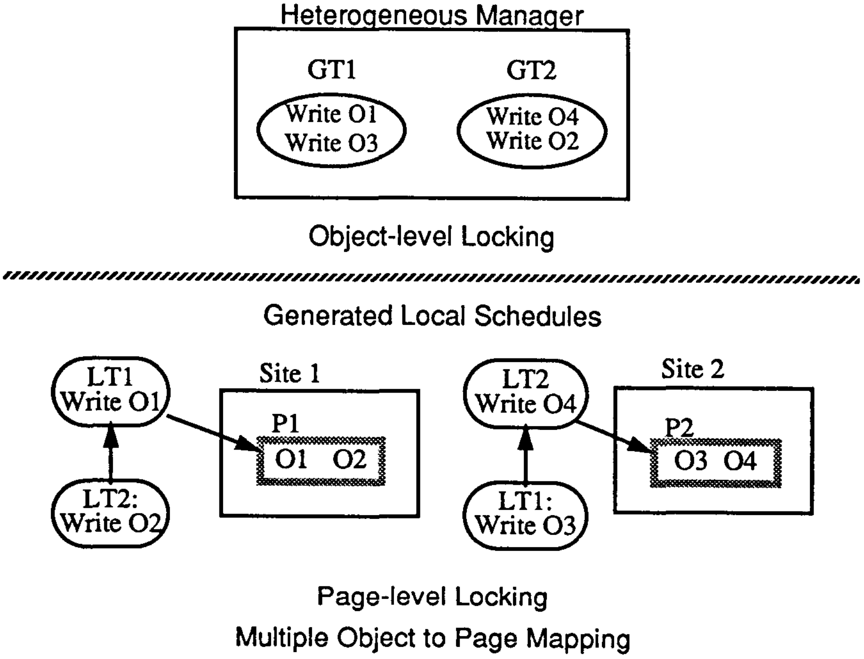


The other dimension of integration is across functional domains; this dimension allows components of different functional domains, like access management and concurrency control management, to be integrated together. This dimension of integration supports the flexible mixing and matching of different types of functions in a heterogeneous configuration and enforces strict notions of layered architectures. By integrating these reusable components in different combinations and in different dimensions, a user can tailor a heterogeneous architecture to the requirements of a particular environment.

5.2 Integrating Protocols and Parameters

Integrating protocols are used to allow components of disparate implementations to communicate, and to facilitate the mixing of different Library Components to create new behaviors. They are defined in an object-oriented paradigm: each protocol has a set of data types, or *integratingparameters,* and a set of operations, or *integratingprotocols,* that can be performed on those data types. The parameters represent information to be integrated and the operations represent the method by

Figure 5. A Heterogeneous System in Deadlock



which the parameters can be integrated. There can be any number of integrating protocols for each Framework Object. In the next subsections, we give an example for one of them called the *lockable_entity* protocol, and its specification. This example helps illustrate why a decomposition of system functionality and exposure of system interfaces is sometimes required for correct execution in a heterogeneous environment.

5.2.1 Motivating Example. Consider the system shown in Figure 5, comprised of a heterogeneous manager and two autonomous systems: Site 1 and Site 2. All three of these systems provide a strict 2-Phase locking protocol in which transactions will not release their locks until their work has been completed. Site 1 and Site 2, provide page-level locking, but the HMS provides locking for heterogeneous transactions at the object-level. Objects O1 and O2 exist on the same page P1 at Site 1; objects O3 and O4 exists on page P2 at Site 2. Two global transactions have been submitted to the HMS: GT1 and GT2 (shown in the system above the dotted line). GT1 has submitted the operations *write O1, write O3;* GT2 has submitted the operations *write O4, write O2.* Local transactions LT1 and LT2 are created on behalf of GT1 and GT2 and generate the local schedules (shown in the systems below the dotted line), at Site 1, LT1: *Write O1,* LT2: *Write 02;* at Site S2, LT2: *Write O4,* LT1: *Write O3.*

This system is in a state of global deadlock. GT1 is waiting for GT2 to release its lock on P2 so that it can get a lock on O3, and GT2 is waiting for GT1 to release its lock on P1, so that it can get a lock on O2. Further, the HMS created this deadlock unwittingly because it was not aware of the implications of its lock requests in the autonomous systems. Because the pairs, O1/O2 and O3/O4, exist on the same pages at their respective sites, the write locks on them actually conflict and create a deadlock. But, because the HMS does not know how the locking is being implemented in the autonomous systems, it cannot detect or avoid this deadlock. Since there is a one-to-n mapping from pages to objects in this particular configuration, heterogeneous transactions have an effect on one another that the heterogeneous manager cannot detect. Notice also that neither of the autonomous systems can detect this deadlock since the deadlock involves heterogeneous transactions that access different autonomous stores. Since the autonomous stores do not communicate with one another, they cannot be aware of how transactions are being managed by other systems or which transactions are part of the same heterogeneous transaction.

There are a range of solutions to this problem. One is to simply execute a time-out procedure on suspended local transactions that were created on the behalf of the HMS. Another solution is to somehow provide enough information to the heterogeneous manager about the state of the autonomous systems to be able to avoid or detect this scenario. Within this "export-information" class of solutions is a whole range of design, implementation, and performance trade-offs. So far, we have focused our work on the identification of these solutions at the specification level without analyzing the trade-offs of different implementation strategies in detail.

5.2.2 The Lockable_Entity Protocol of Integration. We have defined a *lockable_entity* protocol of integration,[4] shown in Figure 6, which provides a specification-level solution to the problem illustrated by the previous example. It provides a mechanism for a Concurrency Control Framework Object within an HMS to determine whether or not a set of locks conflict within an autonomous system. The *interferes* method shown in this specification aids the HMS in determining the ramifications of locking a particular set of entities. Given a set of lockable entities, the *interferes*

[4] This pseudo code representation borrows from the Eiffel programming language (Meyer, 1988).

Figure 6. Example Specification of Integrating Protocol, Lockable Entity

```
class LOCKABLE_ENTITY[T]
export
    interferes
method
    T: LEVEL_OF_GRANULARITY is_one_of
        OBJECT, PAGE, RECORD, FILE, SEGMENT, DB, NULL;
    interferes (lockable_entity1, lockable_entity2: T): BOOLEAN is
    do
        if lock(lockable_entity1) ¡==¿ lock(lockable_entity2) then
            Result := True
        else
            Result := False
    end; – interferes
end – class LOCKABLE_ENTITY
```

method returns whether or not granting locks on Łthem creates a conflict within an autonomous system. The type of the integrating parameter, T, is the level of granularity that the HMS is using for its internal implementation of locking and can be one of the frequently used implementation types like objects or pages shown in the specification. To integrate a set of autonomous systems, the choice of T must be one for which a mapping from T to the various internal implementations of lockable entities in the autonomous systems can be created.

If this protocol were applied in the system just described, the HMS would have been able to determine that locking O1 would also lock O2 and hence would have been able to avoid or detect the global deadlock. In our example, the level of granularity chosen for the HMS was the object-level; the level of granularity of Sites 1 and 2 was the page-level. The HMS could have invoked the *interferes* protocol with input parameters, O1 and O2, which represent an object-level granularity. Through some mapping process from the HMS representation of lockable entities to Site 1's representation of lockable entities, the *interferes* protocol would have returned TRUE implying that a lock on O1 would interfere with a lock on O2.

The required mapping process could be implemented in several ways. Site 1 could maintain an internal mapping from objects to pages and compute the interferes method itself, or Site 1 could export the mapping and the HMS could compute the protocol. In another solution, Site 1 could export information from its lock table which the HMS could use to compute a solution. In any case, the choice of T can be influenced or determined by the set of autonomous systems that are to be incorporated into a particular architecture. Via this mapping process, a heterogeneous set of systems can emulate a common level of granularity with respect to the HMS for a particular type of function so that the function can be integrated properly.

It is important to note that this particular protocol only needs to be applied between each autonomous system and the heterogeneous manager. The autonomous systems do not need to be integrated with respect to each other. Only the heterogeneous manager needs to be able to interpret each system's integrating parameter.

5.2.3 Uses of Integrating Protocols. The integrating protocols can be used to integrate systems in the different dimensions that are supported by the A la carte Framework depicted in Figure 4. Integrating protocols which integrate parameter types within the *same* functional domain and across a set of systems, as in the *lockable_entity* example, are defined in the Functional Domain Framework Objects. Integrating protocols that allow Framework Objects from *different* functional domains to share data are defined in the Autonomous Manager and Heterogeneous Manager Framework Objects. For example, the Autonomous Manager may need to provide an integrating protocol to map between the entities that an Access Manager manipulates and the entities that a Concurrency Control Manager manipulates. In this case, the Autonomous Manager's primitive, *ReadObject*, invokes both the Access Manager primitive, *ReadObject*, and the Concurrency Control Manager primitive, *GetReadAccess*. If the Access Manager is manipulating *Objects* and the Concurrency Control Manager is manipulating Pages, an integrating protocol which maps one into the other must be implemented by the Autonomous Manager.

There may be other integrating protocols of heterogeneous systems which we have not identified. The definition of these protocols is an evolutionary process. As we gain more experience with this Framework, we expect the protocols to be enhanced and extended. Those that we have specified have proven to be useful tools in understanding some of the ramifications in integrating disparate autonomous systems.

6. An Example Set of Configuration Systems

In this section, we give an overview of the object management systems that we integrated in a heterogeneous configuration and a set of requirements that this configuration has to satisfy. Each of the architectures given here describe the systems as they exist in an autonomous, or native, state without any communication with a heterogeneous manager.

6.1 Object Management in Arcadia

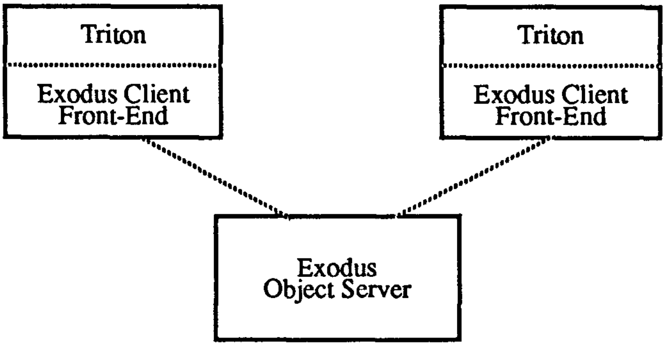
One of the long-range goals of the Arcadia Consortium is to provide an object management infrastructure that supports multiple users with heterogeneous systems. Heterogeneity is a prevalent characteristic of software engineering environments which usually include many diverse software, administrative, and database systems. Process-centered, integrated environments often support applications which require coordinated access to information located across disparate systems. For example, a documentation tool may need to access requirement specifications located in one store and design documents located in another. Similarly, there may be a project management tool that also requires access to these documents. Further, there may also be an analysis tool that operates on requirement specifications only, and requires access to the requirement specifications simultaneously with the documentation and project management tools. This is a common scenario for integrated software engineering environments and provides a motivating example for creating heterogeneous architectures.

In the Arcadia object management infrastructure, two object-management systems, Triton (Heimbigner, 1990) and PGraphite (Wileden et al., 1990), are primarily used to supply persistence for software engineering applications. Each of these systems differ in several aspects, such as models of persistence. The application's programmer can choose to use the one which best suits the requirements of the given application. Each system is composed of a storage manager which is then extended with some additional functionality that the Arcadia Consortium has identified as useful for object management in software engineering environments. We now briefly elaborate on the structure and behavior of each of the component systems.

Triton

One of the autonomous object management systems, Triton, extends the Exodus storage manager and its associated database programming language, E (Richardson and Carey, 1987), with an interpretive environment which provides constructs, like triggers, that are useful in managing a process-oriented software environment. Triggering mechanisms can be used to manage dependencies that exist between various activities in software engineering environments. For example, when some

Figure 7. The Triton/Exodus Architecture in an Autonomous State



portion of a requirements specification is changed, the associated design documents, software, and test harnesses may require some modification. A trigger could be used to automatically enforce this dependency between the related artifacts.

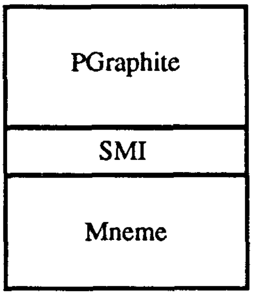
Triton's native architecture is evolving from using a single-user version of the Exodus storage manager to a new, serverized, multi-user implementation of Exodus, as depicted in Figure 7. The new Exodus implementation has a client-server architecture and provides transaction management with a hierarchical, 2-Phase locking protocol (Gray, 1976, 1978) and simple recovery services. Client processes include application code, like Triton, and some Exodus "front-end" software which provides object-manipulation facilities and cache management. Finally, the client processes communicate with the server via a message-passing protocol, depicted by the dashed lines. Currently, Triton processes are single-threaded, so several must be invoked to create a multi-user environment.

PGraphite

PGraphite (Wileden et al., 1990), the other autonomous object management system, provides a data model which is useful for supporting general graph and tree structures, like data flow diagrams, frequently used by software analysis and language processing tools. PGraphite uses its own mechanism for defining persistence models, and has a different data model from Triton.

PGraphite communicates with the underlying storage management system through an intermediary layer of software called the Storage Manager Interface (SMI), as shown in Figure 8. The purpose of the SMI layer is to provide a uniform interface to any number of underlying storage managers so that one can be easily exchanged for another. In its native state this system is implemented within one

Figure 8. The PGraphite/Mneme Architecture in an Autonomous State



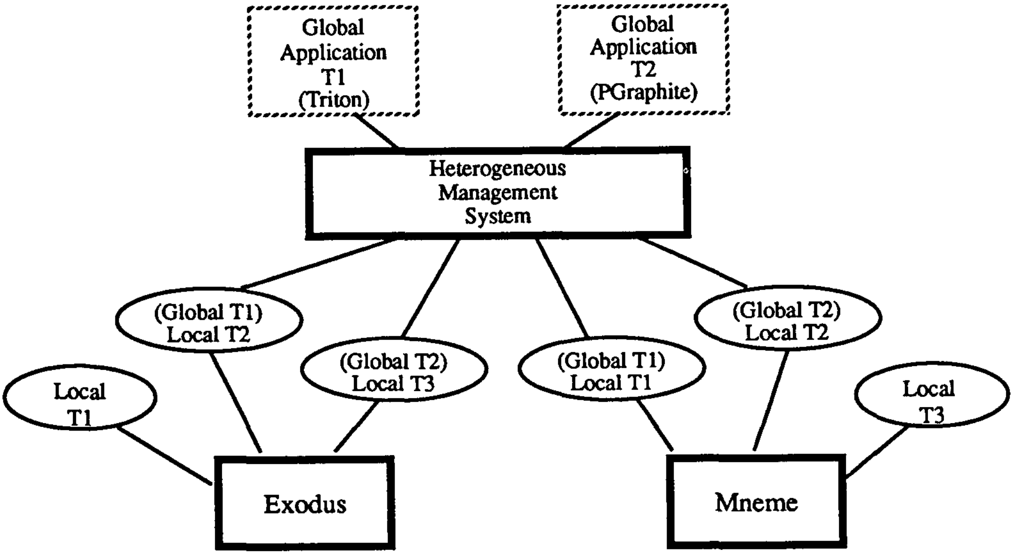
process. SMI mappings have been created for Exodus, Observer (Hornick and Zdonik, 1987), and the system currently in use, Mneme. The current implementation of Mneme provides object manipulation and clustering facilities, but does not support transaction management or recovery. Hence, this version of Mneme is an example system that will have to be combined with A la carte components in order to communicate and operate correctly in a concurrent heterogeneous environment.

6.2 HMS Requirements

The heterogenous management system that integrates these two systems should provide, as a first-cut, some form of transaction management which allows applications to access, within a single global transaction, objects that are located in these heterogeneous stores. Further, other non-heterogeneous, or local, applications should be able to access the autonomous stores simultaneously, and should not have to interact with the autonomous system via the HMS interface. And finally, these global transactions should follow commit protocols which allow for recovery in the event of either a local or global management failure.

The architecture that models the required behavior is depicted in Figure 9. The emboldened boxes present the three major components in the architecture: the HMS and the two autonomous systems. The integrating transaction management system will be introduced at the storage management level of the autonomous systems since it is at this level that the native transaction management facilities of Exodus reside. (The method by which we can introduce local transaction management facilities to Mneme is detailed in the next section.) Transactions are submitted to the HMS by heterogeneous, or global, applications depicted in the dashed boxes. These applications require objects that reside in the different autonomous stores

Figure 9. Model of Required HMS Architecture



and submit object requests to the HMS. The HMS, in turn, submits local liaison transactions, represented by ovals, to the autonomous systems. There are also other local applications simultaneously interacting with the autonomous systems without communicating with the HMS.

There are many different algorithms that could be used to implement this architecture. Our first choice for the classes of algorithms to be implemented by the heterogeneous transaction management system are 2-Phase locking for concurrency management, 2-phase commit and deadlock detection for transaction management, and write-ahead logging for recovery management. We are aware that some software engineering applications may require less traditional transaction management mechanisms like cooperative (Skarra, 1989; 1990; Nodine et al., 1990) or nested transactions (Moss, 1981), but since our initial work is focused on appropriate interface and modularity issues, we have chosen a more accessible and easy-to-implement set of algorithms. Future work can include the investigation of these less traditional types of algorithms in heterogeneous environments.

7. The Arcadia Experiment

Like many existing heterogeneous database environments, each of the autonomous systems, Exodus and Mneme, do not provide complete support for their integration into distributed or heterogeneous architectures in their current native state. We must analyze each system to determine exactly which parts of them are integratable as they stand, which parts of them are integratable with some extensions, and which parts of them are not integratable at all. We will find that some of our initial choices can be implemented relatively easily, while other aspects are more difficult to achieve.

There are several key questions which apply to the integration of heterogeneous configurations that are answered for this set of systems:

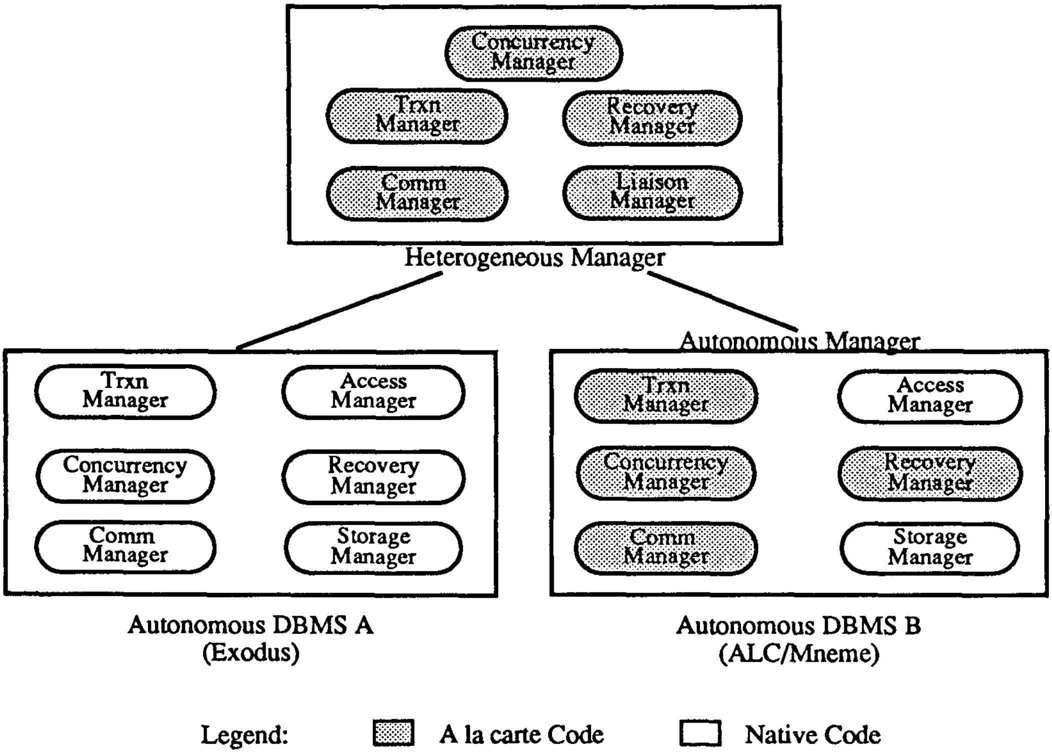
- What Functional Domains are to be integrated?
- Is the required information for integration accessible and, if so, where is it?
- What are the various ways in which to implement an integration mapping?
- What level of "homogeneity" should be emulated in the architecture, and how is the HMS interface affected?
- What are the trade-offs in distributing the HMS functionality in a heterogeneous architecture?

This section describes an integration process in which these questions are answered. First, we detail which Framework Objects are required to achieve the desired behavior for the overall configuration. Second, we analyze how each autonomous system would be best integrated into the architecture. And third, we outline the resulting implementation from our analysis.

7.1 The Required Functional Domains

An A la carte Configuration Model which implements the desired behavior is created by combining Library Components in the heterogeneous transaction management architecture shown in Figure 10. The unfilled ovals depict functionality that the autonomous systems provide in their native state and the filled ovals represent functionality that A la carte Library Components must provide for this particular architecture. The Heterogeneous Manager is implemented entirely of A la carte Library Components and contains a Transaction Manager, a Concurrency Manager, and a Recovery Manager to implement the heterogeneous transaction management algorithms. It also contains a Communications Manager and a Liaison Manager to communicate with the autonomous systems. Each of these functional domains will be integrated with its counterpart in each autonomous system.

Figure 10. An Example Heterogeneous Configuration Model



The autonomous systems are depicted in Figure 10 in terms of the Framework Objects that they implement. At the time of this writing, the Exodus client provides an Access Manager operating on simple *Objects, ComplexObjects,* and *Files;* a Buffer Manager[5] which provides a local cache for objects, and some end-user transaction management operations. The Exodus server is implemented by a Transaction Manager providing most A la carte operations except the *PrepareToCommit* operation, and operating on simple *Objects, ComplexObjects,* and *Files;* a Concurrency Control Manager providing two-phase, hierarchical locking on *Pages* and *Files;* a Recovery Manager providing primarily only a *Restart* operation and

[5] The Buffer Manager Framework Object is another functional domain that is not currently considered a primary A la carte integration domain, but it must be represented to capture the operation of autonomous systems.

operating on the *Database* level; and a Storage Manager[6] operating at the *Byte* level. A Communications Manager provides message passing between clients and a multi-threaded server, and operates on both the *Stream* and *Datagram* level.

Mneme, on the other hand, provides an Access Manager operating on *Objects* and a Storage Manager operating on *Bytes.* Each of the Framework Objects for transaction management, concurrency control, recovery, and communications are not implemented as native functions in Mneme. The functions that Mneme does provide are combined with A la carte Framework Objects to achieve the appropriate local serializable behavior required of it to interoperate properly in a heterogeneous concurrent architecture. This combination process entails "wrapping" Mneme with the Autonomous Manager from the A la carte Framework and combining it with A la carte Library Components which implement concurrency control, transaction management, recovery, and communications.

7.1.1 Integration of Mneme. For each component that A la carte implements in the A la carte/Mneme system, we must choose the Library Component with the appropriate implementation and the type of its integrating parameter. This choice is influenced by the desired behavior of the overall architecture, by the data types accessible in the existing system, and possibly by the integrating parameters of the other systems in a configuration. For example, because serializable behavior is a requirement for this environment, A la carte should provide components which implement a 2-phase locking Concurrency Control Manager and a 2-phase commit Transaction Manager that supports a *PrepareTo-Commit* protocol. Since Mneme provides access to object-level information in its native interface, this level of granularity is a convenient choice for the integrating parameter of the Locking Concurrency Control Manager that A la carte supplies. Also, since Mneme is to be integrated with Exodus, which provides recovery functions at the database-level, this level is considered to be a sufficient for Mneme as well. If, on the other hand, another system that provided recovery at the object-level were to be integrated in this architecture, an end-user application may have also required an object-level recovery scheme for Mneme. And finally, because of a rather unusual architectural design for the overall system to be described below, we found it most convenient to serverize Mneme using the A la carte Communications Manager.

[6] The Storage Manager Framework Object captures all functions, like disk access and buffering, that persistent stores provide. It is not listed as a primary Framework Object in the A la carte Framework since reconfigurable persistence models are not our primary focus in this experiment.

7.1.2 Integration of Exodus. The integration of Exodus raises a different set of issues. In this case, the system provides many of the required Framework Objects in its native state. The integration process for this system mostly requires determining where the functional domains, and their respective integrating parameters, are exposed in the system. Upon careful study of the system's interfaces and design, we were able to integrate most aspects of the required functional domains.

An A la carte view of the function of the Exodus client is to provide a mapping from the integrating parameters of the domains exposed in the interface of the client to those exposed in the interface of the server. For example, the Exodus client decomposes the representation of objects supplied by the Access Manager in the client into their equivalent representation as pages on which the Concurrency Manager in the server operates. The client passes messages to the server requesting operations, like lock requests, for the pages on which an object resides. This decomposition is the equivalent of the integrating protocols associated with the Autonomous Manager Framework Object which provides mappings from one functional domain to another.

A heterogeneous manager can integrate the concurrency control of this system by "intercepting" these service requests as they are passed from the client to the server. In this fashion, an implementation of the *lockable_entity* protocol, described in Section 5, can be achieved. The HMS can integrate its representation of Exodus' locking function by intercepting the real lock requests. The mapping implementation for the protocol is not achieved by some translation from one parameter type to another, but instead, by moving the place in which an HMS typically interacts with an autonomous system.

This approach to integration works well in most cases; however, there are two general types of problems that can arise if a system is not designed with this approach in mind. One problem is that if a client sends a message to the server which is then decomposed into different functional domains *inside of* the server, an "intercepting" HMS may not know into which domains the server may ultimately decompose the message. For example, if a *ReadPage* message from the Buffer Manager of the client is further decomposed into a *GrantReadLock* call to the Concurrency Manager, and a *ReadPage* call to the Storage Manager, inside of the server, an HMS would not be able to intercept the lock message alone. In this case, the heterogeneous system designer must understand which messages will ultimately turn into operations on a functional domain targeted for integration.

The other problem that could arise is that the heterogeneous manager may not be able to integrate a particular domain if the required integrating parameters are not always exposed. In the case of Exodus, there are two interfaces that may have been used to access *lockable_entity* integrating parameters: the end-user interface and the interface between the client and the server. Exodus uses a hierarchical locking mechanism in which file-level locks imply locks on the pages in the file which in turn imply locks on the objects on those pages. Interestingly, this hierarchical locking scheme perpetuates the same problem that the motivating example for the *lockable-entity* protocol illustrated. If a heterogeneous manager were locking at the page-level, and two autonomous systems implemented this hierarchical locking scheme, then the HMS could inadvertently create a deadlock on two pages in the same file and not detect the deadlock.

One implementation of an integrating protocol would track which objects are on which pages and which pages are in which files, so that it could interpret the requests of the client. However, the data types at the client or server interfaces of Exodus would have to expose this information. This information is partially exposed in the end-user interface. The interface between the client and server, on the other hand, provides enough information about lock requests on pages for the HMS to integrate the domain. Hence, the HMS in the Arcadia configuration tracks lock requests on pages made at the interface between the client and the server to integrate the concurrency control functions of Exodus.

Neither of these problems are meant to point out that Exodus is at all poorly designed. To the contrary, because of its clean design, we were able to easily reuse portions of the Exodus software to implement some of the A la carte Framework Objects that supply communications and threading services (for which we are very grateful!). Its generally well-specified interfaces fostered our experimentation with these new approaches to heterogeneous systems integration. These exceptions merely underscore the need for more experience with these approaches so that more specific recommendations and requirements can be made for future systems that are intended to interoperate in heterogeneous and distributed environments.

7.1.3 Transaction Management and Recovery. In addition to integrating concurrency control domains, the HMS must also integrate the transaction management and recovery of the autonomous systems. In this configuration, neither one of the autonomous systems provides the *PrepareToCommit* operation required for a distributed 2-phase commit in their native state. Since Mneme is being extended with A la carte Library Components, it can be configured to provide this call. In this scenario, the HMS implements a commit protocol in which the Mneme stores are transitioned to a semi-permanent state and are “prepared-to-commit” before the Exodus stores are committed. Then, if the Exodus stores commit successfully, the Mneme stores can be committed. If an Exodus store does not commit successfully, the heterogeneous transactions at the Mneme stores can be undone. The worst case scenario is if there is more than one Exodus store being used at a time and one of them crashes after some other Exodus store had been committed; in this case, inconsistency in heterogeneous transactions can arise (Breitbart and Silberschatz, 1988; Du and Elmagarmid, 1989*a*). This characteristic of the configuration (i.e., no *PrepareToCommit* call in more than one of the autonomous systems) identifies one of the constraints that can be triggered by the A la carte Framework during the integration process.

The integration of these domains can also be analyzed to determine *where* their integrating protocols and parameters should be implemented in the architecture. Again, the interfaces to the domains being integrated are provided at both the end-user and server interfaces in each of the autonomous systems. This might lead one to believe that the "intercept messages" approach used for concurrency control is a viable solution for providing the integration mapping for these domains as well.

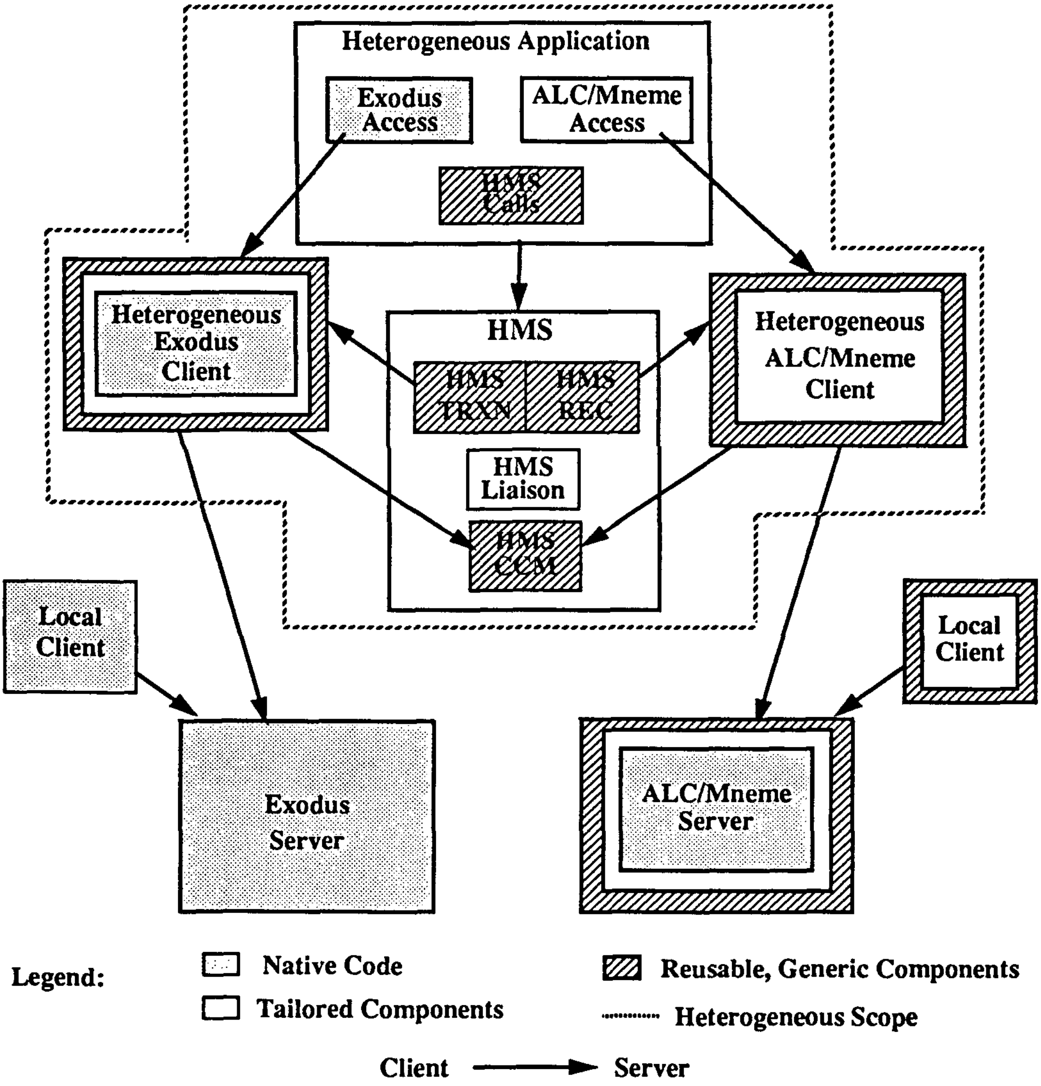
Assume that this approach were taken. Then a heterogeneous application would send local transaction management operations, such as *CommitTrxn,* directly to all of the autonomous systems to which it needs access. As the clients passed the transaction requests to the autonomous servers, the heterogeneous manager would "intercept" the messages to determine if the operation were legal from a global transaction management perspective. The heterogeneous transaction manager could block certain operations, like a lock manager does, if it detected some inconsistent state.

This approach decomposes the operation of the heterogeneous transaction manager too much. The heterogeneous application would have to perform much of the bookkeeping that a heterogeneous transaction manager normally does. For example, the heterogeneous application would have to track which transactions are submitted to which sites and it would have to perform the appropriate recovery operations if one of the autonomous systems did not respond correctly. While the domains would be integrated, the architecture may be considered too visibly heterogeneous for a system that is supposed to integrate the transaction management mechanisms of the autonomous systems. Hence, the Library Components which implement heterogeneous transaction management and recovery are implemented as part of the HMS is executed before submitting local requests to the autonomous systems.

7.2 An Implementation

Figure 11 shows the implementation strategy for the architecture that has resulted from our analysis. This picture shows how the A la carte Library Components

Figure 11. A View of the Underlying Implementations in the Configuration



that implement the Configuration Model shown in Figure 10 are distributed in the Arcadia configuration. The parts of the system separated by the dashed-line box represent the two major scopes of control in this configuration. All of the components within the dashed-line box make up the heterogeneous management system's scope of control and those outside the dashed box (i.e., the Exodus and ALC/Mneme servers and their respective local clients) represent systems with local scopes of control. Hence, the Arcadia configuration is susceptible to inconsistencies that can arise when transactions have access to the same data but operate under different management systems that are not completely coordinated (e.g., a two-phase commit is not readily implementable).

The different types of software in the components of the configuration emphasize which parts of this configuration are applicable to any heterogeneous architecture and which are not. Each of these software types are depicted with a different pattern in Figure 11. Those components made up of native code, like Exodus and parts of the ALC/Mneme server, are shown with the sparsely dotted boxes. Those that are generic, reusable components from the A la carte Library, like most of the components in the HMS, are shown with the diagonally-lined boxes. These pieces of code can be reused in many heterogeneous architectures. Those components that are from the A la carte Library, but that have been have been somewhat tailored to accommodate the integration of a particular system, like the components used to extend Mneme, are depicted as unfilled boxes. These components are mostly reusable in different architectures, however, they do have to be specifically configured to know about the parameter types of the system that they are extending. For example, the Concurrency Control Manager is configured to lock Mneme objects and the Communications Manager is configured to pass messages that will execute Mneme interface calls.

The communication lines of the configuration show how the interfaces of the Framework Objects are exposed to integrate their respective domains. (The communication lines are depicted in Figure 11 as arrowed lines where the arrow points to component acting as a server for a set of messages.) In particular, the separate message passing interface to the Concurrency Control Manager in the HMS is the mechanism used to intercept the locking messages between the heterogeneous clients and autonomous servers so that the concurrency control functions can be integrated properly. The HMS's Transaction Manager and Recovery Manager, on the other hand, provide their interfaces to the heterogeneous applications directly. These different server interfaces are created by defining the interface operations of the desired Framework Object, or Objects, to be the messages accepted by a Communications Manager from the A la carte Library. The integrating parameters of the Framework Object are typically passed as data within in the message. The overall behavior of the configuration is achieved by messages flowing through these interfaces; some examples are described next.

7.2.1 The Configuration's Operation. A heterogeneous application can have concurrent access to objects in both Exodus and Mneme in this configuration. To do this, the heterogeneous application first executes the *BeginHetTrxn* operation of the Heterogeneous Manager which implements the end-user interface of the HMS. The HMS then executes the *BeginExecution* operation of its Liaison Manager to create a heterogeneous liaison client[7] which will execute local transactions on its behalf. Each liaison client in turn connects with its respective autonomous server. The Liaison Manager then executes its *BeginLocalTransaction* operation that instructs the liaison to begin a transaction at the local site. Upon completing this operation, the client returns the local transaction ID for that liaison and its process ID to the HMS. The HMS repeats this process for every local transaction that the heterogeneous transaction has requested to start. When the process is complete, the HMS returns a set of transaction and process ID pairs, each of which uniquely identify a liaison, to the heterogeneous application. Note that if the heterogeneous liaison were multi-threaded, the HMS could start one liaison per autonomous store to handle all heterogeneous requests. If, on the other hand, the client's are not multi-threaded (as in this experiment), a new liaison must exist for each local liaison transaction.

Given a set of liaison IDs, the heterogeneous application can now communicate with the access interface of each liaison directly via the serverized interface of each of the heterogeneous clients. The heterogeneous application knows which transaction it owns at the autonomous site and can now access objects using the native access operations of the autonomous systems. Like the interfaces in the HMS, these interfaces were also constructed by defining the native object access operations of the autonomous systems to be the messages supported by an A la carte Communications Manager. (The same process was also used to serverize the ALC/Mmene autonomous system.) In this fashion, the HMS does not integrate the schemas or access interfaces of the autonomous systems, but it does make the

[7] There are a number of policies that the heterogeneous manager could use to manage creation, execution, and destruction of liaison processes. The most sophisticated policy might keep a set of liaisons around for reuse and adapt its management policy depending on usages. For now, A la carte supports a fixed number of liasons.

interfaces available to a heterogeneous application that cannot execute as local application of an autonomous server.

As the heterogeneous application makes object access requests, each of the autonomous systems decompose the calls into operations of the functional domains that are required to complete the request. For example, a *DeleteObject* operation in a heterogeneous client interface will decompose into several lower level calls, one of which will be a *GetWriteAccess* call to the Concurrency Control Manager. In the case of Exodus, this decomposition was already implemented in the Exodus client code. In the case of Mneme, the A la carte Autonomous Manager implements the decomposition from native Mneme object access calls to internal operations on A la carte components, such as the *GetWriteAccess* operation of the Concurrency Control Manager, and on the Mneme Storage Manager.

As the analysis in the previous section revealed, the concurrency control operations of the autonomous systems should be intercepted by the HMS to provide correct behavior. Hence, whenever a concurrency control operation is invoked in a heterogeneous client, a message is sent to the HMS to execute the respective operation in its Concurrency Control Manager. In our example, the heterogeneous client first passes a *GetWriteAccess* message to the HMS Concurrency Manager before requesting the lock in the autonomous server. This message contains information about the autonomous server lock request, such as what is being locked and which transaction is making the request. The HMS does a look-up in its own internal tables to determine whether or not this local transaction belongs to a heterogeneous transaction. If it does, it then checks to see if the operation is legal with respect to global synchronization and deadlock. If the operation is legal, the HMS records it in its own internal management system and returns control to the liaison which then proceeds to make the original request to the autonomous server. If the request is illegal with respect to global consistency, the HMS blocks the operation until global consistency can be maintained.

Finally, the heterogeneous application commits or aborts its heterogeneous transaction across all of the autonomous systems by executing the *CommitHetTxn* or *AbortHetTxn* operation in the HMS, not in the autonomous systems directly. The HMS will again determine the legality of the request from a global perspective. Based on its determination, it will send the appropriate messages to each liaison either to participate in a distributed commit or to abort. The HMS does not notify the application of the success or failure of the operation until all liaisons have completed their respective tasks.

7.3 General Remarks

Mneme provided us with a good example to investigate approaches to integrate a system that needs some extensions to interoperate with other systems. It underscores the most basic characteristic of heterogeneous environments: the systems to be integrated are typically pre-existing and have been chosen based on criteria other than how well-suited the system is to interoperate with others. An extensible heterogeneous architecture must provide an approach to deal with this characteristic. The integration process applied to this system also emphasizes the notion of incremental systems integration and extensibility. The components with which Mneme is extended could be interchanged as the requirements of an environment changed. It could also be mixed and matched with different components to tailor its behavior from one architecture to another.

The architecture of the serverized Exodus provided an interesting testbed for the analysis of heterogeneous systems integration. Its clean design facilitated our approach. With Exodus, we were able to reveal some of the more subtle issues that need to be addressed in the design of future systems if they are to interoperate in a heterogeneous configuration. Since it's architecture and design is fairly representative of next-generation database systems, we feel that the approach taken and the lessons learned will be applicative to future heterogeneous database environments.

8. Conclusions

We conclude with a description of the current status of the system, some initial recommendations to database designers, and some directions for future work.

8.1 Current Status

The A la carte Framework Objects and approximately 50% of the toolkit environment was implemented in the Eiffel programming language (Meyer, 1988). Many of the A la carte Library Components are implemented and operational as well. Most of the Arcadia configuration is fully implemented except for the heterogenous recovery management components which have only been analyzed for the design of the system. The transaction management extensions for Mneme are under construction and the serverized version of it is currently operational. Finally, the heterogeneous liaisons are also implemented. These components are constructed out of the A la carte Framework Objects, written in Eiffel, and the native programming languages of the autonomous systems: C++ for Exodus and C for Mneme. There are 69 Eiffel classes, all tolled, contributing to the A la carte code in the Arcadia configuration. There are approximately 150 classes which comprise all of the A la carte Framework Objects and the internal management system (i.e., the code which implements prototypes of the creation and constraint managers) in the toolkit environment.

8.2 Summary

The A la carte Framework defines a multi-dimensional model that helps clarify the process of heterogeneous systems integration from a very applicative standpoint. It is intended to help systems integrators understand the implications of certain design decisions and the alternative architectures that are available to them. By uncovering the internal behaviors of autonomous systems, this framework supports the creation of tailorable heterogeneous configurations. Yet, despite its systems-orientation, its meta-level representation of heterogeneous architectures makes it a very flexible and reusable tool.

We believe that this approach to systems integration is not only feasible, but preferable to the more traditional "black-box" approaches to database integration. By truly integrating and managing the disparity between system implementations, we may not only be able to achieve correct systems operations, but we may also be able to optimize these configurations for particular application domains.

8.3 Design Recommendations

Our experience with A la carte leads to some recommendations to designers of future database systems and heterogeneous management systems.

8.3.1 Designing Interoperable, Autonomous Database Systems. Systems that are intended to interoperate in a heterogeneous environment should provide truly open architectures. From an A la carte perspective, "openness" refers to the exposure of a system's key functional interfaces. These interfaces should provide an essential set of operations for each functional domain; the A la carte specifications for the domains of transaction management, concurrency control, recovery, and access seem to be representative. The granularity of the functional components should be chosen carefully. Too fine of a decomposition will make the integration process unmanageable and too large of a granularity can result in inconsistent behavior. Indeed, a concurrency control manager is typically embedded inside of a transaction management system, however, as examples in this article show, access to its interface can allow a heterogeneous manager to integrate a set of systems in a way that avoids undesirable behavior. These interfaces should be exposed at convenient architectural boundaries (e.g. the boundary between a client and a server.) Here, there are trade-offs in where these interfaces are placed and how process boundaries should be split. If two functional domains within a system need to communicate, it may seem more convenient initially to put both domains into one process and not expose the information that they pass to one another. However, this choice may make the integration of those domains more difficult.

The parameters of the functional domains should also be carefully designed. The consistent design of these parameters and exposure of these parameters with their associated domains are as important to building heterogeneous architectures as the exposure of the proper interface calls. Further, it would be most convenient, if the format of these data types could be somewhat standardized. Ironically, some of the more “unruly” aspects of the integration process involve integrating different implementation formats of these data types, not the interface calls of the autonomous systems.

And finally, the interfaces to the functional domains should be used consistently within the autonomous system itself. If the autonomous system does not cleanly decompose its operation into the operations of its functional domains, components of other systems will not be able to be easily integrated with them. Some extra work may be required to track down exceptions and, in some cases, these exceptions may make complete integration impossible.

8.3.2 Designing Extensible Heterogeneous Architectures. Perhaps the most crucial requirement for a heterogeneous management system is the ability to capture the vital operations and data that represent the essence of heterogeneity so that correct execution can be achieved. The A la carte Framework gives an example of how this might be accomplished. There are two parts to satisfying this requirement. One part is to know what these operations and data are. To this end, we have identified some properties and dependencies that represent some of the definitive characteristics of heterogeneous transaction management systems, and a method of decomposing the functions of systems into components that reflect operational heterogeneity. This view can help give some insight into what exactly should be integrated in a heterogeneous system.

The other part to satisfying the requirement is to provide the appropriate extensible representation in which to capture this information. A la carte provides a harness for this information and some examples of how to map between specific database implementations and the harness. By supplying a meta-level harness for integration, new systems can be introduced into existing configurations and others can be removed. The heterogeneous manager is not "hardwired" to the systems in the configuration. This framework also allows for the behavior of the heterogeneous manager itself to be reconfigurable. A heterogeneous manager's behavior may need to change if new demands are placed on it from an end-user community, or if a change in the makeup of the configuration requires a new set of functionality from it. Indeed, given that the harness provided the appropriate functional components, the same framework should be able to be reused to configure either a heterogeneous or autonomous system; the extension of Mneme is an example of this process.

Another desirable feature of a heterogeneous management system is the ability to perform "incremental integrations." Not all users may want an entire monolithic system integrated, particularly in very complex environments. A integrated transaction management system which leaves the language interfaces to the autonomous systems visibly heterogeneous may be all that is required—and easier to supply for a given set of systems. Further, because the system should be extensible, new domains may be added or removed, and mixed and matched, as required.

The Arcadia experiment revealed a requirement for heterogeneous systems that is generally not addressed in the database community. These sophisticated architectures require communications packages that allow the roles of client and server to be as easily mixed, matched, and combined as the other components of the Framework. Special care must be taken in the design of the communications packages so that they are as extensible as the rest of the system components.

Finally, in a distributed architecture where sub-components of management systems may potentially exist in different processes, the ordering of the execution of the overall system may require special analysis to assure that no communications deadlock or incorrect ordering of operations can occur. This underscores the importance of capturing the operational model of a system. It is important to explicitly state how these components interact and what states are legal, otherwise untold inconsistencies might result.

8.4 Future Directions

Future work can include several types of efforts. Other database systems, both commercial and research prototypes, can be integrated using the A la carte Framework. The Framework may also be extended to include the representation of other functional domains like schema representation, query processing and optimization, and security. Similarly, the Library can be extended to include other less "traditional" transaction management schemes. While we do not expect this new functionality to dramatically impact the specification of the Framework Objects, it would most likely have some impact on the properties and dependencies represented in the Meta-Model. Finally, we could perform empirical analysis on different architectures and functional distribution schemes. Our approach has been to try to understand the integration process based on practical experience. As more types of systems are integrated using this Framework, the formal aspects of the Meta-Model can be enhanced.

Acknowledgments

We gratefully acknowledge the Arcadia researchers for their participation in defining the heterogeneous configuration described in this article. In particular, Leon Osterweil provided helpful insights into the role of interoperability in integrated software environments. Jack Wileden, Lori Clark, Stan Sutton, Peri Tarr, and Mark Maybee also participated in many discussions with us about the various aspects of object management in Arcadia. Thanks also go to the Exodus researchers and project team, especially Dave DeWitt, Mike Zwilling, and Dave Haight, who were instrumental in helping us understand the internals of the Exodus Client/Server architecture. Our thanks go to Patrick Pfeffer for many very helpful discussions about heterogeneous architectures and for his work in the design of the Arcadia configuration. And finally, the other members of the A la carte implementation team, Martin Cibulka, Dennis Colarelli, Jean Claude Franchitti, and Mitch Smith, deserve special acknowledgment for their efforts.

Pamela Drew was supported by US WEST Advanced Technologies. Roger King was supported by ONR, contract NOOO14-88-K-0559, by US WEST Advanced Technologies, contract USWEST-OCG0683B, and by IBM, contract 880450. This material is based upon work sponsored by the Defense Advanced Research Projects Agency (DARPA), grant number MDA972-91-J-1012.

References

Alonso, R., Garcia-Molina, H., and Salem, K. Concurrency control and recovery for global procedures in federated database systems. *IEEE Data Engineering Conference,* Los Angeles, 1987.

Andrews, T. and Harris, C. Combining language and database advances in an object-oriented development environment. *Object-Oriented Programming, Systems, Languages, and Applications,* Orlando, 1987.

Banerjee, J., Chou, H.T., Garza, J., Kim, W., Woelk, D., Ballou, N., and Kim, H.J. Data model issues for object-oriented applications. *Transactions on Office Information Systems,* 1:3–26, 1987.

Batory, D.S., Barnett, J.R., Garza, J.F., Smith, K.P., Tsukuda, K., Twichell, B.C., and Wise, T.E. Genesis: An extensible database management system. In: Zdonik, S. and Maier, D., eds. *Readings in Object-Oriented Database Systems.* San Mateo, CA: Morgan-Kaufmann, 1990, pp. 500–518.

Bernstein, P.A., Hadzilacos, V., and Goodman, N. *Concurrency Control and Recovery in Database Systems.* Reading, MA: Addison-Wesley, 1987.

Bershad, B.N., Ching, D., Lazowska, E., Sanislo, J., and Schwartz, M. A remote procedure call facility for interconnecting heterogeneous computer systems. *IEEE Transactions on Software Engineering,* 10:738–753, 1984.

Bershad, B.N., Lazowska, E., Levy, H., and Wagner, B. An open environment for building parallel programming systems. *ACM SIGPLAN Notices Conference on Parallel Programming: Experience with Applictions, Languages, and Systems,* New Haven, CT, 1988*a*.

Bershad, B.N., Lazowska, E., and Levy, H. PRESTO: A system for object-oriented parallel programming. *Software—Practice and Experience,* 18:713–732, 1988*b*.

Breitbart, Y. and Silberschatz, A. Multidatabase update issues. *SIGMOD Record,* 17:135–142, 1988.

Breitbart, Y., Silberschatz, A., and Thompson, G. Reliable transaction management in a multidatabase system. *Proceedings of the ACM SIGMOD Conference,* Atlantic City, NJ, 1990.

Buchmann, A.P., Modeling heterogeneous systems as an active object space. *Fourth International Workshop on Persistent Object Systems: Design, Implementation, and Use,* Martha's Vineyard, MA, 1990.

Buneman, O.P., Davidson, S.B., and Watters, A. Federated approximations for heterogeneous databases. *NSF Workshop on Heterogeneous Databases,* Chicago, 1989.

Carey, M.J., DeWitt, D.J., Frank, D., Graefe, G., Richardson, J.E., Shekita, E., and Muralikrishna, M. The architecture of the EXODUS extensible DBMS. *International Workshop on Object-Oriented Database Systems,* Pacific Grove, CA, 1986*a*.

Carey, M.J., DeWitt, D.J., Richardson, J.E., and Shekita, E. Object and file management in the EXODUS extensible database system. *Proceedings of the 11th International Conference on Very Large Databases,* Kyoto, Japan, 1986*b*.

Carey, M.J., DeWitt, D.J., and Vandenburg, S.L. A data model and query language for EXODUS. *Proceedings of the ACM SIGMOD Conference,* Chicago, IL, 1988.

Chrysanthis, P.K. and Ramamritham, K. ACTA: A framework for specifying and reasoning about transaction structure and behavior. *Proceedings of the ACM SIGMOD Conference,* Atlantic City, NJ, 1990.

Chrysanthis, P.K. and Ramamritham, K. A formalism for extended transaction models. *International Conference on Very Large Databases,* Barcelona, 1991.

Drew, P. A la carte: An implementation of a toolkit for the incremental integration of heterogeneous database management systems. Ph.D. Thesis, University of Colorado, Boulder, CO, 1991.

Drew, P. and King, R. An extensible framework for integrating heterogeneous database management systems: A case study, *NSF Workshop on Multidatabases and Semantic Interoperability,* Tulsa, OK, 1990.

Drew, P., King, R., and Bein, J. A la carte: An extensible framework for the tailorable construction of heterogeneous object stores. *Fourth International Workshop on Persistent Object Bases,* Martha's Vineyard, MA, 1990.

Du, W. and Elmagarmid, A.K. A paradigm for concurrency control in heterogeneous distributed systems. *Technical Report 894, Computer Science Dept.,* W. Lafayette, IN: Purdue University, 1989*a.*

Du, W. and Elmagarmid, A.K. Quasi-serializablity: A correctness criterion for global concurrency control in Interbase. *Proceedings of the International Conference on Very Large Databases,* Amsterdam, 1989*b.*

Elmagarmid, A.K. and Leu, Y. An optimistic concurrency control algorithm for heterogeneous distributed database systems. *IEEE Data Engineering Conference,* Los Angeles, 1987.

Gibbons, P.B. A stub generator for multilanguage RPC in heterogeneous environments. *IEEE Transactions on Software Engineering,* 13:12–30, 1987.

Gray, J. Notes on database operating systems. In: Seegmuller, G., ed. *Notes in Computer Science 60, Advanced Course on Operating Systems.* New York: Springer-Verlag, 1978.

Gray, J. Granularity of locks and degrees of consistency in a shared data base. In: Nijssen, G., ed. *Modelling in Database Management Systems.* Amsterdam: North Holland, 1976.

Hayes, R., Hutchinson, N.C., and Schlichting, R.D. Integrating Emerald into a system for mixed-language programming. *Computer Languages,* 15:95–108, 1990.

Heiler, S. Control mechanisms in an object management system for supporting interoperability of heterogeneous components. *NSF Workshop on Heterogeneous Databases,* Chicago, 1989.

Heimbigner, D. Triton reference manual. *Technical Report CU-CS-483-90.* Department of Computer Science, Boulder, CO: University of Colorado, 1990.

Heimbigner, D. and McLeod, D. A federated architecture for information management. *ACM Transactions on Office Information Systems,* 3:253–278, 1985.

Hornick, M. and Zdonik, S. A shared, segmented memory system for an object-oriented database. *ACM Transactions on Office Information Systems,* 5:70–95, 1987.

Jones, M.B., Rashid, R.F., and Thompson, M.R. Matchmaker: An interface specification language for distributed processing. *Proceedings of the 12th ACM Symposium on Principles of Programming Languages,* Washington, DC, 1985.

Landers, T. and Rosenberg, R.L. An Overview of multibase. In: Schneider, H., ed. *Distributed Databases.* New York: North-Holland, 1982.

Linneman, V., Kuspert, K., Pistor, P., Erke, R., Kemper, A., Sudkamp, N., Walch, G., and Wallrath, M. Design and implementation of an extensible database management system supporting user-defined types and functions. *Proceedings of the Fourteenth International Conference on Very Large Databases,* Los Angeles, 1988.

Liskov, B., Bloom, T., Gifford, R., Scheifler, R., and Weihl, W. Communication in the Mercury system. *Programming Methodology Group Memo 59-1,* Cambridge, MA: MIT, 1988.

Litwin, W., Abdellatif, A., Zeroual, A., Nicolas, B., and Vigier, P. MSOL: A multidatabase language. *Information Sciences,* 49:59–101, 1989.

Litwin, W. and Tirri, H. Flexible concurrency control using value dates. *Integration of Information Systems: Bridging Heterogeneous Databases.* New York: IEEE Press, 1989, pp. 43–58.

Maier, D., Stein, J., and Otis, A. Development of an object-oriented DBMS. *Object-Oriented Programming, Systems, Languages, and Applications,* Portland, OR, 1986.

Manola, F. and Dayal, U. PDM: An object-oriented data model. *International Workshop on Object-Oriented Databases,* Pacific Grove, CA, 1986.

Maybee, M. and Dykes, S.D. Q: Towards a multi-lingual interprocess communications model. *Technical Report CU-CS-476-90,* Boulder, CO: University of Colorado, 1990.

Meyer, B. *Object-Oriented Software Construction.* New York: Prentice Hall, 1988.

Moss, J.E.B. Nested transactions: An approach to reliable distributed computing, Ph.D. Thesis, Cambridge, MA: MIT, 1981.

Nodine, M., Skarra, A., and Zdonik, A. Synchronization and recovery in cooperative transactions. *Fourth International Workshop on Persistent Object Systems,* Martha's Vineyard, MA, 1990.

Pu, C. SuperDatabases for composition of heterogeneous databases. *IEEE Data Engineering Conference,* Washington, DC, 1988.

Richardson, J.E. and Carey, M.J. Programming constructs for database system implementation in EXODUS. *Proceedings of the 1987 ACM SIGMOD Conference,* San Francisco, 1987.

Rowe, L. and Stonebraker, M. The POSTGRES data model. *Proceedings of the 13th Conference on Very Large Data Bases,* Brighton, 1987.

Russo, V.F. and Campbell, R.H. Virtual memory and backing storage management in multiprocessor operating systems using object-oriented design techniques. *Object-Oriented Programming, Systems, Languages, and Applications,* New Orleans, 1989.

Schwarz, P., Chang, W., Freytag, J.C., Lohman, G., McPherson, J., Mohan, C., and Pirahesh, H. Extensibility in the starburst database system. *International Workshop on Object-Oriented Database Systems,* Pacific Grove, CA, 1986.

Siegel, M. and Madnick, S.E. Schema integration using metadata. *NSF Workshop on Heterogeneous Databases,* Chicago, 1989.

Skarra, A. Concurrency control for cooperating transactions in an object-oriented database. *SIGPLAN Notices,* 24:145–147, 1989.

Skarra, A. Localized correctness specifications for cooperating transactions in an object-oriented database. *IEEE Bulletin on Office and Knowledge Engineering,* Summer, 1990.

Sterling, L. and Shapiro, E. *The Art of Prolog.* Cambridge: MIT Press, 1986.

Stonebraker, M. and Rowe, L. The design of POSTGRES. *Proceedings of the 1986 ACM SIGMOD Conference,* Washington, DC, 1986.

Taylor, R.N., Blez, F.C., Clarke, L.A., Osterweil, L., Selby, R.W., Wileden, J.C., Wolf, A.L., and Young, M. Foundations for the Arcadia environment architecture. *SIGSOFT Software Engineering Notes,* 13:1–13, 1988.

Templeton, M. Schema translation in Mermaid. *NSF Workshop on Heterogeneous Databases,* Chicago, 1989.

Thatte, S. A modular and open object-oriented database system. *SIGMOD Record,* 20:47–52, 1991*a*.

Thatte, S. Preliminary architecture workshop. *Report on DARPA Open OODB Workshop I,* Dallas, TX, 1991*b*.

Wileden, J.C., Wolf, A.L., Rosenblatt, W.R., and Tarr, P.L. Specification level interoperability. *Proceedings of the 12th International Conference on Software Engineering,* Nice, France, 1990.