Persistent Object State Management in Legion

Draft - March, 1997
legion@virginia.edu

Abstract
The persistence model for the Legion object-oriented wide-area metasystem is described. This model defines the protocols and mechanisms that are used to deactivate, reactivate, and migrate persistent objects within Legion. The persistence model requires the cooperation of three key entities: the persistent object itself, which is responsible for saving and restoring its volatile state on deactivation and reactivation, respectively, Vault Objects, special Legion Core Objects that manage the persistent states of inactive objects, and Legion Class Objects which manage the association between their persistent instances and Vault objects.

1 Introduction

The basic goal of the Legion project[3] is to construct a wide-area, high-performance metasystem[2] capable of providing a usable programmer interface to the increasingly complex nation-wide computing and communications infrastructure. The basic unit of program composition and scheduling in Legion is the active object, and the programming interfaces supported by Legion are object-oriented. Objects in Legion are logically address-space disjoint, and reside in a single unified name-space managed by a set of Legion Core Objects. The primary mode of object interaction is method invocation.

Legion objects may be persistent in nature, retaining state and responding to methods indefinitely beyond the lifetime of the object that created them. The *Persistent State* of a running Legion object can consist of data in memory, pending results of member function invocations on other objects, unserviced member function invocations from other objects, mass storage resources (e.g. open files), graphical display resources (e.g. terminal windows), as well as many other operating system and hardware platform dependent resources. Since Legion objects are active, the state of an object may also include run-time data such as the call stack and register values of the objects threads of control. *Persistent Object State Management* in the context of Legion is defined to be the ability to capture the state of a Legion object into a linearized, transportable, and possibly architecture independent form, and conversely to reconstruct the state of a running object based on a previously captured state.

A general purpose protocol for persistent object state management is central to a number of basic Legion attributes and services. First, the number of persistent objects in the Legion system at any point in time will certainly exceed the number of running objects which could be efficiently supported by the hardware infrastructure available to the system. The ability to move objects from *Active* to *Inert* states will be central to the scalability of Legion object scheduling services. Beyond this basic issue of scalability, a number of important Legion services could utilize object state management facilities. For example, some fault tolerance schemes are based on the ability to checkpoint the state of a running application and later restart needed program elements in case of failure. Another example is dynamic load balancing through object migration, which addresses the issue of achieving good application performance in the face of shared-resource environments.

This document describes the design of the fundamental protocols and mechanisms used within Legion for persistent object state management. In Section 2 we provide a general overview of the Legion persistence model and introduce the key terms and object classes. In Section 3 we examine the role that object's play in managing their own persistent state. In Section 4 we describe the Legion Core Objects known as Legion Vaults and discuss their role in the persistence model. In Section 5 we describe the role that Class Objects play in the persistence model. In Section 6 we examine implementation issues including the design of key Legion library classes that are used to code persistent objects, and the design of Legion Vaults for different environments. Section 7 contains concluding remarks.

We should note that this document assumes familiarity with the basic Legion system design as described in [4]. Also, we will refer the reader to the Legion library implementation report [1] for low-level Legion interface details where appropriate.

2 Overview

At the most abstract level, the Legion persistence model is straightforward. A persistent Legion object can be in one of two different states, *Active* or *Inert*. When an object is Active, it is logically running as a process on a Legion Core Object known as a *Legion Host*, and it can be accessed directly via method calls. When an object is Inert, it is logically stored as a persistent string of data on a Legion Core Object known as a *Legion Vault*. Thus, Legion Hosts are simply objects that provide an abstract interface to processors (for the purpose of supporting active objects), and Legion Vaults are Legion objects that provide an abstract interface to storage (for the purpose of storing inactive objects).

When an object is Inert, its state is stored in persistent storage that is managed by a Legion Vault. This stored state for an object is referred to as its *Object Persistent Representation (OPR)*. The mechanism for creating an OPR based on an active, running object is encoded in the object itself in the form of an object mandatory method called `SaveState()`. In response to a `SaveState()` method invocation, an active object will typically examine its active dynamic state and write any necessary state information to its own OPR. Similarly, the ability to recover an active object's state from an existing OPR is encoded in the object implementation in the form of an object mandatory method called `RestoreState()`. In response to a `RestoreState()` invocation an object typically reads its state from its own OPR and thus recovers any relevant active state.

The location of an inactive object's OPR is described by an *Object Persistent Address (OPA)*. Just as an active object's address is assigned to it by a Legion Host when the object's active representation is moved to that Host, an object's OPA is assigned to it by a Legion Vault when the object's persistent representation (OPR) is moved to that Vault. Just as Legion Hosts control the amount and type of processing power that an object may use, Legion Vaults manage the amount and type of persistent storage that an object may use. Thus, the job of the Vault is to act as an inert object manager, much in the same way that a Host acts as an active object manager.

The natural parallel between Hosts and Vaults extends beyond their functionality and into their relationship to Class Objects. In the case of an object instance that is active, the Class object is responsible for selecting and remembering the Host object that manages the instance. Similarly, in the case of an object instance that is inactive, the Class object is responsible for selecting and remembering the Vault object that manages the instance.

From the above high-level description of the Legion persistence model, we can discern three distinct cooperating entities with distinct roles to play: the persistent object itself (i.e. the object whose persistence is being managed), the Vault object, and the Class of the persistent object. In the following sections we examine the roles and interrelationships of each of these entities in greater detail.

3 The Persistent Object Perspective

The most central entity in the Legion persistence model is the persistent object instance itself. As described in Section 2, object instances are responsible for providing methods to save and restore their persistent state. In this section, we examine the role of the persistent object in the Legion persistence model. In particular, we examine the interface of and implementation issues related to the object mandatory

SaveState() and RestoreState() methods.

3.1 The SaveState Method

Every object in Legion must support some (possibly trivial) implementation of the object mandatory method SaveState(). As we have already alluded, the job of SaveState() is to examine the volatile state of an active object and save a record of that volatile state to persistent storage.

The record of an object's dynamic state produced by a SaveState() implementation should be written in a self-contained form that can later be used to restore the object to the exact state it was in at the time of capture. Any information in the object's dynamic state necessary for a complete restart should be explicitly saved by SaveState(). For very general SaveState() implementations, this may be quite complex to program. For example, if an object permitted SaveState() invocations to be accepted before completing already running invocations of other methods, the SaveState() implementation would need to save a record of how much progress had been made on the running methods, and to whom the results must be returned on completion. A SaveState() implementation such as this might be quite complex, requiring that call stacks used by the active object be linearized and stored in addition to member variables, records of not-yet-executed method requests, etc. Of course, more simple (but less general) SaveState() implementations are also possible. For example, consider a SaveState() implementation that is usable only while no other methods are being serviced. In this version of SaveState(), only the object's member variables and a record of not-yet-executed method invocations generally needs to be saved. Thus, the SaveState() implementation problem in this case is greatly simplified.

The general observation that we make about the SaveState() implementation problem is that the complexity of SaveState() can be reduced, but this will typically require that the object wait until its dynamic state reaches some well known simple point in execution before it services the SaveState() invocation. Of course, from the SaveState() caller's perspective, it would be best if the method were serviced as soon as possible. Consider a typical usage of SaveState(): the object is about to be deactivated, and thus a request is being made that it save its state before going to sleep. Ideally, such a request should be serviced as soon as possible.

```
class SaveStateRequest {
public:
   char   urgency;
   long   timeSecs;
   long   timeUSecs;
};

class SaveStateReply {
public:
   char   status;
   long   timeSecs;
   long   timeUSecs;
};
```

Figure 1. SaveState parameter and return types.

In order to accommodate the somewhat conflicting requirements of the SaveState() implementor and the SaveState() caller, the interface to SaveState() permits a protocol to be run between a persistent object and its SaveState() caller. A call to SaveState() must provide a parameter of the type SaveStateRequest (see Figure 1) that indicates an urgency type and a time-out value. The urgency type can currently be one of three values:

- SaveState_Normal - The SaveState() call is not urgent. It should be serviced whenever possible.
- SaveState_AndDie - The SaveState() call is not urgent, but it is being called in order to perform an object deactivation. After servicing the SaveState() call (whenever possible), the object should deactivate itself.
- SaveState_DeathImmanent - The SaveState() call is urgent, and is requested to be performed within the time-out value indicated, after which time the object may be involuntarily deactivated.

The final request type (SaveState_DeathImmanent) can be used to indicate an urgent SaveState() request. For example, perhaps the host on which the object is active plans to shut down shortly. Or perhaps the object has utilized more that its allotted processing time on a host. In either case (and in many others), object deactivation is required within a given time frame.

Of course, an object may not be able to accept a SaveState() invocation within the time allotted by the request. The reply type of SaveState is an object of the class `SaveStateReply` (see Figure 1) that indicates whether or not the `SaveState()` request was serviced, and if not, an estimate of the time required before a new `SaveState()` request could be serviced. This reply type allows the object to effectively request more time so that it can reach a state at which the `SaveState()` can be performed.

The above protocol is quite flexible, both from the perspective of the `SaveState()` implementor and the `SaveState()` caller. Ambitious `SaveState()` implementors can code very accommodating `SaveState()` implementations that always save state and return immediately. Simpler `SaveState()` implementations can request more time until a consistent, simple state is reached. From the caller perspective, very generous Host providers may allow `SaveState()` requests to be serviced at at an object's leisure, while stricter Hosts may have varying degrees of tolerance for unresponsive `SaveState()` implementations.

3.2 The RestoreState Method

The `RestoreState()` method is the reverse of the `SaveState()` operation. This method is responsible for reading the saved state of a persistent object from its OPR on stable storage. Based on the OPR, `RestoreState()` reconstructs the appropriate dynamic state of the invoked object.

Typically, `RestoreState()` is invoked immediately after object activation. In such cases, the `RestoreState()` call is performed by the object itself immediately after the Legion library is initialized. Initialization of the Legion library make the object's OPR available to it through the standard Legion library interface. The object can thus on activation initialize the Legion library, restore its state from the OPR, and only then begin to accept method calls.

The interface to `RestoreState()` does not require an invocation protocol like that of `SaveState()`. Thus, `RestoreState()` requests take no parameters and return only a status character indicating whether or not state was restored successfully.

3.3 Using Legion Object Persistent Representations

Thus far, we have described the interfaces to `SaveState()` and `RestoreState()`, and have provided a general overview of their operation. In this section, we examine the key Legion mechanism involved in implementing `SaveState()` and `RestoreState()` methods: the Object Persistent Representation, or OPR.

Legion objects are endowed with an OPR in which they can store volatile state in the event that they must be deactivated during the operation of the system. Objects may also use their OPR to store data structures that are too large to contain in volatile storage (e.g. a "file" object need not keep its entire state, including the contents of the file, in memory - the file contents can be stored directly in the OPR). The persistent representation of an object is referred to as a Legion Object Persistent Representation, or OPR.

The most basic interface to a Legion OPR is provided by the `LegionOPR` C++ object class. Instances of the class `LegionOPR` are constructed based on `LegionOprAddress` C++ object class instances, but this is generally taken care of internally by the Legion library implementation. At the time of activation, objects are passed a `LegionOprAddress` (which can be thought of a description or pointer to a `LegionOPR`) by the responsible Legion Host object so that they can locate and access their persistent representation. When the Legion library is initialized, the OPR Address is automatically converted into a `LegionOPR` instance using `getLegionOPR()`. The programmer can then access the `LegionOPR` instance for a Legion object using the `GetOPR()` method of the `LegionLibraryState` object class. For example:

```
UVaL_Reference<LegionOPR> myOPR;
myOPR = Legion.GetOPR();
```

The interface to the OPR supports accessing the object’s state in two basic forms: *linearized* and *inflated*. For the purposes of object migration, the persistent representation of an object can be gathered into a *linearized form*, suitable for transport. The linearized form of the OPR, which can be accessed via the `getLinearized()` method on the `LegionOPR` class, is typically not important from the persistent object perspective, but is instead generally used by Vault objects. The more important form of the OPR from the persistent object’s perspective is the directly manipulatable form, the *inflated form*. The inflated form of a `LegionOPR` is encapsulated by the `LegionPersistentBufferDir` C++ object class. As its name implies, the `LegionPersistentBufferDir` is a directory of Persistent `LegionBuffer` objects. The inflated form of the OPR is accessed via the `getInflated()` method on the `LegionOPR` class. For example:

```
UVaL_Reference<LegionPersistentBufferDir> myState;
myState = myOPR.getInflated();
```

The `LegionPersistentBufferDir` class implements an association set that maps null terminated character strings to objects of the `LegionBuffer` class and subdirectory objects of the `LegionPersistentBufferDir` class. Objects of this class can be thought of as directories in a file system that contain string named files (persistent `LegionBuffers`) and subdirectories (`LegionPersistentBufferDirs`), although the implementation of these objects need not be based on a file system.

Sample usage of key elements of the `LegionPersistentBufferDir` interface is depicted in Figure 2. This interface contains methods to determine the number of contained buffers and subdirectories, to determine if a given string maps to a contained buffer or subdirectory, to access, add to, or delete from the contained buffers and subdirectories by name, and to iterate over the contained buffers and subdirectories.

```
UVaL_Reference<LegionPersistentBufferDir> myState;
myState = myOPR.getInflated()

// Check the contents of a directory
if (myState.NumSubdirs() == 0) return -1;
if (! myState.IsContainedSubdir(“State”)) return -2;

// Access a subdirectory
UVaL_Reference<LegionPersistentBufferDir> subDir;
subDir = myState.GetSubdir(“State”);

// Access a contained buffer
UVaL_Reference<LegionBuffer> myData;
myData = subDir.GetBuffer(“My Data”);
```

Figure 2. Sample invocations on an object of class LegionPersistentBufferDir.

The LegionBuffers contained in `LegionPersistentBufferDir` objects are persistent in nature. That is, these buffers are based on storage that is contained in the object’s persistent representation and will thus persist after the object is deactivated. Thus, in manipulating these buffers, the object is directly manipulating its persistent state. This means that an object’s OPR is accessed at the most basic level using the familiar LegionBuffer interface (see [1] for more information about LegionBuffers and packable data structures). Data structures that were rendered packable for the purposes of transport in Legion Messages are equally packable into the LegionBuffers obtained as part of the object’s OPR. This leads to a situation where the implementation of an object’s `SaveState()` method is typically a sequence of pack operations on the data structures that make up the object’s state, many of which already needed to be packable (or made up of packable constituents) for the sake of method service and invocation.

During the operation of a Legion object’s `SaveState()` method, or for the purposes of taking checkpoints, the programmer may need to capture the state of the Legion library. This functionality is provided through the `saveState()` method on the `LegionLibraryState` class. To save the state of the Legion library, the programmer simply writes:

```
        LegionLibrary.saveState();
```

To recover the state from the persistent representation, a complementary `restoreState()` operation is provided, e.g.:

```
        LegionLibrary.restoreState();
```

These methods save and recover (among other data structures) information about method requests received by the object but not yet serviced by the user code associated with the object. Generally, all implementations of `SaveState()` and `RestoreState()` should call on `LegionLibrary.saveState()` and `LegionLibrary.restoreState()` to save and restore the Legion library state, respectively.

4 The Vault Perspective

We have described how an object manipulates its persistent representation in the form of an OPR, but we have not yet described how OPRs in the Legion system are obtained, managed, or migrated. Legion Vaults are the Legion Core Objects that are responsible for managing the OPRs of other Legion objects. A Vault has direct access to the OPRs it holds via persistent store mechanisms outside of Legion (e.g., a Unix file or directory). It administers the creation of and access to the OPRs of a set of objects that it is charged with managing.

Vaults have a number of roles throughout the lifetime of an object. For example, when an object is created, a Vault for the object is chosen by the object's Class (see [4] for a complete discussion of active Class objects). The selected Vault creates a new empty OPR for the object, and supplies to the object an OPR address that refers to this new OPR. Another example of the Vault's roles in the system is object migration. If the object's class (or a Placement Mapper on behalf of that class) decides to migrate the object to another Legion Host, the migration may require moving the OPR to a new Vault whose persistent storage is accessible by objects on the new host. In such a case, the class (or Placement Mapper) selects a new Vault, and the OPR is transferred between the Vaults.

```
class LegionVault {
 LegionOprAddress    createOPR(LegionLOID);
 LegionOprAddress    getOPRAddress(LegionLOID);
 LegionLinearizedOpr getOPR(LegionLOID);
 void  giveOPR(LegionLOID, LegionLinearizedOpr);
 void  deleteOPR(LegionLOID);
 int   isManaged(LegionLOID);
 void  markActive(LegionLOID);
 void  markInactive(LegionLOID);
};
```

Figure 3. The LegionVault interface.

The above Vault activities are supported by the basic LegionVault abstract interface depicted in Figure 3. For the purposes of object creation, the Vault provides a `createOPR()` method. This method constructs a new empty object persistent representation, associates this OPR with the given LOID, and returns the address of the new OPR for use by the newly created object. For the purposes of object activation and deactivation, the Vault provides a `getOPRAddress()` method to determine the location of the OPR associated with any of its managed objects. For performing object migration, Vaults support `giveOPR()` and `getOPR()` methods that transfer a linearized (i.e. transportable) OPR to and from Vaults, respectively. The `deleteOPR()` can be used to terminate a given Vault's management of an object's persistent representation. The `isManaged()` method can be used to determine if a Vault manages a given object. Finally, `markActive()` and `markInactive()` methods are provided so that the Vault can be notified when an object is active or inactive, respectively. This knowledge allows the Vault to store the OPRs of inactive objects in compressed or encrypted forms for efficiency and security purposes, respectively.

A critical detail to note about the Vault interface is its usage of OPR Addresses to provide access to object persistent representations. When an object wants to access its OPR, it can learn the address of its

OPR from its Vault. The Vault must provide an address that contains enough information embedded in it to not only find, but access the OPR. For example, consider an implementation of Vaults and OPRs that is based on the Unix file system. In such an environment, an OPR might be implemented as Unix directory, and an OPR address might contain a Unix path name corresponding to a Unix directory. In this case, the OPR addresses, besides containing the path name needed to locate the OPR, would also need to contain an type indicator that lets the object know that it should access the OPR in the form of a Unix subdirectory. In a sense, the OPR address constitutes an agreement between a Vault and a managed object about what kind of OPR will be used for the object. After this agreement is established, the object can access its OPR directly without consulting the Vault.

5 The Class Object Perspective

As is clear from the roles of persistent objects and Vaults in the persistence model, the pairing of Vaults and persistent objects is an application dependent function. Some objects may not be interoperable with some Vaults. Certain classes may not accept the security policies employed by a given type of Vault. The natural place for Vault/object matching in Legion is in the Legion class system. Class objects perform a similar matching process to decide which Host objects can manage the active representations of of their instances. For example, a certain class may not be interoperable with Host objects of a given architecture, and may require special security attributes from Hosts. The situation with Vault matching is fundamentally the same.

To perform the functions for which it is responsible (including Vault/instance matching), each class object logically maintains a table whose entries contain fields specifying certain attributes of object instances (identified by LOIDs). For example, the logical table must store the Object Address for the instance if it is active, the Placement Mapper for the instance if one is employed, and so on. For the purposes of object persistence, the Class object must maintain for each instance a *Current Vault Set* and a *Candidate Vault Set*. The Current Vault Set contains a list of Vaults that currently have Object Persistent Representation for an object. Typically, only one Legion Vault will have a copy of the Object Persistent Representation of an object, but more complex schemes are possible. For example, the class may elect to replicate an object's OPR to increase availability or performance. The Candidate Vault Set field indicates the Vaults that may be given responsibility for the object. This field can be implemented as a simple list, but typically it needs to encapsulate more sophisticated information, such as “no restriction” or “all Vaults with a given security policy.”

In practice, the class object may employ other Legion objects, such as database servers, to maintain some or all of the information that class objects are required to maintain in what we refer to as the “logical table.” Objects may be given the opportunity by their class to directly manipulate the table fields in a manner reminiscent of reflective architectures.

6 Implementation Issues

Given the above general model for object persistence in Legion, we are still left with a number of basic implementation issues. Of particular interest are the details involved in constructing a new kind of Legion Vault based on a given type of persistent storage, and then incorporating the needed functionality into Legion objects to allow them to interoperate with the new Vault. This process involves the following two key steps:

1. The C++ classes needed to support the Legion library interface to an OPR (as described in Section 3, and in greater detail in [1]) must be implemented. The key classes that must be derived from (i.e. implementations of which must be provided) are:
    - `LegionStorage` - This class is the Legion library C++ abstract interface to storage (e.g. memory, file storage, etc.) and is used as one of the fundamental building blocks of LegionBuffers (see [1] for more details about LegionBuffers). In order for Legion

`SaveState()` and `RestoreState()` implementations to use the standard LegionBuffer interface, a version of the LegionStorage class must be implemented over the persistent storage in question. The interface to a LegionStorage is similar to the interface to a Unix file - the file can be logically considered a linear string of bytes that can be read from or written to.

- `LegionPersistentBufferDir` - Recall from Section 3 that the interface to an object's OPR is based on a directory of buffers abstraction to allow different modules in the object to conveniently save their state without overwriting separate data structures. Thus, to support a new persistent storage type as a Legion OPR format, the `LegionPersistentBufferDir` interface must be implemented using the persistent store. The `LegionPersistentBufferDir` interface provides a Unix-like directory service, where directories map string names to LegionBuffers and contained subdirectories. If the persistent storage in question already supports a directory abstraction (e.g. as in the case of most file systems), the `LegionPersistentBufferDir` can typically be a thin wrapper around the existing directory implementation.
- `LegionOprAddress` - While the above classes provide the programmer interface to an object's OPR, initial access to the OPR is encapsulated within the `LegionOprAddress` class. An implementation of the `LegionOprAddress` interface must be provided that implements the `GetOPR()` method. This method encapsulates the task of examining the data contained within the OPR address (e.g. a directory path name), using that data to construct an appropriate `LegionPersistentBufferDir` instance, and returning the resulting OPR handle.

2. The above classes give a Legion object the ability to utilize a given form of OPR. We must next construct a Vault to create and manage OPRs of the new form. As described in Section 4, the Vault must be able to construct new empty OPRs for newly created objects. For example, in the case of a Vault implemented over the Unix file system, the Vault would simply create a new empty subdirectory to make a new empty OPR. The Vault must generate and keep track of the OPR addresses associated with managed objects. For example, if the OPR associated with a given object is migrated to the Vault, the Vault must store that OPR at a newly generated OPR address, and maintain the association between the new managed object and this OPR address. Continuing the example of a Unix file system based Vault, in order to generate new OPR addresses, this Vault might simply generate path names by incrementing a counter and appending its value to a base string path name.

These two key implementation steps (i.e. the implementation of the Legion library OPR interface and the implementation of the Vault) allow a new form of persistent storage to be used by persistent Legion objects. Note, the implementation of persistent Legion objects (in particular, the implementation of their `SaveState()` and `RestoreState()` methods) is not affected. Furthermore, the implementation of Class objects need not be affected - a class must maintain information about which Vault types are acceptable for its instances, but the implementation of this feature is independent of the implementation of Vaults and OPRs.

7 Conclusion

In this paper, we have described the general design of and the most basic implementation details associated with the Legion persistence model. This model allows Legion objects to persist indefinitely over time and migrate between locations in the system. The critical parts of the model are:

- Object self-management of persistent representations - Objects are responsible for managing their own persistent representations through the implementation of `SaveState()` and `RestoreState()` methods. These methods are implemented using an abstract interface that hides the implementation details of actual persistent storage medium used by the object to hold its persistent representation.

- Legion Vaults - These Legion Core objects manage the association between objects and object persistent representations. In the same way that a Legion Host object abstracts processing power (so, for example, Class objects need not be aware of how to activate objects on every kind of host), Legion Vaults abstract persistent storage (so, for example, Class objects need not be aware of all of the types and instances of persistent storage available in the system).
- Class Object pairing of Vaults and instances - The use of a given Vault by an object instance is a policy decision in many dimensions (for example, performance, security, and so on). In Legion, the Vault/instance pairing policy is located within Class objects.

References

[1] A.J. Ferrari, M.J. Lewis, C.L. Viles, A. Nguyen-Tuong, A.S. Grimshaw “Implementation of the Legion Runtime Library,” *University of Virginia CS Technical Report CS-96-16*, November, 1996.

[2] A.S. Grimshaw, J.B.Weissman, E.A. West, and E. Loyot, “Meta Systems: An Approach Combining Parallel Processing And Heterogeneous Distributed Computing Systems,” *Journal of Parallel and Distributed Computing*, pp. 257-270, Vol. 21, No. 3, June 1994.

[3] A.S. Grimshaw, W.A. Wulf, J.C. French, A.C. Weaver, and P.F. Reynolds Jr. “Legion: The Next Logical Step Toward a Nationwide Virtual Computer”, *University of Virginia CS Technical Report CS-94-21*, June 8, 1994.

[4] M.J. Lewis, A.S. Grimshaw, “The Core Legion Object Model,” *Proceedings of IEEE High Performance Distributed Computing 5*, pp. 551-561 Syracuse, NY, August 6-9, 1996.