Dynamically Configurable Distributed Objects

Dissertation Proposal by Michael J. Lewis
Department of Computer Science
University of Virginia
Charlottesville, Virginia
September 19, 1997

1. Synopsis

The Legion wide-area distributed object computing system will consist of many diverse resources owned and controlled by an array of organizations, each with its own characteristics, requirements, and constraints. Our research group is designing and building the initial Legion implementation, but we intend thousands of programmers and users to grow and evolve the system. Any wide-area distributed system must be able to evolve to meet changing environments and user requirements; therefore, Legion objects must be able to evolve efficiently and effectively from one version to the next. Current distributed object computing systems are not well equipped to evolve and change because the run-time representations of objects are too often treated as static monolithic entities—this is the problem that my research is designed to address.

My contribution will be to introduce, develop, and implement the *dynamically configurable distributed object (DCDO) model.* My goal is to demonstrate that the DCDO model is an effective tool for enabling object evolution in wide-area distributed object computing systems.

Using DCDO technology, programmers will be able to evolve *existing* active Legion objects to accept new member functions, to change the interface and behavior of their member functions, and to remove functions from their public or private interface. Programmers will be able to implement these changes *on-the-fly*—without deactivating any part of the system, without replacing binary executables, without interrupting the clients of evolving objects, and without having to know what the changes will be at the time the object is initially compiled and run. I will provide an implementation of the DCDO model, and will apply this implementation to build several different types of services, including *evolution management policies.*

Demonstrating that the DCDO model is an effective tool requires that I define the meaning of the term "effective" in this context. It is my contention that the model and its implementation will be effective if they enable useful services that would be impossible or significantly less convenient to implement without DCDO technology. I have identified five different types of services that I believe qualify as useful, and whose implementation will be enabled or facilitated by the DCDO model. Therefore, successfully designing and building these services will demonstrate the DCDO model to be effective.

The DCDO model and its implementation will constitute an original contribution for several important reasons. First, they address a new problem—the fact that the run-time representations of active objects typically inhibit the objects' ability to evolve efficiently and effectively—in a new environment, namely wide-area distributed object computing systems. Second, the model incorporates existing features from research areas such as mobile agents and transmissible code, object-oriented database evolution, and object-oriented language implementation, but combines them in a way that has yet to be explored. Finally, the DCDO model prescribes that the run-time representations of objects be decomposed, both logically and physically, into multiple components, and that these components be treated as first-class run-time entities. Further, the model specifies a method of managing the components and composing them at run-time into the objects that they combine to represent.

2. Background

The widespread deployment of gigabit networks will provide significant communications bandwith between computing resources, enabling wide-area distributed object computing systems [18]. These systems will consist of many heterogeneous, distributed, unreliable resources. Without significant software support, users will not be able to manage the complexity of this environment. Metasystems software [29]—software that resides "above" physical resources and operating systems and "below" users and applications programs—is needed. Legion [30, 31, 41], Globus [24], and Globe [32, 33, 62] are examples of wide-area metasystems software. Our research group is designing and building Legion, the system on which I will focus my research.

Legion's components will include a run-time system, Legion-aware compilers that target this run-time system, and programming languages that present applications programmers with a high level abstraction of the system. Thus, Legion will allow users to write programs in several different high-level languages, and will transparently create, schedule, and utilize distributed objects to execute the programs.

Legion users will require a wide range of services in many different dimensions, including security, performance, and functionality. No single policy or static set of policies will satisfy every user, so users must be allowed to implement their own solutions and to determine their own trade-offs as much as possible. Legion supports this philosophy by providing the mechanisms for system-level services such as object creation, naming, binding, and migration, and by not mandating these services' policies or implementations.

Legion requires that all of its objects export a common core functional interface, called the *object-mandatory* interface; the implementation of the functions in the interface is determined by the object or its class. Further, Legion specifies the minimum functional interface to a set of core system object types, including *binding agents*, *context objects*, and *host objects* [41]. Again, the implementation of these system objects will vary. Finally, Legion delegates much of what is usually considered system-level responsibility to classes, which are special Legion objects that export the *class-mandatory* interface. Classes are responsible for creating and locating their instances, and for selecting appropriate security and object placement policies. The core system objects provide mechanisms for user-defined class objects to implement policies and algorithms that best match their instances' requirements in terms of performance, security, cost, and functionality.

Legion will contain large numbers of long-lived objects, too many to simultaneously represent all of them as active processes. Therefore, Legion requires a strategy for maintaining and managing the representations of these objects on persistent storage. A Legion object can be in one of two different states, *active* or *inert*. An inert object is represented by an *object persistent representation (OPR)*, which is a set of associated bytes that exist in stable storage somewhere in the Legion system. The OPR contains state information that enables the object to transition to active state. An active object runs as a process that is ready to accept member function invocations; an active object's state is typically maintained in the address space of the process, although this is not strictly necessary.

Object implementations allow Legion objects to run as processes in the system. An object implementation typically contains machine code that is executed when a request to create or activate an object is made; more specifically, object implementations are generally maintained as binary executables that hosts execute when they receive a request to activate or create an object. However, the implementation model does not require that object implementations be maintained as executable files, nor that they even contain architecture-specific machine code. Architecture independent Java byte code, programs written in interpreted languages, and even high level language source code are acceptable object implementations in the model. Furthermore, multiple object implementations can exist for the same Legion object; different object implementations can exist for different machine architectures, and even for different performance tradeoffs on the same architecture (similar to Emerald [12, 36]). The implementation model simply suggests that class objects maintain object implementations for their instances, and that the implementations be appropriate for the host objects that may need to run instances of that class.

Legion objects, including classes, have the following characteristics: *Shared*—Legion contains a single unified namespace, which means that all Legion objects are visible and potentially accessible from all other objects in the system. *Independent*—the source code and object implementations for client objects are not necessarily part of the same software system as those for server objects.[1] Legion client objects can be defined, built, compiled, and run separately from the server objects with which they interact, and clients can bind dynamically to servers. *Distributed*—Legion objects do not necessarily reside in the same address space, process, or machine as their clients.[2] *Active*—a Legion object can be represented by a running process with at least one thread of control. *Persistent*—Legion objects are potentially very long-lived, and can outlive their creators.

3. The problem

Building software, including Legion objects, is a dynamic process. Very seldom does a significant piece of software work perfectly the first time it is run, and then continue, without change, to be as useful over time as it was when it was first deployed. Systems must be able to evolve for a number of reasons, including the following: (1) Software engineering is an imprecise science; software often contains bugs, and sometimes does not immediately meet the requirements of its users. (2) The environment in which the system operates can change unpredictably, thereby invalidating the assumptions that were made when the software was built. (3) Users' requirements can change after software is initially built, requiring that existing operational software evolve to meet the new requirements.

Evolving software systems can present considerable difficulties for software engineers; an entire area of research, namely software change impact analysis [13], employs myriad strategies to detect and manage software evolution. However, these strategies focus on allowing software engineers to make changes to parts of the source code, and determining the impact of these changes on the source code of the rest of a system. The strategies typically assume that the changing software module will be rebuilt (i.e. recompiled and relinked), and that the new version of the system will replace the old version in its entirety.

The fact that Legion objects are shared, independent, distributed, active, and persistent—as described in Section 2 above—complicates their evolution and requires different strategies from those employed in traditional software systems. In particular, any strategy that requires access to the source code of all potentially affected objects would be inappropriate. Legion server objects and their clients can be built in different programming languages, by different programmers, in different parts of the world.

Current distributed computing systems are not well-equipped to evolve from one version to the next, due to at least two prevailing characteristics of these systems:

1. The use of the terms "client" and "server" is somewhat misleading since Legion is not strictly a client/server architecture; Legion supports the construction and execution of *program graphs* [63], which explicitly allow results of invocations to be forwarded to other Legion objects rather than directly back to the caller. The simplest program graph is the traditional RPC graph. For the purposes of this discussion, interactions between objects can be viewed in a client/server (or caller/callee, or invoker/invokee) relationship.

2. Two different meanings for the term "distributed object" exist; sometimes the term indicates that the object resides in a separate logical address space from that of its clients, and sometimes it means that the object comprises fragments that could exist in separate address spaces, processes, or machines. In this document, I use the term in the former sense; a distributed object contains its own logical address space (which may or may not be implemented in a separate actual address space from the caller), but the term implies nothing about whether the object is fragmented across multiple address spaces. For the most part, the discussions in this document are orthogonal to whether or not the object is fragmented over multiple address spaces.

1. *A static monolithic implementation model*: In the current Legion system, and in other distributed computing systems (CORBA [48], DCE [42], Mentat [28], PVM [59], MPI [43], etc.), the machine code that comprises object implementations exists in binary executable files. The mechanism that creates or starts an object, process, or task simply executes the appropriate binary. Within this static monolithic implementation model, changing any part of the object's interface, composition, or implementation—however minor the change may be—requires replacing the entire executable program and restarting it. The object loses any state it maintains in its address space unless the old implementation explicitly stores it before being halted, and unless the new implementation restores it after being started; this is seldom implemented in practice. Further, the effectiveness of dynamic linking, which is a common operating system mechanism for allowing programs to evolve their behavior without having to be rebuilt, is limited because shared libraries must be present in an expected location wherever objects that link them may run. This requires that the underlying systems be more homogeneous, and usually leads either to statically linked binaries or limiting simplifying assumptions.
2. *The absence of evolution strategies*: In existing distributed object systems, when an object's type, interface, or implementation changes, other objects can become incompatible with the new version of the object. Other areas of research, such as software evolution, software change impact analysis, and object-oriented database schema evolution and version management, address this problem. However, distributed object computing systems lack appropriate tools and strategies for enabling and managing evolution. Typically, changes are made to source code, programmers must identify all affected objects and programs, and must continually relink, recompile, or even reprogram them to reflect the current versions of the objects they use.

As a result of these flaws, it is inconvenient to evolve objects in distributed object computing systems. The goal of this work is to provide objects with the ability to evolve and change. Users should be able to alter the structure and behavior of an object in a more efficient and convenient manner than by replacing the binary executable and restarting the object. The mechanism for altering the object should be as straightforward, automatic, and easy-to-use as possible. Further, it should be powerful enough to enable some set of services that would otherwise be either too inconvenient or difficult to implement, or that would not perform as well with a traditional implementation.

4. The solution

A distributed object computing system must allow both its user-level objects and its system-level services to evolve over time to meet changing requirements and environments. In large-scale shared distributed systems, this must be done while considering the effect that the evolution will have on other objects in the system. We will develop and implement a model that will allow existing Legion objects to evolve while they remain up and running, and while other objects continue to interact with them. My contributions will be focused in three areas designed specifically to address the problems identified in Section 3:

1. I will develop the *dynamically configurable distributed object model* as an alternative to the static monolithic model described in Section 3. The DCDO model treats objects as configurable entities whose functionality and composition can change incrementally as the object runs. The DCDO model is well-equipped to support change, and it will enable algorithms and optimizations that would be inconvenient or that would perform too poorly with static monolithic implementations.
2. I will use an implementation of the DCDO model within the Legion class system to build class and object *evolution management policies* for distributed object computing systems. Rather than searching for fundamentally new approaches to evolution, I will borrow existing ideas in database schema evolution, and I will provide new implementation mechanisms that are appropriate for the Legion object model and environment.

3. The benefits of the DCDO model will not be limited to evolution management policies. Dynamically incorporating code into running Legion objects will enable other useful services unrelated to object evolution. I will identify and build several such services, described in Sections 4.5.1 through 4.5.4.

4.1 The dynamically configurable distributed object model

A *dynamically configurable distributed object* is an active distributed object whose implementation can change incrementally after the object is instantiated and running. A DCDO is made up of *implementation components*, each of which can be either (1) a function in the public interface of the object, (2) a private function that is not in the public interface of the object but that can be called from within the object, or (3) an instance variable along with the full set of operations allowed on that variable within the object. An incremental change to a DCDO is the addition, replacement, or removal of one or more of the implementation components of that DCDO. For example, the implementation of a member function in the object's public interface can be replaced with a different implementation. This would allow an optimized implementation to be plugged into an object on-the-fly, without requiring recompilation of the DCDO itself or of the other objects that use it.

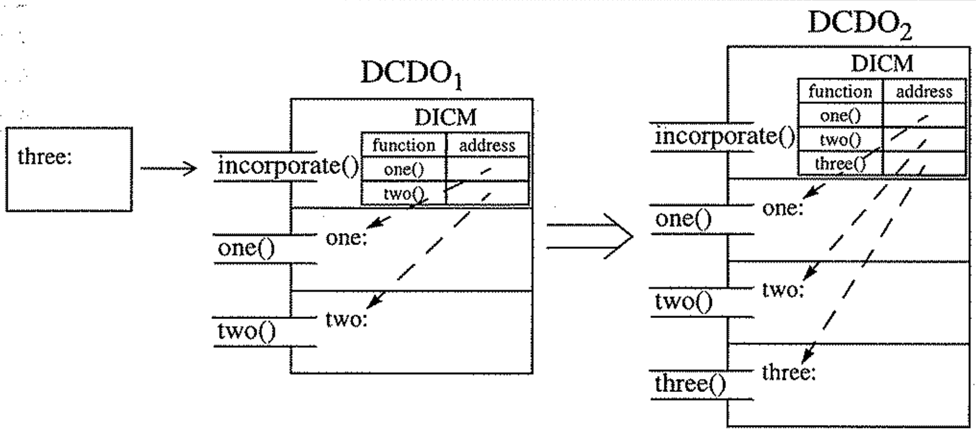

FIGURE 1. *Augmenting the functionality and interface of an object*: DCDO$_1$ initially exports two functions, one() and two(). The implementation component for a third function, three(), is dynamically incorporated into the object by calling its incorporate() member function, thereby creating a new version of the object, DCDO$_2$.

The implementation of a DCDO has two characteristics that allow it to support dynamic configuration through incremental change; (1) it exports a core member function named *incorporate()* which can be used to add, remove, and alter implementation components, and (2) it contains a *dynamic implementation component mapper* (DICM) which maps implementation components to locations within the running image of the object. Other objects can call a DCDO's incorporate() member function[3], which dynamically incorporates new implementation components into the DCDO's running image, and which alters the DCDO's DICM to reflect an implementation change. The DICM manages the DCDO's changing implementation and interface. All references to a DCDO's implementation components—both references from within the object and references made by other objects via remote member function calls—are mapped by the DICM to the correct addresses within the DCDO's running image.

Removing an implementation component requires removing both the component itself from the running DCDO and removing the corresponding map entry from the DICM. Adding a component involves mapping its implementation into the DCDO and updating the DICM to include a map entry for the new

3. Subject to security restrictions implemented in the object [65].

implementation (see Figure 1). Finally, changing a component's implementation entails removing the old implementation, adding the new one, and updating the DICM. These operations are implemented by the incorporate() member function, which takes a list of implementation components as an argument and returns an integer that indicates the success of the operation. If an implementation component in the list is not already present within the DCDO, the component is added to the object. If the implementation component is already present, the current version is replaced by the incoming version. An existing implementation component is removed by passing an implementation component that has the same name, but that contains an empty implementation.

The creator of a DCDO builds the DCDO in two separate steps. First, the creator runs a *seed DCDO*, which contains an empty DICM and which initially exports only the incorporate() member function. Next, the DCDO is configured in one of two different ways; the creator can configure the DCDO by calling incorporate() with the names of new implementation components, or the DCDO can read the names of its implementation components from its OPR, and can essentially call incorporate() on itself. Either way, the process augments the functionality of the object, causes the DICM to be filled in appropriately, and adds functions to the object's interface.

4.2 Dynamically configurable classes

To adhere to the Legion object model, which specifies that class objects define the interface and behavior of their instances, dynamically configuring a DCDO is done in concert with its class. That is, a DCDO does not typically evolve independently of its class, which must be able to answer queries about its instances' interfaces, create new instances, and activate existing inert instances. Legion class objects whose instances are DCDOs are referred to as *dynamically configurable classes* (DCCs). Typical Legion class objects maintain object implementations for their instances as a set of binary executable files. DCCs, on the other hand, maintain an *implementation component store* (ICS) that contains sets of implementation components. Each set comprises one full object implementation for the DCC's instances. Just as normal Legion classes can maintain different executables for different architecture types and performance or security trade-offs, DCCs can maintain different sets of implementation components for the same purposes (see Figure 2).

A compiler or programmer builds a new DCC by first running a *seed DCC* that initially contains an empty ICS. The compiler or programmer then configures the DCC appropriately by calling the *add_component()* member function to augment, alter, or shrink the ICS that the DCC maintains; this step corresponds directly to the process of associating one or more binary executables with normal Legion class objects. The state and contents of the ICS determine the characteristics of the DCC's instances (just as the binary executables associated with normal class objects determine the characteristics of their instances). The user of a DCC can create a new instance of that DCC by calling its *create_instance()* public member function. As described in Section 4.1, DCC class objects will create instances by first running a seed DCDO, and then configuring the DCDO with the appropriate implementation components from its ICS or from the object's initial OPR.

4.3 Enabling change

So far, I have described how normal classes and DCC classes create new instances; each employs a different mechanism that adheres to the Legion model. This section considers the options that the DCDO model and the static monolithic implementation model (as implemented in DCC and normal Legion classes respectively) present to programmers who wish to change the characteristics of their objects.

Since a Legion class defines the characteristics of its instances by maintaining object implementations for them, changing instance characteristics involves changing these implementations; for normal Legion classes, the monolithic binary executable must be replaced in its entirety, and for DCC classes, one or more implementation components must be replaced. Both kinds of classes do equally well in creating new instances after the change—each can simply use its normal mechanism with the new object implementations.

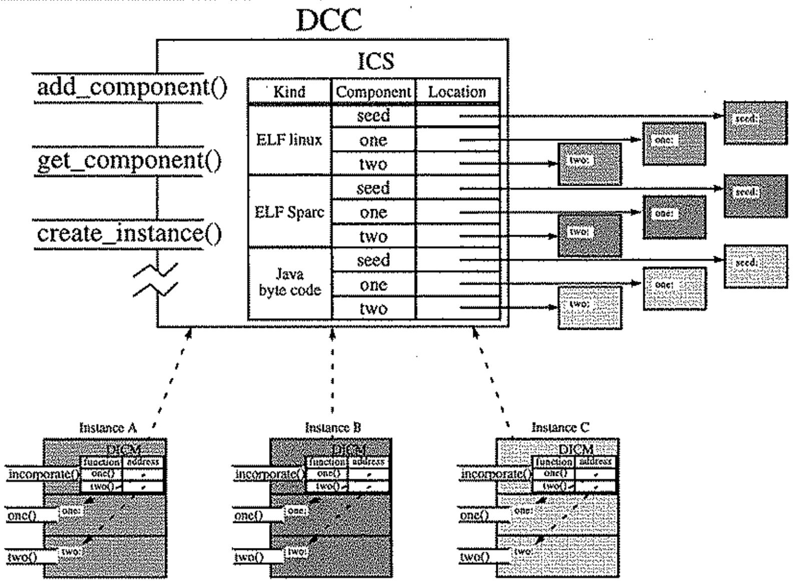

FIGURE 2. *Dynamically configurable classes*: A single DCC containing the ability to instantiate or activate its DCDO objects on three different platforms, PCs running linux, SparcStations running Solaris, and Java Virtual Machine. Implementation components for all three platforms are maintained by the class.

However, existing instances and their clients can be affected by evolving instance characteristics. This is because of two operations that classes must support. First, Legion classes must be able to respond to interface queries from clients of their instances. Therefore, when an implementation evolves, a class must either maintain multiple interfaces, one for each instance version that it defines, or it must ensure that all instances export the same interface so that responses to interface queries will be appropriate for all instances. Second, to enable object migration and fault-tolerance policies based on checkpointing, a class must be able to cause its instances to save and restore their state to and from persistent storage, and to transition its instances between active and inert states [41]. When an object goes from inert to active, the class selects a host and specifies an object implementation that the host should use. When the implementation in the class changes, the instance can have different characteristics after it is re-activated, which means that clients that operate "correctly" before an object becomes inert and then active again, could cease to work afterwards.

The difference between the two models is in how they support change. With a static monolithic object implementation, the entire executable must be replaced. In contrast, only the components that change must be replaced in a DCDO. Thus, DCDOs allow programmers to carry out evolution at a granularity that does a better job of matching that of the change they are making. The benefits of the DCDO approach are at least threefold:

- *Efficiency*: Since a DCDO's implementation is essentially part of its state, once a DCDO has been updated with a new component, it can begin exhibiting its new functionality. Thus, the time to change an object includes the time to transmit the implementation component from the class to the instance, and the time for the component to be incorporated into the running executable. Any state that needs to be maintained from one version of an instance variable to the next typically can be transferred in main memory as the object runs.

With static monolithic implementations, the behavior of the object only changes once the implementation for the object is used. An object must be deactivated and reactivated to cause the new executable to represent the object. Therefore, changing a static monolithic implementation in our current Unix-based Legion implementation requires the following steps: (1) the object must save its state into its OPR, which typically resides on persistent storage, (2) the class must transfer the name of the new object implementation to the host, (3) the host must read the data out of the object implementation (which contains all the bytes in the new statically-linked executable, typically somewhere between 5 and 14 Megabytes), and write it into a new file in the appropriate directory in the file system, (4) the host must fork and "exec()" this executable, requiring the operating system to load it into memory and start it running. Furthermore, upon being reactivated, the object will be assigned a new object address, potentially requiring clients to re-bind to the evolving object, and further increasing the time that the object is unavailable for service.

- *Changes don't require the ability to regenerate the entire executable*: With static monolithic implementations, only programmers who have the ability to completely regenerate the binary executable for an object can change that object. Once changes are made to the source code, the programmer typically must be able to recreate the entire process of generating the executable. In an arbitrarily heterogeneous environment, this requirement can cause considerable difficulties. The programmer must have (1) her own copy of at least part of the source code for an object, (2) all of the tools that are required to transform that source code into an executable (e.g. makefiles, compilers, linkers, libraries, etc.), (3) adequate computing resources for the build, and (4) the ability to recreate the process of building the executable (i.e. libraries must be in the expected directories, arguments to compilers and linkers must be passed correctly, etc.). Thus, the process of regenerating the executable will be inconvenient and often impossible.

  In contrast, since implementation components will typically contain object code, and since an object's interface is available by querying the object or its class, programmers can build implementation components for DCDOs completely independently from the rest of the object's components. In fact, components in the same DCDO could be written in different programming languages. The important interoperability "standard" that allows components to be composed into the same process is a run-time specification of the interface to DCDO and DCC member functions, and the format to which the implementation components adhere. The most complex portion of an implementation component—the machine code—is already specified by a standard format that compilers are used to generating (e.g. *executable and linking format (ELF)* [60]).

- *Instances can evolve independently of one other*: The instances of normal classes typically share an object implementation or set of implementations that are maintained in the class because the class needs them to activate its instances. Therefore, changing an implementation must be done with the cooperation of the class. To make the change for a single instance (and not the others), the class must be designed to maintain different implementations for different instances. On the other hand, since the object implementation that a DCDO runs is a mutable part of its state, programmers can effect a temporary or "unofficial" change by communicating directly with the object; the change need not affect other instances, nor burden the class. (Even though exploiting this property can break a requirement of object-orientation—that the class defines the behavior of its instances—several of the services described in Section 4.5 benefit from being able to evolve instances of the same class independently from one another).

4.4 Evolution management policies

Sections 4.1, 4.2, and 4.3 describe how the dynamic configuration mechanism enables change. However, by itself, the usefulness of the mechanism is limited; it must be used in concert with organized evolution policies and strategies so that programmers can implement and control object evolution in ways that they can comprehend, and that fit the purpose of the evolving object. DCDO mechanism describes how change is effected; *evolution management policies* define the types of changes that are legal, and restrict when the changes can take place. This section describes several different policies, which are distinguished by the degree to which they support and define multiple versions of their instances.

4.4.1 Single-version DCCs

A *single-version DCC* defines exactly one official version of its instances at any given moment in time. All new instances are created to reflect the characteristics of the official version that the ICS in the class defines. When the contents of the ICS change, thereby creating new characteristics for future instances, the class also attempts to bring all existing instances up-to-date to reflect the change. Thus, the class attempts to ensure that all of its instances always reflect the characteristics of the ICS that it maintains.

Within the single-version style, there exist several different strategies for updating the implementations that are in use for existing instances. The problem is analagous to keeping multiple reader caches up to date with the data contained in a centralized data repository. The implementation components in the ICS represent the official copy of the data, and the components currently incorporated in active instances represent cached copies of that data in multiple reader caches. Different DCCs will offer different policies and strategies for keeping instance implementations consistent with the official copies in the DCC.

- In the *proactive update policy*, the DCC will incorporate component changes into all existing instances as soon as the changes are made to the class. This strategy allows instances to be out of date only as long as it takes for the class to propogate the changes to the instances. However, the strategy could incur unnecessary overhead (e.g. if another change is forthcoming before any calls on the instance are made), and it doesn't scale well with the number of active instances. For some small applications that change infrequently, this policy may be appropriate.
- Another option is the *explicit update policy*, in which the class and its instances export a public member function, named *update_implementation()*, that the programmer can call to cause the class to update existing instances. Calling this function on the class tells it to update all instances, or only those that are named as parameters. Calling the function on an instance causes that instance to retrieve the latest version of its components from the class.
- Yet another approach is the *lazy update policy*, in which the instances themselves determine when they get updated. The simplest variation is to enforce strict consistency semantics by having instances consult their class every time they get an invocation request to see if the request can be serviced or if the instance must first be brought up to date with the current implementation version. Other variations allow instances to check for updates less often, perhaps once every *k* member function calls, once every *t* time units, or only when they become inert and active again. Clearly, the different strategies will incur different performance and semantics tradeoffs.

4.4.2 Multi-version DCCs

An alternative to the single-version DCC is the *multi-version DCC*, which defines and allows multiple "official" versions of its instances at the same time; to do so, the DCC maintains multiple ICSs and interface definitions, one per version. Interface queries and activation operations are carried out based on the version to which the instance belongs. For multi-version DCCs, different policies also exist, depending on the status of the versions that are created, and depending on what is done with existing instances.

- In the *no-update* policy, instances are created with a particular version number, which indicates the characteristics of the ICS at the time the instance was created. The instance never evolves to a new version number, but the class continues to fully support the instance. This means that the class must keep multiple copies of interfaces and implementation component sets, one of each for every version for which there are existing instances.
- In the *explicit-update* policy, programmers explicitly cause instances to evolve to later versions by calling a member function on the class; the function takes the name of the instance and the version to which it should be evolved. Programmers can provide the class with functions that indicate how instances should be converted between versions. The class then drives the evolution process appropriately by calling these functions.

Different kinds of DCCs can create new instances to reflect the current version that the class defines, or can allow programmers to specify a version number in the call to create_instance(), thereby allowing them to create instances of old versions.

4.4.3 Multi-version instances

Another evolution management strategy, called *multi-version instances*, allows each instance to export the interface of all versions of its class. This approach enhances *behavioral consistency*, the property that clients continue to operate correctly after servers evolve. Programmers may insert *exception handling routines* into the instances in order to accept calls to functions whose interface (i.e. function name, parameter types, and return types) have changed from previous versions. On every function invocation, the client can specify a version number that it expects to use. If this value does not match the version number of the server, and an appropriate exception handling routine is associated with the object (i.e. a routine that matches the interface in the invocation), then that routine is called to handle the function. The exception handling routine then has several options: (1) it can implement the function and return the appropriate value back to the caller, (2) it can use an existing implementation for the function, and attempt to convert parameters and/or return values to match the types that are expected, (3) it can return a default value that indicates that the function is no longer supported. The different options depend on the evolution policy of the object, and on the kinds of evolution it undergoes.

The evolution management strategies I have described in this section are not intended to cover the entire range of possibilities. Instead, they are intended to represent the kinds of policies that may be useful in a distributed computing environment. I imagine that the set of useful policies includes several that I have not described, and that I have described several that may not turn out to be useful. I will continue to adjust and retool the policies as I carry out the research.

4.5 Applying DCDO technology

DCDO technology will have benefits other than for evolution management. Some services that would be impossible or extremely inconvenient with the static monolithic implementation model have natural implementations that employ and exercise the DCDO mechanism. Several of these services are described below.

4.5.1 Code-accepting server objects

DCDO technology enables server objects that can execute client code within the server's address space, thereby avoiding the cost of remote member function calls. Programmers will be able to write code that accesses a remote object through its exported member functions; but instead of executing this code within the client object that exists in a separate remote address space, thereby paying the cost of a remote procedure call on every member function invocation, the server object will accept the code into its own address space, and object invocations will not have to pay the cost of crossing machine boundaries or even address spaces more than once.

This service will be useful for objects that wish to export an interface that is more fine-grained than is typically appropriate for Legion objects. For instance, file objects can export operations for reading, writing, and seeking. Client functions can be incorporated on-the-fly into file objects to do data-dependent server-side prefetching based on the client-known data access patterns. Without the ability to load the function into the server object, these functions would require multiple remote invocations instead of just one [21].

4.5.2 Object characterization functions

DCDO technology will also benefit scheduling agents that depend on being able to retrieve information about the objects they schedule by calling functions on those objects. These *object characterization functions* might report information about the object's communication and computation behavior in order to make future placement and migration decisions. Normally, a scheduling agent that uses these functions could be used only with objects that were built to export the right set of functions for that scheduler. Other schedulers might require a completely different set of functions and information. This suggests two equally uattractive alternatives; either the entire scheduling community has to agree on a common set of information that objects return, or applications builders must identify *all* the schedulers that should "work with" their objects, and then build their objects to export the right information for these schedulers. Either way, the information that an object returns is set at the time its object implementation is created, and cannot evolve thereafter unless the object implementation is replaced.

With DCDO technology, the characterization functions could be contained in separate implementation components, which could then be sent from scheduling agents to the DCDOs they schedule. This strategy will go a long way toward eliminating the coupling of scheduling agents to objects. Applications programmers will not have to build their objects to cater to particular scheduling agents, and scheduling agents will be able to configure objects to return the appropriate information. Further, when a scheduling agent decides that it has enough information about the communication and computation characteristics of the object, it will be able to remove the functions from the object in order to eliminate, on-the-fly, the cost of the characterization.

4.5.3 Implementation inheritance

Dynamic configuration suggests and enables a unique form of run-time implementation inheritance. Programmers will be able to create a new subclass DCC by calling a member function, *derive()*, on a superclass DCC. In response, the superclass DCC will schedule and execute a seed DCC representing the subclass. The superclass DCC will then transfer an appropriate set of implementation components to the subclass DCC's ICS. This reflects the first step in a typical inheritance process—the subclass inherits the characteristics of its superclass. Once the subclass ICS is created to reflect exactly the characteristics of its superclass, the implementation components in the subclass can be augmented or changed based on the inheritance semantics that are being implemented (most likely as defined in the programming language that the programmer uses to create the class). Multiple inheritance can be implemented by having the new implementation components come from other superclasses.

4.5.4 Co-located objects

In general, objects that communicate frequently (compared to how much they compute) perform best when they are logically "closest" together, that is, when they are connected by the fastest communications link available. Depending upon where two objects are located relative to one another, communication cost consists of the time to traverse a wide-area network (when the objects are located on different sites), to traverse a local-area high-speed network (when objects are located on the same site), or to perform an operating system context switch (when objects are located in different processes on the same host).

A fourth "closest" location option, in which objects can be located within the same address space in a single process, is enabled by DCDO technology. Schedulers could choose to place two objects in the same address space by calling a *join()* function on one of the objects and passing the name of another object that it should join. Implementing join() involves incorporating the implementation components of the two objects together, configuring the process that represents them to accept the member functions of both objects, and dispatching incoming messages to different threads depending on the object for which they are intended. Using DCDO technology, arbitrarily many objects could be joined into a single address space; when these objects communicate with one another they would pay only the price of scheduling a new thread.

4.6 Important issues

I will have to address a number of different issues in order to make the implementation of dynamic configuration successful. Several of these issues are identified and addressed below.

4.6.1 Programmer interface

How a programmer uses the dynamic configuration mechanism is an important consideration, especially because one research goal is to make DCDO technology convenient to use. The important issues include the programmer's mechanism for indicating how source code corresponds to implementation components (i.e., how the programmer specifies the functions and data whose object code should be placed together in a particular implementation component), and the method by which the programmer causes the dynamic configuration member functions to be invoked on DCCs and DCDOs.

For this research, programmers will organize the source code for each individual implementation component in a separate file. The components will be created in two steps: first, an underlying compiler (e.g. g++) will create the object code for the component, and then the programmer will run a tool that "wraps" the object code to make it an implementation component. Programmers will call DCC member functions directly through other tools that expose the DCC's interface. This solution is not fully automatic—it requires some human intervention; therefore, it will work best with browsers that will retrieve and display the current interface of DCCs, implementation components, and DCDOs, so programmers can understand the process and drive it conveniently.

4.6.2 Atomic updates

Dynamic configuration requires a mechanism for ensuring that the updates do not interfere with existing threads of execution within the object. The Legion object model states explicitly that objects may accept and carry out member functions in any order they wish. DCDOs can therefore separate member functions into two categories, those that change the current running implementation, and those that do not. DCDO's will service incorporate() member functions only when no other member functions are active within the object.

4.6.3 Type checking

Legion does not prescribe any single type checking policy for Legion object member function invocations. The objects involved in the invocation—both the client and the server—determine the level and style of type checking that is performed. Policies can differ depending on where (client, server, neither, or both) and when (statically or dynamically) the checking is done. The fact that DCDOs can change their composition and public interface makes dynamic type checking particularly appropriate and useful.

For server-side checking, both implementation components and DCDO objects will contain two sets of signatures, one for functions and data elements that they contain, and one for functions and data elements they call. When the incorporate() function receives an implementation component, it can first consult these tables to ensure type safety. That is, it can make sure that the signatures of the calls match the signatures of the functions that they will access. Failing this check could cause the incorporate() operation to be aborted, a subset of functions in the object to be disabled, or merely a warning message returned back to the user.

4.6.4 Heterogeneity

Since implementation components contain machine code, DCCs and DCDOs will have to recognize and manage multiple sets of implementation components for different architectures. This fact makes programming less convenient, but it is certainly not a fundamental problem. DCCs must ensure that it always incorporates an implementation component into a DCDO seed instance whose machine code types match. To do so, the class can simply remember the current architecture type of each active instance.

4.6.5 Updating the state contained in data components

Section 4.1 states that an implementation component can represent a public or private function, or an instance variable and the operations allowed on that variable. Since instance variables contain state information, DCDOs must allow this state to be initialized or to be transferred from the old version of the instance variable into the new one. To do this, instance variable implementation components are accompanied by *component constructor* functions, whose purpose will be similar to that of C++ constructors. A component constructor will be executed when the new implementation component it represents is incorporated into the object; different constructors will be able to fill the instance variable with its default initial values, or copy the values from the old version of the instance variable into the new one.

5. Enabling technology and preliminary results

To implement dynamic configuration efficiently, a DCDO must meet two fundamental requirements: (1) it must be able to incorporate implementation components by dynamically augmenting and shrinking its own code segment, and by loading and calling new executable code at run-time, and (2) it must be able to alter dynamically the public interface that it exports.

5.1 Run-time code loading

Many systems and languages contain mechanisms that allow programs to load and run new executable code into a process that is already running. Examples include Agent-TCL [27], Aglets [39], Ara [51, 52], Avalon/Common Lisp [20], Eden [4, 10], Emerald [12, 36, 57], Extended Facile [38], Java [25], Kali Scheme [17], LII [11], Mole [58], NCL [21], Obliq [16], Omniware [2], REV [56], Safe-TCL [14], Sumatra [1], TACOMA [35], Telescript [64], Voyager [49], and more. Modern operating systems, including Microsoft Windows via COM/OLE [15, 21, 26], and Unix via shared objects and the run-time linker [60], also enable this functionality. The fact that so many systems contain this kind of functionality indicates that the DCDO model will be effective and implementable for many different languages and underlying platforms. Since the current UVa Legion implementation is built in C++ for Unix systems, I will use shared objects and the run-time linker for my implementation of the DCDO model.

In Unix, the *dlopen()* system call dynamically loads a shared object file into the process that calls the function, and returns an opaque handle that can be used to access the code in that shared object. Internal symbols within the shared object file can then be located using the *dlsym()* function, which takes the opaque handle for the shared object along with the name of the symbol to find, and returns the address of that symbol in that shared object. This address can then be cast to the appropriate type (e.g. a function pointer), and can be used to access the code at that address (e.g. by calling the function named by the symbol).

5.2 Altering an object's interface

An object in a distributed system typically accepts some set of public member functions that is fixed at the time the object is created. Changing the interface that an object exports requires access to the code that dispatches incoming messages to the object's user-defined functions. The event-based UVa Legion run-time library [23, 63], which DCDOs will link, allows this level of access. An object can make function calls at run-time to "enable" a member function that was not enabled when the object first ran. The programmer can specify the address of the code that implements the function, and the method dispatcher will then call that code when invocations for it arrive at the object.

5.3 Prototype implementation

Release 0.0 of the UVa Legion implementation contained a prototype implementation of dynamic configuration. The prototype was designed and implemented to ensure and demonstrate that the Unix run-time linker provides sufficient power to implement the DCDO model. The prototype contained a single example application, a sorter DCDO that contained an integer array and exported a public sort() function that allowed its clients to cause the DCDO to sort its data. The initial sort() function implemented a bubble sort, and the user could "evolve" the object by calling incorporate() to add a reverse_sort() function, the implementation of which was contained in its own implementation component. The user could also replace the bubble sort implementation with a quick sort implementation, and could replace the linked list implementation of the integer array with a one dimensional C++ array. The application demonstrated that the Legion run-time library and the Unix run-time linker could be used together to augment the public interface of a Legion object, to replace an existing function with an updated implementation, and to replace an instance variable implementation component.

6. Research agenda

The goal of my research is to demonstrate that the DCDO model is an effective tool for enabling object evolution in wide-area distributed object computing systems. To achieve this goal, I will complete the following research tasks:

1. *Specify core object interfaces*: I will provide a detailed specification of the DCDO and DCC object interfaces, including the signatures and semantics of the member functions that they export, and the contents and meaning of the member functions' parameters. I will also describe and specify the primary data structures that the objects should contain, including the dynamic implementation component mapper and the implementation component store.

2. *Implement the mechanism*: I will implement the DCDO model within the UVa Legion implementation. The implementation will include seed DCDOs and DCCs that export the full functionality described in Sections 4.1 and 4.2, and will be fully integrated with the Legion implementation and persistence models as implemented in UVa Legion; that is, DCDOs and DCCs will be able to save and restore their state, and Legion host objects will be able to execute DCDOs and to cache implementation components. The implementation will work on multiple Unix platforms (at least two), and will be based on shared objects and the run-time linker, as described in Section 5.

3. *Implement evolution management policies*: I will implement DCCs that employ the five different evolution management policies described in Section 4.4. The policies will include single version DCCs that implement proactive, explicit, and lazy update policies, and multiple version DCCs that implement no-update and explicit-update policies. Other strategies, including multi-version instances may be implemented as well.

4. *Apply dynamic configuration to implement other services*: I will apply dynamic configuration in ways that exercise and benefit from the DCDO mechanism. I will build code-accepting servers, object characterization functions, and run-time support for implementation inheritance, as described in Section 4.5. If the UVa Legion system contains mechanism for co-locating multiple objects within the same address space (by specifying at *compile time* the objects that get compiled into a monolithic implementation), and contains a thread-safe implementation of the library, then I will also provide support for co-locating objects at run-time using the join() function. Otherwise, I will just describe in detail the requirements for an implementation.
5. *Measure and characterize the performance and cost of the implementation*: I will design, run, and report on experiments that will characterize the performance of the DCDO implementation. The experiments will measure at least the following: the time needed to complete a member function in a DCDO as opposed to an object represented by a static monolithic object implementation, the time it takes to create a new instance of a DCC (including the time to configure it with its implementation components), and the time needed to update instances with new implementation components. The experiments will be designed to determine where the cost associated with my implementation of dynamic configuration is located.

   The research agenda is depicted in Figure 3.

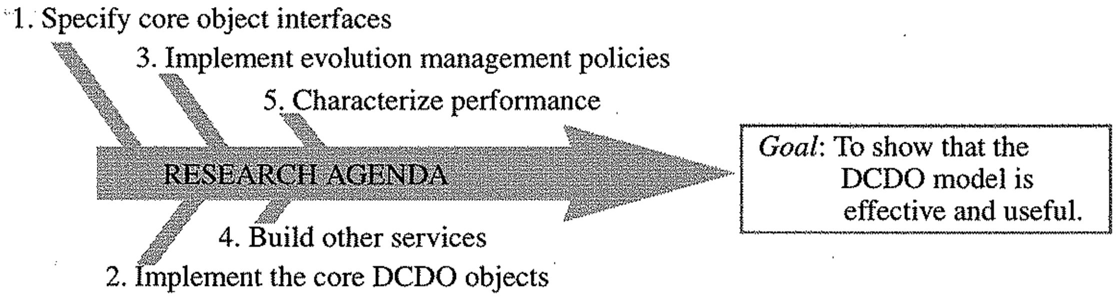
1. Specify core object interfaces
3. Implement evolution management policies
5. Characterize performance
RESEARCH AGENDA
4. Build other services
2. Implement the core DCDO objects
Goal: To show that the DCDO model is effective and useful.

FIGURE 3. *Research agenda*: I will show that the DCDO model is an effective tool for wide-area distributed object computing systems.

Successful completion of items 1 through 4 above will demonstrate that the DCDO model is effective and useful for enabling objects to evolve. Item 5 will illustrate where the technology can be improved, and the types of services and uses for which it will (and won't) be useful.

7. Summary

In this document, I have introduced the dynamically configurable distributed object model; I have described the roles of classes and objects within the model and I have outlined its benefits. The model is designed as an alternative to static monolithic implementations, which can change only in their entirety. I have described several useful services that DCDOs enable and facilitate, including evolution management policies. Building these applications will demonstrate the model to be useful and effective.

8. Related work

Current distributed computing systems with characteristics similar to Legion (e.g. active objects or processes, support for parallel processing, networks of workstations as the underlying compute engine, etc.) do not support mechanisms that have the same goals and characteristics of dynamically configurable distributed objects. This is because existing systems almost exclusively use static monolithic binary executables to create processes. For example, CORBA (Common Object Request Broker Architecture) [48] and DCE (Distributed Computing Environment) [42] both use *interface definition language (IDL)* compilers to generate high level language stubs for an object's functions; the stubs are then filled in by programmers and compiled together into a single binary executable that implements the object.[4] Like CORBA and DCE, PVM (Parallel Virtual Machine) [59] and MPI (Message Passing Interface) [43] both create tasks by running binary executables that exist in system or user "bin" directories. Likewise, Mentat [28] objects are represented by executable binaries created by the Mentat compiler. In all five systems, and nearly all others like them, changing the behavior of a program, task, or object, requires replacing the corresponding binary executable in its entirety.

However, the DCDO model and its proposed implementation address problems similar to those faced in other areas or research, and DCDO technology contains several implementation aspects that exist in other systems. Therefore DCDOs are related to several other projects and technologies. In this section, I identify the most important of these, and I describe how my work with DCDOs differs.

Section 8.1 describes the mechanisms and systems that most closely resemble the fundamental underlying evolution mechanism of DCDOs. Section 8.2 then describes schema evolution techniques of *object-oriented database management systems (OODBMSs)*. This work is relevant because Legion shares several fundamental characteristics with OODBMSs—they both contain persistent, shared, distributed objects and classes. Therefore, Legion faces similar problems in terms of evolution. However, Legion is also very different from OODBMSs in several important ways—Legion objects are represented by active processes, and the code for object methods is contained and executed within the objects, not in applications programs. Therefore, policies, goals, and approaches can be borrowed from OODBMS schema evolution techniques, but mechanisms for implementation must be appropriate for the Legion environment.

8.1 Dynamic configuration of programs

One of the most important aspects of DCDO mechanism is its ability to alter the contents of a running program's code segment on the fly. Other systems and approaches exist for implementing similar mechanism, several of which are described in this section.

8.1.1 Traditional dynamic linking

Traditional dynamic linking technology allows programmers to load and execute *shared libraries* (sometimes known as *shared objects*) of executable code into their program at run-time. The operating system's *link editor*, which is typically called by a high-level language compiler, creates an executable program that contains calls to the *run-time linker* to load automatically the code for the libraries when the functions that they contain are accessed by the program. When a library changes and is replaced by a new version with the same name, programs executed after the change will load and access the functions in the new library; these programs do not have to be recompiled or relinked. This is a considerable advantage for applications linked against libraries whose interface remains constant across a change, but whose imple-

4. CORBA does not require that object implementations exist as binary executable programs; the descriptions of the "execution model" and "construction model" (pp. 1-7 and 1-8 of the CORBA specification document [48]) are worded vaguely enough to allow other implementation styles. However, all concrete CORBA implementations of which I am aware use binary executables for object implementations.

mentation is improved. However, when a library's interface changes, (e.g. when a function is removed or renamed, or when the types of its parameters or return value are changed), then programs that were built and linked against the old version of the library can break. This problem is typically not detected until the program is run.

To guarantee that programs built to expect a certain library interface will continue to behave correctly after a change to a library is made, operating systems typically allow libraries to be versioned. For example, in Unix, when a programmer creates a new version of a library that could break existing applications, the new version can be given a name with a suffix that identifies the version number; the old version can then be left in place for old applications to continue to use. So, for example, `libMyLibrary.so.1` could be supplemented with `libMyLibrary.so.2.`, which could define a new interface. Applications linked against `libMyLibrary.so.1` can continue to load and access all of the functions of the interface in that version of the library, even after the new version has been created. New applications can be linked against the new interface contained in `libMyLibrary.so.2`.

The paragraphs above describe "traditional" dynamic linking—dynamic linking that is enabled through the operating system's link editor, which embeds calls to the run-time linker within the dynamically linked executable that it generates.

Traditional dynamic linking might seem to be sufficient for enabling object evolution in wide-area distributed object computing systems like Legion. Suppose that all of an object's member functions were contained in separate shared library files, and that the appropriate functions were dyamically linked against a core program to produce an object implementation in the form of a dynamically linked executable that has the appropriate characteristics for the object it intends to represent. Member functions could then be evolved by creating new versions of the shared libraries in which the functions reside, and by allowing the operating system's built-in dynamic linking mechanism to reflect the changes in instances. The higher-level evolution management policies described in Section 4.4 could then presumably be implemented by replacing and versioning shared library files appropriately.

However, traditional dynamic linking by itself is *not* sufficient to enable object evolution in Legion because the technique described in the paragraph above (and others like it) makes assumptions about the underlying system that clearly cannot be enforced in a system with Legion's intended characteristics. For traditional dynamic linking to work, the link-time and run-time environments must match one another. The link-time environment contains the names of shared libraries and the directories in which they reside (these names are typically provided as flags to the link editor); for the dynamically linked executable to run correctly, the shared libraries must exist at run-time with the same name, in an appropriate directory, and with the same contents, as the corresponding shared libraries in the link-time environment.

In a heterogeneous system that is intended to comprise many autonomous organizations, the requirement described above causes several problems. A single file system visible from all Legion objects will not exist; therefore, some mechanism for placing replicates of libraries on all the file systems over which the object might run would have to exist. This replication mechanism could not be static (i.e. it could not create at link-time a copy of each library on all file systems in Legion) because such a mechanism would not scale with the large number of file systems that Legion will contain. The mechanism also could not be truly "dynamic" (i.e. it could not download from Legion the shared libraries for the methods on demand). To do so would require that the program trap accesses to member functions implemented in the shared libraries. The trap would then have to map the function call to the appropriate shared library file in Legion space, and download that file onto the local file system into an appropriate directory, with an appropriate name, to match the link-time environment of the dynamically linked executable. However, traditional dynamic linking does not support the ability to trap calls to functions implemented in libraries in this way, and objects cannot be sure in general that they can recreate the link-time environment of the dynamically linked executable.

These limitations, along with the fact that tranditionally dynamically linked executables must be re-executed in order to exhibit the functionality contained in new versions of the shared libraries, limit the effectiveness of traditional dynamic linking for enabling object evolution. User-level access to the run-time linker (or "non-traditional" dynamic linking), as described in Section 5.1, allows the programmer to solve the problems identified in this section, and is therefore well suited to support the DCDO model. In particular, new libraries can be incorporated into running programs without them having to be re-executed.

8.1.2 Component Object Model (COM)

Microsoft's *Component Object Model (COM)* [15, 26, 44] is an innovation that tackles the problem of software evolution. COM is a programming model and binary interoperability standard that allows different components of a running application to evolve separately from one another without causing undefined function errors.[5] Each component is contained in a separate *dynamically linked library (DLL)* file that the application loads on demand, and contains an interface which is implemented as a table of pointers (called a *vtable*) to the functions that the component contains. A COM interface is similar to—and the component it represents is binary-compatible with—a C++ abstract class. All COM components contain a *QueryInterface* function that returns a pointer to the interface of a named component. In order to call a function in a component, the caller must obtain a pointer to the component's interface from QueryInterface, and must call the function through that pointer. Thus, the COM programming model forces the calling code to be ready to deal with the case that the interface and component are not contained in the running application. This distinguishes COM applications from others because most applications assume that a function is implemented, present, and callable if the application that contains the call to the function compiles and links successfully.

Requiring indirect function calls allows components of an application to evolve separately from one another without creating undefined references within the running application. A new component can be added to an application while it is up and running because the application will look for and load the component's DLL file only when it is referenced. To install a new "version" of an interface, a programmer creates a completely new component with a completely new *Globally Unique Identifier (GUID)*. New versions of existing interfaces are reflected instantly within running applications without them having to be relinked or rerun. The behavior and functionality of the application may change due to an evolved component, but it will not fail unpredictably because the COM programming model enourages the existence of client-side code that is designed to handle evolution—and missing functions and interfaces—gracefully.

The strongest similarity between a COM application and a DCDO is the fact that they both use dynamic virtual tables to map function calls to the function implementations that service those calls. Evolving the application or object involves altering entries in this table. Furthermore, both technologies are intended to be built on top of dynamic linking facilities of host operating systems.

However, the focus of COM technology is on evolving sequential applications independently of one another. Applications of the same type have different maintainers wherever they are installed. One programmer can upgrade a running COM application to reflect the existance of a new component simply by

5. In COM, the terms "object" and "component" are sometimes used interchangeably—each can refer to part of a running software entity. In order to avoid confusion, I use the term "component" to refer to the sub-parts of a running software entity, and reserve the term "object" for use in the Legion sense. I use the term "application" in place of "process" because COM supports "local/remote transparency," which means that the concepts discussed here are not restricted to applications that are implemented within a single address space or even on the same machine.

replacing the appropriate DLL file on her local file system, but this upgrade has no effect on other applications installed at different sites. No mechanism for managing the evolution of all applications of the same type exists (nor should it, given the characteristics of the COM environment). In sharp contrast, DCDO technology is built to work within a wide-area object-based distributed system, where an object's implementation characteristics are determined by its Legion class object, not the configuration of the local environment in which the object runs. Therefore, the mechanism for evolving objects—as implemented in DCCs—can take advantage of the fact that objects are associated with one another in this way, and can drive the evolution of all the objects of a particular class appropriately. This is related to another important difference between the two technologies—the way that programmers effect evolution. As described above, COM programmers replace local DLL files to effect change. DCDO programmers, on the other hand, employ the official first-class Legion communications mechanism—they invoke member functions on the objects involved in the change. This difference elevates implementations and implementation components to first-class status in the system.

Another important difference between the two technologies is the abiility of DCDOs to alter the public interface of an object. Again, this difference is a reflection of the different environments for which the technologies are intended. COM applications typically contain a main application and a set of components that implement support services. The presence of certain implementation components within a DCDO typically affects the public interface of the object, and determines how entities outside the DCDO interact with that object. COM is concerned only with client and server components within the same application.

The underyling COM and DCDO technologies for implementing evolution within a single object are similar, but in the case of DCDOs, this mechanism is only part of a higher level service that must be provided for effective object evolution in distributed computing systems. Therefore, COM should be thought of as a very effective enabling technology for implementing dynamic configuration as described in Section 4, but not sufficient on its own to solve the problems that DCDO technology is designed to solve.

8.1.3 Mobile agents and transmissible code

A *mobile agent* is a "code containing object that may be transmited between communicating participants in a distributed system" [38]. Systems that support mobile agents must contain a mechanism for representing code so that it can be transferred from one program to another, over a network, and executed by or within the receiving program. Systems that provide this ability can be characterized along several dimensions, including the format or style of transmitted code, and the initiator of the transmission.

Knabe [38] identifies four different kinds of transmissible code representations—source code, interpreted intermediate representation, compiled intermediate representation, and machine code. REV [56], Avalon/Common Lisp [20], TACOMA [35], and Safe-TCL [14] transmit source code, which is executed via interpretation by the receiving node. Obliq [16], Telescript [64], and Java [25] transmit interpreted intermediate representations, and Omniware utilizes a compiled intermediate representation. Finally, Emerald sends machine code to match the architecture of the receiving node. Extended Facile [38] implements a hybrid approach that allows the transmission of both an architecture independent representation and a machine code representation for the same agent.

Acharya et. al. [1] characterize systems according to the entity that initiates the transmission of code. Omniware [2], Safe-TCL, and Java allow programs to download code from a remote site and execute it locally. Avalon/Common Lisp, REV, NCL [21], and Obliq allow programs to send code to a remote site, and receive the results of the subsequent computation. Finally, Agent-TCL [27], Emerald [57], Mole [58], Aglets [39], TACOMA, and Telescript allow programs to move themselves from one node to another.

The DCDO *model* does not technically specify the format of implementation components—different implementations could have different styles of representation. However, as described earlier, my initial implementation will represent the components as machine code to be dynamically linked into the recieving DCDO. The initiator of the movement of implementation components from DCCs to DCDOs depends on the evolution management strategy. Different strategies allow the DCC, the DCDO instances, or a separate remote object to initiate the tranmission of an implementation component and its incorporation into a DCDO.

The primary difference between DCDO technology and typical mobile agent systems is in the *intent* of the code transmission. The systems that transmit functions to be evaluated on a remote node and then wait for the results to come back, do not typically execute the function on the remote site more than once. That is, the server node receives the function, executes it to completion, and returns the results. Systems whose programs download code into their address space are driven by parts of the program that can take advantage of that code. Thus, the benefits of retrieving the code for local execution is realized by the retrieving program itself. In contrast to these approaches, the purpose of transmitting DCDO implementation components and incorporating them into different objects in the system is to evolve the functionality of the objects so that other entities in the system can take advantage of this new functionality. Thus, tranmitting code improves the services that receiving objects can provide, and this improved service persists beyond a single execution; new member functions can be added to a DCDO, and "third-party" objects can immediately take advantage of these functions.

8.1.4 Kali-Scheme and Erlang

Kali-Scheme is a distributed system that supports remote execution of Scheme code. Kali-Scheme allows programmers to create new address spaces in the system via the `make-aspace` function, and to execute functions on other address spaces by calling `remote-run!`. Kali Scheme provides the ability to transmit continuations between address spaces, and generates a new thread on the appropriate node in response to `remote-run!` invocations. Threads on different nodes communicate using explicit message passing. The system also provides the ability to transmit "templates" (not to be confused with C++ templates), which contain procedure code segments. This allows applications to be incrementally modified.

Kali-Scheme shares some of the same goals and benefits of DCDOs. However, the implementation of Kali Scheme does not scale well since every address space created in the system must have an open connection to every other. Furthermore, Kali-Scheme is a single-language solution for a relatively small system. DCDOs can use multiple programming languages and is appropriate for wide-area distributed computing.

Erlang is a concurrent language that allows programmers to alter the behavior of the system on the fly. Erlang supports *modules*, which contain functions that can be explicitly exported, and data. Object implementations are first-class in the sense that Erang supports functions that operate on the object code for modules. When a module is compiled using the Elang command `c(ModuleName)`, the object code for that module is automatically registered with the system; by default, this registration is implemented by copying the object code into a well known directory, naming the file based on the module name, and loading it into the process. Object code can also be loaded by calling the `load_module(ModuleName)` function. Erlang can contain exactly two versions of the object code for a single module, the *old* version and the *current* version. Another version cannot be created unless the old version is purged, which can be done explicitly by calling the `purge_module(ModuleName)` function, or automatically by the code management system. When a version of the object code is purged, all processes that are currently using that version are killed.

Distributed Erlang systems that comprise multiple *nodes* can be created by running a `net_kernel` process on each of the nodes. Erlang's primitives for concurrency are similar to PVM's—Erlang supports `spawn`, `send`, and `receive` functions. Programmers can spawn processes on other nodes in the system by including a machine name in the call to spawn. In systems that share both a file system and a set of directories that contain object code for modules, modules can be automatically loaded by remote machines when the modules are accessed. This is because the object code for a module will be found in the same directory on the remote node as it will be on the calling node. When the object code for a module does not exist on the remote node, the programmer must explicitly transfer and register the object code at the remote node in order to execute the module there.

Erland differs from DCDOs in the following ways: (1) it is a single language solution, (2) it allows at most two versions of a module's object code at the same time, (3) it assumes either a single file system or programmer support for transferring object code between nodes, and (4) it is not an object-based system in which user-owned classes define the characteristics of their instances; instead, "the system" maintains the object code that defines the behavior of a module. Erlang is similar to DCDOs in that it allows the object code for a module to change while the system is up and running, and the processes that are using that module do not need to be halted and restarted in order to reflect this change.

8.2 Object-oriented database systems

Object-oriented database management systems (OODBMS) store their data in objects, each of which is an instance of a class. As in all object-oriented systems, a class defines both the type of the data that is stored in its instances, and the methods that are allowed to operate on the data. The set of classes in an OODBMS, and their relationships to one another (i.e. the *inheritance hierarchy*), are called the database's *class lattice* or its *schema*. To adjust a schema to allow the database to react to changing roles over time, the schema must be able to evolve; *schema evolution* is the process of changing the class lattice that the database defines. Examples of the kinds of evolution that can take place are adding, deleting, or changing an instance variable or method, moving a class within the class lattice (i.e. changing its superclasses or subclasses), and adding or removing a class.[6] Schema evolution approaches can be grouped into one of two different categories, *class modification* and *class versioning*. Systems that support class modification define a single version of each class in the schema; class versioning systems support classes with multiple versions.

When a class in a schema evolves, three different types of problems can occur: (1) instances can become out-of-date with the definition in the class, (2) subclasses and superclasses of the changed class can become out-of-date, and (3) existing applications programs that were created with the previous schema can become obsolete. Database systems employ various *instance adaptation* strategies to update existing instances to reflect the characteristics defined in the changing class. Instance adaptation strategies can provide *structural consistency*—the property that schema changes are reflected in persistent instances in the database. Other strategies address the problem of providing *behavioral consistency*—ensuring that existing application programs continue to perform properly after schema updates.

Obviously, evolving Legion objects presents problems that have analogues in OODBMSs. Therefore, it is natural that the solutions presented in Section 4.4 have analogues as well. The rest of this section characterizes existing schema evolution techniques in order to demonstrate from whence the evolution management strategies of Section 4.4 have been adapted. I focus on strategies that deal with the effects that changing a class has on existing instances and applications programs (problems (1) and (3) above). I ignore the affects on other classes in the system (problem (2) above) because Legion does not prescribe an inheritance model; several different inheritance semantics and implementations can exists simultaneously. I

6. The schema evolution literature contains several examples of more complete and detailed taxonomies for different OODBMSs [6, 53, 40].

could describe strategies for a particular inheritance scheme, such as the one described in Section 4.5.3, but we have decided that this is beyond the scope of this research.

8.2.1 Class modification

Class modification is the process of changing a single version of a database schema, and updating the contents of the database to reflect that schema. Examples of OODBMSs that support class modification include Orion, GemStone, OTGen, and O_2.

Orion [6] implements schema modification by defining (1) a *taxonomy* of allowable schema evolution operations (e.g. adding a class, changing the data or operations defined by a class, etc.), (2) a set of *invariants* that define the consistency requirements of the class lattice (e.g. the lattice is a directed acyclic graph, subclasses inherit all characteristics from their superclasses, etc.), and (3) a set of *rules* that determine how the invariants are maintained when the evolution operations are performed. The schema designer uses the evolution operations to specify how the schema should change, and then the system verifies the legality of the change using the invariants and rules. Orion updates instances *lazily*; that is, an instance is updated only when the particular schema change requires it, and only when the instance is fetched by an application program. For example, when a schema change removes an attribute from a class definition, the instances are not immediately updated; however, when an instance is fetched, the system *screens* the old value from the view of the application program that fetches it.

GemStone [53] implements schema evolution very similarly to Orion; both systems define a set of evolution operations and schema invariants, and maintain the invariants when the operations are applied. GemStone differs from Orion mainly in its implementation of instance adaptation. Rather than screening instances, GemStone converts them immediately after their class is changed. Thus, the overhead of modifying a class can be considerably greater at the time the modification is made, but then the system does not need to continue to screen instances for out-of-date contents every time they are used.

OTGen [40] also resembles Orion's approach to schema evolution. OTGen extends the class modification functionality with a mechanism for implementing database reorganization. Basically, the schema designer manipulates a transformation table that defines how one version of the database should be transformed into the next. This allows the designer to override the default semantics of an evolution operation, and to accompany schema evolution operations with database changes that are more sophisticated than are typically available (e.g. new database objects can be created, instance variables can be moved from one class to another, etc.).

O_2 [22] allows programmers to specify *conversion functions* that indicate how instances get converted after a class is modified. *Default conversion functions* are provided automatically by the system, and *user-defined conversion functions* can override the behavior that would otherwise be defined in the default functions. Default functions apply a set of rules for converting between the basic attribute types supported by the system; these rules are a combination of C casting rules and conversion routines (e.g. *atoi()* and *sprintf()*). O_2 implements both *immediate* and *deferred* conversion strategies, in which instances are updated immediately after a schema change is made, and only when they are actually used, respectively. The selection of the appropriate strategy is left up to the user.

Section 4.4.1 describes three different instance adaptation strategies for classes that allow modification but not versioning. The semantics of the proactive update policy correspond directly to the semantics of the approach taken by GemStone and the immediate conversion strategy of O_2. The semantics of the DCC lazy update variation in which each instance checks with the class for the latest version on every member function invocation, corresponds to the semantics of the Orion screening mechanism, and to the O_2 deferred conversion strategy.

8.2.2 Class versioning

The earliest and most important effort to support class versioning was done in the Encore [55] OODBMS. Encore allows programmers to create multiple versions of classes. The set of all versions of a class is called that class's *version set*, and the most recently created version is denoted the *current-version*. New versions of a class can be created by altering the set of attributes (methods and instance variables) defined by the class. A *version set interface* exists for every versioned class in the system; a version set interface is a virtual class that provides an "inclusive summary" of all the attributes in all versions of the class. Attributes are never removed from the version set interface, which means that existing programs cannot become inconsistent with types they were built to expect; thus, behavioral consistency is achieved.

All applications programs are bound to a single version of each class they use. When a new version of an object restricts the range of data for an attribute, an existing application program could potentially retrieve a value that it could not handle because that value falls outside the range of values that the program was built to accept. An analagous problem exists when an attribute is removed, or when its range is restricted in a new version of the class; in this case, an existing application program could attempt to write a value that falls outside the new range. To solve these problems, users can provide *exception handling routines* to resolve version conflicts between a program and an object that it references. *Read handlers* can be used to ensure that values returned from an object are in the range that the program expects (i.e. the attribute's range as defined in the version to which the application program is bound), and *write handlers* can ensure that values written into an object fall within the range defined in the current-version.

The primary limitation of the Encore project is that evolution operations that require additional storage within existing instances (e.g. operations that add a new attribute to a class definition) can do no better than to include a handler that returns a default value for the new attribute in existing instances. This is because Encore does not provide a mechanism for adding space to the storage structure of existing objects. Likewise, if an evolution operation removes an attribute from a class, new instances of that class will not contain storage space for that attribute, and application programs bound to a previous version of the class cannot retrieve from a new instance anything but a default value defined by a handler.

CLOSQL [45, 46] is an OODBMS that gives programmers more control over how instances are converted from one version to the next, and more flexibility in converting instances to different versions of their type. CLOSQL implements class versioning, and allows programmers to associate *update* and *backdate* functions that specify how instances are converted between consecutive versions of their class. By default, a program will access instances as defined in the latest version of a class. However, programs can be written to retrieve and access instances according to a specific version of the instances' class. When this happens, the system converts the instance to the version that is requested by invoking the update and backdate operations until the appropriate version is achieved. Attribute values that are removed from one version to the next are maintained by the system in case the instance needs to revert back to a previous version in which they were defined.

Clamen [19] developed a scheme in which creating a new version of a class causes the database to create a new version of every instance of that class. The instance versions are called *facets*. An implementation of the scheme would not necesarily have to copy all the attributes of all the facets, they could be shared between different facets of the same instance.

The no-update policy of multi-version DCCs most closely resembles the semantics of Encore, which also binds a client to a single version of the class. However, since a multi-version DCC does not restrict the kinds of evolution operations it allows (more specifically, multi-version DCCs allow the removal of instance variables and methods), the analogue of a "version set" that is compatible with clients that are bound to any version of the class cannot be maintained in general; thus, full behavioral consistency is not achieved. The explicit-update policy is similar to the use of CLOSQL update functions; I do not describe a policy that converts instances to previous versions of their class. Finally, the multi-version instances strategy that contains exception handling routines corresponds to a combination of Clamen's facets and Encore's exception handling routines.

8.2.3 OODBMSs vs. Legion

At the end of Sections 8.2.1 and 8.2.2, I described how the *semantics* of various database systems correspond to the semantics of the different policies described in Section 4.4. However, since the data model and implementation characteristics of OODBMSs are fundamentally different from those of Legion, Legion requires very different implementation mechanisms to achieve similar semantics. In particular, Legion objects are represented by active entities, and the code that represents member functions is embedded within the objects themselves, rather than in other objects or programs. Therefore, evolving an object's member functions requires updating these functions within all existing instances of the object. Some OODBMSs must alter the structure of objects to add or remove instance variables, but method implementations are part of the schema, not physically part of the objects themselves. Therefore, evolving the implementation of operations does not involve coordination among all the instances of the class. So while the semantics of the policies described in Section 4.4 have analogues in OODBMSs, the DCDO evolution mechanism must work with active processes, and must alter code segments on the fly, unlike OODBMS schema evolution implementation techniques.

8.3 Summary

The underlying mechanisms that DCDOs will employ to implement evolution exist in other systems—modern operating systems provide user-level access to the run-time linker, and systems and languages like COM, Erlang, Java, and Kali-Scheme allow programs to evolve their functionality as they run. Furthermore, the object-oriented database community has developed a myriad of different schema evolution strategies that allow programmers to modify or evolve classes and their instances; my work will borrow many of these ideas, and will adapt them to the Legion environment. Thus, no individual aspect or component of the DCDO model and its implementation is completely new. However, the DCDO mechanism addresses a new problem in a new environment, and combines these existing approaches and techniques in a way that has yet to be explored. The DCDO model separates implementations into composable components, treats each component as a first-class entity in the system, and provides a strategy for maintaining, managing, and utilizing them to enable object evolution in a shared, persistent, independent, active, distributed environment.

References

[1] Acharya, A., Ranganathan, M., Saltz, J., "Sumatra: a language for resource-aware mobile programs," University of Maryland, ???.

[2] Adl-Tabatabai, A., Langdale, G., Lucco, S., Wahbe, R., "Efficient and language-independent mobile programs," *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation*, pp. 127-36, May 1996.

[3] Agrawal, R., Buroff, S., Gehani, N., Shasha, D., "Object versioning in Ode," *Proceedings of the 7th IEEE International Conference on Data Engineering*, pp. 446-455, April 1991.

[4] Almes, G.T., Black, A.P., Lazowska, E.D., Noe, J.D., "The Eden system: a technical review," *IEEE Transactions on Software Engineering*," pp. 43-59, January 1995. TK6540.I48SE

[5] Bal, H.E., Steiner, J.G., Tanenbaum, A.S., "Programming languages for distributed computing systems," *ACM Computing Surveys*, vol. 21, no. 3, September 1989.

[6] Banerjee, J., Kim, W., Kim, H.-J.-, Korth, H.F., "Semantics and implementation of schema evolution in object-oriented databases," *ACM SIGMOD '87*, vol. 16, no. 3, pp. 311-22, December 1987.

[7] Bergstein, P.L., Hursch, W.L., "Maintaining behavioral consistency during schema evolution," *First JSSST International Symposium on Object Technologies for Advanced Software*, pp. 543, 176-93, Kanazawa, Japan, November 4-6, 1993.

[8] Bergstein, P., "Object-preserving class transformations," *OOPSLA '91: Object-Oriented Programming Systems, Languages, and Applications, Special issue of SIGPLAN Notices*, vol. 26, no. 11, pp. 299-313, Phoenix, Arizona, October 1991.

[9] Betz, M., "Interoperable objects," *Dr. Dobb's Journal*, pp. 18-39, October 1994.

[10] Black, A.P., "Supporting distributed applications: experience with Eden," *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pp. 181-93, December 1985. QA76.6.S9197a

[11] Black, A.P., Artsy, Y., "Implementing location independent invocation," ???.

[12] Black, A.P., Hutchinson, N., Jul, E., Levy, H., "Object structure in the Emerald system," *OOPSLA '86: Object-Oriented Programming Systems Languages, and Architectures*, pp. 78-86, September 1986.

[13] Bohner, S.A., Arnold, R.S., Software Change Impact Analysis, IEEE Computer Society Press, Los Alamitos, CA, 1996.

[14] Borenstein, N., "Email with a mind of its own: The Safe-TCL language for enabled mail," *Proceedings of IFIP Working Group 6.5 International Conference*, pp. 389-402, June 1994.

[15] Brockschmidt, K., "What OLE is really about," Microsoft Corporation, July 1996.

[16] Cardelli, L. "A language with distributed scope," ???, Digital Equipment Corporation, May 1995.

[17] Cejtin, H., Jagannathan, S., Kelsey, R., "Higher-order distributed objects," *ACM Transactions on Programming Languages and Systems*, vol. 17, no. 5, pp. 704-739, September 1995.

[18] Chin, R.S., Chanson, S.T., "Distributed object-based programming systems," *ACM Computing Surveys*, vol. 23, no. 1, pp. 91-124, March 1991.

[19] Clamen, S.M., "Schema evolution and integration," *Distributed and Parallel Databases*, vol. 2, no. 1, pp. 101-26.

[20] Clamen, S. Leibengood, L., Nettles, S., Wing, J., "Reliable distributed computing with Avalon/Common Lisp," *Proceedings of the International Conference on Computer Languages*, pp. 169-79, 1990.

[21] Falcone, J., "A programmable interface language for heterogeneous systems," *ACM Transactions on Computer Systems*, vol. 5, no. 4, pp. 330-51, November 1987. QA76.9.I58A35

[22] Ferrandina, F., Ferran, G., Mdec, J., Meyer, T., Zicari, R., "Database evolution in the O_2 database system," *Proceedings of the 21st International Conference on Very Large Databases*, pp. 170-81, Zurich, Switzerland, September 1995.

[23] Ferrari, A.J., Lewis, M.J., Viles, C.L., Nguyen-Tuong, A., Grimshaw, A.S., "Implementation of the Legion library," University of Virginia Computer Science Technical Report CS-96-16, November 1996.

[24] Foster, I., Kesselman, C., "Globus: A metacomputing infrastructure toolkit," *International Journal of Supercomputer Applications* (to appear).

[25] Gosling, J., McGilton, H., "The Java language environment: a white paper," Sun Microsystems Computer Company, Mountain View, CA, October 1995.

[26] Goswell, C., "The COM programmer's cookbook," Microsoft Corporation, Spring 1995.

[27] Gray, R., Cybenko, G., Kotz, D., Rus, D., "Agent TCL," in Cockayne, W., Zypa, M., eds. Itinerant Agents: Explanations and Examples with CDROM, Manning Publishing, 1997.

[28] Grimshaw, A.S., "Easy-to-use object-oriented parallel processing with Mentat," *IEEE Computer*, pp. 39-51, May 1993.

[29] Grimshaw, A.S., Weissman, J.B., West, E.A., Loyot, E., "Metasystems: an approach combining parallel processing and heterogeneous distributed computing systems," *Journal of Parallel and Distributed Computing,* vol. 21, no. 3, pp. 257-69, June 1994.

[30] Grimshaw, A.S., Wulf, W.A., the Legion team, "The Legion vision of a worldwide virtual computer," *Communications of the ACM*, vol. 40, no. 1, January 1997.

[31] Grimshaw, A.S., Wulf, W.A., "Legion—a view from 50,000 feet," *Proceedings of the Fifth IEEE Symposium on High Performance Distributed Computing*, IEEE Computer Society Press, Syracuse, NY, August 6-9, 1996.

[32] Homburg, P., van Doorn, L., van Seen, M., Tanenbaum, A.S., de Jonge, W., "An object model for flexible distributed systems," ???, Vrije Universiteit.

[33] Homburg, P., van Steen, M., Tanenbaum, A.S., "An architecture for a scalable wide area distributed system," ???, Vrije Universiteit.

[34] Johansen, D., Sudmann, N.P., van Renesse, R., "Performance issues in TACOMA," *Third Workshop on Mobile Object Systems, 11th Europeean Conference on Object-Oriented Programming*, Jyvaskyla, Finland, June 9-13, 1997.

[35] Johansen, D., van Renesse, R., Schneider, F., "An introduction to the TACOMA distributed system, version 1.0," Technical Report 95-23, University of Tromso, 1995.

[36] Jul, E., Levy, H., Hutchinson, N., Black, A., "Fine grained mobility in the Emerald system," *ACM Transactions on Computer Systems*, vol. 6, no. 1, pp. 109-133, February 1988.

[37] Kim, W., Chou, H.-T., "Versions of schema for object-oriented databases," *Proceedings of the 14th International Conference on Very Large Databases*, pp. 148-59, 1988.

[38] Knabe, F.C., "Language Support for Mobile Agents," PhD Dissertation CMU-CS-95-223, Carnegie Mellon University, December 1995.

[39] Lange, D., Oshima, M., Programming Mobile Agents in Java, ???

[40] Lerner, B.S., Haberman, A.N., "Beyond schema evolution to database reorganization," *OOPSLA '90: Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pp. 67-76, Ottawa, Canada, October 21-25, 1990.

[41] Lewis, M., Grimshaw, A.S., "The core Legion object model," *Proceedings of the Fifth IEEE Symposium on High Performance Distributed Computing*, IEEE Computer Society Press, Syracuse, NY, August 6-9, 1996.

[42] Lockhart, Jr., H.W., *OSF DCE Guide to Developing Distributed Applications*, McGraw-Hill, Inc. New York 1994.

[43] Message Passing Interface Forum, "MPI: A message-passing interface standard," May 1994.

[44] Microsoft Corporation, "The Component Object Model specification," Version 0.9, Microsoft Corporation, October 24, 1995.

[45] Monk, S., Sommerville, I., "A model for versioning of classes in object-oriented databases," *BNCOD 10, Advances in Databases: Proceedings of the 10th British National Conference on Databases*, Aberdeen, Scotland, pp 42-58, July 6-8, 1992.

[46] Monk, S., Sommerville, I., "Schema evolution in OODBs using class versioning," *SIGMOD Record*, vol. 22, no. 3, pp. 16-22, September 1993.

[47] Morsi, M.M.A., Navathe, S.B., Shilling, J., "On behavioral schema evolution in object-oriented databases," *Advances in Database Technology - EDBT '94: Proceedings of the 4th International Conference on Extending Database Technology*, Cambridge, UK, pp. 173-86, March 1994.

[48] Object Management Group, "The Common Object Request Broker: Architecture and Specification," Revision 2.0, July 1995 (updated July 1996).

[49] ObjectSpace, "Agent-enhanced distributed computing for Java," Voyager core package technical overview, available via http://www.objectspace.com.

[50] Palsberg, J., Xiao, C., Lieberherr, K., "Efficient implementation of adaptive software," *ACM Transactions on Programming Languages and Systems*, vol. 17, no. 2, pp. 264-292, March 1995.

[51] Peine, H. "An introduction to mobile agent programming and the Ara system," ZRI-Report 1/97, Department of Computer Science, University of Kaiserslautern, Germany, 1997.

[52] Peine, H. Stolpmann, T. "The architecture of the Ara platform for mobile agents," In Kurt Rothermel, K., Popescu-Zeletin, R. (Eds.), *Proceedings of the First International Workshop on Mobile Agents: MA'97*, Berlin, Germany, April 7-8, 1997. Lecture Notes in Computer Science No. 1219, Springer Verlag, ISBN 3-540-62803-7.

[53] Penney, D.J., Stein, J., "Class modification in the GemStone object-oriented DBMS," *OOPSLA '87: Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pp. 111-17, 1987.

[54] Schwarz, P., Shoens, K., "Managing change in the Rufus system," *Proceedings of 1994 IEEE 10th International Conference on Data Engineering*, pp. 170-9, Houston, TX, USA, February 14-18, 1994.

[55] Skarra, A.H., Zdonik, S.B., "The management of changing types in an object-oriented database," *Proceedings of the 1986 Object-Oriented Programming Systems Languages and Applications*, pp. 483-495, September 1986.

[56] Stamos, J.W., Gifford, D.K., "Implementing remote evaluation," *IEEE Transactions on Software Engineering*, vol. 16, no. 7, July 1990.

[57] Steensgaard, B., Jul, E., "Object and native code thread mobility among heterogeneous computers," *15th ACM Symposium on Operating Systems Principles (SOSP)*, ???, December 1995. QA76.6.S9197a

[58] Straber, M., Baumann, J., Hohl, F., "Mole—a Java based mobile agent system," *Proceedings of the ECOOP '96 Workshop on Mobile Object Systems*, 1996.

[59] Sunderam, V.S., "PVM: A framework for parallel distributed computing," *Concurrency: Practice and Experience*, vol. 2(4), pp. 315-339, December, 1990.

[60] SunSoft, SunOS 5.3 Linker and Libraries Manual, Sun Microsystems, Inc., Mountainview, California, 1993.

[61] Taivalsaari, A., "A critical view of inheritance and reusability in object-oriented programming," Jyvaskyla Studies in Computer Science, Economics, and Statistics No. 23, Salminen, A., ed., University of Jyvaskyla, 1993.

[62] van Steen, M., Homburg, P., Tanenbaum, A.S., "The architectural design of Globe: a wide-area distributed system," Internal report IR-422, Vrije Universiteit.

[63] Viles, C.L., Lewis, M.J., Ferrari, A.J., Nguyen-Tuong, A., Grimshaw, A.S., "Enabling flexiblity in the Legion run-time library," *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*, pp. 265-274. Las Vegas, Nevada, June 30 — July 2, 1997.

[64] White, J., "Mobile agents white paper," General Magic, http://www.genmagic.com/agents/Whitepaper/whitepaper.html, 1996.

[65] Wulf, W.A., Wang, C., Kienzle, D., "A new model of security for distributed systems," UVa CS Technical Report CS-95-34, August 1995.

Glossary

DCC: (*dynamically configurable class*) A class object whose instances are Dynamically Configurable Distributed Objects.

DCDO: (*dynamically configurable distributed object*) A distributed object whose implementation can change incrementally as the object runs.

DICM: (*dynamic implementation component mapper*) The table, maintained in a dynamically configurable distributed object, that maps implementation component names to addresses within the running executable image of the object.

ICS: (*implementation component store*) The logical data structure—maintained in dynamically configurable classes—that contains sets of implementation components for the instances of that class.

Implementation: A set of bytes, typically machine code, that can be used to activate a Legion object.

Implementation component: A piece of a dynamically configurable distributed object implementation which can be added, removed, or replaced via the object's incorporate() member function.

OPR: (*object persistent representation*) A set of associated bytes that represents a Legion object in its inert state.

Seed DCC: A dynamically configurable class whose implementation component store is initially empty.

Seed DCDO: A dynamically configurable distributed object that exports only the incorporate() member function and default implementations of object-mandatory functions.