

Dynamically Configurable Distributed Objects in Legion

Michael J. Lewis

Department of Computer Science
Binghamton University - SUNY
Binghamton, NY 13902
mlewis@binghamton.edu

Abstract

The dynamically configurable distributed object (DCDO) model helps enable object evolution and facilitate the development of distributed objects from multiple independent implementation components. Using DCDOs, programmers can evolve existing active objects to accept new member functions, to change the interface and behavior of their member functions, and to remove member functions from their public or private interface. Programmers can make 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 objects are initially compiled and run. The model supports evolution management strategies that define when and how object types evolve from one version to the next, and determine when a type change is propagated to existing instances.

DCDOs represent a useful way to evolve active distributed objects in a wide area distributed object computing system, and to enable component-based object composition via the mechanisms of the host distributed system itself. The DCDO model describes the functionality and roles of the object types necessary for an implementation of dynamic configurability. The model is simple, defining the interfaces to just three different object types, it is independent of any particular programming language, operating system, or architecture, and it is flexible enough to support a range of different evolution management policies and strategies, a representative set of which is described in this paper.

The complete DCDO model has been fully implemented within the current version of the Legion wide-area distributed object computing system. A performance study of this implementation measures the run-time overhead associated with the DCDO mechanism, and compares the performance of evolving DCDO objects with that of evolving normal Legion objects. This study demonstrates that dynamic configurability can be implemented efficiently, with a small amount of added overhead for method function invocation and object creation, and with a marked improvement in the cost of evolution.

1. Introduction

The continued growth and widespread deployment of gigabit networks and high-speed network protocols will provide significant communications bandwidth between increasingly powerful computers, enabling systems containing billions of objects and millions of machines. Harnessing the power of this vast array of raw hardware requires well-organized complex software, highly scalable algorithms and services, and effective abstraction of complexity. *Grid computing* [3] research aims to provide this software and these algorithms and services. Wide area distributed object computing systems such as *Legion* [4, 8], which models the system's components—applications, hardware, programs, users, etc.—as objects, are capable of meeting this challenge.

An important subclass of grid computing applications will be required to be constantly operational, because they are critical to the correct operation of the other components of the system. Unfortunately, the requirement that an application run constantly does not preclude the possibility that it might need to change its behavior and implementation, due to changing user requirements, an evolving environment, or bugs in the original deployment of the software. Dynamic evolution is the ability of an application to change its behavior while it runs, and is enabled by the core mechanism of the dynamically configurable distributed object (DCDO) model. Section 2 describes the details of this model, including the roles of the three main object types that it defines, namely DCDO's themselves, DCDO Managers, and Implementation Component Objects. Section 3 then identifies some of the problems that evolving distributed objects can cause for clients in the system, proposes a set of solutions for restricting evolution in useful ways, and describes a representative set of evolution management strategies that have been implemented using these restrictions. Section 4 briefly describes the results of a performance study, the details of which are available in [9]. Finally, Section 5 compares the DCDO model to related work, and Section 6 presents a summary and conclusions.

2. The DCDO Model

The *dynamically configurable distributed object (DCDO) model* helps enable object evolution and facilitate the development of distributed objects from multiple independent implementation components. Using DCDOs, programmers can

evolve existing active objects to accept new member functions, to change the interface and behavior of their member functions, and to remove member functions from their external (public) or internal (private) interface. Programmers can make 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 objects are initially compiled and run. The model supports evolution management strategies that define when and how object types evolve from one version to the next, and determine when a type change is propagated to existing instances.

The DCDO model is simple, defining the interfaces to just three different object types, it is independent of any particular programming language, operating system, or architecture, and it is flexible enough to support a range of different evolution management policies and strategies. The model has been fully implemented within the Legion wide-area distributed object computing system. A performance study demonstrates that dynamic configurability can be implemented efficiently, with a small amount of added overhead for method function invocation and object creation, and with a marked improvement in the cost of evolution.

A distributed object's implementation defines its behavior, which is generally fixed at compile and link time. That is, in traditional distributed object computing systems, an object implementation is defined by a static monolithic executable, potentially linked against an unchanging set of runtime libraries that is defined at link time. In contrast, a DCDO's implementation is fragmented into parts that can be replaced as the object runs, and these parts contain functions that can be turned on and off dynamically. Dynamic configurability gives a programmer the ability to effect changes to an object's implementation at run-time, allowing the programmer to change an object's behavior after it has been deployed and is running.

A DCDO consists of a set of *implementation components*, each of which contains the implementation of a set of *dynamic functions*. A dynamic function can be *exported*, in which case it can be invoked from another distributed object, or *internal*, in which case it may be called only from within the object in which it resides. The full set of exported dynamic functions comprises the implementation component's interface. Dynamic functions can be in one of two states, *enabled* or *disabled*. If a dynamic function is enabled, then some thread's flow of control may enter the function; if it is disabled, the object disallows all calls to the function. Implementation components may also contain a set of internal data structures, but these data structures must be accessed from outside the component by calling the component's exported dynamic functions. Every DCDO maintains a set of implementation components that are *incorporated* into the object. Once a DCDO incorporates an implementation component, the functions that the component defines may be enabled and then called.

A DCDO evolves its behavior in two ways: (1) by enabling and disabling its dynamic functions, and (2) by growing and shrinking the implementation component set that it maintains. Both kinds of evolution are made possible by a *dynamic function mapper* (DFM), a data structure maintained within every DCDO. A DFM contains an entry for every dynamic function that is currently contained in the object, and keeps track of whether the function is exported or internal, and whether it is currently enabled or disabled. A DFM serves as a centralized table through which all calls to dynamic functions must go. Thus, dynamic functions are not invoked directly using only the mechanisms of the programming language(s) with which an object is built. Instead, before calling a dynamic function, a caller must obtain from the DFM the ability to call the function, similar to the *QueryInterface* function in COM [11]. This "ability" to call a dynamic function typically takes the form of the function's address, but the exact mechanism depends on the implementation language. Thus, changing only the DFM (without changing the calling code) can result in a different implementation of the function being executed. This very simple technique of adding to function calls a single level of indirection, which has been applied to countless problems in computer science, is the basis and the key enabler of dynamic configurability.

2.1 Version Identifiers and Implementation Types

Since a DCDO's implementation is intended to change, dynamic configurability provides a mechanism for tracking changes. A *version identifier* is an array of positive integers that identifies some version of an object type's implementation. Every DCDO knows and contains the version identifier for its current implementation. Version identifiers are unique only within a particular object type, they are not necessarily globally unique across types. If two DCDOs of the same type are both of version 1 . 2 . 3, then their implementations are the same—that is, the same components are incorporated into the two objects, and the DFMs of the objects are functionally equivalent to one another (i.e. the same function implementations are enabled and exported).

Dynamic configurability allows different implementations of an object that are functionally equivalent to one another, and that can be used interchangeably to represent the same object. The most important reason for this is so that a system can employ compiled, architecture-specific, executable code in a heterogeneous environment, and still allow objects to migrate from one node to another, even if the architectures of the two nodes are different. Dynamic configurability supports the notion of *implementation types*, which identify the characteristics of different kinds of implementations. Every

implementation component has an associated implementation type, which describes properties such as the component's architecture, its object code format, and (if important) the programming language with which it was built.

2.2 DCDOs

A DCDO contains three different categories of functions: (1) *configuration functions* are used to evolve the implementation of the DCDO, (2) *status reporting functions* return the status of the DCDO's current implementation, but do not change the implementation, and (3) *user-defined functions* depend on the programmer's specific use for the DCDO, and typically have nothing to do with the dynamic configurability of the DCDO. The model concerns itself with only the first two kinds of functions. Thus, an object's external interface is the mechanism that is used to evolve its implementation, by affecting the components it incorporates and the functions that it keeps enabled and disabled. Examples of functions in this interface include `incorporateComponent()`, `removeComponent()`, `enableFunction()`, and `disableFunction()`.

2.3 Implementation Component Objects

An *implementation component object* (ICO) is an active distributed object that maintains an implementation component's data—the executable code that comprises the component, the descriptor that describes the contents of the executable code, and the component's implementation type. A DCDO incorporates a component by reading the data from the ICO and by using the appropriate operating-system-specific mechanism for mapping it into the DCDO's address space, so that it can be used as part of the DCDO's implementation.

An implementation component is maintained within an ICO primarily so that dynamic configurability can benefit from the global namespace defined by the host system; implementation components can be named using whatever scheme exists for naming objects in the system. A separate mechanism for managing a component namespace need not be implemented, and the component's (potentially large amount of) data need not travel with the component whenever it is referenced.

2.4 DCDO Managers

A DCDO Manager is in charge of maintaining implementation components for a particular object type, and for evolving the DCDOs that it manages. A DCDO does not evolve from one version to the next independently; instead, objects of the same type evolve more or less together, under the supervision of their common manager. The degree to which a DCDO's implementation can be out of synch with the version that it "should" reflect, or with the other DCDOs of the same type, is a characteristic of the object type's *evolution management policy*, which is largely implemented within a DCDO Manager.

A DCDO Manager maintains two primary data structures, a *DFM store*, and a *DCDO table*. The DFM store contains a set of *DFM descriptors* that define different versions of the object type that the manager represents. A DFM descriptor's structure mirrors that of a DFM, but it is not used to map function calls to their implementations; instead DFM descriptors are used by the DCDO Manager to configure its DCDOs appropriately; this is done when a DCDO is created, when it migrates to a host, or when it evolves to a new version. The DCDO Manager provides functions for deriving new versions from existing ones, and for configuring the new versions; these functions are similar to a DCDO's configuration functions.

Each version maintained within a DFM store is marked as either *instantiable* or *configurable*. If a version is instantiable, then a new DCDO can be created with the corresponding DFM descriptor, and existing DCDOs can evolve to that version. However, the DFM descriptor of an instantiable version cannot be changed any further. A configurable version and its corresponding DFM descriptor can be evolved and configured, but it cannot be used to create a new DCDO, or to evolve an existing DCDO, until the version is marked instantiable. Typically, a programmer uses a DCDO Manager's interface to create a new configurable version by logically copying an existing instantiable one, to configure the new configurable version appropriately, and then eventually to mark it instantiable so that it can be used. Distinguishing between instantiable and configurable versions ensures that all objects created with a certain version number by some particular DCDO Manager, have the same functionality; in other words it allows the <DCDO Manager, Version Id> pair to uniquely identify an object's interface and implementation.

The DCDO Manager also maintains a logical table of the DCDOs under its control. The DCDO table contains information that includes the version identifier and the implementation type that correspond to each object's current implementation. The DCDO Manager uses this information when deciding when and how to evolve its DCDOs from one version to the next, and exports functions to return this information to other objects.

3. Evolution

Section 2 largely describes the mechanism of the DCDO model; that is, it defines how the DCDO object types allow dynamic changes to implementations. This mechanism, however, has ramifications on the object model. If programmers were allowed to change the implementation of any type in the system, at any time, then the benefit of the "contract" between the objects of that type and their clients would be lost. Our philosophy with regard to this problem is to provide the fully general evolution mechanism as described above, but to also allow programmers to throttle and disable this

mechanism in some cases, i.e. for some of their object types. In Section 3.1 we identify in more detail the problems that evolving existing active objects can cause, and in Section 3.2 we propose some solutions, all of which have been implemented, for these problems. Section 3.3 then describes some possible organized applications of these solutions into evolution management strategies, which define when and how objects of a particular type are allowed to evolve from one version to the next.

3.1 New Problems

Dynamic configurability enables functionality that traditional objects do not allow, namely the alteration of an object's implementation as the object runs. Several types of problems can arise if DCDO configuration functions, particularly those that remove functions from the implementation, can be arbitrarily interleaved with calls on user-defined dynamic functions. The problems are described below.

The Disappearing Exported Function Problem: The disappearing exported function problem can occur when a DCDO's client obtains the interface of the object, and builds an invocation for an enabled exported dynamic function, say F_j , which the client finds in that interface. Then, before the invocation arrives at the DCDO, F_j is disabled and no replacement for F_j is subsequently enabled. The client's invocation will fail, even though the client got the interface from the object, and built an invocation that was correct according to the interface at the time it was obtained. Note that adding functions to a public interface, or changing the implementation of a function while keeping its signature the same do *not* cause problems of this type; clients' calls will not fail in the same way that they will if a dynamic function is removed from the interface.

The Missing Internal Function Problem: The missing internal function problem can occur when one dynamic function, say F_1 , in a DCDO, contains a call to another function F_2 , in the same DCDO, and when F_2 is not enabled in the object's DFM. F_1 can then reach a call to function F_2 that it cannot carry out. Normally, this is not a problem because compilers and linkers can check for missing functions when an executable is built, and the executable does not change after the check is made. When functions can be disabled and removed from an implementation, as dynamic configurability allows, a DCDO can attempt to call a function that no longer exists in its implementation.

The Disappearing Internal Function Problem: The disappearing internal function problem is a special case of the missing internal function problem. Suppose a thread is executing in the DCDO, for example inside of function F_j , but the thread is currently inactive, perhaps because it is blocked on an outcall waiting for a result. In the meantime, a call to a configuration function disables some other function F_2 . Then, the result arrives from the outcall, the thread in function F_j is awakened and tries to call F_2 , which no longer exists. In this case, the thread is affected by a configuration function that is called after the thread starts, but before it completes.

The Disappearing Component Problem: The disappearing component problem can happen when a thread is executing in some component C , and while the thread is inactive, a configuration operation removes component C from the object. When the thread is awakened, it has nowhere to execute.

3.2 Preliminary Solutions

The DCDO model contains several aspects designed specifically to help solve the problems identified above. The following subsections describe *mandatory* and *permanent* dynamic functions, *function dependencies*, and *function activity monitoring*, and explain how they help address the problems described in Section 3.1.

Mandatory and Permanent Functions

The DCDO model calls for dynamic functions to be marked in the DCDO Manager's DFM descriptor as *mandatory*, *permanent*, or *fully dynamic*. When a function is mandatory, some implementation of that function must be present in the DCDO. Further, an implementation of a mandatory function must be present in any instantiable version of the DFM descriptor that is derived from a version in which the function is marked mandatory. If the DFM descriptor contains a mandatory dynamic function with no enabled implementation, the version will not be allowed to be marked instantiable. Thus, once a DCDO evolves to a version that contains a function F that is marked as mandatory, all DCDOs that reflect that version, and all future versions to which those DCDOs will evolve, will contain some implementation of function F .

When a dynamic function is marked permanent, the implementation of that function is frozen. In other words, the dynamic function is essentially no longer dynamic, since the ability to change its implementation or to disable it has been shut off. Once a DCDO evolves to a version that contains a permanent function F implemented in component C , component C 's implementation of function F will be present in all derived versions of the type.

Programmers can mark a dynamic function as mandatory (or permanent) within a descriptor that is maintained with the component itself. This indicates that the function needs to be mandatory (or permanent) in *any* DCDO into which the component is incorporated. If this constraint cannot be met (for example, if a programmer attempts to incorporate compo-

nent C that contains permanent function F , into a DFM descriptor that contains another component with its own permanent implementation of function F), then the attempt to incorporate component C fails. Programmers can also mark dynamic functions as mandatory or permanent within some version of an object's DFM descriptor, using functions exported by DCDO Managers.

Supporting mandatory functions gives programmers a tool to help alleviate the missing function problem and both disappearing function problems. If a public dynamic function is mandatory in an interface obtained by a potential client, then that client can be assured that some function in the object's interface will implement that function for the lifetime of the object, as long as it only evolves to versions derived from the one in which it was marked mandatory. Likewise, potential callers within the object can be given the same assurance. When a dynamic function is permanent, callers can have a stronger assurance, namely that the *implementation* of the function will not change for the lifetime of the object, again assuming that the object only evolves to versions derived from one in which the function is marked permanent.

Essentially, allowing dynamic functions to be marked mandatory or permanent restricts dynamic configurability on a function by function basis, giving callers more indication about the future behavior of the object. If the programmer decides that a function should be allowed to continue to evolve arbitrarily (i.e. she does not mark the function mandatory or permanent), then the missing and disappearing function problems can persist, since the function could be removed "out from under" a call. In this case, it is up to the caller to deal with the problem gracefully. Thus, invocations on a dynamic function should be written to expect the absence of the function. Clients calling a DCDO should time out or catch an exception (depending on the mechanisms of the host system) that indicates that the function they tried to invoke was not present, and callers within the DCDO should check for the absence of a function in the DFM.

The rationale behind allowing functions to be marked as mandatory or permanent, is to provide programmers with the appropriate granularity of control over object evolution, and to directly address the missing and disappearing internal function problems. Since the DCDO mechanism enables evolution at the granularity of functions (which can be enabled and disabled), it is natural to provide tools to restrict evolution at the same granularity, which mandatory and permanent functions do.

Function Dependencies

Dealing with missing and disappearing function problems by using only a strategy of marking them as mandatory or permanent has drawbacks. In particular, the strategy requires the programmer to identify the functions that should be so marked. Essentially, this asks the programmer to decide (and predict) whether it will be more important to allow a DCDO to evolve arbitrarily, or to make assurances to clients about the object's future interface and implementation. Consider the following scenario, which highlights the problem. A programmer marks internal function F_2 as mandatory because it is called by some enabled implementation of function F_1 , and the programmer does not want F_1 to break if F_2 is disabled. Then, F_1 is disabled and removed from the object's implementation (after all, F_1 was not necessarily marked as mandatory or permanent). Now, the programmer is left with F_2 being marked mandatory, but the main reason for marking it mandatory no longer applies, and the object's ability to be dynamically configured has been restricted; in particular, the object cannot evolve to a version that does not contain an implementation of function F_2 .

Dynamic configurability does more about the missing and disappearing *internal* function problems than simply allowing programmers to mark functions as mandatory or permanent. Programmers can also indicate that certain dynamic functions "depend on" other functions in an interface or an implementation. Programmers may indicate two different kinds of dependencies, *structural* dependencies and *behavioral* dependencies. A structural dependency exists when, as described above, one dynamic function contains a call to another. If dynamic function F_1 calls another function F_2 , then F_1 depends structurally on F_2 ; that is, some implementation of F_2 must exist in order for F_1 to execute without potentially encountering a call to a function (F_2) that it won't be able to carry out.

A behavioral dependency is a stronger claim; if a programmer indicates that F_1 depends behaviorally on the implementation of F_2 in component C , then she indicates that if F_1 is enabled, then the implementation of F_2 in component C must remain enabled, not just some implementation of function F_2 , as is the case when she indicates a structural dependency. Behavioral dependencies are designed to shield functions from the behavioral effects of changing the implementation of other functions. For instance, suppose function "Integer[] sort(Integer[])" calls another function "Integer compare(Integer, Integer)" the current implementation of which returns the smaller of two integers. In general, it is possible to replace `compare()` with a different implementation that has the same signature, but that instead returns the larger of the two numbers. This change would *not* cause `sort()` to fail due to a violated structural dependency, since an implementation of `compare()` would still be found in the object; but the change would alter `sort()`'s output—the order of the sorted array would be reversed. The provider of `sort()` may want to ensure that this doesn't hap-

pen; to do so, she can set a behavioral dependency in the DCDO Manager that states that `sort()` depends behaviorally on some particular implementation of `compare()`.

The following kinds of dependencies can be specified:

- **Type A:** $[F_1, C_1] \rightarrow [F_2]$: This expresses a structural dependency of the implementation of dynamic function F_1 in component C_1 on function F_2 , but does not express a behavioral dependency. If the implementation of F_1 found in C_1 is enabled, then some implementation of F_2 must also be enabled. This dependency alone does not preclude the possibility of replacing the implementation of F_2 and having F_1 call the new implementation, but it does guard against the possibility that no implementation of F_2 will exist for F_1 to use.
- **Type B:** $[F_1, C_1] \rightarrow [F_2, C_2]$: This expresses a behavioral dependency of the implementation of function F_1 in component C_1 on the implementation of function F_2 in component C_2 . If the implementation of F_1 found in C_1 is enabled, then the implementation of F_2 in C_2 must also be enabled.
- **Type C:** $[F_1] \rightarrow [F_2, C_2]$: This expresses a behavioral dependency of any implementation of function F_1 on the implementation of function F_2 in component C_2 . If any implementation of F_1 is enabled, then the implementation of F_2 in C_2 must also be enabled.
- **Type D:** $[F_1] \rightarrow [F_2]$: This expresses a structural dependency of any implementation of function F_1 on any implementation of function F_2 . If any implementation of F_1 is enabled, then some implementation of F_2 must also be enabled.

These four dependencies can be added to a DFM descriptor using functions in a DCDO Manager's exported interface.

Dependency Type A is designed to be used with functions that contain calls to other functions within the object, but that still want to benefit from the potential upgrading of those functions. Thus, this dependency type is designed to address directly the disappearing internal function problem, while still reaping a primary benefit of dynamic configurability. The Type B restriction (which allows functions to depend on particular implementations of other functions) is designed for those functions whose behavior, in the estimation of the programmer, should remain as constant as possible. Note that neither the Type A nor the Type B dependency restricts the evolution of the dependent function (F_1) itself. Instead, the dependencies state that as long as the function *is* enabled then the others it depends on must be as well.

To ensure completely that an exported function F_1 will never call a function that does not exist, it is up to the programmer to create the appropriate dependency chain. That is, F_1 should be annotated to depend on all dynamic functions that it calls, and these functions should be marked to depend on the functions that they call, etc. If this is done in such a way that F_1 could never lead to a call to a dynamic function that is not depended on by some other function in the call chain, then calling F_1 will never lead to a broken call. The Type A and Type B dependencies provide the mechanism for programmers to make this assurance, but it is up to the programmer to set the dependencies appropriately for their application.

Type C and Type D dependencies state that *any* implementation of some function can require the presence of some other function. This is perhaps less useful in combating missing and disappearing functions because the dependence of one function on some other is a property of the particular implementation of that function, rather than its signature. But allowing these latter two dependency types could be useful for other purposes. For example, a function F_1 may require that a security function F_2 be enabled to restrict access to F_1 . In this case F_1 may not call F_2 , but still requires that it be present. These dependency types (C and D) do not specifically address the problems of Section 3.1.

Function dependencies represent a flexible mechanism for allowing DCDO builders to restrict the way that DCDOs can be dynamically configured, based on the characteristics of the particular objects being built. Essentially, a structural dependency indicates that the function being depended on should be treated as if it were mandatory, and a behavior dependency indicates that the function being depended on be treated as permanent. However, unlike mandatory or permanent functions, dependencies can evolve along with the implementation, so a dynamic function's "mandatory" or "permanent" status can be essentially retracted when dependencies on it are removed, which can happen when dependent functions are disabled, replaced, or removed.

It is likely that creating structural dependencies could be automated via static analysis of source code by whatever entity builds implementation components and function implementations. If dynamic function F_1 contains a call to dynamic function F_2 , a relationship that can (for the most part) be detected by analyzing the source code for F_1 's implementation, then F_1 depends structurally on F_2 . Behavioral dependencies, on the other hand, are largely opinions that must be expressed somehow by programmers. A compiler cannot in general tell on its own that some dynamic function should require a particular implementation of some other function; programmers must indicate this directly.

Thread Activity Monitoring

The approaches described in Sections and do not address the disappearing component problem described in Section 3.1. To deal with this problem, a DCDO can monitor the threads that execute within itself, keeping track of the dynamic function implementations that have threads inside them. This is possible to do because all calls to dynamic functions go through the DFM. Thus, a counter for each dynamic function can be maintained within the DFM entry for that function.

When a request to remove a component arrives at the DCDO, the object can check to make sure all functions in the component have active thread counts of zero, thereby eliminating the possibility of removing the code for a component out from under a thread. Upon receiving a request to deactivate a component that contains active threads, a DCDO can do different things depending on the policy that it implements; it can return an error, it can delay handling the request until all thread counts go to zero, or it can simply go ahead with the operation after some time-out period that gives the thread a chance to complete.

It might seem important to make sure that a dynamic function's active thread count is zero before deactivating that function, not just before removing the function's component from the DCDO. However, there is no reason why a thread cannot proceed inside a deactivated function; after all, the code for that function still exists in the DCDO. Deactivating a function means that future calls to it are disallowed by the DFM; thus, in some sense it only matters what the status of the function is at the time the call is initiated, not whether the function remains activated throughout the lifetime of the execution of the function. On the other hand, active thread counts can be used in concert with function dependencies to avoid disappearing internal function problems. For example, if function F_1 depends on F_2 , and a thread is executing in F_1 , then the DCDO can postpone any request to disable F_2 until the active thread count for F_1 (and for all other functions that depend on F_2) goes to zero. Finally, by indicating that a function depends on itself, a programmer can ensure that recursive functions are not changed or removed while they are executing.

Again, the three mechanisms described in this section are designed specifically to combat the problems described in Section 3.1. They provide tools for programmers who anticipate these being particularly important problems for their application (or for the clients that use the application) to reduce the circumstances in which the problems arise, or to eliminate the problems altogether. This requires knowledge of both the application and of the interface to the DCDO Manager that enables the mechanism. Thus, in some sense, the mechanisms provides an extra burden on the application developer, but they also allow her to alleviate problems enabled by the DCDO model.

3.3 Evolution Management Strategies

The DCDO model defines an approach for providing mechanism that enables object evolution; it is not a detailed specification of some particular implementation of the idea. The approach is intended to be useful for a range of applications and problem areas. Therefore, no single evolution policy—which defines when and how an object evolves from one version to the next via dynamic configurability—will be appropriate for all applications, because each will potentially have its own unique requirements. The main object types' interfaces are designed to support an extensible set of different evolution management policies. This section describes several representative policies, and discusses how they can be implemented within the structure defined by the objects described above. The policies described here do not make up the complete set enabled by the model, rather they serve simply to help illustrate that the model is flexible and extensible enough to support a wide range of such policies.

Sections 3.1.1 and 3.3.2 describe functions whose job is to evolve DCDOs. However, by itself, the usefulness of the mechanism is limited; it must be used in concert with organized evolution policies so that programmers can implement and control object evolution in ways that they can comprehend, and that fit the purpose of the evolving objects. Evolution management policies define the types of changes that are legal, and define and restrict when the changes take place. This section describes several different policies, which are distinguished at the highest level by the degree to which they support multiple versions of an object type.

3.4 Single-Version DCDO Managers

A single-version DCDO Manager defines exactly one official version of the DCDOs under its control at any given moment in time. All new DCDOs are created to reflect the characteristics of the designated current version. When a new current version V is designated, the DCDO Manager attempts to evolve all of its DCDOs to reflect the implementation of version V . Within this style, programmers are allowed to create new version descriptions and to mark them as instantiable. However, DCDOs will only evolve to the current version maintained by the DCDO Manager, not to any other version, even if it is marked as instantiable.

Since many DCDOs may be under the control of a single manager, and since they are distributed across the system, updates to the current version cannot be instantaneous. Therefore, within the single-version DCDO Manager style, there exist several strategies for updating existing DCDOs to the current version. The problem is analogous to keeping multiple reader caches up to date with the data contained in a centralized data repository. The DFM descriptor for the current version in the DCDO Manager represents the official copy of the data, and the DFMs in the DCDOs represent cached copies

of that data in multiple reader caches. Different DCDO Managers can offer different policies and strategies for keeping DCDO implementations consistent with the current version as defined by the DCDO Manager.

In the *proactive update* policy, the manager incorporates changes into existing DCDOs as soon as a new current version is set in the manager; in other words, designating a new current version triggers an immediate attempt to update all existing instances. This strategy allows a DCDO to be out of date only as long as it takes for the DCDO Manager to propagate the new DFM descriptor to that DCDO, and for the DCDO to apply the descriptor to its implementation. However, the strategy could incur unnecessary overhead if, for example, another change is forthcoming before any calls on the DCDO are made. Further, this strategy does not scale well with the number of DCDOs managed by a particular DCDO Manager; creating a new current version can become expensive. But for object types with few DCDOs that change relatively infrequently, this policy may be appropriate.

Another option is the *explicit update* policy, in which the DCDO Manager relies on other objects to call to the manager in order to evolve them to the new current version. This option separates the policy for updating instances from the DCDO Manager, and allows it to be made by a different external object. This could be useful when, for example, multiple object types need to be updated in coordination with one another. It also allows clients to discover that a DCDO is out of date, and to initiate the update to the current version before invoking a function on the object.

A third approach is the *lazy update* policy, in which a DCDO itself determines when it gets updated to the current version. The simplest variation of this policy is to enforce strict consistency semantics by having DCDOs consult their class every time they get an invocation request, to see if the request can be serviced, or if the DCDO must first be brought up to date with the current version. Other variations allow a DCDO to check for updates less often, perhaps once every k member function calls, once every t time units, or only when it migrates from one host to another.

3.5 Multi-Version DCDO Managers

An alternative to the single-version managers are *multi-version* DCDO Managers, which allow multiple versions of the same object type to co-exist, rather than trying to keep them all up to date with the current version. This contrasts with the single version manager style, wherein multiple versions can exist only in the lag time that it takes to update objects to the official current version. As with the single version managers, several different update policies exist for multi-version managers.

In the *no-update* policy, each DCDO is created with a particular version number, and never evolves to a different version. This limits (or arguably, completely disables) dynamic configurability, in that deployed running objects do not evolve on the fly. DFM descriptors in DCDO Managers evolve via version management and version configuration functions, but they are applied only to new DCDOs, not to existing ones; once a DFM is set, it doesn't change.

Another option is the *increasing version number* policy, in which a DCDO of version V can only evolve to other versions that are (eventually) derived from V . For example, a version 3.2 DCDO can evolve to version 3.2.1 or to version 3.2.0.4, but not to version 3.3. In general, the set of versions defined for an object type in a DCDO Manager forms a tree, and objects can only evolve to versions that are descendants in that tree. This policy works well with the use of mandatory dynamic functions described in Section 3.2. If a DCDO can only evolve to a derived version, and if a function is mandatory in the interface to that DCDO, then a client can be assured that the function will always exist in all future versions of the object. If the DCDO could evolve to a version not derived from that object's current version, then it could potentially evolve to an implementation that does not contain the function.

A third option among multi-version DCDO Managers is the *general evolution* policy, in which a DCDO can evolve to any other ready version at any time. This undermines the use of mandatory and permanent functions, but clients can still query the interface of the DCDO to determine if a function it needs is still exported by the DCDO, or the client can examine the version number of the object to see if it happens to be derived from a version for which the client knows the interface. A hybrid between the increasing version number and general update policies checks to see if evolving a DCDO to a version violates any rules, such as removing a mandatory function or disabling a permanent function, and disallows any such updates.

Within the multi-version DCDO Managers, slight variations of the proactive, explicit, and lazy update policies can be implemented. For example, within the increasing version number policy, the explicit update policy could be altered to allow any ready version number eventually derived from the DCDO's current version to be specified in the parameter to `updateInstance()`. Likewise, the different variations of the lazy update policy could automatically update DCDOs to the current version, but only for those DCDOs whose version derives the current version. For example, after some time-out period, a DCDO may check to see if a new current version has been set in its DCDO Manager. If so, the DCDO updates its implementation, but only if the new current version is derived from the DCDO's version; otherwise the DCDO remains at its present version.

4. Performance

A full description of the implementation of the DCDO model is beyond the scope of this paper, primarily because it requires an understanding of the Legion implementation. Likewise, a detailed description of the performance study would require many more pages than have been allotted for this paper. Both the implementation and the performance study are described in more detail in [9]. This section simply presents the highlights of the performance study. We summarize results of tests that were conducted on the Legion “Centurion” machine. The testbed subset consisted of 16 Dual Processor 400 MHz Pentium II’s, each with 256 Megabytes of RAM. The machines were connected with a 100 Mbps Switched Ethernet.

In evaluating the performance of the dynamic configurability mechanism, it is important to characterize two different aspects of the implementation, (1) the *overhead* of using the mechanism, and (2) its *cost*. Overhead refers to the performance penalty that results from objects being built so that they can evolve on the fly. Dynamic function calls go through a DFM, which adds a level of indirection and can take longer than calls that are made directly.

Overhead: At the highest level, two different kinds of function calls were measured, (1) intra-object calls that result from a part of a DCDO invoking a function that exists within the same object, and (2) inter-object or remote calls, in which one object calls a function exported by a DCDO. Results showed that a dynamic function takes between 10 and 15 microseconds per call, for self-calls, intra-component calls, and inter-component calls alike. And as expected, remote invocations of DCDO dynamic functions take no longer than calls made on normal Legion objects (since 10-15 microseconds is a small fraction of the overall time needed to complete a remote method invocation), and the roundtrip times are independent of the number of functions and components in a DCDO implementation.

Due to the details of how DCDOs are created (from multiple implementation components rather than a single monolithic executable), it takes longer to create DCDOs than regular Legion objects. For example, incorporating an object with 500 functions separated into 50 components takes about 10 seconds, whereas creating an object with the same 500 functions that reside in a static monolithic executable takes only 2.2 seconds. For more reasonably configured objects (e.g. with fewer components), results are comparable to the static executables. Again, a full set of results are presented in [9].

Cost: The cost of evolving a normal Legion object (i.e., an object that is represented by a static monolithic executable) from one implementation to the next, contains several different parts, including capturing the state of the object, transferring the state to a new machine (if necessary), downloading the new executable that represents the next “version” of the object, creating a new process for the object, reading the state information into the new process, and getting clients to know of the new physical address for the object. The performance study shows that in Legion, (1) it takes objects approximately 25 to 35 seconds to realize that a local binding contains a physical address that the object is no longer using, and (2) a 5.1 Megabyte object implementation (typical for moderately sized Legion objects) takes 15 to 25 seconds to download and that a 550 K implementation takes about 4 seconds to download. State capture and recovery are clearly object-specific parameters that depend on the size and format of the object’s contained data.

The study also shows that the cost of evolving a DCDO from one implementation to another is less than half a second, except for the case when new components need to be incorporated. When new components are incorporated, the cost rises to levels roughly equivalent to the time necessary to create a new object. When the components are cached and available to the DCDO that is evolving, the cost is approximately 200 microseconds per component that needs to be added. When the components need to be downloaded to a location in the file system where the DCDO runs, then the cost of evolution is dominated by the time needed to download the component data. Thus, incorporating larger components takes longer than incorporating smaller ones. Even in these extreme cases, the performance advantage of evolving object on the fly and avoiding the stale binding problem and the need for a full executable download, not to mention state capture and recovery, are dramatic.

5. Related Work

Research into the dynamic configuration of distributed programs and systems has grown from a seminal paper by Kramer and Magee [5], in which the authors describe a system called CONIC, which permits dynamic incremental modification of a running system. In a subsequent paper [6], the authors describe an approach to maintaining node (module) consistency (defined loosely as some invariant or constraint about a node that must be preserved) in the face of configuration changes. The authors’ subsequent work lead to systems called REX [7] and Regis [10]. These approaches deal with many of the same issues and problems that DCDOs are designed to address, but a fundamental difference in the work is that evolving a DCDO involves changing the in-core image of an individual running process, rather than the relationship among multiple threads or processes. Thus, the *implementation* of change is at a more fine-grained level, thereby leading to some different issues and challenges.

As mentioned earlier, the mechanism for Microsoft’s Component Object Model (COM) is similar to that of DCDO’s. In that they both use dynamic virtual tables to map function calls to the function implementations that service those calls.

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. 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 heterogeneous wide-area object-based distributed system, where an object's implementation characteristics are determined by its type. Therefore, COM should be considered as an effective enabling technology for implementing dynamic configurability (although it is not used in the Legion implementation), but not sufficient on its own to solve the problems that DCDO technology is designed to solve.

Finally, the evolution management strategies of Section 3.3 are drawn largely from strategies found in object-oriented database schema evolution research [e.g., 2, 12, 13] (both class modification and class versioning approaches), with the obvious differences in implementation details necessary for database objects and for the active distributed objects represented by running processes that are present in Legion.

6. Summary

This paper has described the dynamically configurable distributed object model, which defines a mechanism for distributed objects to evolve their implementations as they run. A DCDO Manager object coordinates and carries out the evolution of the objects under its control, and changes their implementation by calling member functions that exist in the core interface of all DCDO objects. Allowing objects to evolve with the full generality of the DCDO mechanism (i.e. by adding, removing, or changing the implementation of *any* member function at *any* time) can lead to significant problems, the DCDO model provides mechanisms for restricting evolution, and allows programmers to compose these mechanisms into useful evolution management strategies for their applications and object types. Several representative strategies have been implemented and described. The DCDO model has been implemented within the Legion object-based grid computing system, and the overhead of the mechanism has been shown to be small relative to the cost of invoking member functions on remote distributed objects. The cost of evolving DCDOs dynamically is significantly less than using the traditional mechanism that is otherwise required in Legion.

References

- [1] 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.
- [2] Ferrandina, F., Ferran, G., Mdec, J., Meyer, T., Zicari, R., "Database evolution in the O₂ database system," *Proceedings of the 21st International Conference on Very Large Databases*, pp. 170-81, Zurich, Switzerland, September 1995.
- [3] Foster, I, Kesselman, K. (Editors) *The Grid : Blueprint for a New Computing Infrastructure*, Morgan Kaufmann Publishers.
- [4] 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.
- [5] Kramer, J., and Magee, J., "Dynamic Configuration for Distributed Systems," *IEEE Transactions on Software Engineering*, Volume SE-11, Number 4, April 1985.
- [6] Kramer, J., and Magee, J., "The Evolving Philosophers Problem: Dynamic Change Management," *IEEE Transactions on Software Engineering*, Volume 16, Number 11, November 1990.
- [7] Kramer, J., Magee, J., Sloman, M., Dulay, N., "Configuring object-based distributed programs in REX," *IEE Software Engineering Journal*, Volume 7, Number 2, pp. 95-101, March 1992.
- [8] 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.
- [9] Lewis, Michael J., "Dynamically Configurable Distributed Objects," UVA PhD Dissertation, May 2001.
- [10] Magee, J., Dulay, N., Kramer, J., "A constructive development environment for parallel and distributed programs," *IEEE*, pp. 4-14, 1994.
- [11] Microsoft Corporation, "The Component Object Model specification," Version 0.9, Microsoft Corporation, October 24, 1995.
- [12] 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.
- [13] 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.