Dynamically Configurable Distributed Objects

Michael J. Lewis and Andrew S. Grimshaw
{mlewis, grimshaw}@cs.virginia.edu
Legion Research Group
Department of Computer Science
University of Virginia

Abstract

Programmers require the ability to evolve the behavior of their software objects because the objects may contain bugs, they may cease to fit their changing environment, or users' requirements may change. Evolving an object to a new version requires changing that object's implementation, which in distributed object computing systems typically exists as a binary executable file. Dynamic configurability, as described in this paper, represents a departure from the use of binary executables. Using dynamically configurable distributed objects, programmers can evolve existing active objects to accept new member functions, to change the interface and behavior of member functions, and to remove functions from public and private interfaces. Programmers can implement these changes on the fly, without deactivating any part of the system (including the object being evolved), without replacing executables, without necessarily 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. A fully-functional implementation of dynamic configurability has been built into the Legion wide-area distributed object computing system, and has proven to be an efficient and effective mechanism for enabling dynamic distributed object evolution.

This paper was submitted to *ACM Principles of Distributed Computing*, on November 3, 1998.

1. Introduction

Programmers require the ability to evolve the behavior of their software objects because objects may contain bugs, they may cease to fit their changing environment, and users' requirements may change. Object evolution requires programmers to change object implementations. In distributed object computing systems, implementations are typically static monolithic binary executables. Programmers register their executables with the system, which executes them at users' requests. Changing an object's interface, composition, or implementation–however minor the change–requires replacing an entire executable. In large-scale decentralized systems, which necessarily employ implementation caching and replication, object evolution can become expensive and inconvenient. Moreover, evolution is seldom transparent to other objects, which can become incompatible with new versions.

Suppose a wide-area system contains an object type that serves as a generic container of data. Objects of this type serve the same purpose as operating system files, but are accessible from any of the potentially many sites with multiple disjoint underlying file systems, thereby providing a unified global file system. Each global file object would support member functions for reading and writing data, and an access control list for protection. Global file objects provides a basic service that is likely to be useful in a wide variety of applications; therefore multiple users may create many different instances of the object type. Suppose now that the author of the object type creates a new version that reflects an implementation enhancement or fixes a bug. For example, the implementation could be altered to support an improved pre-fetching algorithm.

Consider the difficulty in deploying the new implementation; each object must be deactivated and restarted using the new executable. This requires that all global file objects be identified, and that all cached copies of the implementation be removed. The data contained in the objects must be transferred from one process to the next, which could be expensive and difficult. Furthermore, the protocol for making the transfer would have to be built into the initial version of the object. Also, clients would have to learn about the objects' new physical addresses; that is, although the high level object names may remain unchanged, clients must communicate with new physical processes that represent the objects. Although in most systems this would be automatic and functionally transparent, the new processes would not be discovered instantaneously; therefore, clients would be slowed as they wait for calls to old processes to time out. All of these performance problems could negate any improvement made by the prefetching algorithm, thereby inhibiting the ability of global file objects to evolve.

The problems described above stem from having to deploy a completely new executable to make even the smallest implementation change. A better mechanism would be able to propagate *only the change* to the objects, and would not require new processes to be created for the objects. Enabling efficient and effective distributed object evolution requires (1) a departure from static monolithic implementations, and (2) strategies for managing change and controlling its effect on other objects. The dynamically configurable distributed object (DCDO) model meets both requirements and enables dynamic distributed object evolution.

Using the DCDO model, programmers can evolve an existing active object to accept new member functions, to change its interface and behavior, and to remove functions from its public and private interfaces. Programmers make these changes on the fly, without deactivating any part of the system (including

the objects being evolved), without replacing executables, without interrupting clients, and without knowing what the changes will be when the object is initially compiled and run.

The DCDO model addresses a new problem–the fact that run-time representations of distributed objects inhibit their ability to evolve efficiently and effectively–in a new environment, namely wide-area heterogeneous distributed object computing systems. The model incorporates existing features from other research areas, and combines them in a new way. In particular, the uniqueness of the research is due to the combination of three characteristics: (1) an object's behavior is altered at run-time by loading code dynamically, and by changing a "dynamic function mapper", (2) control of the function mapper is exported to an object type's manager through the object's public interface, and (3) strategies for controlling evolution are defined by the manager, and can be configured and restricted to meet the object type's needs. Recent research has focused on the dynamic extensibility and configurability of operating systems (e.g. Spin [6], Exokernel [12]), network protocols (e.g., *x*-kernel [17]), and individual applications such as Web browsers. This paper reports on work that brings similar extensibility to distributed objects and metasystems.

2. Related work

Current distributed computing systems do not support mechanisms that have the same goals and characteristics as dynamic configurability. This is because existing systems almost exclusively use static executables to create processes or tasks. For example, CORBA [28] and DCE [23] both use IDL compilers to generate high level language function stubs, which are then filled in by programmers and compiled together into a single binary executable. PVM [32] and MPI [24] create tasks by running executables from system or user "bin" directories. Likewise, Mentat [16] objects are represented by binaries created by a special compiler. In these and other working systems, changing the behavior of a program, task, or object, requires replacing an executable in its entirety.

Several languages and systems do begin to eliminate the dependence on static monolithic implementations. The Java class-loader allows programs to load new code at run-time [14], and other languages like Kali-Scheme [9] and Erlang [] contain similar mechanisms. The COM programming model and binary interoperability standard allows different components of a running application to evolve separately from one another without causing undefined function errors [8, 15, 25]. Dynamically linked libraries partition a program into parts that can be built and changed separately [33]. And mobile agent systems transfer runnable code between active entities in a distributed system [1, 2, 7, 18, 19, 29, 35].

However, these approaches alone are inappropriate for heterogeneous distributed object computing systems. Each requires a single common source language (Java, Kali-Scheme, Erlang, and mobile agents), computer architecture (COM), or software environment (dynamic linking). Furthermore, with COM and traditional dynamic linking, changes take effect only when an object is re-started, not as it runs. Finally, none of the mechanisms are designed specifically for objects communicating with other entities in a distributed system; therefore, no provision exists for evolving public interfaces. The dynamic configurability model explicitly supports multiple languages and architectures, allows changes as an object runs, and updates public interfaces automatically.

Evolution has been addressed in several research areas, most notably object-oriented database schema evolution. Orion [5], GemStone [30], OTGen [20], and O_2 [13] support "class modification," where a single version of a database schema is changed, and the contents of the database are updated to

reflect the change. Encore [31], CLOSQL [26], and Clamen's system [11] support class versioning, in which multiple versions of a single class co-exist, and the systems implement various strategies for updating the rest of the database, including class instances. These approaches offer excellent guidelines for evolving distributed objects, since many of the problems are analogous. Therefore, the evolution management approaches presented in Section 4 are largely based on these techniques. However, schema evolution *implementation* techniques do not apply to distributed objects, because they fail to consider that objects are shared, persistent, active, independent, and distributed. Evolving passive objects in databases that are under a single administrator's control is different from evolving active distributed objects in wide-area systems. Thus, dynamic configurability utilizes dynamic code loading technology to apply and adapt database schema evolution techniques.

3. The Model

The dynamic configurability model provides an approach for building distributed objects that can evolve their implementation and behavior on the fly, without being deactivated or restarted. The model provides a high-level implementation prescription, and is independent of any particular source language or host distributed system. This section begins with a description of the unrestricted dynamic configurability mechanism, Section 3.4 describes several problems that arise from this unrestricted mechanism, and Section 3.5 extends the model to include solutions.

3.1 DCDOs

A distributed object's behavior is generally fixed at compile and link time, since it is determined by an executable. In contrast, DCDO implementations consist of parts that can be replaced as the object runs, and the parts contain functions that can be turned on and off dynamically. Dynamic configurability allows programmers to update an object's implementation at run-time, thereby changing the object's behavior after it has been deployed.

A DCDO comprises multiple *implementation components*, each of which contains the implementation of *dynamic functions*. An *exported* dynamic function can be invoked from another distributed object, whereas *internal* functions may be called only from within the object in which they reside. Dynamic functions can also be *enabled* or *disabled*; an object only allows its threads to enter enabled functions. In static implementations, all functions are implicitly enabled, otherwise there would be no need to include them in the implementation. Designating functions as enabled or disabled allows dynamic configurability to alter an implementation at run-time. Every DCDO maintains a set of implementation components that are *incorporated* into the object. Once a DCDO incorporates an implementation component, the component's functions may be enabled and called.

DCDOs evolve their behavior in two ways: (1) by enabling and disabling their dynamic functions, and (2) by growing and shrinking the implementation component sets they maintain. Both kinds of evolution are enabled by a *dynamic function mapper (DFM)* maintained within the DCDO. A DFM contains an entry for every dynamic function currently contained in the object, and indicates whether the function is exported or internal, and enabled or disabled. A DFM is a centralized table through which all dynamic function calls coming from within an object 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 (i.e. another function contained in the object) must obtain from the

DFM the ability to call the function, e.g. by retrieving the function's address. Changing only the DFM, without changing the calling code, can result in the execution of a different function implementation. The calling code in remote clients (i.e. other distributed objects) remains unchanged, but remote invocations also go through the DFM because the DCDO's function dispatcher uses the DFM. Thus, changing the DFM can also alter the behavior of an object from the perspective of other distributed objects. This very simple technique of adding a level of indirection to all dynamic function invocations is the basis for dynamic configurability.

A DCDO exports two different categories of functions, as shown in Figure 1. Configuration functions provide the DCDO's manager with an interface to the DCDO's implementation. This interface includes functions that enable and disable the object's dynamic functions, that incorporate new components, and that remove components. Configuration functions allow the manager to propagate new implementations and versions to its objects. Status reporting functions allow other objects to retrieve information about a DCDO's implementation. Different status reporting functions return descriptions of the functions and components in the current implementation, the object's current version identifier, and its implementation type.

Configuration Functions
- int incorporateComponent(ImplementationComponentId)
- int removeComponent(ImplementationComponentId)
- int enableFunction(FunctionId,ImplementationComponentId)
- int disableFunction(FunctionId)

Status Reporting Functions
- ImplementationStatus getImplementationStatus()
- VersionId getVersion()
- ImplementationType getImplementationType()

Figure 1: A DCDO's Core Interface

A *version identifier* is an array of positive integers that identifies some version of an object type's implementation. Every DCDO contains the version identifier for its current implementation. If two DCDOs of the same type are both of version *1.2.3*, then they incorporate the same components, and their DFMs are functionally equivalent. Most versions are derived from another existing version of that type. The first time a new version *Y* of an object type is derived from some existing version *X*, version *Y*'s identifier is derived from *X*'s by adding another integer field and setting its value to *1*. In future derivations, this field counts the number of times a new version has been created directly from version *X*.

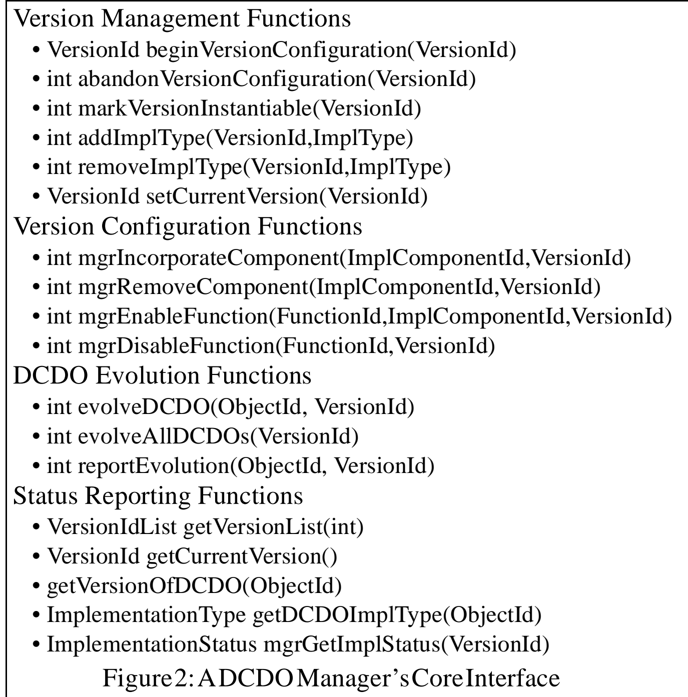
Implementation types identify the characteristics of different kinds of implementations, such as the component's architecture, object code format, and source programming language. Implementation types allow dynamic configurability to work well in systems that support multiple implementations for the same object (perhaps one per architecture), and allow multiple dynamic code loading technologies to co-exist within the same system.

3.2 Implementation Component Objects

An *implementation component object (ICO)* is an active distributed object that maintains a component's data–the executable code comprising the component, the descriptor describing the code's contents, and the implementation type. A DCDO incorporates a component by reading the data from the ICO and by mapping it into the DCDO's address space to be used in the DCDO's implementation. Implementation components are maintained within distributed objects primarily so that dynamic configurability can benefit from the host system's global namespace; a separate component namespace need not be implemented, and the component's data need not travel with the component whenever it is referenced.

3.3 DCDO Managers

A DCDO manager maintains implementation components for an object type, and evolves the DCDOs that it manages. DCDOs do not evolve independently; instead, objects of like type evolve 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 type's evolution management policy (Section 4), defined by its DCDO manager. A DCDO manager maintains a *DFM store* and a *DCDO table*. The DFM store contains a set of *DFM descriptors* that define the different versions that the manager represents. A DFM descriptor's structure mirrors a DFM's, but is not used to map function calls to their implementations; instead DCDO managers use DFM descriptors to configure DCDOs appropriately.


Version Management Functions
- VersionId beginVersionConfiguration(VersionId)
- int abandonVersionConfiguration(VersionId)
- int markVersionInstantiable(VersionId)
- int addImplType(VersionId,ImplType)
- int removeImplType(VersionId,ImplType)
- VersionId setCurrentVersion(VersionId)

Version Configuration Functions
- int mgrIncorporateComponent(ImplComponentId,VersionId)
- int mgrRemoveComponent(ImplComponentId,VersionId)
- int mgrEnableFunction(FunctionId,ImplComponentId,VersionId)
- int mgrDisableFunction(FunctionId,VersionId)

DCDO Evolution Functions
- int evolveDCDO(ObjectId, VersionId)
- int evolveAllDCDOs(VersionId)
- int reportEvolution(ObjectId, VersionId)

Status Reporting Functions
- VersionIdList getVersionList(int)
- VersionId getCurrentVersion()
- getVersionOfDCDO(ObjectId)
- ImplementationType getDCDOImplType(ObjectId)
- ImplementationStatus mgrGetImplStatus(VersionId)

Figure 2: A DCDO Manager's Core Interface

The manager exports functions for deriving new versions from existing ones, and for configuring the new versions, as shown in Figure 2. Each version is *instantiable* or *configurable*. Instantiable versions can be used to create new DCDOs, and to update existing DCDOs. However, an instantiable version's DFM descriptor cannot be changed directly. A configurable DFM descriptor can be evolved and configured, but it cannot be used to create a new DCDO, nor to evolve an existing DCDO, until the version is marked instantiable. Typically, a programmer creates a new configurable version from an existing instantiable one, configures the new version appropriately, and then marks it instantiable.

A DCDO table contains an entry for each DCDO under the manager's control. The table contains the version identifier and the implementation type for each object's current implementation. The manager uses this information to decide when and how to evolve its DCDOs, and exports functions to return this information to other objects.

3.4 New Problems

If the dynamic configurability model consisted of only the mechanisms described above, it would be too flexible and open. In broadening the traditional notion of how objects are built and how their implementations can change, dynamic configurability introduces new problems that are absent from systems whose objects have a more restricted ability to evolve. In traditional systems, the process of building a program is kept separate from the process of running it. Dynamic configurability blurs the line between the two, allowing programs to evolve while they run. Although this approach affords the programmer considerable flexibility, it can also be more dangerous and difficult to manage. The problems that arise when DCDO configuration functions can be arbitrarily interleaved with calls on user-defined dynamic functions are

described below. All of the problems are variations on the same theme–a DCDO may attempt to use a part of the implementation that is no longer available.

The Disappearing Exported Function Problem: Suppose a client obtains a DCDO's interface, and builds an invocation for enabled exported dynamic function F_1 in that interface. Before the invocation arrives at the DCDO, F_1 is disabled and no replacement for F_1 is subsequently enabled. The invocation will fail, even though the client built an invocation that was correct according to the interface it obtained.

The Missing Internal Function Problem: One dynamic function, F_1, contains a call to another function F_2, in the same DCDO, and 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, compilers and linkers can check for missing functions when building an executable, which does not change after the check is made. When functions can be disabled and removed from an implementation, a DCDO can attempt to call a function that no longer exists in its implementation.

The Disappearing Component Problem: The disappearing component problem can happen when a thread is executing in some component *C*, and a configuration operation removes component *C* from the object; the thread then has nowhere to execute.

3.5 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*.

Mandatory and Permanent Functions: Dynamic functions are marked in the DCDO manager's DFM descriptor as *mandatory*, *permanent*, or *fully dynamic*. Some implementation of every mandatory 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 cannot be marked instantiable. Thus, once a DCDO evolves to a version that contains a mandatory function F_m, all DCDOs that reflect that version (and all future versions to which those DCDOs will evolve) will contain some implementation of function F_m. When a dynamic function is marked permanent, the function's implementation freezes. Once a DCDO contains a permanent function F_p implemented in component *C*, component *C*'s implementation of function F_p will be present in all future versions of the object.

Programmers can mark a dynamic function as mandatory (or permanent) within the component's own descriptor, thereby indicating that the function must be mandatory (or permanent) in *any* DCDO into which it is incorporated. If this constraint cannot be met, 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 (Figure 3).

Mandatory functions help programmers alleviate the missing and disappearing function problems. If function F_m is mandatory in an interface obtained by a client, then that client can be assured that some function will implement F_m for the lifetime of the object, as long as it only evolves to versions derived from the one in which it was marked mandatory. When a dynamic function is permanent, callers can be

assured that the *implementation* of the function will not change for the lifetime of the object. Essentially, marking functions as mandatory or permanent restricts dynamic configurability on a function by function basis, giving callers more indication about the future behavior of the object.

Function Dependencies: Marking dynamic functions as mandatory or permanent has drawbacks. In particular, it requires the programmer to identify the functions that should be so marked. Essentially, this asks the programmer to decide 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. Now, the programmer is left with a mandatory F_2. But the main reason for marking F_2 mandatory no longer applies, and the object's ability to be dynamically configured has been restricted–the object cannot evolve to a version that does not contain an implementation of function F_2.

To better deal with missing internal function problems, programmers can indicate that certain dynamic functions "depend on" other functions in an interface or an implementation. Programmers may indicate *structural* dependencies and *behavioral* dependencies. If dynamic function F_1 calls 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.

Mandatory and Permanent Functions
- int markFunctionMandatory(FunctionId, VersionId)
- int markFunctionPermanent(
  FunctionId,ImplementationComponentId, VersionId)

Function Dependency Functions
- int addFunctionDependency(VersionId, ImplementationType,
  FunctionId, ImplementationComponentId,
  FunctionId, ImplementationComponentId)
- int removeFunctionDependency(VersionId, ImplementationType,
  FunctionId, ImplementationComponentId,
  FunctionId, ImplementationComponentId)

Figure 3: Setting Function Status and Dependencies.

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 if F_1 is enabled, the particular implementation of F_2 in component C must remain enabled. Behavioral dependencies are designed to shield functions from the behavioral effects of changing the implementation of other functions. 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 returns the larger of the 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 certainly alter sort()'s output. The provider of sort() may want to ensure that this doesn't happen; 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().

Function dependencies allow DCDO builders to restrict the way that DCDOs can be dynamically configured, based on the characteristics of the 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 man-

datory or permanent functions, dependencies can evolve along with the implementation, so a dynamic function's "mandatory" or "permanent" status can be 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. Behavioral dependencies, on the other hand, are semantic information that cannot be deduced by a compiler.

Thread Activity Monitoring: The approaches described above don't address the disappearing component problem. To deal with this problem, a DCDO can monitor its threads, keeping track of the function implementations that have threads inside them. This can be done because all calls to dynamic functions go through the DFM. Thus, the DFM can maintain a counter for each function. When a request to remove a component arrives, the DCDO can make sure all the component's functions 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 could return an error, it could delay handling the request until all thread counts go to zero, or it could simply go ahead with the operation after some time-out period that gives the thread a chance to complete.

Active thread counts can be used in concert with function dependencies to avoid missing 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. Furthermore, by indicating that a function depends on itself, programmers can ensure that recursive functions do not change or disappear while they execute.

4. Evolution Management

The dynamic configurability mechanism must be used in concert with organized evolution policies so that programmers can implement and control object evolution. Evolution management policies define the types of legal changes, 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.

4.1 Single-Version DCDO Managers

A single-version DCDO manager defines exactly one official version at any given moment in time. New DCDOs 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*. Programmers can still create new version descriptions and mark them as instantiable, but DCDOs only evolve to the one current version maintained by the manager.

Since many DCDOs may be under the control of a single manager, and since they are distributed across the system, DCDO updates cannot be instantaneous. Therefore, several strategies update existing DCDOs to the current version differently. In the *proactive update* policy, the manager incorporates changes into DCDOs as soon as a new current version is set. A DCDO can be out of date only as long as it takes for the manager to propagate the new DFM descriptor, and for the DCDO to apply the descriptor. However, the strategy could incur unnecessary overhead if another change is forthcoming before any calls on the DCDO

are made. Further, this strategy does not scale well with the number of DCDOs. For object types with few DCDOs that change relatively infrequently, this policy may be appropriate.

In the *explicit update* policy, the DCDO manager relies on other objects to evolve the DCDOs to the current version. This 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 DCDO must be brought up to date with the current version. Other variations allow a DCDO to check for updates once every *k* member function calls, once every *t* time units, or only when it migrates from one host to another.

4.2 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; again, several different update policies exist. In the *no-update* policy, each DCDO is created with a particular version number, and never evolves to a different version. This limits 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. Therefore, for the no-update policy, the DCDO manager's DCDO evolution functions can be disabled or ignored.

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* but not to version *3.3*. In general, an object type's versions form a tree, and objects can only evolve to versions that are descendants from their current version. This policy works well with the use of mandatory dynamic functions, since clients can be assured that mandatory functions will exist in all future versions of an object. A third option is the *general evolution* policy, in which a DCDO can evolve to any other instantiable version at any time. This undermines the use of mandatory and permanent functions. A hybrid between the increasing version number and general evolution policies checks to see if evolving a DCDO to a new 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 exist. For example, within the increasing version number policy, the explicit update policy could be altered to allow an object to evolve to any instantiable version number eventually derived from the DCDO's current version. 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.

5. Implementation

We have implemented dynamic configurability, as described in this paper, within the latest release (Version 1.4) of the Legion wide-area distributed object computing system. The implementation includes separate libraries for creating DCDOs, DCDO managers (which exist as class objects in Legion), and ICOs. Dynamic configurability objects are fully interoperable with other Legion objects; they are persistent and can be migrated in accordance with the Legion model. The implementation also contains a library that

objects can use to call the functions of DCDOs, DCDO managers, and ICOs using a C++ API. A set of command line utility programs allows programmers to create implementation components from native object code, to create new DCDOs and DCDO managers, and to evolve objects via dynamic configurability.

In the current implementation, the source code and compilation process for implementation components closely resemble those for Legion objects that are programmed to use the Legion run-time library directly. The minor differences include the addition of calls to dynamic configurability library initialization routines, and the linking of the ICO library. The source code is run through a native C++ compiler, and the resulting object code is turned into an ICO by the appropriate command line utility, which requires its user to specify the names of the component's dynamic functions and their corresponding labels within the object code. The tool then registers the component with Legion.

Programmers create Legion DCDO objects similarly–by linking the source code for their objects against the DCDO library, and by adding calls to initialization routines. A DCDO implements the incorporateComponent() function by using the explicit API for managing dynamically loaded object files provided by most modern operating systems. This basic mechanism is used to populate and configure the DFM appropriately. To alter the public interface of DCDOs, dynamic configurability uses the Legion run-time library, which gives programmers access to an object's function dispatcher; when an exported function is enabled, it is registered with the function dispatcher, when it is disabled it is removed.

DCDO managers are built into Legion class objects by linking and initializing the DCDO manager library, which sets up the manager's appropriate data structures, and overloads several Legion's class mandatory functions. A dynamic configurability system initialization phase creates different meta-classes that exhibit the evolution management policies described in Section 4. Programmers can select from among the strategies by creating a new DCDO manager as an instance of the appropriate meta-class. These implementation characteristics keep dynamic configurability compatible with Legion's model.

Initial performance results indicate that the overhead of dynamic configurability is negligible. One level of indirection on dynamic function calls is insignificant when compared to the other costs of remote member function invocation, and only the finest grained functions are affected by the overhead on calls from within the object. Furthermore, because components are smaller than executables, propagating them throughout the system is faster than propogating implementations [22].

6. Summary

Dynamic configurability allows objects to evolve their implementations on the fly, without being deactivated or restarted. The model specifies the roles and basic functionality of DCDOs, their managers, and implementation components. Mechanisms are built into DCDOs to reduce or eliminate new problems that arise from dynamic configurability–objects trying to use functions or components that no longer exist. Different evolution management strategies, which define when and how objects are updated with new versions, are implemented within different DCDO managers to meet object types' varying needs. The model has been fully implemented within Legion, and initial performance results are promising.

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] Armstrong, J., Virding, R. Wikstrom, C., Williams, M., Concurrent Programming in Erlang, Second Edition, Prentice Hall, Englewood Cliffs, NJ 07632.

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

[5] 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.

[6] Bershad, B., Savage, S., Pardyak, P., Sirer, E.G., Becker, D., Fiuczynski, M., Chambers, C., Eggers, S., "Extensibility, Safety and Performance in the SPIN Operating System," *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP-15)*, pp. 267 - 284. October 1997.

[7] 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.

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

[9] 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.

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

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

[12] Engler, D.R., Kaashoek, M.F., O'Toole, J.W., "Exokernel: An operating system architecture for application-level resource management," *Proceedings of the 13th ACM Symposium on Operating System Principles*, October 1995.

[13] 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.

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

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

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

[17] Hutchinson, N.C., Peterson, L.L., "The *x*-kernel: an architecture for implementing network protocols," *IEEE Transactions on Software Engineering*, vol. 17, no. 1, January 1991.

[18] 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.

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

[20] 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.

[21] Lewis, M.J., 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.

[22] Lewis, M.J., Grimshaw, A.S., "Performance of dynamically configurable distributed objects in Legion," UVa Computer Science Technical Report, *in progress*.

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

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

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

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

[27] 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.

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

[29] 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.

[30] 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.

[31] 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.

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

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

[34] 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.

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