Live Upgrades of CORBA Applications Using Object Replication*

L. A. Tewksbury, L. E. Moser, P. M. Melliar-Smith

Department of Electrical and Computer Engineering
University of California, Santa Barbara, CA 93106

tewks@alpha.ece.ucsb.edu, moser@ece.ucsb.edu, pmms@ece.ucsb.edu

Abstract

In a distributed system, software modification to correct programmer errors and to enhance functionality is often necessary, but incurring the downtime required to perform software upgrades can be prohibitively expensive or logistically infeasible. The Eternal Evolution Manager addresses this problem by enabling CORBA applications to be upgraded while they continue to provide service. The off-line analysis in preparation for the live upgrade is largely automatic, and the upgrade itself is fully automatic. The Eternal Evolution Manager insulates the application programmer from the difficult problems inherent to software evolution.

1 Introduction

The world is becoming increasingly reliant on the correct and continuous operation of its computers, and distributed systems offer the potential for greater reliability and availability. Critical computer systems (such as those in hospitals and air traffic control towers) must operate continuously, without interruption of service to ensure the safety of their users. International enterprise operations cannot incur downtime without prohibitive loss of revenue and reputation.

The computers being used for these distributed systems provide faster processing and have more memory than their predecessors, and the software that is being developed to run on them is more complex. Inevitably, coding errors are present in the software. Despite forward planning in the design phase, unanticipated code modifications will be required to improve both the reliability and the functionality of the software. Typically, distributed systems incur disruptive planned downtime to allow for software upgrades. *Live upgrades* of the software eliminate downtime by allowing program code to be modified while the system continues to provide service.

*This research has been supported by the DARPA/ONR Contract N00174-95-K-0083, DARPA/AFOSR Contract F3602-97-1-0248 and MURI/AFOSR Contract F49620-00-1-0330.

1.1 The Common Object Request Broker Architecture

The Common Object Request Broker Architecture (CORBA) [13], established by the Object Management Group (OMG), is a standard for distributed object computing that allows the interoperation of distributed object programs, regardless of the byte order, platform, operating system or programming language that they are using. Within the CORBA architecture, the Object Request Broker (ORB) routes invocations and responses between client and server objects typically located on different computers within the distributed system. Clients statically or dynamically invoke the methods of a server server that are defined by its interface definition language (IDL) interface. Static invocation requires that the client has compile-time knowledge of a particular server's interface.

CORBA's Dynamic Invocation Interface (DII) allows client objects to invoke server objects without compile-time knowledge of a server's IDL interface. A client "learns" about a server's interface through the Interface Repository, and constructs a method invocation to that server by filling in the proper parameters for the invocation. It might appear that the DII is a plausible way to achieve live upgrades: when a server is upgraded, its entry in the Interface Repository is changed. The fundamental problem is how an unchanged client object can invoke a new method on an upgraded server object and know which values to provide for the parameters. The answer frequently involves manual resolution or intervention by a human

who must be present to provide the proper parameters for the method invocation. CORBA programs should evolve transparently, thus, the DII is not a solution to the live upgrade problem. Consequently, the evolution framework described in this paper uses CORBA's Static Invocation Interface.

1.2 The Eternal System

The Eternal system [11, 12] enhances CORBA applications with transparent fault tolerance and live upgrades. Recognizing the lack of a CORBA framework to perform live upgrades, the OMG has recently released a Request for Information concerning online upgrades [14].

In the Eternal system, the client and server objects of the CORBA application are replicated, and the replicas are distributed across the system. The replicas comprise a single logical object group, and, assuming active replication, invocations received by the object group are received and responded to by all of the individual replicas of the object. Duplicate invocations and responses from the replicas of an object are detected and suppressed to preserve strong replica consistency. Replica consistency is also facilitated by ensuring that all object replicas reliably receive messages in the same total order. The replication mechanisms provide strong replica consistency with low overheads, and without requiring the modification of either the application or the CORBA ORB.

The structure of the Eternal system is shown in Figure 1. The Eternal system conveys the IIOP messages of the CORBA application using the reliable totally-ordered multicast messages of the underlying Totem system [10]. The Interceptor captures the IIOP messages (containing the client's requests and the server's replies), which are intended by the ORB for TCP/IP, and diverts them instead to the Replication Mechanisms for multicasting via Totem. The Replication Mechanisms, together with the Logging-Recovery Mechanisms, maintain strong consistency of the replicas, and provide recovery from faults. The Evolution Manager exploits object replication to support live upgrades of CORBA application objects. [17, 18] The Evolution Manager has been developed on Solaris 2.x Sun workstations running Inprise's Visibroker 3.x, Vertel's e*ORB 2.1 and Iona's Orbix 2.x ORBs, for C++ application code.

2 Related Work

Kramer and Magee [9] state that evolution is "dynamic" if it does not stop or disturb those parts of the system that are unaffected by the change. In other words, for an upgrade to be dynamic, only those parts of the system **not** being upgraded must be unaffected. They passivate all of the objects being upgraded, and all of the objects that can invoke these objects. In systems permitting nested operations, this set includes all nodes that can initiate a nested operation that involves the node to be upgraded. In a highly connected system (the authors explicitly state that their technique was designed for systems with loosely-connected components), this method could significantly impact performance. Instead of passivating the entire object, Bidan et al [1] only passivates the communication link between any object and the object being upgraded. The current implementation of the Evolution Manager requires the quiescence of all of the components being upgraded only during the brief atomic switchover. Our technique significantly reduces the quiescence requirements on the application being upgraded.

Hauptmann and Wasel [6] break up multiple component upgrades into a sequence of independent upgrades, but do not indicate how this is accomplished while maintaining program consistency. In contrast, Feiler and Li [3] maintain that substantial off-line analysis is required to determine which replacements can actually be upgraded independently. Senivongse [19] uses off-line analysis to generate mapping operators that mask the evolution of servers from their clients. These mapping operators transform old invocations into new invocations, and thus resemble our wrapper functions, although our wrapper functions exist temporarily in the application and are only used as a optimization to our primary, wrapperless live upgrade technique.

Podus [15] loads new code into memory and changes the binding of procedures to perform live upgrades. Signature changes are facilitated by interprocedures and mapper procedures that convert between two versions of state. The drawback of this method, and similar methods, is that the mechanisms used to achieve the dynamic upgrade are complicated and require specialized compilers and linkers.

Many evolution techniques developed by other researchers place stringent limitations on the types of applications that can be upgraded, and the types of changes that can be made; we have addressed many of these limitations in the design of the Evolution Manager. Some do not allow method signature changes [5]; many require that the new component must be a superclass of the old component [5, 20]. Hursch and Seiter [8] have developed a framework (using an experimental version of Schema, S++) that automatically preserves a system's consistency while it undergoes schema evolution, as long as the upgrades preserve information. It

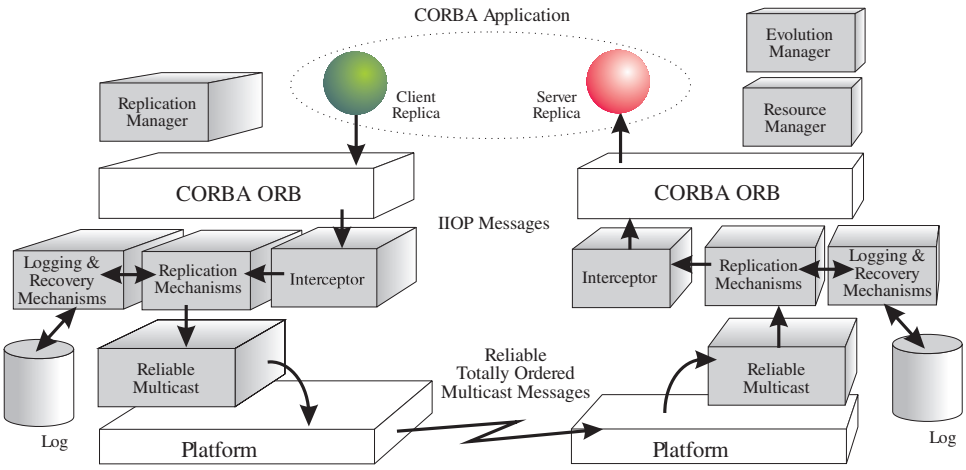

Figure 1. The Eternal System.

handles the situation in which the programmer wishes to rename a class, or add an abstract superclass to existing code, while maintaining the consistency of other program components. The Argus system [2] supports live upgrades of applications written in the Argus language, but the types of upgrades it can perform are constrained by a lack of programming language support for subtyping.

Like Eternal, Polylith [7] deals with evolution, load balancing and fault tolerance but, to achieve an upgrade using Polylith, the application programmer must provide substantial assistance by explicitly identifying reconfiguration points.

The Simplex architecture [4] executes multiple versions of the program simultaneously and shields the old version of the code from the new version until it has been established that the new version works properly. In this way, correct program behavior is maintained despite upgrading to an incorrect new version of the code. Despite the limitations on the types of upgrades Simplex can handle, their error-checking is a feature that we hope to incorporate into Eternal.

3 The Eternal Evolution Manager

We have designed the Evolution Manager so that the behavior of an application is negligibly impacted when no upgrades are taking place, and only minimally impacted while an upgrade is underway. The steps that must be performed during an upgrade are complicated, and would be tedious and error-prone for the human. We automate most of these steps, and for those steps that require the interaction of the application programmer, we provide a graphical user interface to aid the programmer, and to minimize the possibility of error.

To make our live upgrade approach applicable to a wide range of application programs, we operate within the CORBA standard and use the C++ programming language (although the implementation is generalizable to other distributed programming environments, such as DCOM and Java RMI based infrastructures, and other programming languages.) The application programmer is not constrained to maintain aspects of the old code (such as the old IDL interfaces) when writing a new version of the code. We aim to handle all *reasonable* code modifications, though substantial changes to a program might require several incremental steps. Reasonable code modifications include those to an object's interface, method implementations, and instance variables, etc. An example of an unreasonable code modification would be one that changes the inside of an executing infinite event loop. With our current implementation, we would not be able to perform a live upgrade on such an object because it will never become quiescent. Additionally, replacing the old version of the code with a new and completely unrelated version would be unreasonable as there would be no way to transfer the state between such "versions".

The Evolution Manager is responsible for upgrading CORBA applications in a way that does not require the application to stop operating. The components of the Evolution Manager are the Preparer and the Upgrader, shown in Figure 2. Performing a live upgrade requires the completion of the following three steps:

1. The application programmer writes a modified version P" of the original code P.

2. The Preparer, with programmer assistance, performs off-line analysis of the two code versions.

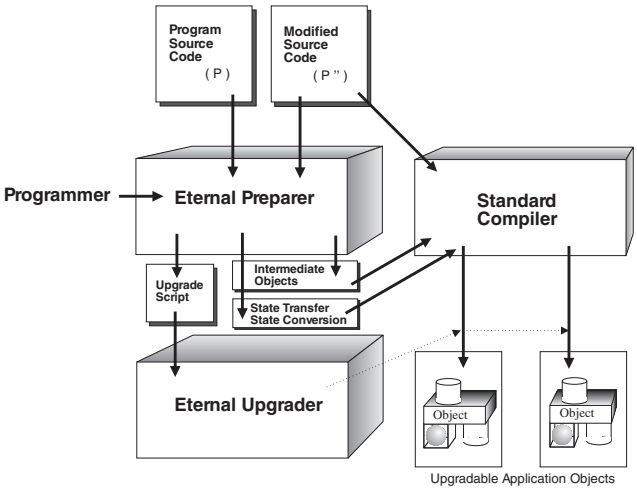

Figure 2. The Eternal Evolution Manager.

3. The Upgrader performs a fully-automatic upgrade while the application continues to provide service.

The Evolution Manager does not concern itself with the first portion of the upgrade task and the modified source code is assumed to implement correctly the intended program modifications. The Evolution Manager handles the second and third steps.

3.1 The Eternal Preparer

The Preparer performs offline analysis, comparing P and P" in preparation for the live upgrade. This analysis includes the automatic generation of state transfer code and, with input from the application programmer, generation of state conversion code [16]. The intermediate code P', a superset of P and P", is also generated automatically, and is run during the transition between the original version P of the code and the new version P". State conversion code is necessary because the structural representation of the state of the old version of the code and the structural representation of the state of the new version may differ.

The intermediate code contains new objects with new interfaces coexisting with old objects and their old interfaces which both exist together inside of intermediate objects. The intermediate objects behave as either the old or the new objects, depending on whether the object has been *switched over*. By encapsulating the old and new functionality within a single object, switching between the executing code versions is expedited.

Additionally, the Preparer analyzes the differences between the two code versions to determine the steps that the Upgrader needs to perform to complete the upgrade. Does the upgrade require an interface change, or are the changes encapsulated in the method implementations? If an interface changes, the Preparer must deduce which methods are new, which have been omitted in the new version, and whether a new method is meant to replace an old one. The upgrade of a single object that involves an interface change might involve a ripple effect of upgrades to other objects in the program. All of the code that invokes an upgraded method must be upgraded simultaneously to provide the correct parameters for the new version of the method invocation. The Preparer uses a CORBA method invocation graph to determine the propagation effects of an upgrade of one object and identifies the nested invocations (in which an invocation on one object results in further object invocations) in the system; this information is required to determine when an object is quiescent. As we shall explain later, an object cannot be upgraded unless it is quiescent.

The Preparer also generates a reverse sequence of changes that can be employed if, for any reason, it is necessary to undo an upgrade. Undoing an upgrade would be necessary if the new version of the code performed inadequately. In the future, we will incorporate testing of the new code into the upgrade process.

3.2 The Eternal Upgrader

After the offline analysis is completed, and while the application continues to provide service, the Upgrader performs an automatic upgrade from P to P". The general strategy of our approach is that any object being upgraded is replicated. If live upgrades are required but fault tolerance is not, the objects are replicated only while they are being upgraded. Because the Eternal infrastructure already replicates application objects to achieve fault tolerance, and because an application that requires no downtime from an upgrade will most likely require no downtime due to faults, replication for live upgrades incurs no additional overhead beyond that required for fault tolerance.

We achieve upgrades without interrupting the executing application by performing a series of *individual replacements* which "nudge" the application towards an upgraded state but which do not affect the behavior of the program. The replicas being upgraded are replaced, one at a time,[1] by their intermediate versions, which continue to execute the old methods. Once all of these intermediate versions are in place, we effect an *atomic switchover* after which only the new methods are invoked. Then, to clean up the program, we re-

[1] The replicas are replaced individually because we preserve the fault tolerance of the system during the upgrade.

place the intermediate objects one at a time with final versions containing only the new version of the code, while executing the new methods.

4 Interface-Preserving Upgrades

The upgrade process is simplest when the new version of an object has an interface that is syntactically and semantically identical to the interface of the old version. In this situation, the object being upgraded can be considered separately from the other application objects and the change can be isolated from the rest of the program.

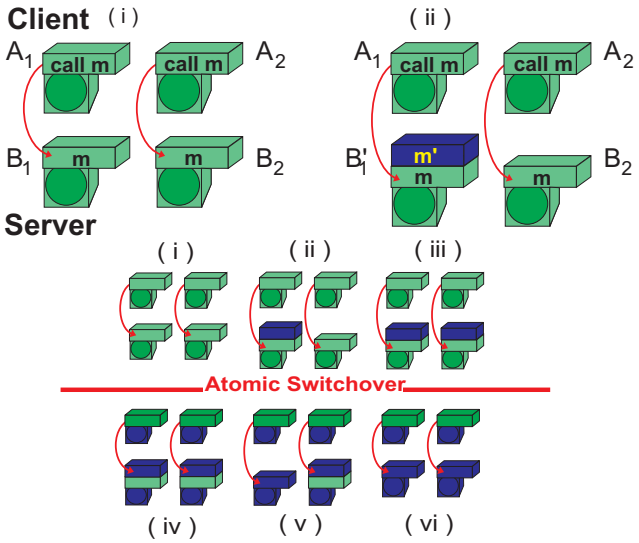

Figure 3. Upgrade without an interface change.

Consider Figure 3 in which the server object *B* is being upgraded to server object *B"*. The original server contains a method *B.m()*, which is replaced with a new method *B.m'()*. In this example, the new and the old methods have the same names (so that the client *A* does not need to be upgraded). For clarity of exposition, we refer to the new method *B.m()* as *B.m'()*.

- Step *(i)* shows the initial configuration.
- Step *(ii)* shows the first server replica *B1* being replaced by an intermediate server replica *B1'* that contains both the *B.m()* and *B.m'()* methods, while *B2* continues to provide uninterrupted "old" service. State is transferred from *B2* to *B1'*.
- Step *(iii)* shows the second server replica *B2* being replaced by an intermediate server replica *B2'* that contains both the *B.m()* and *B.m'()* methods, while *B1'* continues to provide uninterrupted "old" service. State is transferred from *B1'* to *B2'*.
- We now perform an atomic switchover and convert from the old state of *B'* to the new state of *B'*. Step *(iv)* shows that all invocations by the clients are now routed to the new *B.m'()* methods.
- Step *(v)* shows the intermediate server replica *B1'* being replaced by a final server replica *B1"* that contains only the new *B.m'()* method, while *B2'* continues to provide uninterrupted "new" service. State is transferred from *B2'* to *B1"*.
- Step *(vi)* shows the intermediate server replica *B2'* being replaced by a final server replica *B2"* that contains only the new *B.m'()* method, while *B1"* continues to provide uninterrupted "new" service. State is transferred from *B1"* to *B2"*.

Because of the underlying reliable totally-ordered multicasts, the atomic switchover messages are received by all of the *B'* replicas at the same logical delivery time and, thus, the state of the server replicas remains consistent.

5 Coordinated Upgrades

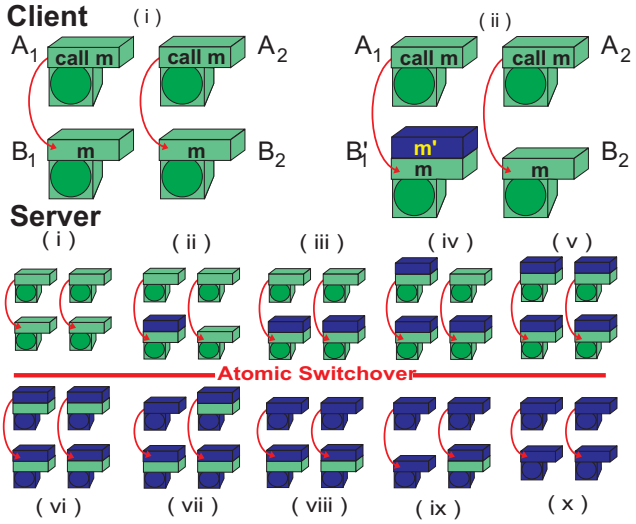

Figure 4. Upgrade with an interface change (a coordinated upgrade).

When an object's interface changes as the result of an upgrade, the situation becomes more complex, as shown in Figure 4 . Suppose that the upgrade of *B* to

B" involves a signature change in method *B.m()*. If we were to upgrade only *B* before upgrading the client objects that can invoke it, any invocations of *B.m()* after *B'* is switched over would produce an error.

We considered two potential solutions to this problem. The first is to use wrapper functions to translate the invocation and response types of the old method into types appropriate for the new method. The second is to allow objects to exist temporarily with both interfaces and to use context information to determine which interface should be invoked. Because there are many method signature changes that could not be accommodated by a simple transformation, we chose the second alternative. However, using wrapper functions to translate between old and new signatures can improve performance.

For the general case, we use *coordinated upgrade sets*, groups of objects that are evolved "together", to upgrade multiple objects at the same time. The mechanics of a coordinated upgrade strongly resemble those used in an interface-preserving upgrade.

Consider Figure 4 in which the server object *B* is being upgraded. The original server contains a method *B.m()* which is replaced with a new method *B.m'()*, where *B.m()* and *B.m'()* have different signatures. It should be noted that this example has been simplified for the sake of clarity, and is technically incorrect. IDL does not allow overloading of method names, so the actual upgrade requires two intermediate versions of the *B*.

- Step *(i)* shows the initial configuration.
- Step *(ii)* shows the first server replica *B1* being replaced by an intermediate server replica *B1'* that contains both the *B.m()* and *B.m'()* methods, while B2 continues to provide "old" uninterrupted service. State is transferred from *B2* to *B1'*.
- Step *(iii)* shows the second server replica *B2* being replaced by an intermediate server replica *B2'* that contains both the *B.m()* and *B.m'()* methods, while *B1'* continues to provide "old" uninterrupted service. State is transferred from *B1'* to *B2'*.
- Step *(iv)* shows the first client replica *A1* being replaced by an intermediate client replica *A1'* that contains code to invoke both the *B.m()* and *B.m'()* methods. A2 continues to invoke *B.m()* on the server replicas and state is transferred from *A2* to *A1'*.
- Step *(v)* shows the second client replica *A2* being replaced by an intermediate client replica *A2'* that contains code to invoke both the *B.m()* and *B.m'()* methods. *A1'* continues to invoke *B.m()* on the server replicas and state is transferred from *A1'* to *A2'*.
- We now perform an atomic switchover and convert the state of *B* to *B'* and the state of *A* to *A'*. Step *(vi)* shows that the clients now invoke the new *B.m'()* methods.
- Step *(vii)* shows the intermediate client replica *A1'* being replaced by a final client replica *A1"* that contains only the code to invoke the new *B.m'()* method. *A2'* continues to invoke *B.m'()* on the server replicas and state is transferred from *A2'* to *A1"*.
- Step *(viii)* shows the intermediate client replica *A2'* being replaced by a final client replica *A2"* that contains only the code to invoke the new *B.m'()* method. *A2"* continues to invoke *B'm()* on the server replicas and state is transferred from *A1"* to *A2"*.
- Step *(ix)* shows the intermediate server replica *B1'* being replaced by a final server replica *B1"* that contains only the new *B.m'()* method, while *B2'* continues to provide "new" uninterrupted service. State is transferred from *B2'* to *B1"*.
- Step *(x)* shows the intermediate server replica *B2'* being replaced by a final server replica *B2"* that contains only the new *B.m'()* method, while *B1"* continues to provide "new" uninterrupted service. State is transferred from *B1"* to *B2"*.

6 The Need for Object Quiescence

A *quiescent* object is one that is not executing any of its methods. For example, consider a method *A.m()* that invokes a method *B.n()*. If *B.n()* has been invoked but has not completed, then *B* is not quiescent, because it is executing, and *A* is not quiescent because the invocation that it initiated on *B* is yet to complete and its progress has been stalled waiting for a response from *B*. Note that the invocations being discussed here are synchronous. We cannot ensure the quiescence of an object which contains one-way, or asynchronous, operations unless the completion of a one-way operation is signaled somewhere to the infrastructure.

The issue of quiescence is important during two different periods in the upgrade of an application. First, an object must be quiescent when it undergoes an invisible replacement so that the states of the replicas remain consistent after the state transfer. To see why this condition is necessary, consider extracting the state of

Replica A1	Replica A2	Replica A3
aMethod() {	aMethod() {	aMethod() {
a++; ⇐	a++;	a++;
b = a + b;	b = a + b; ⇐	b = a + b;
c = a*2+b;	c = a*2+b;	c = a*2+b; ⇐
...	...	...
}	}	}

Figure 5. Transferring state without waiting for quiesence yields indeterminate results.

three nonquiescent replicas *A1*, *A2* and *A3* of the same object while they are executing a method, *aMethod()*. Replicas execute at their own pace, so it is not possible to determine which line of *aMethod()* a particular replica is currently executing. For the sake of exposition, let us assume that the state transfer occurs when the replicas have executed the lines of code indicated by the arrows in Figure 5.

Replica *A1* has updated variable `a`, replica *A2* has updated variables `a` and `b`, and replica *A3* has updated variables `a`, `b` and `c`. Because the underlying Eternal infrastructure provides duplicate message detection and suppression, only one of the replicas' *get_state()* responses will return, and it is unpredictable which replica's *get_state()* response will get through. Thus, there is no way to return a deterministic set of values for the state of a nonquiescent replica. Similarly, calling *set_state()* on non-quiescent replicas can cause replica inconsistency.

Quiescence is also important during the atomic switchover. In general, all members of a coordinated upgrade set must be quiescent when the switchover occurs. Although the Eternal infrastructure ensures the quiescence of individual replicas to which they deliver messages, coordinating the quiescence of multiple objects on different hosts is a non-trivial task. We have designed several coordinated quiescence algorithms and have tested them by upgrading simple applications with small coordinated upgrade sets. A three-tier application running full-throttle experiences a 15% throughput degradation during a live upgrade.[2] It is much more difficult to quiesce the coordinated upgrade set of a large application undergoing a complicated upgrade. A large coordinated upgrade set might not be quiescent very often during an application's execution. Furthermore, imposing quiescence on the entire coordinated upgrade set can impact the performance of the system. Fortunately, the quiescence requirement can be loosened, as discussed below.

[2]Detailed information about coordinating the quiescence of multiple objects can be found in [17].

6.1 Wrapper Functions

The quiescence requirements of the switchover can be relaxed by utilizing wrapper functions. A wrapper function transforms a method of an old code version into a format acceptable to the corresponding method of the new code version.

Original Method	New Method	Wrapper Possible?
aMethod(int i, int r)	aMethod(int i)	probably
aMethod(float i)	aMethod(int i)	probably
aMethod()	aMethod(int i)	questionable

Figure 6. Wrapper function feasibility.

If a wrapper function can be written for each of the changed methods of an object and incorporated into the automatically generated intermediate code, the object can be switched over after the switchover of the rest of the objects of the coordinated upgrade set. Wrapper functions must handle syntactic differences between the methods (parameter and return types) as well as semantic differences and, thus, their synthesis requires significant programmer assistance. Some upgrades that involve more substantial changes to methods might not be suitable as wrapper functions. The following examples illustrate the feasibility of the wrapper function approach for different types of method upgrades.

The first entry in Figure 6 is a method upgrade that drops the parameter `r` in the new version of the method. After receiving verification from the user that the two `i` variables are equivalent, the translation between the two method versions is achieved by not passing the second parameter to the new method. The wrapper function that is automatically generated to handle this translation is shown in Figure 7.

```
class InterObj {
 void aMethod(int i, int r) {
   if (switchFlag == 0)
    /* old aMethod implementation */
   else if (switchFlag == 1)
    aMethod(i);
 }
 void aMethod(int i) {
    /* new aMethod implementation */
 }
}
```

Figure 7. An upgrade from *aMethod(int i, int r)* to *aMethod(int i)* can be masked by a wrapper function located in an intermediate object.

The method upgrade shown in the second row of Figure 6 involves a method parameter type change from `float` to `int`. It is simple to convert from a variable of type float to a variable of type integer, and, with input from the application programmer, translation between less obvious types changes is tractable.

It is much more difficult to write a wrapper function for a method upgrade that augments the new method with additional information. The Preparer cannot automatically generate wrapper code for the third method upgrade shown in Figure 6 to initialize `i`, although the application programmer might know how to generate an appropriate value for `i`, and could then provide an initialization routine for it within a wrapper function.

By utilizing wrapper functions whenever possible, we can increase the frequency with which coordinated upgrade sets can be switched over. Objects with a full set of wrapper functions can be handled after the main switchover occurs.

7 Evolution Mechanisms

7.1 Upgrade Code Generation

original Count class	new Count class

class Count {
private:
  int countVal;
  int countDir;
public:
  void updateCount() {
  if (countDir == 1)
  countVal++;
  else if (countDir == -1)
  countVal--;
  }
  void changeDir(**int** dir) {
  countDir = dir;
  }
  void set_state(State* s) {
  /* *for Count* */
  }
  State* get_state() {
  /* *for Count* */
  }
}
	class Count_new {
	private:
	  int countVal;
	  int countDir;
	public:
	  void updateCount(**int** i) {
	  if (countDir == 1)
	  countVal += i;
	  else if (countDir == -1)
	  countVal -= i;
	  }
	  void changeDir(**int** dir) {
	  countDir = dir;
	  }
	  void set_state(State *s) {
	  /* *for Count_new* */
	  }
	  State* get_state() {
	  /* *for Count_new* */
	  }
	}

Figure 8. The old and new versions of the `Count` class.

Cconsider a class `Count` that increments or decrements a variable `countVar` when *updateCount()* is invoked.

```
class Count_inter {
private:
    unsigned short switchFlag; /* init to 0 */
    int countVal_old, countDir_old;
    int countVal_new, countDir_new;
public:
    void updateCount() {
        if (switchFlag == 0) {
            if (countDir_old == 1) countVal_old++;
            else if (countDir_old == -1) countVal_old--;
        }
    }
    void updateCount(int inc) {
        if (switchFlag == 1) {
            if (countDir_new == 1) countVal_new += inc;
            else if (countDir_new == -1)
              countVal_new -= inc;
        }
    }
    void changeDir(int dir) {
        if (switchFlag == 0) countDir_old = dir;
        else countDir_new = dir;
    }
    void switchover() {
        switchFlag = 1;
        /* No state conversion is needed in this example */
        countVal_new = countVal_old;
        countDir_new = countDir_old;
    }
    void set_state(State *state) {
        if (switchFlag == 0) /* Count set_state() code */
        else /* Count_new set_state() code */
    }
    State *get_state() {
        if (switchFlag == 0) /* Count get_state() code */
        else /* Count_new get_state() code */
    }
}
```

Figure 9. Automatically generated intermediate code for an upgrade of the `Count` object.

The new version, `Count_new`, of this class increases or decreases `countVar` by an amount specified in the parameter of the new *updateCount(int inc)* function. Both classes are shown in Figure 8. The Preparer parses these two class definitions and automatically generates the intermediate class `Count_inter`, shown in Figure 9. `Count_inter` contains a member variable `switchFlag` that indicates whether the switchover has occurred and that determines whether incoming messages should be "handled" by the old or the new versions of the code. The methods defined in the intermediate object are a superset of those defined in the old and the new versions of the object. Within the intermediate object, the state of the old version is kept distinct from the state of the new version by appending the state variables with either the *_old* or the *_new* suffix.

7.2 State Transfer and Conversion

The Evolution Manager transfers state during each of the invisible replacements, as well as during the atomic switchover. In the current implementation, the invisible replacement entails the following steps:

1. Get the state of the object group.
2. Kill the replica that is being upgraded.
3. Start up the new version of the replica.
4. Transfer state to this new replica.
5. Deliver to the new replica any messages that were queued while it received the current state.

The state transfer process might be time-consuming if the state is large. While state is being transferred to a replica, the replica's messages are queued, and its progress is stalled. This delay is not a huge concern during the invisible replacements. Only one replica is being delayed at a time, so the other replicas can continue to provide service. The situation is different during an atomic switchover, because all of the replicas of the objects in the coordinated upgrade set are simultaneously delayed by the transfer of state, which could result in a substantial performance lag.

However, an optimization is used for state transfer during an atomic switchover, because only intermediate versions of the code, containing both the old and the new objects, are switched over. Instead of performing the lengthy process of encoding the state of the old object, transferring it to the new object, and then decoding the state (which is the state transfer technique used during invisible replacements), the state is transferred more directly. Within the *switchover()* method, the old state variables are assigned directly to the new state variables, as shown in Figure 9. When state conversion is needed, the direct assignment is replaced by a conversion routine.

8 Conclusions and Future Work

We have presented a system, the Eternal Evolution Manager, that enables live upgrades of CORBA applications in a distrubuted object system. Our evolution algorithms allow an upgrade to be performed while the system continues to provide service by exploiting object replication and a reliable totally-ordered multicast protocol. The aim is to automate as much of the process as possible both to reduce programmer errors and to encourage application upgrades that might not have been possible using more complicated schemes. In addition to the infrastructure that automatically handles the precise sequence of operations necessary to perform an upgrade, we have developed a substantial amount of "helper" code, which automatically generates state transfer and state conversion code for the application.

Once we begin upgrading complicated applications with large coordinated upgrade sets, quiescence will likely become the system's biggest performance bottleneck. There might exist upgrades whose coordinated upgrade sets rarely enter an upgradable state. We might need to preempt executing replicas (without sacrificing replica consistency) to perform the live upgrade. Another possible technique allows an incremental switchover of the coordinated upgrade set, obtaining information about how different program components can work together from the application programmer. We also plan to investigate programming styles and techniques that minimize the amount of time that the Upgrader must wait for a coordinated upgrade set to become quiescent.

References

[1] C. Bidan, V. Issarny, T. Saridakisand A. Zarras, "A dynamic reconfiguration service for CORBA," *Proceedings of the 4th International Conference on Configurable Distributed Systems*, Annapolis, MD (May 1998), pp. 35-42.

[2] T. Bloom and M. Day, "Reconfiguration and module replacement in Argus: theory and practice," *Software Engineering Journal* (March, 1993), pp. 102-108.

[3] P. Feiler and J. Li, "Consistency in dynamic reconfiguration," *Proceedings of the 4th International Conference on Configurable Distributed Systems*, Annapolis, MD (May 1998), pp. 189-196.

[4] M. Gagliardi, R. Rajkumar and L. Sha, "Designing for evolvability: Building blocks for evolvable real-time systems," *Proceedings of the IEEE 1996 Real-Time Technology and Applications Symposium*, Brookline, MA (June 1996), pp 100-109.

[5] D. Gupta and P. Jalote, "On-line software version change using state transfer between processes," *Software - Practice and Experience* (September 1993), pp. 949-964.

[6] S. Hauptmann and J. Wasel, "On-line maintenance with on-the-fly software replacement," *Proceedings of the 3rd International Conference on*

Configurable Distributed Systems, Annapolis, MD (May 1998), pp. 70-80.

[7] C. Hofmeister and J. Purtilo, "Dynamic reconfiguration in distributed systems: adapting software modules for replacement," *Proceedings of the IEEE 13th International Conference on Distributed Computing Systems,* Pittsburg, PA (May 1993), pp. 101-110.

[8] W. L. Hursch and L. M. Seiter, "Automating the evolution of object-oriented systems," Proceedings of the International Symposium on Object Technologies for Advanced Software, Kanazawa, Japan (March 1996), pp. 2-21.

[9] J. Kramer and J. Magee, "The evolving philosophers problem: Dynamic change management," *IEEE Transactions On Software Engineering*, vol. 16, no. 11 (November 1990), pp. 1293-1306.

[10] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia and C. A. Lingley-Papadopoulos , "Totem: A fault-tolerant multicast group communication system" *Communications of the ACM*, vol. 39 , no. 4 (1996), pp. 54-63.

[11] L. E. Moser, P. M. Melliar-Smith and P. Narasimhan, "Consistent object replication in the Eternal system," *Theory and Practice of Object Systems* vol. 4, no. 2, (1998), pp. 81-92.

[12] P. Narasimhan, L. E. Moser and P. M. Melliar-Smith, "Replica consistency of CORBA objects in partitionable distributed systems," *Distributed Systems Engineering Journal*, vol. 4, no. 3 (September 1997), pp 139-150.

[13] Object Management Group, "The Common Object Request Broker: Architecture and Specification," Revision 2.3.1, OMG Technical Document Formal/99-10-07, Object Management Group (October 1999).

[14] Object Management Group, "Online Upgrades Request for Information," OMG Technical Document realtime/2000-08-01 (August 2000).

[15] M. Segal and O. Frieder, "On-the-fly program modification: Systems for dynamic updating," *IEEE Software* (March 1993), pp. 53-65.

[16] L. A. Tewksbury, L. E. Moser and P. M. Melliar-Smith, "Automatically generated state transfer and conversion code to facilitate software upgrades," *Proceedings of the Maintenance and Reliability Conference*, Gatlinsburg, TN (May 2001).

[17] L. A. Tewksbury, L. E. Moser and P. M. Melliar-Smith, "Coordinating the simultaneous upgrade of multiple CORBA application objects," to appear in the *Proceedings of the 3rd International Symposium on Distributed Objects and Applications*, Rome, Italy (September 2001).

[18] L. A. Tewksbury, L. E. Moser and P. M. Melliar-Smith, "Live upgrade techniques for CORBA applications," to appear in the *Proceedings of the 3rd International IFIP Working Conference on Distributed Applications and Interoperable Systems*, Krakow, Poland (September 2001).

[19] T. Senivongse, "Enabling flexible cross-version Interoperability for distributed services," *Proceedings of the International Symposium on Distributed Objects and Applications*, Edinburgh, Scotland (1999), pp. 201-210.

[20] I. Warren and I. Sommerville, "A model for dynamic configuration which preserves application integrity," *Proceedings of the 3rd International Conference on Configurable Distributed Systems*, Annapolis, MD (May 1996), pp. 81-88.