Preventing Recursion Deadlock in Concurrent Object-Oriented Systems

by

Eric A. Brewer   Carl A. Waldspurger

Technical Report MIT/LCS/TR-526

January 1992

Abstract

This paper presents solutions to the problem of deadlock due to recursion in concurrent object-oriented programming languages. Two language-independent, system-level mechanisms for solving this problem are proposed: a novel technique using *multi-ported objects*, and a *named-threads* scheme that borrows from previous work in distributed computing. We compare the solutions and present an analysis of their relative merits.

Keywords: deadlock, recursion, object-oriented systems, programming languages, concurrency

This work was supported in part by the National Science Foundation under grant CCR-8716884, by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-89-J-1988, and by an equipment grant from Digital Equipment Corporation. Eric A. Brewer was supported by an Office of Naval Research Fellowship.

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts 02139

1 Introduction

Recursion is a powerful programming technique that allows straightforward expression of many algorithms. Unfortunately, recursion often leads to deadlock in contemporary concurrent object-oriented systems. In many systems [Ame87, Yon87, Man87, Chi90], a method that modifies an object's state cannot even call itself recursively. This occurs because the object (as sender) is blocked waiting for its call to complete, but the called method never executes because the object (as receiver) is blocked. In general, *recursion deadlock* occurs whenever an object is blocked pending a result whose computation requires the invocation of additional methods on that same object.

We present two transparent solutions that allow *general* recursion without deadlock. These solutions are transparent in that programs suffering from recursion deadlock will run correctly without change if either solution is incorporated into the underlying system. The first solution is based on *multi-ported objects*, and uses separate communication ports to identify recursive calls. The second solution, *named threads*, draws on previous work in distributed computing, and generates a unique name for each thread in order to detect recursive calls.

A combination of three factors leads to recursion deadlock. First, an object must hold a lock on some state. In systems with at most one active thread per object [Ame87, Yon87, Man87, Chi90], there is a single implicit lock for the entire object state. Second, the object must make a blocking call to some object (possibly itself), holding the lock while it waits for the response to the call. Finally, the resulting call graph must contain a call back to the locked object that requires access to the locked state, forming a cycle. When these criteria are met, every object involved in the cycle is waiting for a reply, but none of the objects can make any progress. The objects on the cycle are deadlocked.

Aside from simple recursive methods, many patterns of message-passing can lead to recursion deadlock. A common practice in programming sequential object-oriented systems is to use a method as a modular "subroutine" from within another method defined on the same object. An attempt to do this in a concurrent system, however, will result in deadlock. More sophisticated programs that manipulate cyclic data structures, use call-backs while responding to unusual or exceptional conditions, or implement dynamic sharing mechanisms are all candidates for recursion deadlock. Some programming styles are also prone to recursion deadlock. For example, using *inheritance by delegation* [Lie86, Lie87], an object can delegate the handling of a method to another object dynamically. In a delegated call, references to `self` should be resolved to the original object that performed the delegation.[1] Thus, all calls to `self` involve

[1] The term `self` may be somewhat misleading in this context. Whenever a message is delegated, it must

call-backs that could lead to recursion deadlock.

In the next section, we examine existing work related to the recursion deadlock problem. Our basic model of concurrent object-oriented systems is defined in section 3. Section 4 discusses the semantics of recursion without deadlock. In section 5, we outline our general solution framework, and we present the solutions in sections 6 and 7. An illustrative example appears in section 8. Finally, we compare the solutions in section 9, and then present our conclusions.

2 Related Work

A variety of *partial* solutions exist for handling some cases of recursion deadlock. The simplest partial solutions handle only *direct recursion* involving a single object. These amount to releasing the object lock and ensuring that the next method invoked is the recursive call (e.g., by prepending it to the incoming message queue), which will reacquire the lock. This approach is implicit in languages such as Vulcan [Kah87], which ensure that calls to `self` are processed before other incoming messages.

Another solution for direct recursion is to provide *procedures* in addition to methods [Yon87]. Unlike methods, procedures are stateless and need not be associated with an object. Since procedures are stateless, they do not require locks. Using this approach, a method calls a procedure and the procedure handles the recursion. This avoids deadlock because there is no lock acquisition, assuming the procedures never invoke any methods. (Recursion deadlock is possible if the procedures invoke methods or otherwise acquire locks.) Since procedures only have access to the current object's state, multiple-object recursion is not possible.

A partial solution for *fixed-depth* recursion is the use of selective message acceptance constructs [Yon87, Ame87]. For example, ABCL/1 allows calls to be accepted in the body of a method if the object enters a selective "waiting mode". In this case, the recursive call is handled in the body and need not acquire the object lock. An explicit waiting mode must be introduced for each level of recursion; if there are too many recursive calls the system will deadlock.

If recursive calling patterns are completely known in advance, deadlock can be avoided in actor systems by using replacement actors. By cleverly specifying *insensitive actors* that buffer most incoming messages while responding to a few special messages (such as `become`), programmers can write code that explicitly avoids potential deadlocks [Agh86]. Insensitive actors are automatically generated by compilers for actor languages to support the acceptance of

include a reference to the *client* object that originally received the message. The term client is used because the object that is the target of the delegation can be thought of as performing a service for the original object [Lie86].

replies to messages sent by a locked actor.[2] Although actor replacement provides the flexibility required to write deadlock-free code, the complexity of explicitly introducing insensitive actors and behaviors for all possible recursive calling patterns is daunting. In fact, these low-level actor mechanisms could be used to *implement* the solutions we propose without system-level changes. In general, systems that provide linguistic control of message acceptance, such as *enabled-sets* [Tom89] or *protocols* [Bos89], can be used to implement our language-independent solutions.

Techniques for *deadlock detection* from the distributed systems and database literature [Gli80, Sin89] could also be used to address the recursion deadlock problem. Deadlock detection algorithms examine process and resource interactions to find cycles (assumed to be relatively infrequent), and usually operate autonomously, separate from normal system activities. Although such schemes theoretically could be used to detect and recover from deadlocks in a concurrent object-oriented system, they would not be practical for fine-grained programs that use recursion. Nevertheless, the general idea of maintaining a list of blocked-process dependencies is related to the deadlock-prevention techniques that we propose.

Finally, recursion deadlock is closely related to the *nested-monitor problem* [Lis77]. In the nested-monitor problem, monitors correspond to objects with a single lock. Nested-monitor calls correspond to blocking calls made while an object holds its lock. Thus, deadlock occurs if there is a cycle in the call chain.[3] Most work on the nested-monitor problem occurred before the advent of concurrent object-oriented languages. Solutions presented during that period amounted to either releasing the lock across the call or forbidding calls entirely [Had77].

3 Computational Model

We make a few basic assumptions about the underlying computational model. The model we assume encompasses most contemporary concurrent object-oriented systems. *Objects* abstractly encapsulate local state with a set of *methods* that can manipulate that state directly. Objects interact only by sending *messages* to invoke methods at other objects. To avoid ambiguity, we define the following terms:

- **Message:** A message is a request from a sender object to a receiver object to perform an operation. A message causes the invocation of a particular method at the receiver; the

[2] In actor parlance, an actor that has not yet computed its replacement behavior.

[3] The nested-monitor problem is not restricted to cyclic calls. At the time, other forms of deadlock were the primary concern.

arguments for that method are passed in the message. Messages may or may not arrive in the order they were sent, but message delivery is guaranteed.

- **Message Queue:** Each object has a message queue that buffers all incoming messages. An object removes messages from its queue and invokes the corresponding methods. We ignore issues of message priority and queue overflow.

- **Send:** We divide messages into two categories, sends and calls. A send is an asynchronous, non-blocking method invocation. It is unidirectional, and has no corresponding reply. After performing a send, the sender immediately continues execution, and does not wait for a reply or for the completion of the invoked method.

- **Call:** A call is a synchronous, blocking method invocation. After performing a call, the sender waits for a reply. This is analogous to normal procedure call semantics. Any locks held by the sender prior to the call are held until the reply is received. Every call has a matching reply; we assume the underlying system handles the matching of call-reply pairs.[4]

  A set of concurrent calls may also be sent such that each of the calls operates in parallel, and the sender waits for replies from all of the calls before continuing execution. This corresponds to a *fork–join* semantics.

- **Lock:** A lock ensures mutual exclusion for some piece of state. A given object may contain several locks. We assume that locks are the underlying primitive synchronization mechanism for mutual exclusion. Lock acquisition may be implicit or explicit. In actor languages, for example, a single object lock is acquired implicitly upon method invocation and is released explicitly via the `ready` construct, which indicates that the method has finished modifying the actor's state.[5]

  To avoid complication, we will assume that there is a single implicit lock per object that provides mutual exclusion for the entire object state, as in most contemporary languages [Ame87, Yon87, Man87, Chi90]. However, the solutions we present can be easily adapted for languages with more sophisticated locking schemes; we briefly discuss this after presenting the solutions for the single-lock case.

- **Thread:** A thread is a single flow of control that performs a sequential computation. A single thread may execute code at several different objects. For example, if object

[4] We do not preclude explicit handling of reply values, which can be useful for forwarding or delegation.

[5] In actor terms, there is no mutation; the actor computes a new "replacement actor" to process subsequent messages.

A calls object B, the sequential flow of control would initially execute some code at A, then proceed to execute the invoked method at B, and finally continue execution back at A. In a sense, the thread travels with the messages between A and B. This view of threads may differ from the low-level details of the underlying implementation, which might involve two distinct "threads" at A and B. Semantically, however, there is only one thread because the computation is sequential.

A thread can also *fork* several distinct subthreads by performing concurrent calls. In this case, the original thread is suspended until its subthreads, or *children*, all reply and *join* with the original parent thread.[6]

4 Recursive Call Semantics

4.1 Existing Sequential Semantics

In sequential object-oriented systems such as Smalltalk-80 [Gol83] and C++ [Str86], there is only a single thread of control, so mutual-exclusion locks are unnecessary. In these systems, if the call chain generated during a method invocation results in a later invocation on the same object, the recursive call is permitted to modify the object's state. Thus, a recursive call may be used to change the state of an object in sequential object-oriented systems, and there is no deadlock issue.

4.2 Proposed Concurrent Semantics

In concurrent systems, a complete lack of mutual exclusion is not satisfactory because of the need to avoid interference between concurrently executing threads. For example, consider a bank account object A with a current balance of \$100 that is accessed concurrently by two different threads, t_1 and t_2. Without provisions for serialization, t_1 and t_2 could concurrently invoke "`withdraw` \$75" methods on A. If both threads happened to read the current balance before either modified it to reflect a withdrawal, the account would have a balance of \$25 instead of being overdrawn. In order to serialize the `withdraw` methods, a mutual-exclusion lock could be associated with the account balance to ensure that each method appears to execute atomically. However, the addition of this lock makes recursion deadlock possible.

Ideally, we would like to preserve atomicity while eliminating the potential for recursion deadlock. Our proposed semantics for concurrent systems is consistent with the "expected"

[6] Joins are relevant only for blocking calls.

behavior in sequential systems: we allow recursive calls to execute without (re)acquiring locks.

However, the resulting behavior is undefined if a thread forks subthreads. When a thread forks several children, which thread gets the lock? If all subthreads have access to the object state, they may interfere with one another, exactly the behavior locks are supposed to prevent. The desired behavior is that any of the descendant threads may access the state, but only one may have access at any given time. Thus, our proposed semantics is to satisfy two properties: first, descendants must be able to acquire locks held by their ancestors, and second, mutual exclusion must be provided among siblings. One way to view this is that a descendant thread must acquire a lock from its ancestor, providing mutual exclusion from its siblings. The lock is returned to the ancestor when the descendant releases it. Once returned another descendant may acquire the lock.

The semantics discussed so far provide mutual exclusion at the granularity of a single method. Atomicity at both smaller and larger grain sizes may be desirable. Finer granularity exclusion can be provided by explicitly acquiring and releasing locks within a method. Larger granularity exclusion requires that threads hold locks across multiple method invocations. Although our work does not address these granularities explicitly, they can be handled by the solutions we present with simple modifications. Furthermore, most concurrent object-oriented languages provide little or no support for fine- or coarse-grain locking.[7] These languages commonly acquire locks upon method invocation and release them at or near completion of the method. This implicitly limits the granularity of mutual exclusion to the method level.

The recursion allowed by the solutions presented in this paper is generated by calls, not by sends. This implies that a method that performs a send and *requires a result generated by that send before it completes* will suffer from deadlock upon recursion. For example, a method that performs a request using send, and then waits for information from a second send, will deadlock if recursion is required to generate the second send. The obvious solution is to use call instead, so that the reply either is the result or guarantees that the required action has taken place. Thus we view send as a mechanism for causing remote actions that need not complete before the current method. This matches the use of call and send in current object-oriented systems.

4.3 Recursion and Message Order

The effect of the proposed mechanisms on *message order* depends on the policy of the underlying language. In some languages, such as Acore [Man87], message ordering is completely nondeterministic. Others, such as ABCL/1 [Yon87], make the *transmission-order preserva-*

[7] Exceptions include Argus [Lis88] and Concurrent Smalltalk [Dal87].

tion assumption (TOPA). TOPA guarantees point-to-point ordering: for any pair of messages sent from an object X to another object Y, the order of reception is identical to the order of transmission.

If message ordering is nondeterministic, there are no ordering constraints that our mechanisms could violate. If TOPA is guaranteed by the underlying message delivery system, then our mechanisms preserve TOPA.

However, recursion introduce a new notion of order. Recursion implies that recursive subtasks are logically part of the current task. In other words, all nested subtasks must complete before the current task can complete. Thus recursive subtasks (spawned by *recursive messages*) must be executed while the main task is waiting for the reply that the subtasks are supposed to generate. The logical order of execution is the order of reception, except that only recursive subtasks execute while the main task blocks on a call. Recursive messages are handled in the order received *relative to other recursive messages*, but before all non-recursive messages.

5 General Solution Framework

We describe two different transparent system-level mechanisms that allow programmers to express computations using recursive methods. No syntactic changes or programmer-visible linguistic mechanisms are necessary.

The solutions presented in this paper have two key aspects in common. First, each message is tagged with identifying information by its sender. Second, each object filters incoming messages, dynamically deciding whether to accept or buffer each message based on the identifying information that it contains. A subset of messages in the message queue are currently *acceptable*. A predicate, called the *accept predicate*, is used to test for membership in this set. When an object is ready to process a new message, it accepts the first message in its message queue that satisfies the accept predicate.

6 Multi-Ported Object Solution

In this section we present a novel solution to the recursion deadlock problem using multiple communication channels, or *ports*, per object. Conventional object-oriented systems assume objects have a single port through which all incoming messages arrive. The traditional notion of an object can be relaxed to allow *several* ports. The use of multiple ports to enable different client capabilities, multiple viewpoints, and secure communications is explored in [Kah89]. We demonstrate that the recursion deadlock problem can be solved by providing objects with the ability to create and select ports dynamically.

The recursion problem is solved by creating a new *current port* for each method invocation. This port receives all replies and recursive calls, and persists until the method terminates. An object accepts incoming messages from its current port, and buffers messages addressed to other ports. The current port for an idle object X is the distinguished top-level port P_0^X.

6.1 Message Acceptance

Messages that arrive at an object X while it is executing some method H are buffered in the message queue associated with X for later processing. If H is blocked pending the arrival of messages that are required for further computation, or if H completes, X enters message reception mode. At this point X may begin processing messages that satisfy the *accept predicate*. Multi-ported objects use the following accept predicate: a message is accepted if and only if it is addressed to the current port.

6.2 Message Handling

Normal object semantics guarantee that for each object, only one method activation exists at a time; objects process messages serially (between state transitions). To permit recursive calls, we weaken this constraint to require that each object maintain a *method activation stack* of pending method activation frames, and guarantee that only the activation at the top of this stack may be actively executing. This corresponds to the stack of procedure call frames found in conventional sequential languages.

The top frame on the method activation stack contains the state of the currently executing computation. Frames other than the top frame contain the state of suspended method invocations that are blocked pending the arrival of reply messages. New frames are pushed on the method activation stack when handling messages other than replies.[8]

When message M is accepted by object X, the following actions are performed:

1. If M is a reply, match the reply value to its corresponding call. If there are no remaining outstanding concurrent calls, resume the associated blocked thread.

2. Otherwise, the following actions are performed:

   (a) A new frame is allocated on top of the method activation stack.

[8] If the current port is the distinguished top-level port, these messages can be top-level calls or sends. Otherwise, these messages will be recursive calls.

(b) A new, locally unique port number P is generated (perhaps by simply incrementing a counter) and associated with the frame. This port becomes the new current port; the set of currently acceptable messages are those addressed to P.

(c) The appropriate method named by M is invoked.

When a method completes, its frame is popped off the method activation stack. The current port is then set to the port associated with the suspended method currently on top of the activation stack.

6.3 Sending Messages

In the following discussion, assume that an object X, during the invocation of its method H in response to a message M, is sending a message M_Y to object Y. The current port for X during its handling of H is denoted by P_h^X.

M_Y is augmented with a *port binding map*, $\mathcal{B}_{M_Y}$, that associates object names with communication ports. In general, the size of a port binding map $\mathcal{B}_M$ is proportional to the number of distinct objects involved in the processing of message M. The procedure for sending M_Y is:

1. If M_Y is a send:

   (a) Set $\mathcal{B}_{M_Y}$ to `nil`.

   (b) Send M_Y to Y at port P_0^Y.

2. If M_Y is a call, compute the destination port and the port binding map to be sent with M:

   (a) The destination port p is computed by searching for the port associated with Y in the port binding map $\mathcal{B}_M$ from message M. If $Y \notin \mathcal{B}_M$, then set p to P_0^Y.

   (b) Set $\mathcal{B}_{M_Y}$ to be the same as $\mathcal{B}_M$ extended[9] (or changed) to map $X \rightarrow P_h^X$.

   (c) Send M_Y to Y at port p, saving the appropriate information to match up the reply from Y.

   (d) If M_Y is one of several concurrent calls, perform the remaining calls. After performing the last call, suspend the current thread pending replies.

[9]This extension (or change) is done at most once per method invocation, not once per call.

6.4 Extension for Multiple Locks

Some languages, such as Concurrent Smalltalk [Dal87], support the addition of explicit locks for object methods as a general mechanism for concurrency control. The multi-ported object solution for recursion deadlock can be adapted to work with such languages.

Basically, the notion of a *current port* is extended to a current port *set*; each lock has an associated current port. Port binding maps associate object names with communication port sets. The accept predicate is modified to accept a message if and only if it is addressed to the current set of ports. When an incoming message is accepted, a new method activation frame is allocated, and an associated unique port number is generated for every lock that the method acquires. When a call message is sent, the current port binding map is extended (or changed) to map `self` to the current port set.

6.5 Summary

In summary, an object creates a port for each method invocation. This port is used exclusively for replies and recursive calls. The port name is propagated in the port binding maps of call messages to objects that perform computations as subtasks on behalf of the current method. Each call is addressed to a specific destination port, and is accepted by an object only if its port matches the port associated with the current method. Section 8 presents an example using the multi-ported object approach to avoid recursion deadlock.

7 Named-Threads Solution

The essential elements of named threads are based on *action ids* from Argus [Lis88, Lis87], a language for robust distributed computing. The *named-threads* approach to avoiding recursion deadlock assigns each thread a unique identifier that travels with it through every object and message. Every object has a current owner, which is its currently executing thread, and every message has a name, which is that of the thread that carries it. In a recursive call, the name of the message matches the name of the owner. This avoids the deadlock that results from attempting to reacquire access to the state.

The simplest cases occur in systems without concurrent calls. In such situations, a thread is given a unique name that it keeps for the duration of its existence. Upon acquiring access to the object's state, the thread marks itself as the owner of the object using its name or *thread id*. Upon recursion, the thread id of the message is checked against the thread id of the owner. If the ids match, the incoming message has access to the state. Because the new task can determine that it already has access, it does not wait for the current task to finish, thus avoiding deadlock.

The essential elements of named threads originated in Argus [Lis88], a language for robust distributed computing. Argus uses *transactions* to provide fault tolerance: a transaction either completes correctly, or if aborted, has no effect on the state of the system. A transaction often involves several objects (possibly on different machines) and several threads within an object. The techniques used to provide fault tolerance over many machines are quite complex and are not all relevant to recursion deadlock. Hidden in those techniques are the use of *action ids*, upon which thread ids are based [Lis87].

7.1 Concurrent Calls

Most concurrent object-oriented languages allow a single thread to create multiple threads. This wreaks havoc with the simple thread-id solution presented above. As discussed in Section 3, two requirements must be met for the desired semantics: descendants must be able to acquire locks held by their ancestors, and mutual exclusion must be provided among siblings. These requirements lead to the following convention for naming threads.

Upon creation, a thread is given a unique identifier. This could be done by using a combination of the id of the creating object and some time-dependent integer, such as a simple counter. As expected, the difficulties arise with concurrent calls. A set of concurrent blocking calls will be referred to as a *call group*. When a thread forks subthreads, we extend its thread id for every member of its call group. The extension is different for every resulting thread. For example, if thread t performs three concurrent calls, the three new threads are: $t.1$, $t.2$, and $t.3$. If $t.1$ then spawns a two-member call group, there are six threads in total: t, $t.1$, $t.1.1$, $t.1.2$, $t.2$, and $t.3$. Note that each thread has a unique thread id, and that a thread id encodes all of the thread's ancestors. A thread is an ancestor of another thread exactly when its id is a prefix of the other's id.

The use of named threads requires some extensions to the basic object model. First, the object lock must have an owner. This field holds the name of the currently executing thread and is referred to as `owner`. Second, there must be a stack of pending call messages (as explained below), referred to as `ownerStack`. Finally, each message must have a field that contains its thread id.

7.2 Message Acceptance

The named-threads approach employs the following *accept predicate*: a message is accepted if and only if it is a descendant of `owner`. That is, `owner` must be a prefix of the thread id associated with the message to be accepted. The accept predicate changes every time `owner` changes. The `owner` field initially contains `nil`, which yields a predicate that accepts all messages. Accepted messages are handled as follows:

1. If the message is a reply, then it is a response to a call spawned by the current owner. (If it did not match the current owner, it would not have been accepted.) The blocked thread is resumed.

2. Otherwise, the message is a recursive call. In this case, the current `owner` is pushed onto the stack `ownerStack`. The thread named by the id of the message becomes the new owner, and the method for the message executes.

7.3 Choosing the Next Thread

Once a thread is started, no new messages are accepted until the thread either ends or performs a call. When the thread ends, the next thread is chosen as follows:

1. The previous owner is popped off of `ownerStack`. This changes the accept predicate, which could make previously unacceptable messages acceptable.

2. The next message is chosen based on the new predicate. There must eventually be a message, namely the matching reply to the pending call.

7.4 Sending Messages

If a thread performs a call, the object starts accepting messages. Because the thread owns the lock, only descendant messages or the reply will be accepted. A send does not pass on the thread id, leaving the id field of the message empty. Thus methods executed in response to sends start new threads.

When concurrent calls are made, the thread id of each member of the call group is extended by a unique number. All of the members are descendants of the original thread, but no member is an ancestor of another member. This guarantees that at most one member of the call group can have access to an object at any given time. The spawning thread retains ownership of the lock until one of the concurrent calls recurses, or until all the replies are received and the method completes.

7.5 Extension for Multiple Locks

As in the multi-ported object solution, the named-thread solution can be modified to handle multiple locks per object. Each object has a set of locks and a method requires a particular subset in order to execute. Each lock has an owner field and a stack of owners. As in the single-lock case, the owner field contains the name of the thread that holds the lock. A message is accepted if it can obtain all of the locks needed by the corresponding method. Every lock the method needs must be either free or owned by an ancestor. If the required subset can be obtained, the current thread becomes the new owner of every lock in the subset. The previous owners are pushed onto the corresponding stacks. Upon release of a lock, the previous owner is restored.

7.6 Summary

In summary, recursion is detected by encoding ancestry in thread ids. Recursive calls are descendants of the blocked call, and are allowed to execute. Names are extended when concurrent calls are made, providing mutual exclusion among the resulting threads. An example is given in the next section.

8 An Illustrative Example

Consider a `do-query` method defined for a `node` object that is connected to other nodes in a graph. When invoked on a node N, this method queries each of N's children concurrently and then returns a function of their replies. The children compute their replies in the same way. As is common in sequential implementations, a node object marks itself visited the first time it is queried, and the `do-query` method returns immediately if an object has already been visited. To illustrate our solutions, we examine this abstract concurrent query algorithm on part of a larger graph. Note that the call tree is isomorphic to the graph; calls are made along the directed edges.

Figure 1(a) presents a graph that does not involve recursion. Figure 1(b) depicts a query performed in this graph. Object X queries object W, which concurrently queries objects Y and Z. All of the queries complete and X returns the result. Figure 1(c) presents a graph that involves recursion; the corresponding query in figure 1(d) suffers from recursion deadlock. Since X is blocked waiting for the query to W to complete, the query from W to X never executes. The method at W will not complete until X replies, and X will not reply until the method at W completes. The following two subsections illustrate how the solutions eliminate recursion

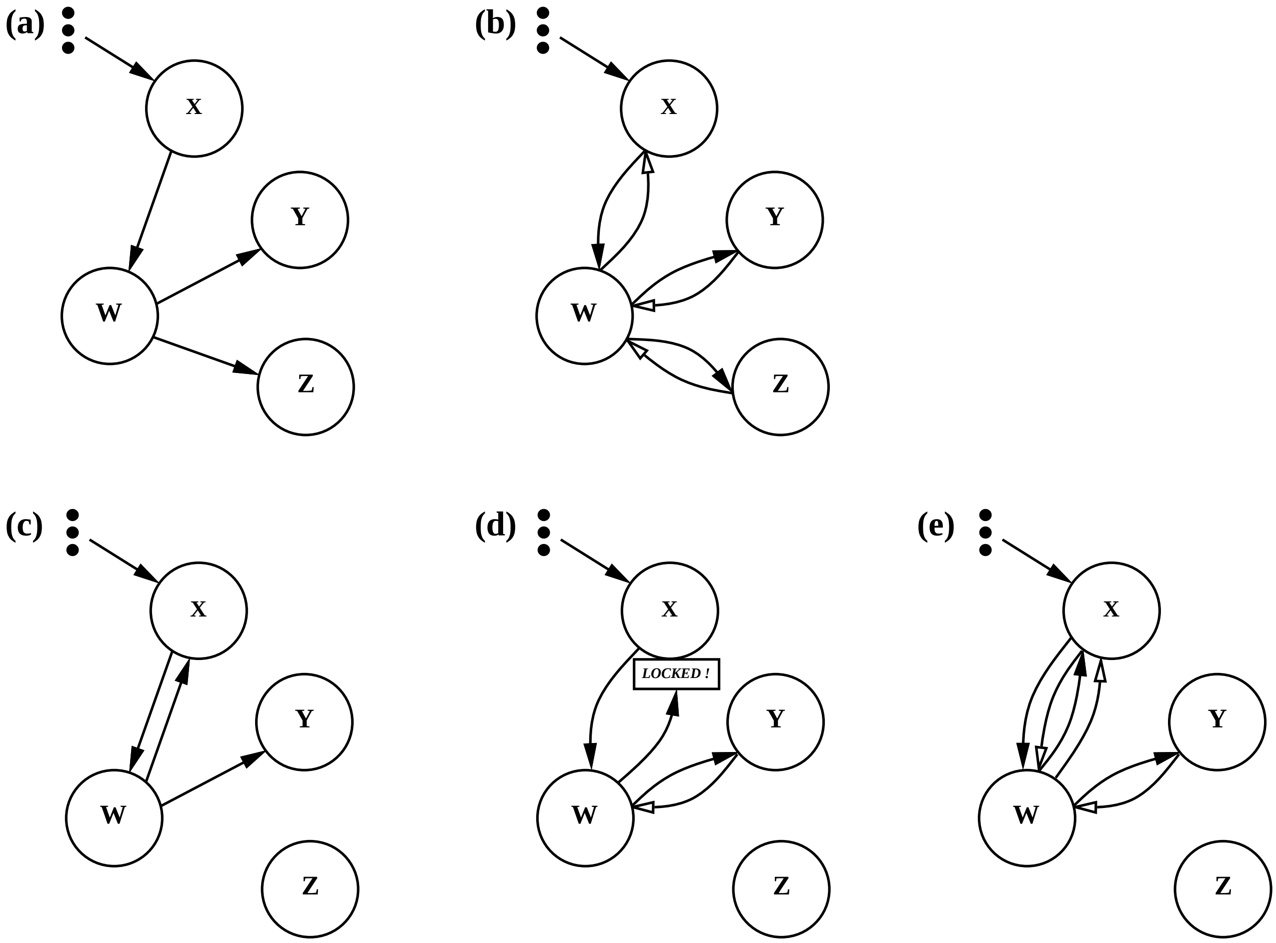

Figure 1: Graph fragments and calling patterns in the abstract query example. In the calling pattern diagrams, call and reply messages are represented by solid and hollow arrowheads respectively. (a) A non-recursive graph fragment. (b) Call pattern in the non-recursive graph fragment. X calls W, which concurrently calls Y and Z. (c) A recursive graph fragment. (d) Call pattern in the recursive graph fragment. X calls W, which concurrently calls X and Y, resulting in deadlock. (e) A deadlock-free call chain, which uses the solution techniques presented in the paper.

deadlock in this example.

8.1 A Multi-Ported Object Solution

The following steps, depicted in Figure 1(e), trace the message-passing activity for the example presented above. Assume that objects W, Y, and Z are initially idle, with current ports P_0^W, P_0^Y, and P_0^Z, respectively. Assume that X calls W while its current frame is X_5, with associated port P_5^X.

1. X calls `do-query` at W on port P_0^W, using the port binding map[10] $\mathcal{B} = \{(X \rightarrow P_5^X)\}$.

2. W receives X's call: W starts a new frame W_1 with associated port P_1^W. W concurrently calls X and Y, using $\mathcal{B} = \{(X \rightarrow P_5^X), (W \rightarrow P_1^W)\}$. The call to X is sent to X's port P_5^X, and the call to Y is sent to Y's port P_0^Y.

3. Concurrently:

   (a) Y receives W's call: Y starts a new frame Y_1 with associated port P_1^Y. Y replies to W, and ends frame Y_1.

   (b) X receives W's call: Note that this is a recursive call involving the path of objects $X \rightarrow$W$\rightarrow X$. X accepts the message since it is addressed to X's current port P_5^X. X starts a new frame X_6 with associated port P_6^X. X replies to W, and then ends frame X_6, restoring X_5 as the current frame.

4. W receives the replies from X and Y.

5. W computes the return value as a function of the information returned from X and Y, and sends a reply containing this value to X. W then ends frame W_1, restoring W_0 as the current frame.

8.2 A Named-Threads Solution

This example can be used to illustrate the named-threads technique as well. Objects will be annotated with their current owner: "$X[t.1]$" implies that object X is owned by thread $t.1$.

1. $X[x]$ calls $W[\,]$, invoking method `do-query`. This leads to $W[x]$.

---

[10]The port binding map $\mathcal{B}$ would contain additional entries if the current call is part of a larger call chain. For clarity, only those entries relevant to the calls in the depicted graph fragment are listed.

2. $W[x]$ concurrently calls $X[x]$ and $Y[\,]$. The message to $X[x]$ has an id of $x.1$, and the message to $Y[\,]$ has an id of $x.2$.

3. Concurrently:

   (a) $Y[\,]$ receives the call from $W[x]$. Note that W is owned by x but the calling thread is $x.2$. This implies $Y[x.2]$. The result is calculated and returned to $W[x]$. After the reply, Y is again unowned, that is, $Y[\,]$.

   (b) $X[x]$ receives the call (with message id $x.1$) from $W[x]$. This is a recursive call. Since x, the current owner of X, is an ancestor of the calling thread, $x.1$, the call is accepted. The current owner, x, is pushed on to `ownerStack`, and $x.1$ becomes the new owner. $X[x.1]$ replies to $W[x]$, and the previous owner is popped off the stack. This implies $X[x]$.

4. $W[x]$ receives the replies from X and Y.

5. W computes the return value of `do-query` based on the replies. The result is returned to $X[x]$, and W is again unowned. The ownership is then: $X[x]$, $Y[\,]$, and $W[\,]$.

9 Analysis and Comparison

The multi-ported object and named-thread solutions affect the underlying system in three major areas: maintenance of a call stack, overhead for message sending and message acceptance, and an increase in message length. The impact is negligible when using only sends, but may be significant in the presence of blocking calls.

A send message incurs negligible overhead because there is no call stack, little extra work for handling messages, and no increase in message length. For the case of blocking calls, each object maintains a stack of pending calls. In the multi-port solution this is the stack of frames, while in the named-threads solution, it is the stack of owners. For general recursion, the size of the pending call stack is unbounded. The height of the stack at X depends on the number of recursive calls in the call chain, that is, the number of calls to X. For single-object recursion, this is the depth of the call chain, while for multiple-object recursion the height will be smaller than the length of the call chain. The stack is not a side effect of these solutions; it is fundamental to recursion. Just as a stack supports recursive procedures in conventional sequential languages, the stacks used here support recursive call messages.

9.1 Comparative Overhead

For the general case, is it useful to define two metrics for call chains: *object depth* and *split depth*. The object depth is the number of distinct objects in a call chain. Thus, if X calls Y calls Z, the object depth is three, while if X calls Y calls X, the object depth is two. The split depth measures the number of splits in a call chain. An unsplit thread is defined to have a split depth of one. Thus if X calls Y, the split depth at Y is one, while if X concurrently calls Y and Z, the split depth at Y is two (one for the original thread and one for the split at X). Using the named-threads notation introduced in section 7, the split depth is the number of fields in the id: "x.1.2" has a split depth of three.

The solutions differ in the performance of choosing destinations for calls and accepting messages. For multi-ported objects, choosing a destination for a call requires scanning the port binding map to identify the exact destination. The average time for this scan is proportional to the length of the port binding map, which is the same as the object depth of the call chain. Thus the time for locating the destination is proportional to the object depth.[11] For the named-threads solution, the destination is known and the cost is constant.

The cost of message acceptance has a dual behavior. The multi-port solution checks the validity of the port with a single comparison. The named-threads solution must compare the id of the message and the id of the owner. In the worst case, this comparison is proportional to the length of the owner id. Since the length of the owner grows with each split, the cost of the acceptance test is proportional to the split depth. Note that if the object is unowned, the message is a send, or the id starts with a different object, then the test completes immediately. In summary, the multi-port solution requires time proportional to the object depth for choosing a destination, but can accept messages in constant time. On the other hand, the named threads solution requires time proportional to the split depth for accepting a message, but can locate message destinations in constant time.

Message length is also affected differently by the two solutions. For multi-ported objects, the message is extended by the port binding map: the length of the extension is proportional to the object depth. For named threads, the message is extended by the id of the calling thread: the length of the extension is proportional to the split depth.

[11] More complex data structures, such as hash tables, could reduce this lookup to constant time asymptotically. However, this would probably be worse in practice due to both higher constant factors and larger space requirements.

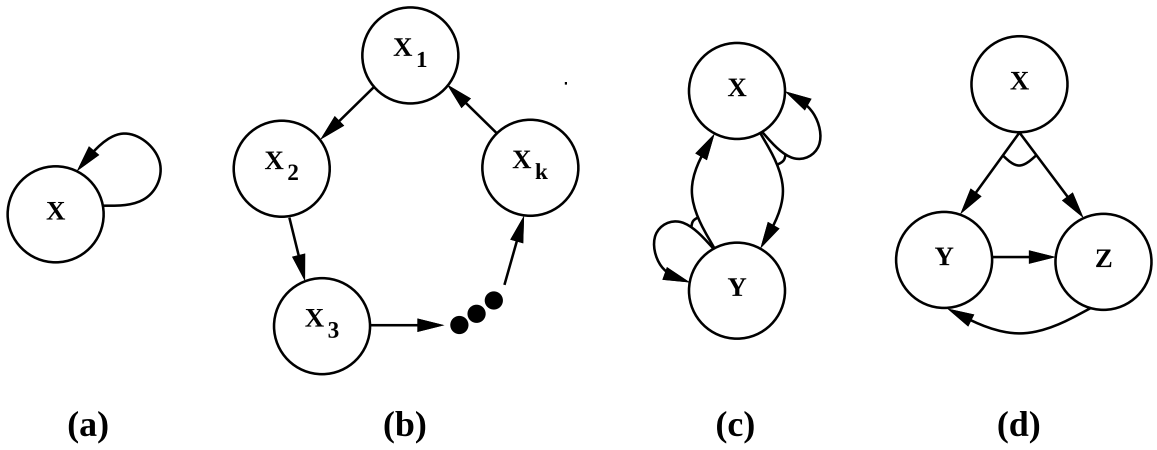

Figure 2: Calling patterns for comparing object depth and split depth. Arrows denote calls, and arcs joining arrows signify concurrent calls.

9.2 Example Calling Patterns

Since the performance of the multi-port solution depends on object depth and that of the named-threads solution depends on split depth, their relative merit depends on the call chains encountered in a given system. Several possible call chains are shown in Figure 2.

The most common case of recursion is direct recursion, represented by the call chain in Figure 2(a). Only one object and one thread are involved, so both the object depth and the split depth are one. Both solutions have low overhead for this case.

A more general form of recursion is shown in Figure 2(b). The call chain is a cycle of k objects. For $k = 2$, this reduces to simple mutual recursion. Since there are k objects involved, the object depth is k, regardless of how many times the call chain loops around the cycle. Since there are no concurrent calls, the split depth is one. Note that the overhead is the same if the last call is not recursive; the split depth is one and the object depth is k. The overhead for the multi-port solution is proportional to k, while the overhead for the named-threads solution is constant. The named-threads approach is superior when there are many objects involved and few concurrent calls.

The multi-port solution is superior in the opposite case: few objects and many concurrent calls. Figure 2(c) depicts a call chain with only two objects, each of which makes a pair of concurrent calls. This kind of call tree might occur in a system that processes data structures with two different types of nodes. For example, consider a computation applied to a binary tree, in which X handles all of the right nodes and Y handles all of the left nodes. Because there are only two objects involved, the object depth is two, regardless of the height of the call tree. Performance is much worse for the named threads solution. After k pairs of concurrent calls, the split depth is k for threads that are the leaves of the call tree. Thus, the overhead for the multi-port technique is constant, while the overhead for the named-threads solution is proportional to k.

In summary, the relative overhead of these solutions depends on the ratio of object depth to split depth. Systems with recursion among small groups of objects and many concurrent calls would perform better using the multi-ported object technique. Systems with recursion among many objects and few concurrent calls would perform better using the named-threads technique. In many applications, call chains involve few objects and few concurrent calls, in which case both solutions perform well.

9.3 A Hybrid Model: Differentiating Recursive Calls

Since the additional overhead in the proposed solutions stems from the requirements of recursive calls, it may be useful to differentiate calls that are allowed to recurse from those that are not. Note that if a call is not allowed to recurse, the situation is identical to current object systems with one exception: non-recursive calls must propagate the recursion information if performed in response to a call that is allowed to recurse. This mixed model yields better performance when a programmer (or compiler) is certain that recursion cannot occur. However, deadlock results if a "non-recursive" call does recurse.

9.4 General Object Deadlock

Although the techniques presented eliminate deadlock due to recursion, the general deadlock problem remains quite serious. Figure 2(d) depicts a call chain that may lead to deadlock. Object X concurrently calls objects Y and Z, which in turn call each other. Deadlock occurs if Y is blocked waiting on Z and Z is blocked waiting on Y. In this case, deadlock can be avoided by serializing the calls at X. Unfortunately, avoiding deadlock in general is quite difficult. Thus, although blocking calls provide clean, simple semantics, it is generally safer to use sends wherever possible. But using sends without eliminating blocking is no better than using calls; the fundamental problem is blocking.

10 Conclusions

Current object systems place severe limits on the use of recursion, reducing expressive power. The two techniques presented allow fully general recursion in a manner that is transparent to the user. The multi-port solution uses ports to distinguish recursive calls, placing the burden on the sender to identify the correct port. The named threads solution names each path in the call tree, encoding ancestors in the name. The test for ancestry is used to detect recursive calls, placing the burden on the receiver to identify ancestors. The relative overhead of these solutions is dependent on application call graphs, in particular, on the ratio of object depth to split depth. Call graphs with many objects and relatively few concurrent calls perform better with the named-threads solution, while the multi-port solution leads to better performance in the opposite case.

Recursion is a powerful and important programming technique that causes deadlock in most concurrent object-oriented systems. The solutions presented in this paper provide simple, effective system-level support for general recursion. They can be used by designers and implementors of concurrent object-oriented systems to avoid severe restrictions on the expression of recursion.

11 Acknowledgements

We are grateful to Barbara Liskov, William Weihl, Adrian Colbrook, Chris Dellarocas, Sanjay Ghemawat, Bob Gruber, Wilson Hsieh, Ken Kahn, and Paul Wang for their comments and assistance.

References

[Agh86] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, 1986.

[Ame87] Pierre America. *POOL-T — A Parallel Object-Oriented Language.* In Akinori Yonezawa and Mario Tokoro, eds., *Object-Oriented Concurrent Programming*, MIT Press, 1987.

[Bos89] Jan van den Bos and Chris Laffra. *PROCOL: A Parallel Object Language with Protocols.* Proceedings of the Fourth ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '89), October 1989.

[Chi90] Andrew A. Chien. *Concurrent Aggregates (CA): An Object-Oriented Language for Fine-Grained Message-Passing Machines.*, PhD thesis, Massachusetts Institute of Technology, July 1990.

[Dal87] William J. Dally. *A VLSI Architecture for Concurrent Data Structures.* Kluwer Academic Publishers, 1987.

[Gli80] Virgil D. Gligor and Susan H. Shattuck. *On Deadlock Detection in Distributed Systems.* IEEE Transactions on Software Engineering 6(5): 435-440, September 1980.

[Gol83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, MA, 1983.

[Had77] Bruce K. Haddon. *Nested Monitor Calls.* Operating Systems Review, vol. 11, no. 3, October 1977.

[Kah87] Kenneth Kahn, Eric Dean Tribble, Mark S. Miller, and Daniel G. Bobrow. *Vulcan: Logical Concurrent Objects.* In Ehud Shapiro, *Concurrent Prolog: Collected Papers*, MIT Press, 1987.

[Kah89] Kenneth Kahn. *Objects: A Fresh Look.* Proceedings of the Third European Conference on Object-Oriented Programming (ECOOP '89), Cambridge University Press, July 1989.

[Lie86] Henry Lieberman. *Using prototypical objects to implement shared behavior in object-oriented systems.* Proceedings of the First ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '86), September 1986.

[Lie87] Henry Lieberman. *Concurrent Object-Oriented Programming in Act1.* In Akinori Yonezawa and Mario Tokoro, eds., *Object-Oriented Concurrent Programming*, MIT Press, 1987.

[Lis87] Barbara Liskov. *Implementation of Argus.* Proceedings of the 11th ACM Symposium on Operating Systems Principles, November 1987.

[Lis88] Barbara Liskov. *Distributed Programming in Argus.* Communications of the ACM, vol. 31, no. 3, March 1988.

[Lis77] Andrew Lister. *The problem of nested monitor calls.* Operating Systems Review, vol. 11, no. 2, July 1977.

[Man87] Carl R. Manning. *Acore: The Design of a Core Actor Language and its Compiler.* Master's thesis, Massachusetts Institute of Technology, August 1987.

[Sin89] Mukesh Singhal. *Deadlock Detection in Distributed Systems.* IEEE Computer, November 1989.

[Str86] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, MA, 1986.

[Tom89] Chris Tomlinson and Vineet Singh. *Inheritance and Synchronization with Enabled-Sets.* Proceedings of the Fourth ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '89), October 1989.

[Yon87] Akinori Yonezawa, Etsuya Shibayama, Toshihiro Takada, and Yasuaki Honda. *Modelling and Programming in an Object-Oriented Concurrent Language ABCL/1.* In Akinori Yonezawa and Mario Tokoro, eds., *Object-Oriented Concurrent Programming*, MIT Press, 1987.