Higher-Order Concurrency in Java*

Erik D. Demaine
Dept. of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
eddemaine@uwaterloo.ca

Abstract. In this paper we examine an extension to Hoare's Communicating Sequential Processes model called *higher-order concurrency*, proposed by Reppy. In this extension, communication algorithms (or *events*) are first-class objects and can be created and manipulated dynamically. In addition, threads are automatically garbage collected and channels are first-class, that is, they can be passed over other channels. We describe the design of a Java package that implements the main features of higher-order concurrency, with similar ease-of-use to Reppy's Concurrent ML system. Our implementation can be easily extended to use a distributed system, which is a major limitation with Concurrent ML. We also hope to bring the idea of higher-order concurrency to a wider audience, since it is extremely powerful and flexible, but currently only well known to the programming-languages community.

1 Introduction

CSP (Communicating Sequential Processes) [6] and its derivative occam [12] initiated the area of *concurrent programming*, which is now a large area of research. In particular, they introduced important concepts, including synchronous communication over channels and non-deterministic choice, that provide a useful abstraction of message passing. Recently, Welch and Wood [13] have explored adding higher-level communication primitives in the KRoC 0.8beta release of occam [14].

Briefly, CSP provides message passing by basic send and receive primitives. They are synchronous in that the send/receive operations wait for matching receive/sends to continue. The destination or source passed to one of these primitives is specified by a *channel*. A channel is a uni-directional connection between two processes. They effectively correspond to the *port* abstraction, that is, one of several message queues on a particular process, but are much more convenient to program with. The non-deterministic choice operation (called `alt` in occam) does one out of a list of sends and receives, whichever has a matching partner first.

Reppy [10] proposed an extension to the basic CSP model called *higher-order concurrency*. One way to view this extension is that synchronous-communication operations, which we call *events*, can be more than just a send, a receive, or a choice of sends and receives. Rather, an event could consist of doing several subevents in sequence. The

*This work was supported by the Natural Sciences and Engineering Research Council (NSERC).

idea is that you can then build up arbitrarily complex events by combining primitive events (sends and receives) with both non-deterministic choice and sequences.

1.1 First-Class Events

In the higher-order-concurrency model, events are first-class objects, that is, events can be dynamically created, stored in variables, and passed to/returned from functions, as if they were more conventional objects like integers or booleans. Instead of writing a function to perform some communication (e.g., implement a protocol), one can write a function that returns an event describing how to do this communication. Events can be executed when desired via the `sync` operation. The advantage of having an "intermediate form" of events is that the event returned by the function can now be manipulated (i.e., combined with other events) before it is executed. If the function carried out the communication directly, there would be no way in general to manipulate it without changing the code.

A simple example is communicating with a distributed database that may replicate data (for example, DNS servers). A program may wish to query a number of database servers for the same information, and use the result that comes first (this corresponds to a non-deterministic choice). Presumably, the distributed database comes with a client function that allows a program to make a query to a server and get a response. If the function returns an event, we can simply call the function once for each server we want to query, use the `choose` combinator[1] (making another event `e`), and call `sync(e)`. On the other hand, if the function sends the request and then waits for a reply (instead of immediately returning an event explaining how to do so), then we would have to modify the routine to support several transactions at once, resulting in messy, difficult-to-understand/debug code. This is especially the case when each server uses a different, complex protocol.

One may question whether complex protocols can actually be described using few event combinators. Indeed, the higher-order-concurrency model provides very few features, but they are provided in an extremely flexible (first-class) way. In his PhD thesis [10], Reppy showed that the following forms of communication could be constructed (of course, the solutions also involve creating threads). In other words, he was able to construct event values *from scratch* that represent the following advanced concurrent features:

- Buffered channels (asynchronous communication)
- Multicast channels and thus multicast operations
- Condition variables (write-once variables)
- Ada-style rendezvous, plus allowing nested transactions
- Locks and semaphores
- Multilisp futures

[1] A *combinator*, from functional-language terminology, is a function that takes values of a particular type and "combines" them into a new value of that type. For example, the compose operation ($\circ$) is a combinator on functions.

Since these constructs are abstracted into events, they can be further manipulated. For example, we can non-deterministically choose between multiple Ada entry calls. We could also obtain lock A or lock B, whichever is available first; this is particularly useful for resolving deadlock. This is clearly not possible with other concurrent languages unless there is built-in support for these specific operations, but it is a fortunate side effect of higher-order concurrency.

The power of typical concurrent-programming languages can be measured by how many features it provides. On the other hand, if a concurrent-programming language provides first-class event abstraction, it does not have to "guess" what concurrent structures the user might want. If the language does not support a needed construct, users can build it themselves. Users do not have to make do with the provided forms of communication; instead, they can build up their own forms from existing ones, and use them as if they were built-in. Assuming they do this in the spirit of higher-order concurrency, they can then make new constructs from others they have built.

Another important feature of the higher-order concurrency model is garbage collection of threads. The system monitors channels, and marks them as *dead* if there are no threads currently "at the other end" of the channel, that is, communications on this channel will necessarily block forever. Whenever a thread executes an event that only involves dead channels, it can be proven that the thread will block forever, and hence it can be discarded. This greatly simplifies non-deterministic communication and end cases; for example, the programmer does not need to worry about sending "kill" messages to so-called "daemon" threads that implement the above constructs.

1.2 Goals

Reppy implemented the higher-order-concurrency model in the Concurrent ML (CML) system [9, 11]. CML is implemented in the sequential language Standard ML (SML) via user-level threads. There are two major disadvantages of implementing higher-order concurrency in this way.

First of all, SML is not in common use except for programming-language research. This has the consequence that CML is not wide-spread, even though it has an incredible amount of flexibility that makes it a top-ranking concurrent language. It would be advantageous to bring the concurrency features to a more popular language such as C or Java.

Second, there are no possibilities for exploiting a parallel computer, even though CML programs often have a high level of concurrency. If we can use system-level threads or operating-system processes, then we can put parts of the program on separate processors. SML has little support for distributed systems, whereas most of the development in this area has been done in languages such as C, and more recently in Java 1.1 with Remote Method Invocation (RMI).

In previous work [1], we implemented several features of higher-order concurrency in the Parallel Virtual Machine (PVM) [3] using a base language of C. Unfortunately, because of the base language, the package is somewhat inconvenient to use because events and channels have to be deleted explicitly. There is also an implicit problem because processes cannot be parameterized, disallowing some things that are possible in CML.

In this paper, we look at how higher-order concurrency can be implemented in Java.

Java [4] provides garbage collection of objects and full subtyping. In addition, it has support for concurrency, and will soon have effective support for distributed systems. As we shall show, this makes it particularly suitable for implementing higher-order concurrency in a way that it is extremely convenient to use. Java is a good "replacement" for ML, since it provides a sufficient number of features, while staying (in principle) more efficient since it avoids expensive features such as closures and continuations.

While we have as yet only implemented a multi-threaded, single-process version of higher-order concurrency, we have designed the package using a system model that is very close to a distributed system. We consider threads to be completely separate in the sense that the only way they can communicate is through mailboxes, which is essentially message passing. Hence, the system should be easy to modify to use multiple computers in parallel.

The main goal of this paper is to bring the capabilities of higher-order concurrency to a larger community. In this way, we hope that more people will realize the useful generality of the approach, and that the ideas will become more widespread. We feel that garbage collection of processes and first-class events and channels offer a powerful and easy-to-use abstraction for concurrent programming that does not significantly harm performance.

Similar to CSP and occam, it is relatively easy to reason theoretically about programs written in the higher-order-concurrency model, using the formal semantics defined by Reppy [11]. Hence, we are also helping to bridge the gap between theoretically based languages and Java.

1.3 Related Work

No one else, to our knowledge, has implemented higher-order concurrency in a language other than ML. Hence, the idea of bringing it to a wider audience is entirely new. While the Distributed ML project [8] has extended Concurrent ML to use multiple processors, it only supports synchronous channels within the same process; one can only use "port groups" (a form of asynchronous multicast channels) for interprocess communication. We feel that changing the available concurrency abstractions, depending on the location of the threads that want to communicate, is an unfortunate approach.

Hilderink et al. [5] examine implementing CSP in Java. While our implementation of higher-order concurrency could be considered a generalization of this work, the approaches are entirely different.

We have developed several protocols to support synchronous communication over one-to-many and many-to-one channels with non-determinism. This is by no means new. The deadlock-free version of the protocol in Section 5 resembles the protocol described in [7], although the latter uses extra threads. Our two algorithms that only use a two-message cycle per user-level message (one deadlock-prone, the other deadlock-free) are new.

1.4 Outline

In the following four sections, we look at various components of higher-order concurrency, and show how they can be implemented in Java. Section 6 gives an example

```
public interface ThreadCode {
   public void threadCode (void);
}
```

Figure 1: *The* `ThreadCode` *interface.*

construction, demonstrating that the described system is easy to use. We conclude in Section 7.

2 Threads

In the higher-order-concurrency model, threads are described by functions that take no parameters and return `void` (in C/Java terminology), which represent their code. Java does not support references to functions; instead, we must place the code for a particular thread in a class, and pass an instance of the class. We have the ability to store state in non-static data of the instance, representing a closure (functional-language terminology for code with a corresponding environment or state).

These "code classes" are represented by implementing the `ThreadCode` interface (Figure 1). The static `HOC.spawn` routine, which creates a thread running the specified code, then simply creates an `HOCThread` object, which extends `java.lang.Thread`, whose `run` routine simply calls the associated `threadCode` function. The reason for this level of indirection (instead of the user simply specifying a `java.lang.Runnable` object), is that we can now call our own initialization code, and call cleanup code (described further in Section 4) when the thread exits. The thread exits either by returning from the `threadCode` function or by throwing `java.lang.ThreadDeath`, which can be caught via `try/catch`.

3 Mailboxes

As mentioned in Section 1.2, we completely separate threads and only allow them to communicate via mailboxes. We define two types of mailboxes: a `CMailbox` for control messages used in the protocol described in Section 5, and a `Mailbox` that is used for sending user-level messages. In this section we describe the latter type, because they do strictly more than `CMailbox`es.

In the next section, we will see that channels must be modified when they are communicated to other threads (through channels); note that channels, like events, are also first-class objects. Since the type `Chan` is a class (as are all non-trivial types in Java), the `Mailbox` can only store a reference to the channel, or whatever object is being communicated.

Java provides run-time type information, so the `Mailbox` can determine if the object is indeed a `Chan`, in which case it can "morph" it into a different channel. We will see that we need a two-phase morphing process; the sending thread must "pre-morph" the channel, and the receiving thread must "morph" it and return the morphed channel. More generally, we provide the `Morphing` interface (Figure 2) which `Chan` implements; if the object being communicated implements `Morphing`, then the `preMorph` and `morph`

```
public interface Morphing {
   public void preMorph (Thread dest);
   public Morphing morph (Thread src);
}
```

Figure 2: *The* `Morphing` *interface.* `preMorph` *is called on the sending thread.* `morph` *(called on the receiving thread) should return an object of the same type as* `this`*.*

methods are called at appropriate points during communication. Note that this is transparent to the user, because it is all done at the `Mailbox` level (unless the user "cheats" and communicates objects without using mailboxes (channels), in which case morphing must be done by hand).

An important problem arises, however. Suppose the user passes a `java.util.Vector` or `java.util.Hashtable` to another thread. These data structures (which store general `java.lang.Objects`) potentially include `Morphing` objects. This can be abstracted as follows: the object being passed is a tree, where each non-leaf is a *container*, and each leaf is not. We wish to examine each leaf, and morph it if it implements `Morphing`, which involves changing the reference stored in the parent container.

We implement this by providing static functions `HOC.morph` and `HOC.preMorph` that can be applied to arbitrary `java.lang.Objects`, which check if the object implements `Morphing`, and if so call its `morph` and `preMorph` methods, respectively. Hence, any object that contains general objects can implement `Morphing`, and simply call `HOC.preMorph` and `HOC.morph` on each subobject it includes. Since we do not want to modify the Java class library, we currently have to handle `java.util.Vector` and related built-in Java types as special cases, by checking if the object is one of these types.

The end result is that passing objects is simple, while creating classes that may contain `Morphing` objects is now more complicated. Such a complication is necessary, however, if we wish to pass channels over channels, as we shall now see.

4 Channels and Ports

While the higher-order-concurrency model provides many-to-many channels, that is, there can be an arbitrary number of senders and receivers on a channel (Figure 3(a)), this is difficult to implement in a distributed system, and would be very inefficient. Hence, we provide many-to-one and one-to-many channels with classes `M2oChan` and `O2mChan`, respectively, which both extend the abstract class `Chan`. Using first-class events, one could build many-to-many channels (Figure 3(b)) and treat them as if they were a built-in feature, except that you would have to use different `transmit` and `receive` functions to create primitive events for them.

The advantage of one-to-many [many-to-one] channels is the notion of a fixed[2] *owner*, namely the unique sender [receiver]. The owner creates the first copy of the channel, and more copies are created by sending the channel to other threads. The *membership*

[2]There is no fundamental reason why the owner cannot be moved around dynamically; indeed, this is supported in some concurrent-programming languages such as Fortran-M [2]. However, it complicates the protocols, and is likely an inefficient and uncommon operation.

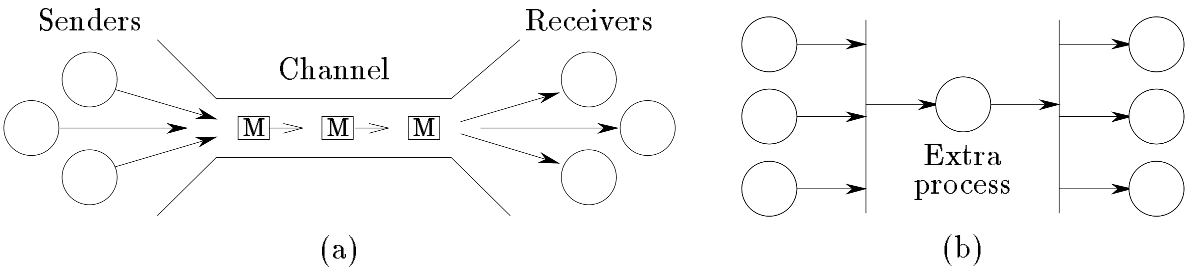

Figure 3: *(a) A many-to-many channel. Attempting to send on a channel will block until one of the receivers attempts to receive a message from the channel, and vice versa. When there are multiple willing senders and/or receivers, the behavior is non-deterministic, although we draw it as a queue here. (b) Implementing a many-to-many channel as a many-to-one and one-to-many channel with an extra process in between.*

of a channel is the set of all other threads that are "connected" to the channel (that is, they have copies of the channel).

There are three major properties that we want channels to possess:

1. The owner must maintain the membership of each channel. First, some of the protocols described in Section 5 require this information. Second, it is important to know when the membership is empty, because at this point the channel is considered *dead*, that is, the owner will never succeed at sending/receiving on it. We use this information to garbage-collect *threads* when they `sync` on an event that only involves dead channels, since it will necessarily wait forever; this is a feature of higher-order concurrency that greatly simplifies programming (an example is given is Section 6).

2. We must detect when the owner has discarded all of its references to the channel, because it means that the members cannot succeed in communicating over the channel. We call the channel dead in this situation as well, since it can no longer be used for communication. Hence, this information is necessary for garbage collection of threads.

3. There must be a mechanism for members to name the channel when communicating to the owner. This is required for protocol reasons; for example, if a thread wishes to send a message on the channel, it must tell the owner. It is important that the owner knows information about the channel given just its name, so that (for example) it can notify the sender if the channel is dead (i.e., the owner has discarded its references).

Let us first demonstrate why it is necessary to morph channels, even with a shared address space. If we do not, the owner and every member will have a reference to a common `Chan` object (Figure 4(a)). In Java, we can only detect when the object has zero references (by overriding the `finalize` method). Hence, conditions (1) and (2) are not satisfied; we can only detect when both the owner has discarded its references and there are no members.

Even with channel morphing, there is still a problem. As shown in Figure 4(b), condition (3) is not satisfied, since there is no way for the owner (thread a) to find its

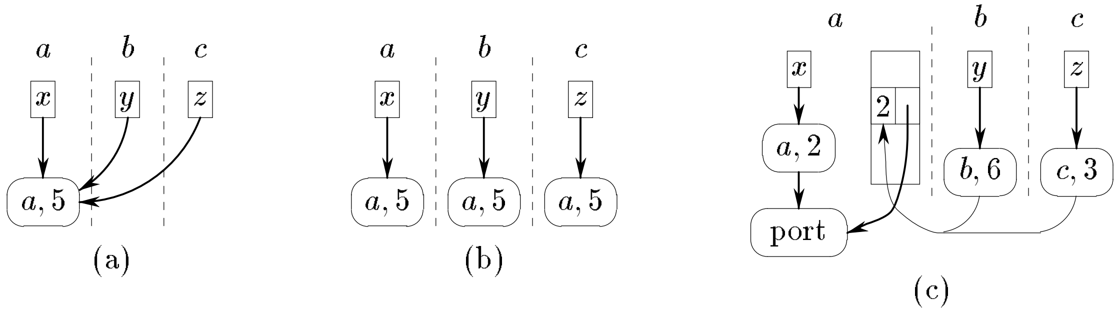

Figure 4: *Possible ways to organize channels. In this example, thread a is the owner of a channel with members b and c. Small rectangles denote variables, and rounded rectangles denote channels. Thick lines denote reference (pointer) association, whereas thin lines denote conceptual linking, that is, channels that know the port's id. (a) No morphing. (b) No ports. (c) Final design with an id-to-port mapping.*

copy of the channel using the id $(a, 5)$. We can add a hash table to provide such an association (as in Figure 4(c)), but this violates condition (2), since there is always a reference to the owner's copy of the channel (stored in the hash table).

Therefore, we need to split up the owner's copy of a channel into two objects, a `Chan` (as usual) and a `Port` (Figure 4(c)). The port maintains membership and persists until there are no copies left of the channel. Condition (3) is satisfied since there is an id-to-port mapping, control messages refer to the port id, and channels store their corresponding port id (and the owner thread). Garbage collection of the `Chans`, that is, the copies of the channel, can be detected, and will cause notifications to be sent to the `Port`. We can hence easily detect if the membership is empty, and can easily determine if the owner has discarded its copy, and therefore conditions (1) and (2) are satisfied. Note that each copy of the channel has its own id so that members can distinguish between requests from the owner even when there are multiple copies of the channel on the same thread.

One assumption that we make is that, if a thread sends a channel to its owner, the owner treats this new copy as if it was a member. This way, once the owner discards the initial copy of the channel, it will never get another one. Hence, the channel is permanently dead once the owner discards its copy, which makes it easy for a member to detect if the channel is dead (the member essentially asks the owner if it has discarded its copy yet).

4.1 Maintaining Channel Membership

Let us discuss the control messages needed to maintain channel membership. If a member's `Chan` is garbage collected, i.e., a member has discarded its copy of the channel, then the member sends a `REMOVE` message to indicate this. We cannot use such a simple method to add members when we communicate channels to other threads, however.

Suppose that threads only told the owner when they received channels. Then the following scenario leads to an unfortunate situation: the sending thread, currently the only member, sends the `Chan` and immediately discards it (causing a `REMOVE` message). Potentially, the `REMOVE` message arrives before the receiver sends its `ADD` message, meaning that the port thinks there are no members for a period of time. Hence, the port may be destroyed, even though there is still a member.

A similar race condition occurs if only the sending thread notifies the owner, assuming arbitrary network delays. Hence, both the sender and receiver have to notify the owner, in such a way that the owner ignores the second `ADD` message that arrives. This involves some precise manipulation of data structures that is purely an implementation detail. Details can be found in [1].

5 Events

Let us first overview the event combinators that we implement. As we have indicated, `choose` returns an event representing the non-deterministic choice of a list of events (either an array or a `java.util.Vector`). The remaining combinators, `wrap`, `guard`, and `wrapAbort`, are for constructing sequences.

`wrap` takes an event and a function with a single parameter and return value of type `java.lang.Object`, and returns a new event. Applying `sync` to the wrapped event corresponds to `sync`ing the subevent, and returning the function's value when given the subevent's result as a parameter. (The result of a `transmit` event is `null`, and the result of a `receive` event is the object received.) If we assume the function performs the "rest of the sequence," then we have implemented a sequence event, where we *commit* to completing the entire sequence once the first part (the subevent) completes. (The notion of commitment is important when the event is placed in a `choose`.)

`guard` takes a function and returns an event representing it. Applying `sync` to the "guarded function" causes the function to be called; the `guard` event is effectively replaced by the event that the function returns, for the duration of that call to `sync`. If the function spawns a thread to do the "previous part of the sequence" and returns the last part, then we commit only after the entire sequence completes. By mixing `wrap` and `guard`, we can place the *commit point* in an arbitrary position in a sequence of events [10].

Finally, `wrapAbort` takes an event and a function, and returns an event that is equivalent to the subevent, unless the event is not completed (e.g., it was in a `choose` event and a different subevent completed first), in which case a thread is spawned to evaluate the function. This is useful for representing non-atomic transactions that consist of a sequence of operations with a commit point near the end. Typically, a guarded function returns a `wrapAbort` event to clean up what it started in the background.

It should be obvious how to define interfaces that represent the closures (functions and state) given to the above combinators. An abort function is equivalent to a `ThreadCode` (Figure 1), and the others are similar.

We represent events by an abstract class `Event`, and represent primitive (communication) events by an abstract class `CommEvent`. The entire class hierarchy below `Event` is shown in Figure 5.

Every event type must define two routines: `possible` and `complete`. `possible` is called before the `sync` operation "starts" (i.e., before it tries to complete primitive events). For primitive events, this must add the event to a specified vector. `complete` is called after a primitive event completes; both a reference to the completed primitive event and its result are passed to the `complete` function. The return value is a pair of a boolean and an object. The boolean should be true if and only if the event is an ancestor of

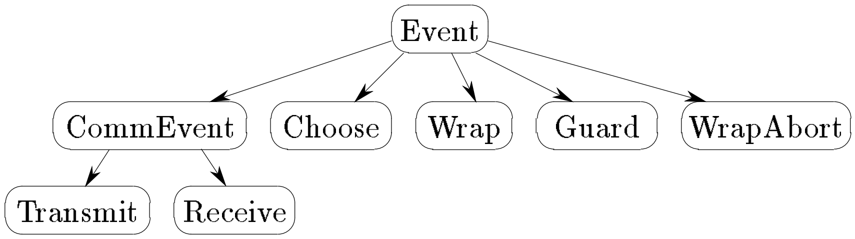

Figure 5: *The class hierarchy below* `Event`*.*

the completed primitive event. The object is the result computed so far. It is easy to define the described event types using `possible` and `complete`.

As mentioned above, the `sync` operation collects a list of the primitive events, that is, the leaves in the "event tree." A primitive event type must implement four additional operations, `notify`, `attempt`, `finish`, and `abort`, which correspond to phases in the protocol that is currently implemented. In the beginning, a `notify` message is sent, for each primitive event executed by member threads, to the channel's owner.

After each control message is received and processed, `sync` "attempts" each primitive event. For primitive events executed by owners, this succeeds if a `notify` message has been received (and it has not been cancelled with an abort message); in this case, the owner half-commits (i.e., it commits to determining if this transaction will succeed) and sends a `request` message. For other primitive events, `attempt` succeeds if a `request` message has been received, in which case the thread sends a `grant` message and fully commits. A `grant` message for an event e causes the owner to fully commit to e, whereas an `abort` message causes it to decommit from a previous half-commitment to e (if one exists). Fully committing to e corresponds to `abort`ing other primitive events and calling `finish` on e (which sends or receives a user-level message). When a thread is half-committed, it does not `attempt` any primitive events.

This protocol is summarized in Figure 6. It is subject to deadlock if the communication pattern is cyclic, and involves three to four control messages for each synchronous communication (two is clearly optimal). We have designed a deadlock-free version, a version that uses only two message cycles per synchronous communication, and a deadlock-free version of the latter protocol. Unfortunately, we do not have sufficient space to describe them in this paper; we expect to have a separate paper ready soon.

6 Example

In this section, we show how general semaphores can be easily implemented in the described system, with events to represent the up and down operations (Figure 7). With this we can perform a non-deterministic choice between downing two semaphores. We could also, for example, non-deterministically either down two semaphores in sequence, or down a different two semaphores in sequence, using `wrapAbort` to up the first semaphore in a failed sequence.

We shall go over the code itself and illustrate its features as we go. A semaphore object (`Semaphore`) is implemented using a thread (`SemaphoreThread`) that is dedicated to maintaining the semaphore count. The `Semaphore` constructor demonstrates how the "first" channels are created (over which we can pass other channels): if the `true` flag

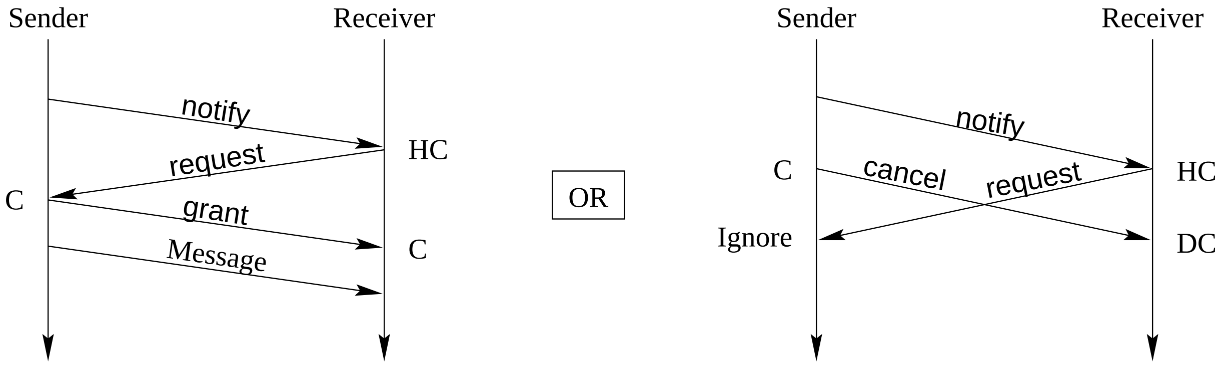

Figure 6: *The successful and unsuccessful scenarios in the described protocol for a many-to-one channel. HC, C, and DC stand for half-commit, commit, and decommit, respectively.*

is passed to `HOC.spawn`, then a channel to and from the child is created, and returned as an array of two `Chans`. The child can retrieve its copies of the channels by calling `HOC.getParentChans`, which returns a (morphed copy of) the same array. The `Semaphore` constructor uses the channel from the child (stored in entry 1) to obtain two many-to-one channels that the `SemaphoreThread` creates, which are stored in the private variables of the `Semaphore` object.

The up and down operations on the `Semaphore` correspond to sending a (null) message on the up and down channels that the `SemaphoreThread` continually tries to receive on. The `SemaphoreThread` uses a `Wrap` event so that `sync` returns the `Integer` 1 when up succeeds, and -1 when down succeeds. Normally, it will `sync` on a non-deterministic choice between the two possibilities of up and down, but when the semaphore value is zero, it only allows an up operation. By splitting the up and down operations into two channels, it is simple to disable the down operation and block threads that attempt to down, without requiring a complex request-reply structure.

One useful property of the `SemaphoreThread` is that it will automatically disappear when no copies of the corresponding `Semaphore` object exist. Thus, garbage collection of threads avoids the need of sending a "kill" message to destroy the `SemaphoreThread` once it is no longer needed.

We chose this example because it shows how easy it is to code with higher-order concurrency. Recently, semaphores were added to the KRoC occam compiler because they are annoying to implement in occam, and difficult to implement efficiently [14]. With higher-order concurrency, it is easy for the user to build concurrency structures and use them as if they were built-in. The distributed-semaphore implementation in Figure 7 is also likely nearly as efficient as possible.

The Java package is somewhat more difficult to use than CML because users must implement `Morphing` for nearly every concurrent structure they build. As we noted earlier, this is a necessary inconvenience, since we need to morph channels that may be nested within structures that the user communicates to other threads. One alternative is to access information available within the Java Virtual Machine (JVM) and traverse the data structures this way. We believe that this is not the right approach unless such information can be accessed in a standard way; this is unlikely to happen, since it endangers data encapsulation.

```
import hoc.*;
public class Semaphore implements Morphing {
   private Chan downCh, upCh;
   public Semaphore (int value) {
      Chan[] child = HOC.spawn (new SemaphoreThread (value), true);
      downCh = child[1].accept ();
      upCh = child[1].accept ();
   }
   private Semaphore (Chan downCh, Chan upCh) {
      this.downCh = downCh;
      this.upCh = upCh;
   }
   public Event down () {
      return new Transmit (downCh, null);
   }
   public Event up () {
      return new Transmit (upCh, null);
   }
   public void preMorph (Thread dest) {
      HOC.preMorph (downCh, dest);
      HOC.preMorph (upCh, dest);
   }
   public Morphing morph (Thread src) {
      return new Semaphore (HOC.morph (downCh, src), HOC.morph (upCh, src));
   }
}
final class SemaphoreThread implements ThreadCode {
   int value;
   SemaphoreThread (int value) {
      this.value = value;
   }
   public void threadCode () {
      Chan[] parent = HOC.getParentChans ();
      Chan downCh = new M2oChan ();
      Chan upCh = new M2oChan ();
      parent[1].send (downCh);
      parent[1].send (upCh);
      Event down = new Wrap (new Receive (downCh), new ReturnInt (-1));
      Event up = new Wrap (new Receive (upCh), new ReturnInt (1));
      Event both = new Choose (down, up);
      while (true) {
         Integer i = (Integer) sync ((value <= 0 ? up : both));
         value += i.intValue ();
      }
   }
}
final class ReturnInt implements WrapFunc {
   private int i;
   ReturnInt (int i) {
      this.i = i;
   }
   Object wrapFunc (Object obj) {
      return new Integer (i);
   }
}
```

Figure 7: *Implementation of semaphores using higher-order-concurrency features.*

7 Conclusion

In this paper we have described the design of an implementation of higher-order concurrency in Java. It provides first-class events and channels, and automatic garbage collection of threads. We have achieved an ease-of-use similar to that in Concurrent ML. The result is a concurrent-programming language where users can build up their own constructs and use them as if they were built-in. It is easily extendible to use a distributed system, unlike CML.

While Java programs are currently slow, the package is still very useful. Concurrency is a highly useful paradigm for building interactive systems [9], which is a major focus with Java. A version supporting threads running on separate computers is necessary when the program is run in a distributed environment, for example groupware applications.

The speed of concurrency support is clearly not relevant, since it depends on the underlying Java Virtual Machine implementation, and currently Java cannot compare to other run-time environments (ignoring concurrency). However, the networking overhead is of particular concern, especially when we consider slow networks. The currently implemented protocol requires three to four messages per user-level message, but we have developed a protocol that achieves two message cycles per user-level message, which is optimal. Hence, the package should be very efficient in a distributed environment.

Acknowledgment

We wish to thank David Taylor for valuable comments on the paper.

References

[1] Erik D. Demaine. Higher-order concurrency in PVM. In *Proceedings of the Cluster Computing Conference*, Atlanta, Georgia, March 1997. World Wide Web. http://www.mathcs.emory.edu/~ccc97.

[2] Ian Foster and K. M. Chandy. Fortran M: A language for modular parallel programming. *Journal of Parallel and Distributed Computing*, 26(1):21–35, 1995.

[3] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine — A User's Guide and Tutorial for Networked Parallel Computing*. The MIT Press, Cambridge, MA, 1994.

[4] James Gosling and Henry McGilton. The Java language environment. White paper, Sun Microsystems, Inc., 1996.

[5] G. H. Hilderink, J. F. Broenink, W. Vervoort, and A. W. P. Bakkers. Communicating Java threads. In *Proceedings of the Parallel Programming and Java Conference (WoTUG20)*. IOS Press (Netherlands), April 1997.

[6] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[7] Frederick Knabe. A distributed protocol for channel-based communication with choice. Technical Report ECRC-92-16, European Computer-Industry Research Centre, München, Germany, 1992.

[8] Clifford Dale Krumvieda. Distributed ML: Abstractions for efficient and fault-tolerant programming. Technical Report TR93-1376, Cornell University, August 1993. PhD thesis.

[9] John H. Reppy. CML: A higher-order concurrent language. In *Proceedings of the SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 293–305, Toronto, June 1991.

[10] John H. Reppy. *Higher-order concurrency.* PhD thesis, Dept. of Computer Science, Cornell University, June 1992.

[11] John H. Reppy. Concurrent ML: Design, application, and semantics. In *Programming Concurrency, Simulation and Automated Reasoning*, Lecture Notes in Computer Science, Berlin, 1993. Springer-Verlag.

[12] SGS-THOMSON Microelectronics Limited. *occam 2 Reference Manual.* Prentice Hall International Ltd., 1988.

[13] P. H. Welch and D. C. Wood. Higher levels of process synchronisation in occam. World Wide Web. http://www.hensa.ac.uk/parallel/occam/projects/occam-for-all/hlps/.

[14] D. C. Wood and P. H. Welch. The Kent Retargetable occam Compiler. In B. O'Neill, editor, *Parallel Processing Developments, Proceedings of the 19th WoTUG Technical Conference*, pages 143–166, Nottingham-Trent University, March 1996. IOS Press (Netherlands).