Kill-Safe Synchronization Abstractions

> "Well, it just so happens that your friend here is only mostly dead.
> There's a big difference between mostly dead and all dead."
> – Miracle Max in *The Princess Bride*

Matthew Flatt
University of Utah

Robert Bruce Findler
University of Chicago

Abstract

When an individual task can be forcefully terminated at any time, cooperating tasks must communicate carefully. For example, if two tasks share an object, and if one task is terminated while it manipulates the object, the object may remain in an inconsistent or frozen state that incapacitates the other task. To support communication among terminable tasks, language run-time systems (and operating systems) provide kill-safe abstractions for inter-task communication. No kill-safe guarantee is available, however, for abstractions that are implemented outside the run-time system.

In this paper, we show how a run-time system can support new kill-safe abstractions without requiring modification to the run-time system, and without requiring the run-time system to trust any new code. Our design frees the run-time implementor to provide only a modest set of synchronization primitives in the trusted computing base, while still allowing tasks to communicate using sophisticated abstractions.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Concurrent programming structures*; D.4.1 [**Operating Systems**]: Process Management—*Synchronization*

General Terms

Languages, Design

1 Introduction

Most modern programming languages offer support for multiple tasks in the form of threads. Support for task *termination* is less widely implemented and generally less understood, but no less useful to programmers. The designers of Java, for example, understood the need for termination, and they included `Thread.stop` and `Thread.destroy` in the language. Flaws in the specification of `Thread.stop` forced its withdrawal, however, and `Thread.destroy` has never been implemented. Meanwhile, various extensions of Java have provided termination in a more controlled form [1, 2, 9, 10, 12], and termination of Java tasks is a driving goal of the new JSR-121 standard [24].

Termination in any language becomes troublesome when tasks share objects. Two tasks may share a queue, for example, and they may require that terminating one task does not corrupt or permanently freeze the queue for the other task. In other words, the tasks require a *kill-safe* queue. Just as *thread-safe* means that an abstraction's operations appear atomic in the presence of interleaved threads of execution, *kill-safe* means that abstraction's operations appear atomic in the presence of thread termination.

Java extensions such as JSR-121 do not allow a programmer to implement kill-safe abstractions. Instead, such extensions restrict sharing among terminable tasks to objects that are managed by the run-time system "kernel," so that only the kernel needs to implement kill safety. In particular, if one task is terminated while manipulating a kernel-maintained queue, the kernel delays termination until it can leave the queue in a consistent state. On the one hand, when the kernel implementor's work is complete, application programmers need not worry about termination when using shared objects. On the other hand, application programmers are restricted to the kernel's abstractions for reliable communication.

In this paper, we show how to extend a run-time system (once and for all) so that programmers can define kill-safe abstractions outside the kernel. Furthermore, with our design, the set of cooperating tasks that share an object need not be defined in advance, and the cooperating tasks need not trust each other; the tasks must trust only the implementation of the shared object.

Our design builds on the observation that termination of a thread is much like indefinite suspension of the thread. By creating a thread to manage a particular synchronization abstraction, and by employing our new primitives to protect the thread, a programmer can ensure that an abstraction instance is suspended when it might otherwise be killed. In other words, when the instance is "killed" as part of a task termination, it turns out to be "only mostly dead," and a surviving task that shares the instance can resurrect it. This technique succeeds because our primitives allow a manager thread to preserve its instance's consistency across suspends and resumes, and this consistency is the essence of kill-safety. At the same time, a rogue task cannot escape termination through accomplices of equal stature, because a resurrected task gains no more privileges than those of its resurrector.

PLDI'04, June 9–11, 2004, Washington, DC, USA.
Copyright 2004 ACM 1-58113-807-5/04/0006 ...$5.00

The latest version of MzScheme [5] implements our design for kill-safe abstractions. MzScheme builds on the primitives of Concurrent ML [21], which enable a programmer to construct synchronization abstractions that have the same first-class status as built-in abstractions. By starting with Concurrent ML's primitives, we ensure that our model and implementation cover a large class of abstractions. In general, MzScheme can express any abstraction that is expressible in Concurrent ML, and we believe that any such abstraction can be made kill-safe.

Section 2 describes our model of task management and kill-safe abstractions. Section 3 presents MzScheme's task-control mechanisms, and Section 4 sketches the implementation of a kill-safe queue. Section 5 reviews MzScheme's embedding of Concurrent ML primitives, mainly for readers who are not familiar with Concurrent ML. Section 6 presents a complete and realistic queue implementation using the Concurrent ML primitives. This full implementation motivates a remaining detail of our design that is covered in Section 7. Section 8 explores in more detail some of our design choices. Sections 9 and 10 summarize related work and conclude.

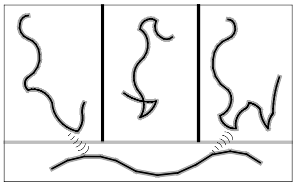

Figure 1.
OS (e.g., Unix)

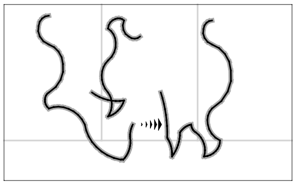

Figure 2.
Safe run-time (e.g., JVM)

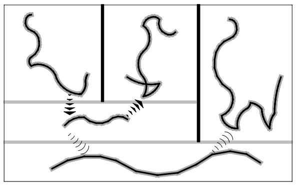

Figure 3.
Nested tasks

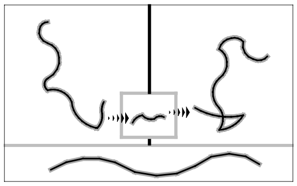

Figure 4.
Kill-safe (MzScheme)

2 Motivating Example

Consider the implementation of a web server with servlets. The system administrator allows certain users to implement servlets, but the administrator reserves the right to terminate any servlet-based session that appears to misbehave (e.g., consumes too much memory).

For various reasons, the servlets for two active sessions might discover each other and wish to communicate. For example, the sessions may share a collaborative document whose implementation is specific to the pair of servlets. The servlet tasks trust the document implementation, but they cannot trust each other to survive, because the server might terminate one or the other session at any time. In short, the servlets need to share a kill-safe document.

2.1 Existing Approaches

If the server and servlets are implemented as processes in a conventional operating system, then they are strictly isolated, as in Figure 1. Each task occasionally communicates across the task boundary (drawn as a thick gray line in the figure) to the kernel task, but a task does not communicate directly to other tasks or cross into another task's space (as indicated in the figure by thick black lines). As a result, a misbehaving session is terminated easily. If two servlets need to communicate, however, they must use the primitives provided by the kernel (depicted in the figure by a telecast from the left task to the right task via the kernel task). The kernel's set of primitives is unlikely to include a document abstraction, so this architecture does not meet the needs of the servlet implementors; the document must reside in one servlet or another, so it will be terminated with the servlet.

If the server and servlet tasks are implemented as threads in a safe programming language, the server can rely on abstract datatypes to protect its data structures from misbehaving servlets. As illustrated in Figure 2, task boundaries fade away, and servlets are free to set up communication abstractions that better match their needs (as depicted in the figure by a close-range telecast in a task's space). With no boundaries between tasks, however, there is no guarantee that a servlet will not be terminated while it is manipulating a shared document, which means that the document is not kill-safe.

A nested-task architecture can avoid the fixed set of kill-safe communication primitives while providing control. As illustrated in Figure 3, nesting effectively allows a programmer to extend the kernel's set of communication primitives by implementing a new nested kernel. In this architecture, the servlets that share a document must run as sub-tasks, controlled by a master servlet that implements a document. The server administrator's needs are met, because servlets can be controlled. The servlet implementors' needs are met for small groups of servlets, but not for large groups. Sessions must be grouped for communication, so the more each session communicates with other sessions, the more all sessions become grouped together and controlled by a monolithic master servlet, which defeats the purpose of servlets.

Figure 4 illustrates an architecture that meets the needs of both the server administrator and servlet implementors. The small gray box in the middle of the figure represents a kill-safe abstraction that the kernel need not trust, but that can be loaded as needed by cooperating servlets.

This pattern cannot be implemented in existing systems—not even systems that support nested tasks or shared memory. With nested tasks, the gray-box task must be a sub-task of the left or right task, and therefore subject to termination along with its parent task. With shared memory, either the left or right task must cross into the small box, and it might get terminated while manipulating structures there. Finally, the problem cannot be solved by allowing the two tasks to enter the gray box and execute atomically, since the two tasks could then collude to starve the rest of the system. For the same reason, the termination of a task cannot be delayed until it leaves the gray box.

2.2 Our Solution

Our solution requires either the left or right task to create the gray-box task initially as a sub-task. Later, when the other task gains access to the gray box, it "promotes" the box as its own sub-task. As a result, the gray-box task becomes more resistant to termination than either the left or right task alone, though no more resistant than than the tasks combined.

If the left task is terminated, the box task is merely suspended. When the right task later accesses the box, it resumes the box task, and safely continues. If both the left and right tasks are terminated, the box task becomes both suspended and inaccessible, and therefore effectively terminated. Thus, the system as a whole can protect itself against malicious (or buggy) collaborations by terminating both collaborators. Meanwhile, communication channels that are provided by the box protect the left and right tasks from each other, much as kernel-supplied channels protect tasks from each other.

This solution combines three lines of work:

- Our earlier work for nested tasks (i.e., the gray boxes) in MzScheme [6] provides the base task model.
- The primitives of Concurrent ML [21] enable tasks to implement communication abstractions that have the same first-class status as kernel abstractions.
- New primitives allow modifying the task hierarchy without threatening bystander tasks.

Technically, our extension of existing task and concurrency models is modest, and the pattern that we show for kill-safety is a straightforward extension of Concurrent ML patterns—once our new primitives are given. Our contribution lies in the synthesis of these elements to provide a simple and expressive mechanism for kill-safe abstractions. To our knowledge, it is the first such combination to solve the servlet problem described in the previous section (and other problems like it).

The servlet problem closely mirrors a problem in the implementation of DrScheme's help system. Help pages are written in HTML, and the help system works by running the PLT web server [7] plus a browser that is connected to the server. Although the server and browser could execute as different OS tasks and communicate through TCP sockets (as web servers and browsers normally do), production OSes make this architecture surprisingly fragile. Therefore, DrScheme implements its own browser, and it runs a web server and a browser directly in its own MzScheme virtual machine. The two parties communicate through a socket-like abstraction whose core is an asynchronous buffered queue. The MzScheme kernel provides no such abstraction, and due to the way that DrScheme is organized, adding an intermediate kernel layer for the web browser and server would be prohibitively difficult. Furthermore, both the server and browser take advantage of termination for internal tasks (e.g., to cancel a browser click), and those tasks are involved in communication. Such terminations then wreak havoc with the queue implementation.

In the new MzScheme, small adjustments to the queue implementation make it kill-safe as well as thread-safe. The help system now works reliably with no additional changes to DrScheme, the server, or the browser. At the same time, the kill-safe abstraction does not (and cannot) compromise task control. In particular, when testing DrScheme within DrScheme, we can terminate the inner DrScheme, and it reliably terminates the associated help system, including any queue-manager threads.

Kill-safe buffered queues are merely the tip of the abstraction iceberg, but we use this example in the following sections to illustrate essential techniques for kill-safe abstractions. See *Concurrent Programming in ML* by John Reppy [21] for many other example abstractions that can be made kill-safe using our technique.

2.3 Abandoned Approaches

Before arriving at MzScheme's current primitives for kill-safe abstractions, we explored two main alternatives.

Restricted atomic sections: As mentioned at the end of Section 2.1, the kernel cannot allow a task to execute arbitrary code atomically, otherwise it might starve the rest of the system. The kernel might, however, allow a task to execute atomically for a short period of time, or to execute code that provably terminates in a short time. We abandoned this approach, because we could not find a way to define "time" that makes sense to a programmer. Dynamic measurements in terms of clock ticks or program operations were too sensitive to small program changes, and static methods, based on limiting the code to certain primitive operations, proved insufficiently expressive.

Transactions with rollbacks and commit points: Although a transaction-oriented approach looked promising, and although Rudys and Wallach have made progress in this direction [23], synchronous channels encode directly the kind of transactions that seem most useful for our purposes. We therefore abandoned this direction and embraced the Concurrent ML primitives as our base.

3 Task Control in MzScheme

To provide a more concrete explanation of kill-safe abstractions, we must first introduce some terminology and basic task constructs. In this section, we present the constructs as they are implemented in MzScheme.[1]

MzScheme's support for tasks encompasses threads of execution, task-specific state, per-task resource control, per-task GUI modalities, and more. Instead of supplying a monolithic "process" construct, however, MzScheme supports the many different facets of a process through many specific constructs [6]. With respect to kill-safe abstractions, the only relevant facets are threads of execution, resource control as it relates to thread termination, and thread-specific state as it relates to determining the resource controller.

3.1 Threads

The `spawn` procedure takes a function of no arguments and calls the function in a new thread of execution. The thread terminates when the function returns. Meanwhile, `spawn` returns a thread descriptor to its caller.

```
(define t1 (spawn (lambda () (printf "Hello"))))
(define t2 (spawn (lambda () (printf "Nihao"))))
;; prints "Hello" and "Nihao", possibly interleaved
```

3.2 Resource Control

A *custodian* is a resource controller in MzScheme. Whenever a thread is created, a network socket is opened, a GUI window is created, or any other primitive resource is allocated, it is placed under the control of the *current custodian*. A thread can create and install a new custodian, but the newly created custodian is a *sub-custodian* that is controlled by the current custodian.

The only operation on a custodian is `custodian-shutdown-all`, which suspends all threads, closes all sockets, destroys all GUI windows, etc. that are controlled by the custodian, and prevents any further resources allocated to the custodian. The shut-down

[1] Some functions are defined in the `(lib "cml.ss")` module.

command also propagates to any controlled sub-custodians. After a custodian is shut down, it drops references to primitive resources, and the memory for such objects can be reclaimed by the garbage collector.[2] Unlike other objects, a thread can have multiple custodians (added with thread-resume, which we describe later, in Section 3.3). A thread is suspended only when all of its controlling custodians are shut down.

The make-custodian procedure creates a new custodian. The parameterize form with current-custodian sets the current custodian during the evaluation of an expression. In particular, parameterize can be used to install a custodian while creating a new thread, and the new thread is then controlled by the custodian.

```
(define cust (make-custodian))
(define (lots-of-work) ....)
(parameterize ([current-custodian cust])
  (spawn lots-of-work))
(custodian-shutdown-all cust) ; stops lots-of-work
```

When a thread is created, it inherits the current custodian from its creating thread. Thus, assuming that *lots-of-work* is not closed over the original custodian, (custodian-shutdown-all *cust*) reliably terminates the task that executes *lots-of-work*—no matter how many threads that it spawns, sockets that it opens, GUI windows that it creates, or sub-custodians that it generates.

The current custodian for a particular thread is not necessarily the same as the thread's controller. The thread's controlling custodian is determined when the thread is spawned, but the current custodian (for controlling newly allocated resources) can be changed by the thread at any time through parameterize.

3.3 Thread Resumption

The thread-resume primitive is the key to implementing kill-safe abstractions. Given a single thread argument, MzScheme's thread-resume function resumes the thread if it is suspended:

```
(define t (spawn lots-of-work))
(thread-suspend t) ; suspends lots-of-work
(thread-resume t) ; resumes lots-of-work
```

The thread-resume function can only resume a thread that has a custodian. If a thread's only custodian has been shut down, the resume request has no effect.

```
(define cust (make-custodian))
(define t (parameterize ([current-custodian cust])
            (spawn lots-of-work)))
(custodian-shutdown-all cust) ; stops lots-of-work
(thread-resume t) ; doesn't resume lots-of-work
```

The thread-resume function accepts an optional second argument to provide a new custodian to the thread before attempting to resume it. The optional argument can be either a custodian, in which case it is added to the thread's set of controllers, or another thread, in which case this other thread's custodians are added to the set of controllers for the first thread.

```
(define cust (make-custodian))
(define cust2 (make-custodian))
(define t1 (parameterize ([current-custodian cust1])
             (spawn lots-of-work)))
(define t2 (parameterize ([current-custodian cust2])
             (spawn lots-of-work)))
(custodian-shutdown-all cust1) ; suspends t1
(thread-resume t1) ; doesn't resume t1
(thread-resume t1 t2) ; resumes t1, adds cust2
(custodian-shutdown-all cust2) ; stops t1 and t2
```

If the second argument to thread-resume is a thread, then in addition to adding a custodian to the first thread, the resume yokes the first thread to second as follows:

- Whenever the second thread is resumed, the first thread is also resumed.
- Whenever the second thread acquires a new custodian, the first thread also acquires the custodian.

The overall effect of (thread-resume *t1 t2*) is to ensure that *t1* survives at least as long as *t2*—assuming that *t1* is suspended only indirectly via custodian-shutdown-all. This effect holds because a custodian-based suspension of *t1* will necessarily also suspend *t2*, since *t1* can only run out of custodians if *t2* also runs out. Meanwhile, if both are suspended and *t2* is resumed, then so is *t1*.

The two-argument thread-resume allows two threads *t1* and *t2* share an object that embeds a thread *t*. In that case, (thread-resume *t t1*) plus (thread-resume *t t2*) makes the embedded thread *t* act as though it has no controlling custodian, at least as far as *t1* and *t2* can tell, since *t* runs whenever *t1* or *t2* runs. This combination is the key to implementing a kill-safe version of the queue abstraction.

Furthermore, the two-argument thread-resume does not enable a set of processes to conspire and escape termination. The processes may share their custodians, but after all of the custodians are shut down, no thread created by the processes can execute without outside help. Indeed, in the absence of outside references, the process's threads will be garbage-collected.

4 Sketch for a Kill-Safe Queue

The primitive synchronization abstraction in MzScheme is a synchronous channel [11, 15], which allows two tasks to rendezvous and exchange a single value. This built-in abstraction is kill-safe, in that the termination of a task on one end of the channel does not endanger the task on the other end of the channel—though, obviously, no further communication will take place.

In this section, we consider the implementation of a kill-safe queue (a.k.a. asynchronous buffered channel). Values sent into the queue are parceled out one-by-one to receivers. A send to a queue never blocks, except to synchronize access to the internal list of queued items. A receive blocks only when the queue is empty, or to synchronize internal access.

```
(define q (queue))
(queue-send q "Hello")
(queue-send q "Bye")
(queue-recv q) ; ⇒ "Hello"
(queue-recv q) ; ⇒ "Bye"
```

Figure 5 sketches an implementation of queues. Each queue consists of a channel *in-ch* for putting items into the queue, a channel *out-ch* for getting items out of the queue, and a manager thread running *serve* to pipe items from *in-ch* to *out-ch*.

Even with only this sketch, we can see that the queue is not yet kill-safe. Suppose that a thread *t1* creates *q* by calling (*queue*).

[2] Shutting down a resource removes internal references, which frees most of the memory associated with the resource.

Suppose further that *t1* is controlled by custodian *c1*, and that *q* is made available to a thread *t2* controlled by custodian *c2*:

```
;; t1, controlled by c1  | ;; t2, controlled by c2
(define q (queue))       |
(send-to-other q)        | (define q (get-from-other))
;; suspend threads of c1 |
                         | ;; stuck — q was suspended by c1
                         | (queue-send q 10)
```

Since *t1* creates *q*, the queue's internal thread *mgr-t* is controlled by *c1*. Suspending all threads of *c1* suspends both *t1* and *mgr-t*. As a result, the send in *t2* gets stuck—and a send into a buffered queue should never get stuck.

We might attempt to fix the problem with a resume of the queue's thread before each queue operation. A simple resume is not enough, however; between the time that *t2* resumes *mgr-t* and the time that it performs its action on the queue, another thread might re-suspend all threads of *c1*, thus re-suspending *mgr-t*.

The solution is `(thread-resume mgr-t t2)`, which not only resumes the queue thread, but also adds *t2*'s custodian, *c2*, as a controller of *mgr-t*. Afterward, a mere suspension of *c1*'s threads does not suspend *mgr-t*, since it is also controlled by *c2*.

Since *mgr-t* is not accessible outside the queue implementation, it can be suspended only by shutting down both of its custodians, *c1* and *c2*. In that case, then both *t1* and *t2* will be suspended as well as *mgr-t*—which is precisely the desired behavior if the custodians were shut down to terminate tasks *t1* and *t2*. In other words, *mgr-t* acquires no more priviledge to run than the sum of *t1* and *t2*'s priviledges.

If the suspension is not intended to terminate the task, and if *t2* is later resumed, then *mgr-t* is also resumed, due to the chaining installed by `(thread-resume mgr-t t2)`. More generally, by guarding each queue operation with `(thread-resume mgr-t (current-thread))` we ensure that *mgr-t* runs whenever a queue-using thread runs. Figure 6 shows revisions of the queue implementation with these guards, which make it kill-safe.

This example demonstrates both how kill-safe abstractions are possible, and how abstractions can be made kill-safe with relative ease. Nevertheless, it does not demonstrate the full power of our primitives for defining kill-safe abstractions. For such a demonstration, we must introduce MzScheme's embedding of the Concurrent ML primitives, so that we can build more flexible abstractions.

5 Review of Concurrent ML

This section provides a brief tutorial on MzScheme's embedding of the Concurrent ML [21] primitives. The tutorial is intended mainly for readers who are unfamiliar with Concurrent ML.

The primitives support synchronization among tasks via first-class events. A few kinds of events are built in, such as events for sending or receiving values through a channel. More importantly, the primitives enable the construction of entirely new kinds of events that have the same first-class status as the built-in events.

- `sync` : α-`event` → α

The `sync` procedure takes an *event* and blocks until the event is ready to supply a value. Some primitives provide a source of events. For example, `thread-done-evt` takes a thread descriptor and returns an event that is ready (with a void value) when the thread has terminated.

```
;; thread-done-evt : thread → void-event

(define t1 (spawn (lambda () (printf "Hello"))))
(define t2 (spawn (lambda () (printf "Nihao"))))
(sync (thread-done-evt t1)) ; waits until t1 is done
(sync (thread-done-evt t2)) ; waits until t2 is done
(printf "Bye")
;; prints “Hello” and “Nihao” interleaved, then “Bye”
```

```
;; queue : → α-queue
;; queue-send : α-queue α → void
;; queue-recv : α-queue → α

;; Declare an opaque q record with two fields; the constructor
;; is make-q, and the field selectors are q-in-ch and q-out-ch
(define-struct q (in-ch out-ch))
;; make-q : α-channel α-channel → α-queue

(define (queue)
  (define in-ch (channel)) ; to accept sends into queue
  (define out-ch (channel)) ; to supply recvs from queue
  ;; A manager thread loops with serve
  (define (serve items)
     ;; Handle sends and recvs
     ....
     ;; Loop with new queue items:
     (serve new-items))
  ;; Create the manager thread
  (spawn (lambda () (serve (list))))
  ;; Return a queue as an opaque q record
  (make-q in-ch out-ch))

(define (queue-send q v)
  ;; Send v to (q-in-ch q)
  ....)

(define (queue-recv q)
  ;; Receive from (q-out-ch q)
  ....)
```

Figure 5. Implementation sketch for a queue

```
(define-struct q (in-ch out-ch mgr-t))
;; make-q : α-channel α-channel thread → α-queue

(define (queue)
  ....
  ;; Create a manager thread
  (define mgr-t (spawn (lambda () (serve (list)))))
  ;; The q record now refers to the manager thread
  (make-q in-ch out-ch mgr-t))

(define (queue-send q v)
  ;; Make sure the manager thread runs
  (thread-resume (q-mgr-t q) (current-thread))
  ;; Send v to (q-in-ch q)
  ....)

(define (queue-recv q)
  ;; Make sure the manager thread runs
  (thread-resume (q-mgr-t q) (current-thread))
  ;; Receive from (q-out-ch q)
  ....)
```

Figure 6. A kill-safe queue, revises Figure 5

- `channel` : → α-`channel`
  `channel-recv-evt` : α-`channel` → α-`event`
  `channel-send-evt` : α-`channel` α → `void-event`

The `channel` procedure takes no arguments and returns a channel descriptor. A channel's only purpose is to generate events; the `channel-recv-evt` and `channel-send-evt` procedures create events for receiving values from the channel and sending values into the channel, respectively. The result of a receive event is a value sent through the channel, and the result of a send event is void. A send event is created with a specific value to put into the channel, and the event is ready only when a receive event can accept the value simultaneously. Similarly, a receive event is ready only when a send event can provide a value simultaneously.

```
(define c (channel))
(spawn (lambda () (sync (channel-send-evt c "Hello"))))
(sync (channel-recv-evt c)) ; ⇒ "Hello"
```

Multiple threads can attempt to send or receive through a particular channel concurrently. In that case, the system selects threads arbitrarily (but fairly) to form a send–receive pair.

```
(define c (channel))
(spawn (lambda () (sync (channel-send-evt c "Hello"))))
(spawn (lambda () (sync (channel-send-evt c "Nihao"))))
(sync (channel-recv-evt c)) ; ⇒ "Hello" or "Nihao"
(sync (channel-recv-evt c)) ; ⇒ the other string,
                            ;    "Nihao" or "Hello"
```

- `choice-evt` : α-`event` ... α-`event` → α-`event`

The `choice-evt` procedure takes any number of events and combines them into a single event. The combining event is ready when one of the original events is ready. If multiple events are ready, one is chosen arbitrarily (but fairly), and the value produced by the combining event is the value produced by the chosen event.

```
(define c1 (channel))
(define c2 (channel))
(spawn (lambda () (sync (channel-send-evt c1 "Hello"))))
(spawn (lambda () (sync (channel-send-evt c2 "Nihao"))))
(define cc (choice-evt (channel-recv-evt c1)
                       (channel-recv-evt c2)))
(sync cc) ; ⇒ "Hello" or "Nihao"
(sync cc) ; ⇒ the other string, "Nihao" or "Hello"
```

In the above example, even if both sending threads are ready when the main thread first calls `sync`, only one receive event in *cc* is chosen, and so it is matched with only one sending thread. The other sending thread remains blocked until the second `(sync cc)`.

- `wrap-evt` : α-`event` (α → β) → β-`event`

The `wrap-evt` function takes an event and a transformer procedure of one argument, and it produces a new event. The new event is ready when the given event is ready, and its value is the result of the transformer procedure applied to the original event's value.

```
(define c1 (channel))
(define c2 (channel))
(spawn (lambda () (sync (channel-send-evt c1 "Hello"))))
(spawn (lambda () (sync (channel-send-evt c2 "Nihao"))))
(sync (choice-evt
       (wrap-evt (channel-recv-evt c1)
                 (lambda (x) (list x "from 1")))
       (wrap-evt (channel-recv-evt c2)
                 (lambda (x) (list x "from 2")))))
;; ⇒ (list "Hello" "from 1") or (list "Nihao" "from 2")
```

- `guard-evt` : (→ α-`event`) → α-`event`

An event created by `guard-evt` encapsulates a procedure that is called when `sync` is applied to the event. The procedure's result is an event to use to in place of the guard event for the `sync`. For example, assume that `current-time` produces the current time, and that `time-evt` produces an event that is ready at a given absolute time. Then, `guard-evt` can be used to construct a timeout event.

```
;; current-time : → num
;; time-evt : num → event

(define one-sec-timeout
  (guard-evt (lambda ()
               (time-evt (+ 1 (current-time))))))
(sync one-sec-timeout) ; ⇒ void, one second later
(sync one-sec-timeout) ; ⇒ void, another second later
```

The result from `guard-evt` might be best described as an "event generator" instead of an event, but this generator can be used anywhere than an event can be used. Event generation is important for *one-sec-timeout*, which must construct an alarm time based on the time that *one-sec-timeout* is used, not when *one-sec-timeout* is created.

- `nack-guard-evt` : (`void-event` → α-`event`) → α-`event`

The `nack-guard-evt` function generalizes `guard-evt`. For `nack-guard-evt`, the given guard procedure must accept a single argument. The argument is a "Negative ACKnowledgment" event that becomes ready if the guard-generated event is not chosen by `sync`—usually because the event is combined with others using `choice-evt`.

```
(sync (choice-evt
       (wrap-evt one-sec-timeout
                 (lambda (void) "Hello"))
       (nack-guard-evt
         (lambda (nack)
           ;; Start a thread to watch nack
           (spawn (lambda ()
                    (sync nack) (printf "nack")))
           ;; This event is never ready
           (channel-recv-evt (channel)))))) ; ⇒ "Hello"
;; Meanwhile, "nack" is printed
```

Each time `sync` is applied to a NACK-guarded event, the guard procedure is called with a newly generated NACK event. Thus, a NACK event becomes ready only when a specific guard-generated event is not chosen in a specific `sync` call.

We defer a complete definition of "not chosen" to Section 7, following a motivating example.

6 Queue: Complete and Improved

Having reviewed the Concurrent ML primitives, we are almost ready to complete the implementation sketch of queues from Section 4. First, however, we refine the queue abstraction to better match the programming idioms of Concurrent ML. This refinement helps demonstrate that our strategy for kill-safety applies to other Concurrent ML abstractions. After showing the implementation of the first improved queue abstraction, we improve the abstraction one step further to demonstrate an additional key idiom.

```
;; queue : → α-queue
;; queue-send-evt : α-queue α → void-event
;; queue-recv-evt : α-queue → α-event

(define-struct q (in-ch out-ch mgr-t))
;; make-q : α-channel α-channel thread → α-queue

(define (queue)
  (define in-ch (channel)) ; to accept sends into queue
  (define out-ch (channel)) ; to supply recvs from queue
  ;; A manager thread loops with serve
  (define (serve items)
    (if (null? items)
        ;; Nothing to supply a recv until we accept a send
        (serve (list (sync (channel-recv-evt in-ch))))
        ;; Accept a send or supply a recv, whichever is ready
        (sync (choice-evt
               (wrap-evt
                (channel-recv-evt in-ch)
                (lambda (v)
                  ;; Accepted a send; enqueue it
                  (serve (append items (list v)))))
               (wrap-evt
                (channel-send-evt out-ch (car items))
                (lambda (void)
                  ;; Supplied a recv; dequeue it
                  (serve (cdr items))))))))
  ;; Create the manager thread
  (define mgr-t (spawn (lambda () (serve (list)))))
  ;; Return a queue as an opaque q record
  (make-q in-ch out-ch mgr-t))

(define (queue-send-evt q v)
  (guard-evt
   (lambda ()
     ;; Make sure the manager thread runs
     (thread-resume (q-mgr-t q) (current-thread))
     ;; Channel send
     (channel-send-evt (q-in-ch q) v))))

(define (queue-recv-evt q)
  (guard-evt
   (lambda ()
     ;; Make sure the manager thread runs
     (thread-resume (q-mgr-t q) (current-thread))
     ;; Channel receive
     (channel-recv-evt (q-out-ch q)))))
```

Figure 7. Implementation of a kill-safe queue

6.1 Queue Actions as Events

Our original queue sketch provided *queue-send* and *queue-recv* functions that block until the corresponding action completes. We should instead provide *queue-send-evt* and *queue-recv-evt* functions that generate events. With events, a programmer can incorporate queues in future synchronization abstractions, which may need to select among multiple blocking actions.

```
(define q (queue))
(sync (queue-send-evt q "Hello"))
(sync (queue-send-evt q "Bye"))
(sync (queue-recv-evt q)) ; ⇒ "Hello"
(sync (queue-recv-evt q)) ; ⇒ "Bye"
```

Figure 7 shows the complete implementation of improved, kill-safe queues:

- The *queue* function creates a thread to manage the internal list of values. Access to the internal list is thus implicitly single-threaded, avoiding race conditions.
- When the queue is neither empty nor full, the queue-managing thread uses `choice-evt` to select among the send and receive actions. If both actions become enabled at once, one or the other is chosen atomically and fairly.
- In the manager thread, `wrap-evt` meshes with `choice-evt` to implement a dispatch for whichever action becomes ready.
- The *queue-send-evt* function guards its result event with a use of `thread-resume`. The guard ensures that the manager thread runs to service the send. The *queue-recv-evt* similarly guards its result.

If a queue becomes unreachable, its manager thread is garbage collected. More generally, when a thread becomes permanently blocked because all objects that can unblock it become unreachable, the thread itself becomes unreachable, and its resources can be reclaimed by the garbage collector.

To a consumer of the abstraction, the values produced by *queue*, *queue-recv-evt*, and *queue-send-evt* have the same first-class status as values produced by `channel`, `channel-recv-evt`, and `channel-send-evt`. For example, queue send and receive events can be multiplexed with other events (using `choice-evt`) in building additional abstractions.

6.2 Selective Dequeue

In DrScheme's help system, a *queue* is used in place of a socket that listens for connections. The *queue* abstraction might also be useful for handling messages to GUI objects, such as a mouse-click messages and refresh messages. A GUI message queue, however, must support a selective dequeuing. For example, a task might wish to handle only refresh messages posted to the queue, leaving mouse-click messages intact.

Unfortunately, selective dequeue cannot be implemented by dequeuing a message, applying a predicate, and then re-posting the message if the predicate fails; re-posting the unwanted message changes its order in the queue with respect to other messages.

To support selective dequeue, we must modify the server so that it accepts dequeue requests with a corresponding predicate, and then satisfies a request only when an item in the queue matches the predicate. On the client side, the selective receive event must be guarded so that it sends a request to the server, then accepts a result through a newly created channel. The new channel ties together the request and the result, so that a result is sent to the correct receiver.

Figure 8 shows a revision of the queue implementation to support selective dequeue. The manager thread still accepts *send*s through *in-ch*, but it no longer supplies queued items to a fixed *out-ch* channel. Instead, the manager thread accepts receive requests through *req-ch*, and it keeps a list of the requests. While the manager waits for sends and additional receive requests, it also services requests for which a matching item is available.

One problem with this implementation is that the manager thread executes an arbitrary predicate procedure that is supplied by a client of the queue. A client could supply a predicate that does not return or that suspends the current thread, thus incapacitating the server thread. One solution to this problem is to change *msg-queue-recv-evt* so that it accepts only simple predicates,

```
;; msg-queue : → α-msg-queue
;; msg-queue-send-evt : α-msg-queue α → void-event
;; msg-queue-recv-evt : α-msg-queue (α → bool) → α-event

(define-struct q (in-ch req-ch mgr-t))
;; make-q : α-channel α–req-channel thread → α-queue

(define-struct req (pred out-ch))
;; make-req : (α → bool) α-channel → α-req

(define (msg-queue)
  (define in-ch (channel))
  (define req-ch (channel))
  (define never-evt (channel-recv-evt (channel)))
  (define (serve items reqs)
    (sync (apply
           choice-evt
           ;; Maybe accept a send
           (wrap-evt
            (channel-recv-evt in-ch)
            (lambda (v)
              ;; Accepted a send; enqueue it
              (serve (append items (list v)) reqs)))
           ;; Maybe accept a recv request
           (wrap-evt
            (channel-recv-evt req-ch)
            (lambda (req)
              ;; Accepted a recv request; add it
              (serve items (cons req reqs))))
           ;; Maybe service a recv request in reqs
           (map (make-service-evt items reqs) reqs))))
  (define (make-service-evt items reqs)
    (lambda (req)
      ;; Search queue items using pred
      (find-first-item (req-pred req) items
       (lambda (item)
         ;; Found an item; try to service req
         (wrap-evt
          (channel-send-evt (req-out-ch req) item)
          (lambda (void)
            ;; Serviced, so remove item and request
            (serve (remove item items)
                   (remove req reqs)))))
       (lambda ()
         ;; No matching item to service req
         never-evt))))
  (define mgr-t
    (spawn (lambda () (serve (list) (list)))))
  (make-q in-ch req-ch mgr-t))

(define (msg-queue-send-evt q v)
  ;; Same as queue-send-evt in Figure 7
  ....)

(define (msg-queue-recv-evt q pred)
  (guard-evt
   (lambda ()
     (define out-ch (channel))
     ;; Make sure the manager thread runs
     (thread-resume (q-mgr-t q) (current-thread))
     ;; Request for an item matching pred with reply to out-ch
     (sync (channel-send-evt (q-req-ch q)
                             (make-req pred out-ch)))
     ;; Result arrives on out-ch
     (channel-recv-evt out-ch))))
```

Figure 8. Queue with selective dequeue, first attempt

such as odd? and even?, that are known to be harmless. We show how to allow arbitrary predicates in Section 8.1.

Even if we constrain *pred*, the implementation of selective dequeue contains a space leak. The following example illustrates the problem:

```
(define q (msg-queue))
(sync (msg-queue-send-evt q 1))
(sync (msg-queue-send-evt q 2))
(sync (choice-evt
       (msg-queue-recv-evt q odd?)
       (msg-queue-recv-evt q even?)))
```

The sync call sends two requests to the server. One is serviced, and the program continues. Meanwhile, a leftover request remains with the server. The request will never be successfully serviced, because no sync waits on the associated *out-ch*. Still, the request is stuck in the internal *reqs* list, and leftover requests can pile up over time, degrading performance and wasting resources. A similar problem occurs if the thread making a request is terminated.

To avoid this problem, the server needs to know when a client sync has abandoned a dequeue request. Figure 9 shows how nack-guard-evt can provide this information. The *msg-queue-recv-evt* function now sends the manager a "gave up" event in addition to a result channel. The manager thread uses the new event to keep the request list clean.

The *msg-queue* example illustrates a particular Concurrent ML idiom: a client–server protocol where the client sends a request to the server, but may withdraw the request before it can be satisfied. Withdrawal reliably prevents acceptance and vice-versa, due to the rendezvous associated with a channel transfer (i.e., the sender and receiver must simultaneously agree to the transfer of a result).

The request idiom poses an extra challenge for kill-safety. A client can be terminated at any point in the request cycle, so we must define nack-guard-evt so that it handles this possibility. The next section completes our explanation of MzScheme's primitives with a suitable definition of nack-guard-evt.

7 Termination and NACKs

Recall that the event provided to a guard procedure by nack-guard-evt becomes ready if the guard-generated event is not chosen. MzScheme extends the Concurrent ML definition of "not chosen" so that it includes all of the following cases, which cover all of the ways that a thread can abandon an event:

- The sync call chooses an event other than the one returned by the guard.
- Control escapes from the sync call through an exception or continuation jump. The exception or jump may have been triggered through a break signal (discussed further in Section 8.2), by another guard involved in the same sync, or even by the guard procedure that received the NACK event. Continuation jumps back into a guard are always blocked by our definition of nack-guard-evt, so multiple escapes are not possible.
- The syncing thread terminates (i.e., it is suspended and unreachable).

In the code from Figure 9, the event produced by *msg-queue-recv-evt* can be used in an arbitrary client context, so all of the above cases are possible. For example, the context might begin a `sync` on the event in a particular thread, then terminate the thread during the `sync`. If the guard procedure is executing at the time of termination, and if termination occurs before the nested `sync` completes a send to `(q-req-ch q)`, then the server never becomes aware of the request, so the server is left in a consistent state. If termination occurs after the nested `sync` but before the outer `sync` chooses the guard's result `(channel-recv-evt out-ch)`, the server has already received `gave-up-evt`, and therefore knows to abandon the request (because `gave-up-evt` becomes ready). Finally, if termination occurs after a successful outer `sync` on `(channel-recv-evt out-ch)`, then the server has completed the request, so termination does not affect the server.

MzScheme's `nack-guard-evt` corresponds to Concurrent ML's `withNack`. An earlier version [19] of Concurrent ML offered `wrapAbort`, instead, and a later presentation [21] explains how `withNack` can be implemented with `wrapAbort`. Our definition of "not chosen" does not allow such an implementation, and thus strengthens the argument that `withNack` is the right operation to designate as primitive.

8 Beyond Kill-Safety

To further explain our design choices, we show in the following sections how custodians and events solve problems besides kill-safety, and how they interact with cooperative termination.

8.1 Custodians and Events At Work, Again

As noted in Section 6.2, our initial implementation of selective dequeue is unsafe; the server executes arbitrary predicate procedures that are supplied by clients, which might damage the server thread. For example, a thread can provide a predicate to the server that suspends the current thread, thus disabling the queue.

```
(define q (msg-queue))
(sync (msg-queue-send-evt q 1))
(spawn (lambda ()
        (define (die x)
          (thread-suspend (current-thread)))
        (sync (msg-queue-recv-evt q die))))
(sync (msg-queue-recv-evt q odd?)) ; probably stuck
```

To avoid this problem, the server can run the predicate in a new thread, which prevents the predicate from harming the server thread. Moreover, the new thread should be executed under a custodian that is supplied by the client thread (as part of a request), which means that the predicate-running thread can execute only when the client is still allowed to execute. This arrangement is analogous to a remote procedure call, except that "remoteness" is implemented by using another process's custodian.

The server cannot simply wait in turn for each remote thread to complete its work, because a predicate might not terminate. Instead, the server must `sync` on an event that corresponds to completion of the remote thread. When the event delivers a list of acceptable items, they can be added to the request. In later iterations, a remote thread is started for the request only if the list of known acceptable items becomes empty.

To implement this change, we modify only `make-service-evt` and `msg-queue-recv-evt` as shown in Figure 10. The change to `msg-queue-recv-evt` adds a custodian and empty list (of known acceptable items) to the request. The new `make-service-evt` checks whether a particular request has at least one known acceptable item. If the request has none, then `service-evt` uses `ok-items-evt` to call the predicate remotely on the current list of items; if those items are received from the remote call, the request's list is updated. Otherwise, the request has known acceptable items, and `make-service-evt` creates an event to service the request with the first acceptable item. When an item is delivered in this way, it is removed from the current list of queued items, and also removed from every remaining request's list of acceptable items.

```
(define-struct req (pred out-ch gave-up-evt))
;; make-req : (α → bool) α-channel void-event → α-req

(define (msg-queue)
  ....
  (define (serve items reqs)
    ....
         ;; Add make-abandon-evt events
         (append
          (map (make-service-evt items reqs) reqs)
          (map (make-abandon-evt items reqs) reqs)))))
  ....
  (define (make-abandon-evt items reqs)
    (lambda (req)
      ;; Event to detect that the receiver gives up
      (wrap-evt (req-gave-up-evt req)
               (lambda (void)
                 ;; Receiver gave up; remove request
                 (serve items (remove req reqs))))))
....

(define (msg-queue-recv-evt q pred)
  (nack-guard-evt
   (lambda (gave-up-evt)
     (define out-ch (channel))
     (thread-resume (q-mgr-t q) (current-thread))
     ;; As before, but also send the server gave-up-evt
     (sync (channel-send-evt
           (q-req-ch q)
           (make-req pred out-ch gave-up-evt)))
     (channel-recv-evt out-ch))))
```

Figure 9. Revision to Figure 8

This example illustrates how custodians and Concurrent ML's primitives complement each other beyond kill-safety. These additional uses increase our confidence that the combination is general, and therefore a good approach to the specific problem of kill-safety.

8.2 Cooperative Termination

Cooperative termination allows a thread to execute clean-up actions (e.g., flush a file buffer, update a shared GUI display) before terminating. Cooperative termination is useful for many of the same reasons as uncooperative termination, though only when a process can be trusted to exit quickly.

MzScheme's `break-thread` function enables cooperative termination. This function takes a thread descriptor and sends the thread a break signal. The signal is analogous to a Unix process signal, but the break signal is manifest in the target thread as an asynchronous exception, much as in Concurrent Haskell [14].

```
(define t (spawn lots-of-work))
(break-thread t) ; possibly interrupts lots-of-work
```

```
(define-struct req (pred out-ch gave-up-evt
                    cust ok-items))
;; make-req : (α → bool) α-channel void-event
;;            custodian α-list → α-req

(define (msg-queue)
  ....
  (define (make-service-evt items reqs)
    (lambda (req)
      (if (null? (req-ok-items req))
          ;; Look for items acceptable to pred
          (wrap-evt
           (ok-items-evt req items)
           (lambda (ok-items)
             ;; Got a list of acceptable items, so update req
             (serve items
                    (cons (new-ok-items req ok-items)
                          (remove req reqs)))))
          ;; Use first acceptable item to service req
          (wrap-evt
           (channel-send-evt (req-out-ch req)
                             (car (req-ok-items req)))
           (lambda (void)
             ;; Serviced, so remove item and request
             (define item (car (req-ok-items req)))
             (serve (remove item items)
                    (map
                     (remove-ok-item item)
                     (remove req reqs))))))))
  (define (ok-items-evt req items)
    ;; New thread runs pred and delivers a list to items-ch
    (define items-ch (channel))
    (parameterize ([current-custodian (req-cust req)])
      (spawn
       (lambda ()
         (define ok-items (filter (req-pred req) items))
         (sync (channel-send-evt items-ch ok-items)))))
    (channel-recv-evt items-ch))
  (define (remove-ok-item item)
    ;; Given a req, remove item from its list of acceptable items
    (lambda (req)
      (new-ok-items req
                    (remove item (req-ok-items req)))))
  (define (new-ok-items req ok-items)
    (make-req (req-pred req) (req-out-ch req)
              (req-gave-up-evt req) (req-cust req)
              ok-items))
....

(define (msg-queue-recv-evt q pred)
  (nack-guard-evt
   (lambda (gave-up-evt)
     (define out-ch (channel))
     (thread-resume (q-mgr-t q) (current-thread))
     ;; Include a custodian and an initially empty list of
     ;; known acceptable items
     (sync (channel-send-evt
            (q-req-ch q)
            (make-req pred out-ch gave-up-evt
                      (current-custodian) (list))))
     ;; Result arrives on out-ch
     (channel-recv-evt out-ch))))
```

Figure 10. Revision to Figure 9 for arbitrary predicates

```
;; swap-channel : → α-swap-channel
;; swap-evt : α-swap-channel α → α

(define-struct sc (ch))
;; make-sc : α-req-channel → α-swap-channel

(define-struct req (v ch))
;; make-req : α α-channel → α-req

(define (swap-channel) (make-sc (channel)))

(define (swap-evt sc v)
  (guard-evt
   (lambda ()
     (define in-ch (channel))
     (choice-evt
      ;; Maybe act as server and receive req
      (wrap-evt
       (channel-recv-evt (sc-ch sc))
       (lambda (req)
         ;; Reply to req
         (sync (channel-send-evt (req-ch req) v))
         (req-v req)))
      ;; Maybe act as client and send req
      (wrap-evt
       (channel-send-evt (sc-ch sc) (make-req v in-ch))
       (lambda (void)
         ;; Receive answer to req
         (sync (channel-recv-evt in-ch))))))))
```

Figure 11. A break-safe implementation of swap channels

Since breaks are for cooperative termination, a thread is allowed to disable breaks by using `parameterize` with `break-enabled`. If a break signal is sent to a thread when breaks are disabled, the signal is delayed until breaks are re-enabled in the thread. A break signal has no effect if the target thread has a delayed break already.

MzScheme automatically disables breaks in certain contexts, so that many synchronization abstractions are naturally break-safe. One such abstraction is a swap channel [21, pg. 59], as shown in Figure 11. A swap channel is like a channel, except that both synchronizing threads provide a value to the other. Implementing this two-way channel with one-way channels requires two phases. In the first phase, the thread that is elected to act as the client sends its value to the other thread, which is elected as the server. In the second phase, the server thread sends its value back to the client thread. The client and server threads are elected through `sync`'s non-deterministic choice.

When two threads complete the first phase, they are committed to the swap. In other words, a break should not interrupt the second phase. In Figure 11's implementation of *`swap-evt`*, a break cannot interrupt the second phase, because the second phase is in a wrap procedure. MzScheme implicitly disables breaks from time that `sync` chooses an event until the event's wrap procedure completes.

Although the body of a wrap procedure may explicitly re-enable events, this example also illustrates why breaks are not implicitly enabled by `sync`, unlike Concurrent Haskell's `takeMVar`. If `sync` implicitly enabled breaks, then the second phase of a swap might be skipped after the threads have committed to swapping.

For cases where a programmer would like to re-enable breaks while `sync` blocks, MzScheme provides a separate `sync/enable-break` function. This function enables breaks such that either a break exception is raised or an event is chosen, but not both. (Merely wrapping a `sync` with `parameterize` to enable breaks does not achieve sync/enable-break's exclusive-or behavior. With parameterize and sync, the break may occur after an event is chosen but before breaks are re-disabled, thus allowing both choice and a break.) In our experience, sync/enable-break's exclusive-or guarantee is important for building break-safe code on top of synchronization abstractions.

In the same way that a synchronization abstraction is responsible for providing an appropriate commit point—so that events can be combined with others through choice-evt—each synchronization abstraction is responsible for preserving sync/enable-break's exclusive-or guarantee. For example, adding (parameterize ([break-enabled #t]) #f) to Figure 11 after (sync (channel-send-evt (*req-ch req*) *v*)) would defeat sync/enable-break's exclusive-or guarantee, though without damaging the swap abstraction in any other way.

Many kill-safe synchronization abstractions are naturally break-safe, too. For example, the queue implementations of Figure 7 and Figure 10 interact perfectly with cooperative termination, including sync/enable-break.

Unfortunately, sync/enable-break's exclusive-or guarantee is not preserved by a kill-safe version of the swap abstraction. In the kill-safe implementation, which is shown in Figure 12, a manager thread pairs swapping clients and delivers a value to each client. From the manager perspective, two clients are committed to swap as soon as they are both known; from the client perspective, the swap commits when the manager delivers a value. The mismatch means that a client may be interrupted between the time that the manager commits the swap and the time that the clients receive values, thus defeating an exclusive-or guarantee for sync/enable-break (although a break still does not damage the abstraction in any other way).

This example illustrates how break-safety is not necessarily easier than kill-safety, and vice-versa. Although break-safety seems intuitively easier, because a thread can disable breaks during sensitive operations, continuing to execute after a break introduces concerns that are absent with immediate termination. Future work may suggest a way to reconcile these forms of termination, or, in the case of swap channels, they may be irreconcilable due to the limitations of channel rendezvous [17].

```
(define-struct sc (ch mgr-t))
;; make-sc : α-req-channel thread → α-swap-channel

(define-struct req (v ch gave-up))
;; make-req : α α-channel void-event → α-req

(define (swap-channel)
  (define ch (channel))
  (define (serve-first)
    ;; Get first thread for swap
    (sync (wrap-evt
          (channel-recv-evt ch)
          serve-second)))
  (define (serve-second a)
    ;; Try to get second thread for swap
    (sync
     (choice-evt
      ;; Possibility 1 — got second thread, so swap
      (wrap-evt
       (channel-recv-evt ch)
       (lambda (b)
         ;; Send each thread the other's value
         (send-eventually (req-ch a) (req-v b))
         (send-eventually (req-ch b) (req-v a))
         (serve-first)))
      ;; Possibility 2 — first gave up, so start over
      (wrap-evt
       (req-gave-up a)
       (lambda (void) (serve-first))))))
  (define (send-eventually ch v)
    ;; Spawn a thread, in case ch's thread isn't ready
    (spawn (lambda ()
             (sync (channel-send-evt ch v)))))
  (make-sc ch (spawn serve-first)))

(define (swap-evt sc v)
  (nack-guard-evt
   (lambda (gave-up)
     (define in-ch (channel))
     (thread-resume (sc-mgr-t sc) (current-thread))
     (sync
      (wrap-evt
       (channel-send-evt (sc-ch sc)
                        (make-req v in-ch gave-up))
       (lambda (void) in-ch))))))
```

Figure 12. A kill-safe implementation of swap channels

9 Related Work

Many systems provide a mechanism for cooperative interruption of a process, such as thread cancellation in Posix [16], alerts in Modula-3 [8], and asynchronous exceptions in Haskell [14] or MzScheme [5]. For applications like our servlets example, however, forced termination is necessary, and our work is concerned with kill-safety with respect to forced termination.

Some previous work addresses the interaction between termination and synchronization for specific primitives. Examples include work on monitors in Pilot [18] and remote pointers in Luna [10]. To our knowledge, no previous work addresses the problem of termination with respect to *programmer-defined* synchronization abstractions. Indeed, the problem makes sense only after programmers are given significant abstraction capability, which is why our work depends on Concurrent ML [19, 20, 21].

The idea of managing a resource through a designated thread appears in many contexts, notably in microkernels [4]. Argus's guardians [13] reflect a similar idea in the area of persistent, distributed computing. Our specific use of the thread-manager pattern is typical of Concurrent ML programs, but also reminiscent of the J-Kernel [9] approach, which creates a thread when crossing a trust boundary to defend against termination. We extend this idea by adding a mechanism to adjust a thread's execution capability relative to other threads.

MzScheme does not provide a way to revoke access to an object, as in Luna [10]. It also provides no way to disable code that is associated with a task, as in Rudys et al.'s soft termination [22]. Given a mechanism for disabling code, we conjecture that code fragments could be connected to the custodian hierarchy to prevent a shared abstraction's code from being disabled prematurely.

The architectures in Figure 1 and 2 (in Section 2) illustrate a trade-off between ease of communication and ease of termination, but they are merely the extremes. A variation of the OS-style architecture can improve communication by enriching the kernel's set of primitives, as in KaffeOS [1], Alta [2], SPIN [3], J-Kernel [9], Luna [10], and Nemesis [12]. In each case, however, the system offers a fixed set of kill-safe primitives to applications, and our goal is to allow programmer-defined kill-safe abstractions.

10 Conclusion

Many real-world programming tasks require concurrency. Nevertheless, certain programming models make correct concurrency difficult, so that programmers speak of *thread-safe* implementations and work hard to ensure that an abstraction is thread-safe. Languages that are specifically designed for concurrency, (e.g., Erlang, Concurrent ML) can encourage thread-safe implementations, making them the rule rather than the exception. Indeed, in such languages, concurrency tends to simplify implementations, rather than complicate them.

Many real-world programming tasks with concurrency also benefit from termination. Nevertheless, programming systems have not been designed to encourage (or even enable) *kill-safe* abstractions. We have taken a step in this direction, extending constructs for thread-safe abstractions to enable kill-safe abstractions, thus increasing the potential of termination to simplify implementations.

Acknowledgements
We would like to thank John Reppy and the anonymous reviewers for their comments and suggestions.

The code in this paper is available at the following URL:

`http : //www.cs.utah.edu/plt/kill − safe/`

11 References

[1] G. Back, W. Hsieh, and J. Lepreau. Processes in KaffeOS: Isolation, resource management, and sharing in Java. In *Proc. USENIX Conference on Operating Systems Design and Implementation*, pages 333–346, Oct. 2000.

[2] G. Back, P. Tullmann, L. Stoller, W. C. Hsieh, and J. Lepreau. Java operating systems: Design and implementation. In *Proceedings of the USENIX 2000 Technical Conference*, pages 197–210, San Diego, CA, June 2000.

[3] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proc. ACM Symposium on Operating Systems Principles*, pages 267–284, Dec. 1995.

[4] D. L. Black, D. B. Golub, D. P. Julin, R. F. Rashid, R. P. Draves, R. W. Dean, A. Forin, J. Barrera, H. Tokuda, G.-R. Malan, and D. Bohman. Microkernel operating system architecture and Mach. *Journal of Information Processing*, 14(4):442–453, 1991.

[5] M. Flatt. *PLT MzScheme: Language Manual*, 2004. `www.mzscheme.org`.

[6] M. Flatt, R. B. Findler, S. Krishnamurthi, and M. Felleisen. Programming languages as operating systems (*or* revenge of the son of the Lisp machine). In *Proc. ACM International Conference on Functional Programming*, pages 138–147, Sept. 1999.

[7] P. Graunke, S. Krishnamurthi, S. V. D. Hoeven, and M. Felleisen. Programming the Web with high-level programming languages. In *Proc. European Symposium on Programming*, volume 2028 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[8] S. P. Harbison. *Modula-3*. Prentice Hall, 1991.

[9] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing multiple protection domains in Java. In *Proc. of USENIX Annual Technical Conference*, pages 259–270, June 1998.

[10] C. Hawblitzel and T. von Eicken. Luna: a flexible Java protection system. In *Proc. USENIX Conference on Operating Systems Design and Implementation*, Oct. 2002.

[11] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, NJ, 1985.

[12] I. M. Leslie, D. McAuley, R. J. Black, T. Roscoe, P. R. Barham, D. M. Evers, R. Fairburns, and E. A. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal on Selected Areas in Communications*, 14(7):1280–1297, Sept. 1996.

[13] B. Liskov and R. Scheifler. Guardians and actions: Linguistics support for robust, distributed systems. *ACM Transactions on Computing Systems*, 5(3):381–404, 1983.

[14] S. Marlow, S. L. Peyton Jones, A. Moran, and J. H. Reppy. Asynchronous exceptions in Haskell. In *Proc. ACM Conference on Programming Language Design and Implementation*, pages 274–285, 2001.

[15] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.

[16] National Institute of Standards and Technology (U.S.). *POSIX: portable operating system interface for computer environments*, Sept. 1988.

[17] P. Panangaden and J. H. Reppy. The essence of Concurrent ML. In F. Nielson, editor, *ML with Concurrency: Design, Analysis, Implementation and Application*, Monographs in Computer Science, pages 5–29. Springer-Verlag, 1997.

[18] D. Redell, Y. Dalal, T. Horsley, H. Lauer, W. Lynch, P. McJones, H. Murray, and S. Purcell. Pilot: An operating system for a personal computer. *Communications of the ACM*, 23(2):81–92, Feb. 1980.

[19] J. H. Reppy. Synchronous operations as first-class values. In *Proc. ACM Conference on Programming Language Design and Implementation*, pages 250–259, 1988.

[20] J. H. Reppy. *Higher–Order Concurrency*. PhD thesis, Cornell University, 1992.

[21] J. H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[22] A. Rudys, J. Clements, and D. S. Wallach. Termination in language-based systems. *ACM Transactions on Information and System Security*, 5(3):138–168, 2002.

[23] A. Rudys and D. S. Wallach. Transactional rollback for language-based systems. In *Proc. International Conference on Dependable Systems and Networks*, June 2002.

[24] Soper, P., specification lead. JSR 121: Application isolation API specification, 2003. `http://www.jcp.org/`.