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

DRAFT: March 29, 2003

Matthew Flatt
University of Utah

Robert Bruce Findler
University of Chicago

Abstract

When an individual task can be 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 kernel.

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

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 [2, 3, 10, 11, 14], and termination of Java tasks is a driving goal of the new JSR-121 standard [23].

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. Java extensions such as JSR-121 solve the problem in the same way as conventional operating systems: they restrict sharing among terminable tasks to objects that are managed by the run-time kernel. Thus, 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 for the other task.

The conventional design places full responsibility on the kernel implementor to manage objects and termination correctly. 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; new communication abstractions require extensions to the kernel.

Our latest version of MzScheme [7] avoids this restriction through a novel set of primitives for managing tasks. Using the primitives, a programmer can implement a new kill-safe abstraction without modifying MzScheme’s kernel. Crucially, the set of cooperating tasks that share an object need not be defined in advance, and the cooperating tasks need not trust each other. (They must trust only the implementation of the shared object.)

The new MzScheme primitives build on the observation that thread termination is merely indefinite suspension plus garbage collection. By creating a thread to manage a particular synchronization abstraction, a programmer can ensure that the abstraction instance is suspended when it might otherwise be killed. In short, when the instance object is “killed” as part of a task termination, it turns out to be “only mostly dead.” A surviving task that shares the object can resurrect it. This technique succeeds because MzScheme’s primitives allow a manager thread to preserve its object’s consistency across suspends and resumes.

MzScheme also builds on the primitives of Concurrent ML [20], 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 covers 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 with small adjustments.

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 im-
plementation using the Concurrent ML primitives. This full implementation motivates a remaining detail of our design that is covered in Section 7. Section 8 compares some of our design choices to some alternatives. Section 9 summarizes related work.

2 Termination and Cooperation

Conventional OSes govern a realm of intense distrust. Each task trusts only the kernel, and the kernel trusts no one. To maintain order, the kernel strictly isolates tasks, as illustrated in Figure 1. Each task occasionally communicates across the “red line” (drawn as a thick gray line in the figure) to the kernel task, but a tasks do 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 haywire task at worst corrupts its own space (imagine a task in the figure growing uncontrollably to fill its own space), and the task is therefore terminated easily. The cost of easy termination is a restriction on communication. If two tasks need to communicate, 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).

In contrast, the run-time kernel of a safe programming language presides over a realm that is free from bit-bashing vandals. In the same way that a task relies on abstract datatypes for internal protection, the task can rely on abstract datatypes for protection from other tasks. The kernel therefore allows task boundaries to fade away, as illustrated in Figure 2. Compared to a conventional OS, tasks are free to set up new communication abstractions that better match their needs (as depicted in the figure by a close-range telecast in a task’s space). However, a haywire task can now bring down the entire system, not by mangling data structures, but by consuming too many resources (again, imagine a task growing uncontrollably, but this time it fills the entire system, squeezing out other tasks).

2.1 The Best of Both Worlds

The architectures in Figure 1 and 2 demonstrate a trade-off between ease of communication versus 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, possibly exploiting language safety in the design of the new primitives. Indeed, many language-based systems take this approach, including Alta [3], SPIN [4], J-Kernel [10], Nemesis [14], KaffeOS [2], and Luna [11]. In each case, however, the system offers a fixed set of primitives to applications. Inevitably, some set of tasks would benefit from a new communication abstraction.

A hybrid architecture can avoid the fixed set of primitives, and its implementation may consist simply of running one system inside another. By running, say, KaffeOS under Unix, KaffeOS programs can take advantage of KaffeOS’s primitives, while Unix-level tasks continue to communicate with Unix primitives. This layering of kernels, illustrated in Figure 3, effectively allows a programmer to extend the kernel’s set of communication primitives by implementing a new kernel. Depending on the base, implementing a new kernel may be difficult, but our previous work on MzScheme/MrEd [8] shows how a language can help. Indeed, the DrScheme programming environment [6] depends on such support from MzScheme, since DrScheme acts as a kernel to the programs that it executes for debugging.

2.2 Remaining Problem: Discovering Cooperation

The drawback of the kernel-layers approach is that the layers must be defined in advance, and tasks must be associated with layers in advance. In practice, two already-running tasks may discover each other and wish to communicate through an abstraction that is not provided by their shared kernel.

For example, servlets for two different sessions might discover each other and wish to share a cache whose implementation is specific to the servlets. Although the servlet tasks can trust the cache implementation, they cannot trust each other to survive, because the server might terminate one or the other at any time.

This configuration is illustrated in Figure 4. In the figure, the left and right tasks do not trust each other, but they trust the small gray
box in the middle (in the same way that they trust the kernel). The core of the socket-like abstraction abstraction is an asynchronous buffered queue. The MzScheme kernel, however, provides no such abstraction, and 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 Reppy’s book [20] for many other example abstractions that can be made kill-safe using our technique.

3 Task Control in MzScheme

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 [8]. 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.

```scheme
(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 windows 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. The shut-down command also propagates to any controlled sub-custodians. After a custodian is shut down, it drops references to primitive resources such as sockets, GUI windows, etc., and the memory for such objects can be reclaimed by the garbage collector (when they become otherwise unreachable). The custodian itself remains available to control new objects. It also retains a weak reference to each suspend thread, in case the thread is later resumed. If a thread is resumed, a future custodian-shutdown-all once again suspends the thread.

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 installed custodian.

In practice, shutting down a resource removes internal references, which frees most of the memory associated with the resource.
(define cust (make-custodian))
(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 Control

The custodian-shutdown-all procedure provides one way to suspend a thread. MzScheme also provides thread-suspend for suspending a specific thread.

Suspending a thread stops a computation uncooperatively, but such heavy-handed termination is not always necessary. For cooperative termination, MzScheme provides the thread-break function, which takes a thread descriptor and sends the thread a break signal. This signal is analogous to a Unix process signal, but the break signal is manifest in the target thread as a asynchronous exception, much as in Concurrent Haskell [16].

(define cust (make-custodian))
(define t (parameterize ([current-custodian cust])
 (spawn lots-of-work)))
(thread-break t) ; possibly interrupts lots-of-work

Asynchronous breaks add only a minor challenge to the implementation of kill-safe abstractions, and we address this challenge in Section 7. For completeness, we offer a few additional details concerning breaks in MzScheme, but the disinterested reader may safely skip to the next subsection.

Break exceptions can be enabled and disabled by using parameterize with break-enabled. Breaks are always implicitly disabled during exception handling and during the evaluation of dynamic-wind pre- and post-actions. If a break signal is delivered to a thread that has disabled breaks, the signal is delayed until breaks are re-enabled in the thread. A break signal has no effect if the target thread already has a delayed break.

Unlike Concurrent Haskell’s take!Var, a blocking sync does not implicitly enable breaks while it blocks. Such implicit enabling is unnecessary to encourage interruption when thread-suspend is available is a back-up, and in our experience, implicit enabling of breaks makes a system fragile. A separate sync/enable-breaks function enables breaks such that either a break execution is raised or an event is selected, but not both. (Merely wrapping a sync with a parameterize to enable breaks does not achieve this exclusive-or behavior, because the break may occur after an event is selected but before breaks are re-disabled.)

3.4 Thread Resumption

Given a single thread argument, MzScheme’s thread-resume function resumes the thread if it is suspended.

(define cust (make-custodian))
(define t (parameterize ([current-custodian cust])
 (spawn lots-of-work)))
(custodian-shutdown-all cust) ; stops lots-of-work
(resume-thread t) ; resumes lots-of-work

MzScheme provides no operation for resuming all controlled objects in a custodian. Indeed, for most kinds of controlled objects, a shut-down action is terminal. Thus, a thread such as t above may resume and discover that its resources have been closed. For kill-safe abstractions, the implication is that an abstraction cannot rely on controlled resources other than threads.

The thread-resume function accepts an optional second argument. The second argument, which also must be a thread descriptor, plays two roles:

• It yokes the first thread to second for resumes, in that the first thread is resumed whenever the second thread is resumed.
• It potentially “strengthens” the first thread by changing the thread’s controlling custodian to a more senior custodian. The more senior custodian is determined by comparing the two thread’s controlling custodians, and finding the least-senior common super-custodian.

The overall effect of (thread-resume t t2) is to ensure that t is suspended only when t2 is also suspended—assuming that t is suspended only indirectly via custodian-shutdown-all. This effect holds because an indirect suspension of t will also suspend t2, and if t2 is resumed later, so is t.

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. This combination was the key to implementing kill-safe versions of the queue abstractions.

4 Sketch for a Kill-Safe Queue

The primitive synchronization abstraction in MzScheme is a CSP-style synchronous channel [12], 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.

The implementation is not yet kill-safe. For example, suppose that
a thread t1 creates q by calling (queue). Suppose further that t1 is controlled by custodian c1, and that q is made available to a thread t2 controlled by custodian c2:

(a thread t1 creates q by calling (queue). Suppose further that t1 is controlled by custodian c1, and that q is made available to a thread t2 controlled by custodian c2:

\[ t1, \text{controlled by } c1 \quad \text{(send-to-other q)} \]

\[ (\text{send-to-other q}) \quad \text{(define q (get-from-other))} \]

Since t1 creates q, the queue’s internal thread t is controlled by c1. Suspending all threads of c1 suspends both t1 and 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 t and the time that it performs its action on the queue, another thread might re-suspend all threads of c1, thus re-suspending t.

The solution is (thread-resume t t2), which not only resumes the queue thread, but also changes the controlling custodian of t to a super-custodian of both c1 and c2. afterward, a mere suspension of c1’s threads does not suspend t.

Since t is not accessible outside the queue implementation, it can be suspended only through its custodian. Furthermore, since t’s custodian is a common ancestor of t1 and t2, both of those threads will be suspended as well, and it does not matter that t is suspended. Indeed, t must be suspended, because the suspension may be intended to terminate the set of tasks t1 and t2.

If the suspension is not intended to terminate the task, and if t2 is later resumed, then t is also resumed, due to the chaining installed by (thread-resume t t2). More generally, by guarding each queue operation with (thread-resume t (current-thread)) we ensure that 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 MzScheme’s 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 Concurrent ML [20] primitives that are listed in Figure 7. 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.

\[ \text{(define t1 (spawn (lambda () (println "Hello")))}) \]

\[ \text{(define t2 (spawn (lambda () (println "Nihao")))}) \]

\[ \text{(sync (thread-done-evt t1)); waits until t1 is done} \]

\[ \text{(sync (thread-done-evt t2)); waits until t2 is done} \]

\[ \text{(print "Bye")} \]

\[ \text{prints "Hello" and "Nihao" interleaved, then "Bye"} \]

---

2 The functions are defined in the (lib "cml.ss") module.
The channel procedure takes no arguments and returns a channel descriptor. A channel’s only purpose is to generate events; the channel-send-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 is available. channel can provide a value simultaneously.

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.

\[
\text{sync} : \alpha \rightarrow \alpha \\
\text{channel} : \rightarrow \alpha \rightarrow \alpha \\
\text{channel-recv-evt} : \alpha \rightarrow \alpha \\
\text{channel-send-evt} : \alpha \rightarrow \alpha \rightarrow \alpha \\
\text{choice-evt} : \alpha \rightarrow \alpha \\
\text{wrap-evt} : \alpha \rightarrow \alpha \\
\text{guard-evt} : \rightarrow \alpha \rightarrow \alpha \\
\text{nack-guard-evt} : \rightarrow \alpha \rightarrow \alpha \\
\text{Figure 7. Concurrent ML primitives in MzScheme}
\]

- \text{channel} : \rightarrow \alpha \rightarrow \alpha
- \text{channel-recv-evt} : \alpha \rightarrow \alpha
- \text{channel-send-evt} : \alpha \rightarrow \alpha \rightarrow \alpha
- \text{choice-evt} : \alpha \rightarrow \alpha
- \text{wrap-evt} : \alpha \rightarrow \alpha
- \text{guard-evt} : \rightarrow \alpha \rightarrow \alpha
- \text{nack-guard-evt} : \rightarrow \alpha \rightarrow \alpha

The \text{choice-evt} procedure takes any number of events and composes them into a single event. The composite event is ready when one of the composed events is ready. If multiple composed events are ready, one is chosen arbitrarily but fairly, and the value produced by the composite event is the value produced by the chosen event.

\[
\begin{align*}
\text{(define c1 (channel))} \\
\text{(define c2 (channel))} \\
\text{(spawn (lambda () (sync (channel-send-evt c1 "Hello"))))} \\
\text{(spawn (lambda () (sync (channel-send-evt c2 "Nihao"))))} \\
\text{(define cc (choice-evt (channel-recv-evt c1) (channel-recv-evt c2)))} \\
\text{(sync cc) ;} \Rightarrow \text{"Hello" or "Nihao"} \\
\text{(sync cc) ;} \Rightarrow \text{"Nihao" or "Hello"}
\end{align*}
\]

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).

- \text{wrap-evt} : \alpha \rightarrow \alpha
- \text{guard-evt} : \rightarrow \alpha \rightarrow \alpha

The \text{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.

\[
\begin{align*}
\text{(define cc1 (channel))} \\
\text{(define cc2 (channel))} \\
\text{(spawn (lambda () (sync (channel-send-evt cc1 "Hello"))))} \\
\text{(spawn (lambda () (sync (channel-send-evt cc2 "Nihao"))))} \\
\text{(sync (choice-evt (wrap-evt (channel-recv-evt cc1)) (nack-guard-evt (lambda (x) (list x "from 1")))))} \\
\text{(wrap-evt (channel-recv-evt cc2))} \\
\text{(lambda (x) (list x "from 2")))} \\
\text{); } \Rightarrow \text{(list "Hello" "from 1") or (list "Nihao" "from 2")}
\end{align*}
\]

The \text{guard-evt} function is the dual of \text{wrap-evt}. Whereas \text{wrap-evt} supports post-processing on the result of an event, \text{guard-evt} supports pre-processing to generate an event for synchronization. For example, assume that \text{current-time} produces the current time, and that \text{time-evt} produces an event that is ready at a given absolute time. Then, \text{guard-evt} can be used to construct a timeout event.

\[
\begin{align*}
\text{;; current-time :} \rightarrow \text{num} \\
\text{;; time-evt : num \rightarrow event} \\
\text{(define one-sec-timeout (guard-evt (lambda () (time-evt (+ 1 (current-time))))) \;
\text{(sync one-sec-timeout)} ; \Rightarrow \text{void, one second later} \\
\text{(sync one-sec-timeout)} ; \Rightarrow \text{void, another second later}
\end{align*}
\]

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

- \text{nack-guard-evt} : \rightarrow \alpha \rightarrow \alpha

The \text{nack-guard-evt} function generalizes \text{guard-evt}. For \text{nack-guard-evt}, the given guard procedure must accept a single argument. The argument is a “Negative ACKnowledgement” event that becomes ready if the guard-generated event is not chosen by \text{sync}.

\[
\begin{align*}
\text{(define c (channel))} \\
\text{(spawn (lambda () (sync (channel-send-evt c "Hello"))))} \\
\text{(sync (choice-evt (nack-guard-evt (lambda (nack) \;
\text{;; Start a thread to watch NACK} \\
\text{(spawn (lambda () \;
\text{(sync nack) (printf "not first"))} \;
\text{;; This event is never ready} \\
\text{(channel-recv-evt (channel)))} \;
\text{;; This event is ready eventually} \\
\text{(channel-recv-evt c1))} ; \Rightarrow \text{"Hello"} \\
\text{;; Meanwhile, “not first” is printed}
\end{align*}
\]

Each time \text{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 \text{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 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.

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"
```

Figures 8 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.
- The wrap-evt function meshes with choice-evt to implement a dispatch for the ready action within the manager thread.
- 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 recv 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 de-queuing a message, applying a predicate, 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 9 shows a revision of the queue implementation to support selective dequeue. The manager thread still accepts sends through in-ch, but it no longer supplies queued items to a fixed
(define (msg-queue)
  (define in-ch (channel))
  (define req-ch (channel))
  (define never-evt (channel-recv-evt channel))
  (define (serve items reqs)
    (sync (apply (channel-recv-evt in-ch)
      (lambda ()
        ;; May accept a send
        (wrap-evt (channel-recv-evt in-ch)
          (lambda (v)
            ;; Accepted a send; enqueue it
            (serve (append items (list v)) reqs))))
      ;; May accept a recv request
      (wrap-evt (channel-recv-evt req-ch)
        (lambda (req)
          ;; Accepted a recv request; add it
          (serve (cons req reqs))))
      ;; Maybe service a recv request in reqs
      (map (make-service-event items reqs)
        reqs))))
  (define (make-service-event items reqs)
    (lambda (req)
      (define pred (car req))
      (define out-ch (cadr req))
      ;; Search queue items using pred
      (find-first-item pred items
        (lambda (item)
          ;; Found an item; try to service req
          (wrap-evt (channel-send-evt out-ch 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 8
    ....)
  (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))
      ;; Send the server a request for an item matching
      ;; pred, and request reply to out-ch
      (sync (channel-send-evt (q req-ch q)
        (list pred out-ch))
      ;; Result arrives on out-ch
      (channel-recv-evt out-ch))))
)(define (msg-queue)
  ....)
  (define (serve queue reqs)
    ....)
  (define (make-service/abandon-event queue reqs)
    (lambda (req)
      (choice-event
        ;; Service event, as before
        (make-service-event queue reqs req)
        ;; Add event to detect that the receiver gives up
        (wrap-evt (caddr req) ; gave-up-event
          (lambda (void)
            ;; Receiver gave up; remove request
            (serve queue (remove req reqs))))))
  (define (msg-queue-recv-evt q pred)
    (nack-guard-event
      (lambda (gave-up-evt)
        (define out-ch (channel))
        ;; As before, but also tell the server that gave-up-event
        ;; will become ready if we give up
        (thread-resume (gave-up-evt q) (current-thread))
        (sync (channel-send-evt出来了 (q req-ch q)
          (list pred out-ch gave-up-event))))
      ;; Result arrives on out-ch
      (channel-recv-evt out-ch)))))

Figure 9. Queue with selective dequeue, first attempt

Figure 10. Revision to Figure 9

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.

The initial implementation of selective dequeue, however, contains a space leak. The following example illustrates the problem:

;;; 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 q (msg-queue))
(sync (msg-queue-send-evt q 1))
(sync (msg-queue-send-evt q 2))
(sync (choice-event
  (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 sync has abandoned a receive request. Figure 10 shows how nack-guard-event provides 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 “gave up” event to keep the request list clean.
The mag–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 “not chosen” for nack-guard-evt so that it handles this possibility. The next section completes our explanation of MzScheme’s primitives with this definition of “not chosen”.

7 Termination and NACKs
Recall that the event provided to a guard procedure by nack-guard-evt becomes “not chosen” 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 async call chose an event other than the one returned by the guard.
- Control escaped from the async call through an exception or continuation jump. The exception or jump may have been triggered through a break signal, by another guard involved in the same async, or even by the guard procedure that received the NACK event. (Continuation jumps back into a guard are always blocked, so multiple escapes are not possible.)
- The asyncing thread terminated (i.e., it is suspended and unreachable).

MzScheme’s nack-guard-evt corresponds to Concurrent ML’s withNack. An earlier version [19] of Concurrent ML offers wrapAbort, instead, and a later presentation [20] 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 correct primitive.

8 Design Considerations
Before arriving at MzScheme’s current primitives for kill-safe abstractions, we explored two main alternatives:

- **Restricted atomic sections**: As mentioned in Section 2.2, 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 “short time” that made much sense to the 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 [22], 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.

Two aspects of the current design merit further review. The first concerns the usefulness of immediate termination independent of reachability. The second relates to the transitivity of thread operations.

8.1 Terminated versus Suspended+Unreachable
The spawn function described in this paper has the primitive name thread/suspend-to-kill in MzScheme. MzScheme also provides a thread primitive, which creates a thread that is immediately terminated when it is shut down by a custodian. Immediate termination acts as hint to the run-time system, so that it can reclaim the thread’s resources.

In our implementation, the hint appears to be worthwhile. Indeed, we have used thread/suspend-to-kill so far only to implement kill-safe abstractions. Nevertheless, the hint’s usefulness most likely reflects weakness in our memory manager. In principle, immediate termination can be implemented in terms of spawn by introducing an indirecttion to the thread descriptor that is severed when the thread is shut down (so that the underlying thread descriptor become inaccessible).

8.2 Transitive Resume and Non-Transitive Promotion
The thread-resume function can yoke one thread to another for resumes. If a chain of threads becomes yoked, a resume for the initial thread propagates through the entire chain. This transitivity does not apply to the promotion of a thread to a more superior custodian.

The lack of transitivity for promotion could create a problem in the following situation. Suppose that a manager thread within an object creates several helper threads when it starts. As the object is used by different tasks, the manager thread is promoted to the control of a superior custodian, but its helpers are not automatically promoted, and therefore they may become suspended when the manager thread keeps running.

This pattern has not appeared in any implementation so far. If it did, the problem could be solved by changing the manager thread’s role so that its only job is to resume and promote helper threads; the manager’s original work would become the job of a new helper. Without concrete experience, we cannot judge whether this conversion is reasonable, or whether it is so difficult that our primitives should be refined to support transitive promotion.

9 Related Work
Much previous work addresses the interaction between termination and synchronization for specific primitives. Examples include work on monitors in Pilot [17] and remote pointers in Luna [11]. 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 [18, 19, 20].

The idea of managing a resource through an designated thread appears in many contexts, notably in microkernels [5]. Argus’s guardians [15] 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 [10] 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.

CSP-style communication [12] is especially popular for purely functional languages such as Erlang [1] and Concurrent Haskell [13]. Our handling of termination applies to a purely functional setting as well as it does for Scheme and ML. (No mutations appear in our examples.)

MzScheme does not provide a way to revoke access to an object, as in Luna [11]. It also provides no way to disable code that is associated with a task, as in Rudys and Wallach’s soft termination [21]. 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.

10 Conclusion

MzScheme provides the run-time system for the DrScheme programming environment, the PLT web server, and other applications that naturally consist of cooperating tasks. To better support such applications, we experiment with task-management constructs in MzScheme that ease cooperation without abandoning control. In the long run, we hope to build ever larger applications by composing other applications, and we hope to make this composition as simple as composing libraries.

This paper reports the latest step in our experiment, which allows a set of applications to trust a communication abstraction without trusting all clients of the abstraction, and without requiring the kernel to trust the abstraction. In effect, we have freed application programmers in MzScheme to pursue their own experiments for improving cooperation.

11 References