Concurrency Among Strangers

Programming in E as Plan Coordination

Mark S. Miller[1,2], E. Dean Tribble, and Jonathan Shapiro[1]

[1] Johns Hopkins University
[2] Hewlett Packard Laboratories

Abstract. Programmers write programs, expressing plans for machines to execute. When composed so that they may cooperate, plans may instead interfere with each other in unanticipated ways. *Plan coordination* is the art of simultaneously enabling plans to cooperate, while avoiding hazards of destructive plan interference. For sequential computation within a single machine, object programming supports plan coordination well. For concurrent computation, this paper shows how hard it is to use locking to prevent plans from interfering without also destroying their ability to cooperate.

In Internet-scale computing, machines proceed concurrently, interact across barriers of large latencies and partial failure, and encounter each other's misbehavior. Each dimension presents new plan coordination challenges. This paper explains how the E language addresses these joint challenges by changing only a few concepts of conventional sequential object programming. Several projects are adapting these insights to existing platforms.

1 Introduction

The fundamental constraint we face as programmers is complexity. It might seem that we could successfully formulate plans only for systems we can understand. Instead, every day, programmers successfully contribute code towards working systems too complex for anyone to understand *as a whole*. We make use of modularity and abstraction mechanisms to construct systems whose component plans we can understand piecemeal, and whose compositions we can understand without fully understanding each plan being composed.

> Programmers are not to be measured by their ingenuity and their logic but by the completeness of their case analysis.
>
> —Alan Perlis

In the human world, when you plan for yourself, you make assumptions about future situations in which your plan will unfold. Occasionally, someone else's plan may interfere with yours, invalidating the assumptions on which your plan is based. To plan successfully, you need some sense of which assumptions are usually safe from such disruption. You do not need to anticipate every possible contingency, however. If someone does something you did not expect, you will probably be better able to figure out how to cope at that time anyway.

To formulate plans for machines to execute, programmers must also make assumptions. When separately formulated plans are composed, conflicting assumptions can cause the run-time situation to become *inconsistent* with a given plan's assumptions, leading it awry. By dividing the state of a computational system into separately encapsulated objects, and by giving objects limited access to each other, we limit outside interference and extend the range of assumptions our programs may safely rely upon.[1] Beyond these assumptions, correct programs must handle all relevant contingencies. By abstraction, we limit one object's need for knowledge of others, reducing the number of cases which are relevant. However, even under sequential and benign conditions, the remaining case analysis can still be quite painful.

Under concurrency, an object's own plans may destructively interfere with each other. In distributed programming, asynchrony and partial failure limit an object's local knowledge of relevant facts, increasing the number of relevant cases it must consider. In secure programming, we carefully distinguish those objects whose good behavior we rely on from those we don't, but we seek to cooperate with both. Confidentiality further constrains local knowledge; deceit and malice are further sources of possible plan interference. Each of these dimensions threatens an explosion of new cases we must consider. To succeed, we must find ways of reducing the size of the resulting case analysis.

Previous papers have focused on E's support for limited trust within the constraints of distributed systems [MMF00, MYS03, MS03, MTS04]. This paper focuses on E's support for concurrent and distributed programming within the constraints of limited trust.

2 Overview

Throughout this paper, we do not seek universal solutions to coordination problems, but rather, abstraction mechanisms adequate to craft diverse solutions adapted to the needs of many applications. We illustrate many of our points with a simple example, a "statusHolder" object implementing the listener pattern.

The Sequential StatusHolder introduces the statusHolder and examines its hazards in a sequential environment.

Why Not Shared-state Concurrency shows several attempts at a conventionally thread-safe statusHolder in Java and the ways each suffers from plan interference.

A Taste of E shows a statusHolder written in E and explains E's eventual-send operator in the context of a single thread of control.

[1] This view of encapsulation and composition parallels Hayek's explanation of how property rights protect human plans from interference and how trade brings about their cooperative alignment [vH45]. See [MD88, TM02] for more.

Communicating Event-Loops explains how the statusHolder handles concurrency and distribution under benign conditions.

Protection from Misbehavior examines how the plans coordinated by our statusHolder are and are not vulnerable to each other.

Promise Pipelining introduces promises for the results of eventually-sent messages, and shows how pipelining helps programs tolerate latency and how broken promise contagion lets programs handle eventually-thrown exceptions.

Partial Failure shows how statusHolder's clients can regain access following a partition or crash and explains the issues involved in regaining distributed consistency.

The When-Catch Expression explains how to turn data-flow back into control-flow.

From Objects to Actors and Back Again presents a brief history of E's concurrency control.

Related Work discusses other systems with similar goals, as well as current projects adapting these insights to existing platforms.

Discussion and Conclusions summarizes current status, what remains to be done, and lessons learned.

3 The Sequential StatusHolder

Throughout the paper, we will examine different forms of the listener pattern [Eng97]. The code below is representative of the basic sequential listener pattern.[2] In it, a statusHolder object is used to coordinate a changing status between *publishers* and *subscribers*. A subscriber can ask for the current status of a statusHolder by calling `getStatus`, or can subscribe to receive notifications when the status changes by calling `addListener` with a listener object. A publisher changes the status in a statusHolder by calling `setStatus` with the new value. This in turn will call `statusChanged` on all subscribed listeners. In this way, publishers can communicate status updates to subscribers without knowing of each individual subscriber.

We can use this pattern to coordinate several loosely coupled plans. For example, in a simple application, a bank account manager publishes an account balance to an analysis spreadsheet and a financial application. Deposits and withdrawals cause a new balance to be published. The spreadsheet adds a listener that will update the display to show the current balance. The finance application adds a listener to begin trading activities when the balance falls below some threshold. Although these clients interact cooperatively, they know very little about each other.

[2] The listener pattern [Eng97] is similar to the observer pattern [GHJV94]. However, the analysis which follows would be quite different if we were starting from the observer pattern.

```
public class StatusHolder {
    private Object myStatus;
    private final ArrayList<Listener> myListeners
                    = new ArrayList();

    public StatusHolder(Object status) {
        myStatus = status;
    }
    public void addListener(Listener newListener) {
        myListeners.add(newListener);
    }
    public Object getStatus() {
        return myStatus;
    }
    public void setStatus(Object newStatus) {
        myStatus = newStatus;
        for (Listener listener: myListeners) {
            listener.statusChanged(newStatus);
        }
    }
}
```

Even under sequential and benign conditions, this pattern creates plan interference hazards.

Aborting the wrong plan: If a listener throws an exception, this prevents some other listeners from being notified of the new status and possibly aborts the publisher's plan. In the above example, the spreadsheet's inability to display the new balance should not impact either the finance application or the bank account manager.

Nested subscription: The actions of a listener could cause a new listener to be subscribed. For example, to bring a lowered balance back up, the finance application might initiate a stock trade operation, which adds its own listener. Whether that new listener sees the current event, fails to see the current event, or fails to be subscribed depends on minor details of the listener implementation.

Nested publication: Similarly, a listener may cause a publisher to publish a new status, possibly unknowingly due to aliasing. For example, during an update, the invocation of `setStatus` notifies the finance application, which deposits money into the account. A new update to the balance is published and an inner invocation of `setStatus` notifies all listeners of the new balance. After that inner invocation returns, the outer invocation of `setStatus` continues notifying listeners of the older, pre-deposit balance. Some of the listeners would receive the notifications *out of order.* As a result, the spreadsheet might leave the display showing the wrong balance, or worse, the finance application might initiate transactions based on incorrect information.

The nested publication hazard is especially striking because it reveals that problems typically associated with concurrency may arise even in a simple sequential example. This is why we draw attention to *plans*, rather than programs or processes. The statusHolder, by running each subscriber's plan during a step of a publisher's plan, has provoked plan interference: these largely independent plans now interact in surprising ways, creating numerous new cases that are difficult to identify, prevent, or test. Although these hazards are real, experience suggests that programmers can usually find ways to avoid them in sequential programs under benign conditions.

4 Why Not Shared-State Concurrency

With genuine concurrency, interacting plans unfold in parallel. To manipulate state and preserve consistency, a plan needs to ensure others are not manipulating that same state at the same time. This section explores the plan coordination problem in the context of the conventional shared-state concurrency-control paradigm [VH04], also known as shared-memory multi-threading. We present several attempts at a conventionally *thread-safe* statusHolder—searching for one that prevents its clients from interfering without preventing them from cooperating.

In the absence of real-time concerns, we can analyze concurrency without thinking about genuine parallelism. Instead, we can model the effects of concurrency as the non-deterministic interleaving of atomic units of operation. We can roughly characterize a concurrency-control paradigm with the answers to two questions:

Serializability: What are the coarsest-grain units of operation, such that we can account for all visible effects of concurrency as equivalent to some fully ordered interleaving of these units [IBM68]? For shared-state concurrency, this unit is generally no larger than a memory access, instruction, or system call—which is often finer than the "primitives" provided by our programming languages [Boe05]. For databases, this unit is the transaction.

Mutual exclusion: What mechanisms can eliminate the possibility of some interleavings, so as to preclude the hazards associated with them? For shared-state concurrency, the two dominant answers are monitors [Hoa74, BH93] and rendezvous [Hoa78]. For distributed programming, many systems restrict the orders in which messages may be delivered [BJ87, Ami95, Lam98].

Java is loosely in the monitor tradition. Ada, Concurrent ML, and the synchronous π-calculus are loosely in the rendezvous tradition. With minor adjustments, the following comments apply to both.

4.1 Preserving Consistency

If we place our sequential statusHolder into a concurrent environment, publishers or subscribers may call it from different threads. The resulting interleaving of

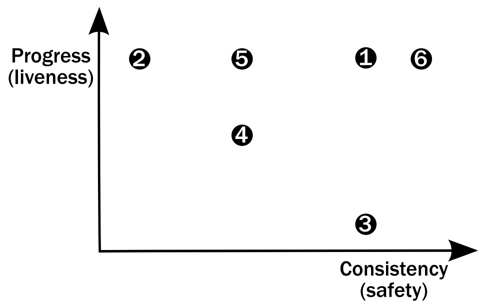

Fig. 1. A correct program must both remain consistent and continue to make progress. The sequence above represents our search for a statusHolder which supports both well: (1) The sequential statusHolder. (2) The sequential statusHolder in a concurrent environment. (3) The fully synchronized statusHolder. (4) Placing the for-loop outside the synchronized block. (5) Spawning a new thread per listener notification. (6) Using communicating event-loops.

operations might, for example, mutate the `myListeners` list while the for-loop is in progress.

Adding the "`synchronized`" keyword to all methods of the above code causes it to resemble a monitor. This fully synchronized statusHolder eliminates exactly those cases where multiple plans interleave within the statusHolder. It is as good at preserving its own consistency as our original sequential statusHolder was.

However, it is generally recommended that Java programmers avoid this fully synchronized pattern because it is prone to deadlock [Eng97]. Although each listener is called from some publisher's thread, its purpose may be to contribute to a plan unfolding in its subscriber's thread. To defend itself against such concurrent entry, the objects at this boundary may themselves be synchronized. If a `statusChanged` notification gets blocked here, waiting on that subscriber's thread, it blocks the statusHolder, as well as any other objects whose locks are held by that publisher's thread. If the subscriber's thread is itself waiting on one of these objects, we have a classic deadly embrace.

Although we have eliminated interleavings that lead to inconsistency, some of the interleavings we eliminated were necessary to make progress.

4.2 Avoiding Deadlock

To avoid this problem, [Eng97] recommends changing the `setStatus` method to clone the listeners list within the synchronized block, and then to exit the block before entering the for-loop, as shown by the code below. This pattern avoids holding a lock during notification and thus avoids the obvious deadlock described above between a publisher and a subscriber. It does not avoid the underlying hazard, however, because the publisher may hold other locks.

```
public void setStatus(Object newStatus) {
    ArrayList<Listener> listeners;
    synchronized (this) {
        myStatus = newStatus;
        listeners = (ArrayList<Listener>)myListeners.clone();
    }
    for (Listener listener: listeners) {
        listener.statusChanged(newStatus);
    }
}
```

For example, if the account manager holds a lock on the bank account during a withdrawal, a deposit attempt by the finance application thread may result in an equivalent deadlock, with the account manager waiting for the notification of the finance application to complete, and the finance application waiting for the account to unlock. The result is that all the associated objects are locked and other subscribers will never hear about this update. Thus, the underlying hazard remains.

In this approach, some interleavings needed for progress are still eliminated, and as we will see, some newly-allowed interleavings lead to inconsistency.

4.3 Race Conditions

The approach above has a consistency hazard: if `setStatus` is called from two threads, the order in which they update `myStatus` will be the order they enter the synchronized block above. However, the for-loop notifying listeners of a later status may race ahead of one that will notify them of an earlier status. As a result, even a single subscriber may see updates out of order, so the spreadsheet may leave the display showing the wrong balance, even in the absence of any nested publication.

It is possible to adjust for these remaining problems. The style recommended for some rendezvous-based languages, like Concurrent ML and the π-calculus, corresponds to spawning a separate thread to perform each notification. This avoids using the producer's thread to notify the subscribers and thus avoids the deadlock hazard—it allows all interleavings needed for progress. However, this style still suffers from the same race condition hazards and so still fails to eliminate the right interleavings. We could compensate for this by adding a counter to the statusHolder and to the notification API, and by modifying the logic of all listeners to reorder notifications. But a formerly trivial pattern has now exploded into a case-analysis minefield. Actual systems contain thousands of patterns more complex than the statusHolder. Some of these will suffer from less obvious minefields.

> This is "Multi-Threaded Hell". As your application evolves, or as different programmers encounter the sporadic and non-reproducible corruption or deadlock bugs, they will add or remove locks around different

data structures, causing your code base to veer back and forth . . . , erring first on the side of more deadlocking, and then on the side of more corruption. This kind of thrashing is bad for the quality of the code, bad for the forward progress of the project, and bad for morale.

—An experience report from the development of Mojo Nation [WO01]

5 A Taste of E

Before revisiting the issues above, let's first use this example to briefly explain E as a sequential object language. (For a more complete explanation of E, see [Sti04].) Here is the same statusHolder as defined in E.

```
def makeStatusHolder(var myStatus) {
    def myListeners := [].diverge()
    def statusHolder {
        to addListener(newListener) {
            myListeners.push(newListener)
        }
        to getStatus() { return myStatus }
        to setStatus(newStatus) {
            myStatus := newStatus
            for listener in myListeners {
                listener.statusChanged(newStatus)
            }
        }
    }
    return statusHolder
}
```

E has no classes. Instead, the expression beginning with "`def statusHolder`" is an object definition expression. It creates a new object with the enclosed method definitions and binds the new `statusHolder` variable to this object. An invocation, such as "`statusHolder.setStatus(33)`", causes a message to be delivered to an object. When an object receives a message, it reacts according to the code of its matching method. As with Smalltalk [GR83] or Actors [HBS73], all values are objects, and all computation proceeds only by delivering messages to objects.

From a λ-calculus perspective, an object definition expression is a lambda expression, in which the (implicit) parameter is bound to the incoming message and the body selects a method to run according to the message. The delivery of a message to an object is the application of an object-as-closure to a message-as-argument. An object's behavior is indeed a function of the message it is applied to. This view of objects goes back to Smalltalk-72 [GK76] and Actors, and is hinted at earlier in [Hoa65]. Also see [SS04].

Unlike a class definition, an object definition does not declare its instance variables. Instead, the instance variables of an object are simply the variables used freely within the object definition (which therefore must be defined in some lexically enclosing scope). The instance variables of statusHolder are `myStatus` and `myListeners`. Variables are unassignable by default; the "`var`" keyword defines `myStatus` as an assignable variable. Square brackets evaluate to an immutable list containing the values of the subexpressions (the empty-list in the example). Lists respond to the "`diverge()`" message by returning a new mutable list whose initial contents are a snapshot of the diverged list. Thus, `myListeners` is initialized to a new, empty, mutable list, which acts much like an `ArrayList`.

E provides syntactic shorthands to use objects that define a "`run`" method as if they were functions. The syntax for `makeStatusHolder` is a shorthand for defining an object with a single "`run`" method. It expands to:

```
def makeStatusHolder {
    to run(var myStatus) { ...
```

The corresponding function call syntax, "`makeStatusHolder(44)`", is shorthand which expands to "`makeStatusHolder.run(44)`". Each time `makeStatusHolder` is called, it defines and returns a new statusHolder.

5.1 Two Ways to Postpone Plans

The E code for statusHolder above retains the simplicity and hazards of the sequential Java version. To address these hazards requires examining the underlying issues. When the statusHolder—or any agent—is executing plan X and discovers the need to engage in plan Y, in a sequential system, it has two simple alternatives of when to do Y:

Immediately: Put X aside, work on Y until complete, then go back to X.
Eventually: Put Y on a "to-do" list and work on it after X is complete.

The "immediate" option corresponds to conventional, sequential call-return control flow (or strict applicative-order evaluation), and is represented by the "`.`" or *immediate-call* operator, which delivers the message immediately. Above, statusHolder's `addListener` method tells `myListeners` to push the `newListener` *immediately*. When `addListener` proceeds past this point, it may assume that all side effects it requested are done.

For the statusHolder example, all of the sequential hazards (e.g., Nested Publication) and many of the concurrent hazards (deadlock) occur because the `statusChanged` method is also invoked immediately: the publisher's plan is set aside to pursue the listener's plan (which might then abort, change the state further, etc.).

The "eventual" option corresponds to the human notion of a "to-do" list: the item is queued for later execution. E provides direct support for this asynchronous messaging option, represented by the "`<-`" or *eventual-send* operator. Using eventual-send, the `setStatus` method can ensure that each listener will be notified of the changed status in such a way that it does not interfere with the statusHolder's current plan. To accomplish this in E, the `setStatus` method becomes:

```
to setStatus(newStatus) {
    myStatus := newStatus
    for listener in myListeners {
        listener <- statusChanged(newStatus)
    }
}
```

As a result of using eventual-send above, all of the sequential hazards are addressed. Errors, new subscriptions, and additional status changes caused by listeners will all take place after all notifications for a published event have been scheduled. Publishers' plans and subscribers' plans are temporally isolated—so these plans may unfold with fewer unintended interactions. For example, it can no longer matter whether `myStatus` is assigned before or after the for-loop.

5.2 Simple E Execution

This section describes how temporal isolation is achieved within a single thread of control. The next section describes how it is achieved in the face of concurrency and distribution.

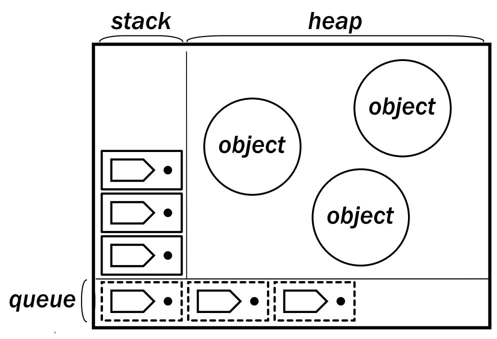

Fig. 2. An E vat consists of a heap of objects and a thread of control. The stack and queue together record the postponed plans the thread needs to process. An immediate-call pushes a new frame on top of the stack, representing the delivery of a message (*arrow*) to a target object (*dot*). An eventual-send enqueues a new pending delivery on the right end of the queue. The thread proceeds from top to bottom and then from left to right.

In E, an eventual-send creates and queues a *pending delivery*, which represents the eventual delivery of a particular message to a particular object. Within a single thread of control, E has both a normal execution stack for immediate call-return and a queue containing all the pending deliveries. Execution proceeds by taking a pending-delivery from the queue, delivering its message to its object, and processing all the resulting immediate-calls in conventional call-return order. This is called a *turn*. When a pending delivery completes, the next one is dequeued, and so forth. This is the classic event-loop model, in which all of the events are pending deliveries. Because each event's turn runs to completion before the next is serviced, they are temporally isolated.

Additional mechanisms to process results and exceptions from eventual-sends will be discussed in further sections below.

The combination of a stack, a pending delivery queue, and the heap of objects they operate on is called a *vat*, illustrated in Figure 2.[3] Each E object lives in exactly one vat and a vat may host many objects. Each vat lives on one machine at a time and a machine may host many vats. The vat is also the minimum unit of persistence, migration, partial failure, resource control, and defense from denial of service. We will return to some of these topics below.

6 Communicating Event-Loops

We now consider the case where our account (including account manager and its statusHolder) runs in VatA on one machine, and our spreadsheet (including its listener) runs in VatS on another machine.

In E, we distinguish several reference-states. A direct reference between two objects in the same vat is a *near reference*.[4] As we have seen, near references carry both immediate-calls and eventual-sends. Only *eventual references* may cross vat boundaries, so the spreadsheet holds an eventual reference to the statusHolder, which in turns holds an eventual reference to the spreadsheet's listener. Eventual references are first class—they can be passed as arguments, returned as results, and stored in data structures, just like near references. However, eventual references carry only eventual-sends, not immediate-calls—an immediate-call on an eventual reference throws an exception. Our statusHolder is compatible with this constraint, since it stores, retrieves, and eventual-sends to its listeners, but never immediate-calls them. Figure 3 shows what happens when a message is sent between vats.

When the statusHolder in VatA performs an eventual-send of the `statusChanged` message to the spreadsheet's listener in VatS, VatA creates a pending delivery as before, recording the need to deliver this message to this listener. Pending deliveries need to be queued on the pending delivery queue of the vat hosting the object that will receive the message—in this case, VatS. VatA serializes (marshals) the pending delivery onto an encrypted, order-preserving byte stream read by VatS. Should it ever arrive at VatS, VatS will unserialize it and queue it on its own pending delivery queue.

Since each vat runs concurrently with all other vats, turns in different vats no longer have actual temporal isolation. If VatS is otherwise idle, it may service this delivery, notifying the spreadhseet's listener of the new balance, while the original turn is still in progress in VatA. But so what? These two turns can

[3] Figures 2–5 were created by Ka-Ping Yee with input from the e-lang community.

[4] For brevity, we generally do not distinguish a near reference from the object it designates.

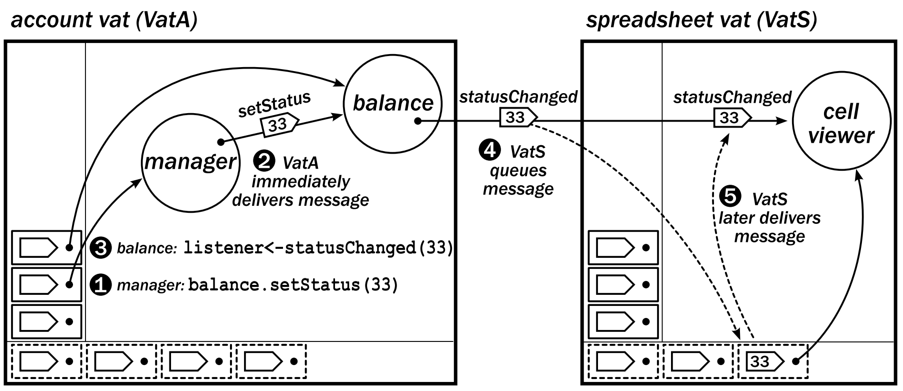
account vat (VatA)
spreadsheet vat (VatS)
balance
setStatus
33
manager
❷ VatA immediately delivers message
❸ balance: listener<-statusChanged(33)
❶ manager: balance.setStatus(33)
statusChanged
33
❹ VatS queues message
statusChanged
33
cell viewer
❺ VatS later delivers message
33

Fig. 3. If the account manager and the spreadsheet are in separate vats, when the account manager (1) tells the statusHolder that represents its balance to immediately update, this (2) transfers control to the statusHolder, which (3) notes that its listeners should eventually be notified. The message is (4) sent to the spreadsheet's vat, which queues it on arrival and eventually (5) delivers it to the listener, which updates the display of the spreadsheet cell.

only execute simultaneously when they are in different vats. In this case, the spreadsheet cannot affect the account manager's turn-in-progress. Because only eventual references span between vats, the spreadsheet can only affect VatA by eventual-sending to objects hosted by VatA. This cannot affect any turn already in progress in VatA—VatA only queues the pending delivery, and will service it sometime after the current turn and turns for previously queued pending deliveries, complete.

Only near references provide one object synchronous access to another. Therefore an object has synchronous access to state only within its own vat. Taken together, these rules guarantee that a running turn—a sequential call-return program—has mutually exclusive access to everything to which it has synchronous access. In the absence of real-time concerns, this provides all the isolation that was achieved by temporal isolation in the single-threaded case.

The net effect is that a turn is E's unit of operation. We can faithfully account for the visible effects of concurrency without any interleaving of the steps within a turn. Any actual multi-vat computation is equivalent to some fully ordered interleaving of turns.[5] Because E has no explicit locking constructs, computation

[5] An E turn may never terminate, which is hard to account for within this simple model of serializability. There are formal models of asynchronous systems that can account for non-terminating events [CL85]. Within the scope of this paper, we can safely ignore this issue.

The actual E system does provide synchronous file I/O operations. When these files are local, prompt, and private to the vat accessing them, this does not violate turn isolation, but since files may be remote, non-prompt, or shared, the availability of these synchronous I/O operations does violate the E model.

within a turn can never block—it can only run, to completion or forever. A vat as a whole is either processing pending deliveries, or is idle when there are no pending deliveries to service. Because computation never blocks, it cannot deadlock. Other lost progress hazards are discussed in the section on "Datalock" below.

As with database transactions, the length of an E turn is not predetermined. It is a tradeoff left for the developer to decide. How the object graph is carved up into vats and how computation is carved up into turns will determine which interleaving cases are eliminated, and which must be handled explicitly by the programmer. For example, when the spreadsheet was co-located with the statusHolder, it could immediate-call both `getStatus` and `addListener` in order to ensure that the spreadsheet's cell sees exactly the updates to an initial valid state. But when it can only eventual-send these messages, they may arrive at the statusHolder interleaved with other messages. To relieve potentially remote clients of this burden, the statusHolder should send an initial notification to newly subscribed listeners:

```
to addListener(newListener) {
    myListeners.push(newListener)
    newListener <- statusChanged(myStatus)
}
```

6.1 Issues with Event-Loops

This architecture imposes some strong constraints on programming (e.g., no threads or coroutines), which can impede certain useful patterns of plan cooperation. In particular, recursive algorithms, such as recursive-descent parsers, must a) happen entirely within a single turn, b) be redesigned (e.g., as a table-driven parser), or c) if it needs external non-prompt input (e.g., a stream from the user), be run in a dedicated vat. E programs have used each of these approaches.

Thread-based coordination patterns can typically be adapted to vat granularity. For example, rather than adding the complexity of a priority queue for pending deliveries, different vats would simply run at different processor priorities. For example, if a user-interaction vat *could* proceed (has pending deliveries in its queue), it should; a helper "background" vat (e.g., spelling check) should consume processor resources only if no user-directed action could proceed. A divide-and-conquer approach for multi-processing could run a vat on each processor and divide the problem among them. The event-loop approach is unsuitable for problems that cannot easily be adapted to a message-passing hardware architecture, such as fluid dynamics computation.

7 Protection from Misbehavior

When using a language that supports shared-state concurrency, one can choose to avoid it and adopt the event-loop style instead. Indeed, several Java libraries, such as AWT, were initially designed to be thread-safe, and were then redesigned around event-loops. Using event-loops, one can easily write a Java class equivalent to our `makeStatusHolder`. If one can so easily choose to avoid shared-state concurrency, does E actually need to prohibit it?

E uses the event-loop approach to simplify the task of preserving consistency while maintaining progress. Preserving consistency stays simple for the statusHolder only if it executes in at most one thread at a time. As we discussed previously, the possibility of multiple threads would necessitate complex locking. If one of its clients *could* create a new thread and call it, then the simple version of the statusHolder could not preserve consistency (i.e., it would need to perform the complex locking mentioned in the previous section).

In the extreme case, one object may actively intend to disrupt the plans of another. This leads us to examine plan coordination in the presence of malicious behavior. The topic is of interest both because large and distributed systems in practice need to handle potentially malicious components, and because analysis of the malicious case can help uncover hazards that are already present in the non-malicious case.

7.1 Defensive Correctness

If a user browsing a webserver were able to cause incorrect pages to be displayed to other users, we would likely consider it a bug in the webserver—we expect it to remain correct regardless of the client's behavior. We call this property *defensive correctness*: a program P is defensively correct if it remains correct despite arbitrary behavior on the part of its clients. Before this definition can be useful, we need to pin down what we mean by "arbitrary" behavior.

When we say that a program P is correct, this normally means that we have a specification in mind, and that P behaves according to that specification. There are some implicit caveats in that assertion. For example, P cannot behave at all unless it is run on a machine; if the machine operates incorrectly, P on that machine may behave in ways that deviate from its specification. We do not consider this to be a bug in P, because P's correctness implicitly depends on the machine's correctness. If P's correctness depends on another component R's correctness, we will say that P *relies upon* R. For example, a typical webserver relies on the underlying machine and on operating system features such as files and sockets. We will refer to the set of all elements on which P relies as P's *reliance set*.[6]

We define Q's *authority* as the set of effects Q could cause. With regard to P's correctness, Q's *relevant authority* is bounded by the assumption that everything in P's reliance set is correct, since P was defined under this assumption.

[6] The set of all things that P relies on is similar in concept to P's "Trusted Computing Base" or TCB. "Rely" articulates the objective situation (P is vulnerable to R), and so avoids confusions engendered by the word "trust".

While the focus in this paper is on correctness, a similar "reliance" analysis could be applied to other program properties, such as promptness [Har85].

For example, if a user could cause a webserver to show the wrong page to other browsers by replacing a file through an operating system exploit, then the underlying operating system would be incorrect, not the webserver. We say that P *protects against* Q if P remains correct despite any of the effects in Q's relevant authority, that is, despite any possible actions by Q, assuming the correctness of P's reliance set.

Now we can speak more precisely about defensive correctness. The "arbitrary behavior" mentioned earlier is the combined relevant authority of an object's clients. P is *defensively correct* if it protects against all of its clients. The focus is on *clients* in particular in order to enable the composition of correct components into larger correct systems. If P relies on R, then P also relies on all of R's other clients *unless* R is defensively correct. If R does not protect against its other clients, P cannot prevent them from interfering with its own plan, which makes it infeasible for P to ensure its own correctness. By not relying on its clients, R enables them to avoid relying on each other.

This explains why it is important for E to forbid the spawning of threads. As we saw earlier, it can be very difficult to write programs in which threads protect against each other. Removing threads eliminates a key obstacle to defensive correctness.

Correctness can be divided into consistency (safety) and progress (liveness). An object that is vulnerable to denial-of-service by its clients may nevertheless be *defensively consistent.* Given that all the objects it relies on themselves remain consistent, a defensively consistent object will never give incorrect service to well-behaved clients, but it may be prevented from giving them any service. While a defensively correct object is invulnerable to its clients, a defensively consistent object is merely incorruptible by its clients.

Different security properties are feasible at different granularities. Some conventional operating systems attempt to provide support for protecting users from each other's misbehavior. But because programs are normally run with their user's full authority, all software run under the same account is mutually reliant: since each is granted the authority to corrupt the others via underlying components on which they all rely, they cannot usefully protect against such "friendly fire".[7] Some operating system designs [DH65] support process-granularity defensive consistency. Others, by providing principled controls over computational resource rights [Har85, SSF99], can also protect against denial of service. Among machines distributed over today's Internet, cryptographic protocols help support defensive consistency, but defensive correctness remains infeasible.

In most programming languages, all objects in the same process are mutually reliant. A secure language is one which supports some useful form of protections within a process. Among objects in the same vat, E supports defensive consistency: Any object may go into an infinite loop, thereby preventing the progress of all other objects within their vat. Therefore, within E's architecture, defensive correctness *within* a vat is impossible. With respect to progress, all objects within

[7] See [SKYM04] for an unconventional way to use conventional OSes to provide greater security.

the same vat are mutually reliant. In many situations, defensive consistency is adequate—a potential adversary often has more to gain from corruption than denial of service. This is especially so in iterated relationships, since corruption may misdirect plans but go undetected, while loss of progress is quite noticeable.

7.2 Principle of Least Authority (POLA)

Our statusHolder itself is now defensively consistent, but is it a good abstraction for the account manager to rely on to build its own defensively consistent plans? In our example scenario, we have been assuming that the account manager acts only as a publisher and that the finance application and spreadsheet act only as subscribers. However either subscriber *could* invoke the `setStatus` method. If the finance application calls `setStatus` with a bogus balance, the spreadsheet will dutifully render it.

This is a problem of access control. The statusHolder, by bundling two kinds of authority into one object, encouraged patterns where both kinds of authority were provided to objects that only needed one. This can be addressed by grouping these methods into separate objects, each of which represents a sensible bundle of authority.

```
def makeStatusPair(var myStatus) {
    def myListeners := [].diverge()
    def statusGetter {
        to addListener(newListener) {
            myListeners.push(newListener)
            newListener <- statusChanged(myStatus)
        }
        to getStatus() { return myStatus }
    }
    def statusSetter {
        to setStatus(newStatus) {
            myStatus := newStatus
            for listener in myListeners {
                listener <- statusChanged(newStatus)
            }
        }
    }
    return [statusGetter, statusSetter]
}
```

Now the account manager can make use of `makeStatusPair` as follows:

```
def [sGetter, sSetter] := makeStatusPair(33)
```

The call to `makeStatusPair` on the right side makes four objects—an object representing the `myStatus` variable, a mutable `myListeners` list, a `statusGetter`, and a `statusSetter`. The last two each share access to the first two. The call to `makeStatusPair` returns a list holding these last two objects. The left side pattern-matches this list, binding `sGetter` to the new `statusGetter`, and binding `sSetter` to the new `statusSetter`.

The account manager can now keep the new `statusSetter` for itself and give the spreadsheet and the finance application access only to the new `statusGetter`. More generally, we may now describe publishers as those with access to `statusSetter` and subscribers as those with access to `statusGetter`. The account manager can now provide consistent balance reports to its clients because it has denied them the possibility of corrupting this service.

As with concurrency control, the key to access control is to allow the possibilities needed for cooperation, while limiting the possibilities that would allow for plan interference. We wish to provide objects the authority needed to carry out their proper duties—publishers gotta publish—but little more. This is known as *POLA*, the *Principle of Least Authority* (See [MS03] for the relationship between POLA and the Principle of Least Privilege [SS75]). By not granting its subscribers the authority to publish a bogus balance, the account manager no longer needs to worry about what would happen if they did. This discipline helps us compose plans so as to allow well-intentioned plans to successfully cooperate, while minimizing the kinds of plan interference they must defend against.

7.3 A Taste of E Across a Network

E's computational model extends across the network. An eventual reference in a vat can refer to an object in a vat on another machine; eventual-sends to that reference are sent across an encrypted, authenticated link and posted as pending deliveries for the target object on the remote vat.

E's network protocol, Pluribus, actually runs between vats, not between machines. Therefore, we can ignore the distinction between vats and machines without loss of generality. An incorrect machine is, from our perspective, simply a set of incorrect vats; i.e., vats that do not implement the language and/or protocol correctly. The design of Pluribus is beyond the scope of this document, but a few words are in order.

Pluribus enforces characteristics of the E computational model, such as reference integrity, so that E programs can rely on those properties between vats and therefore between machines. Even if a remote vat runs its objects in an unsafe language like C++, other vats could still view it from a correctness point of view as a set of (possibly incorrect) objects written in E. From the perspective of other vats, the objects in the remote vat could collude and act arbitrarily within the union of the authorities granted to any of them, but they cannot feasibly[8] manufacture new authorities. Thus, if an object relies on another object in a remote vat, then it also relies on that remote vat (because the remote object relies on that vat).

[8] Pluribus relies on the standard cryptographic assumptions that large random numbers are not feasibly guessable, and that well-accepted algorithms are immune to feasible cryptanalysis.

8 Promise Pipelining

The eventual-send examples so far were carefully selected to be evaluated only for their effects, with no use made of the value of these expressions. This section discusses the handling of return results and exceptions produced by eventual-sends.

8.1 Promises

As discussed previously, eventual-sends queue a pending delivery and complete immediately. The return value from an eventual-send operation is called a *promise* for the eventual result. The promise is not a near reference for the result of the eventual-send because the eventual-send cannot have happened yet (i.e., it will happen in a later turn). Instead, the promise is an eventual-reference for the result. A pending delivery, in addition to the message and reference to the target object, includes a *resolver* for the promise, which provides the right to choose what the promise designates. When the turn spawned by the eventual-send completes, its vat reports the outcome to the resolver, *resolving* the promise so that the promise eventually becomes a reference designating that outcome, called the *resolution*.

Once resolved, the promise is equivalent to its resolution. Thus, if it resolves to an eventual-reference for an object in another vat, then the promise becomes that eventual reference. If it resolves to an object that can be passed by copy between vats, then it becomes a near-reference to that object.

Because the promise starts out as an eventual reference, messages can be eventually-sent to it even *before* it is resolved. Messages sent to the promise cannot be delivered until the promise is resolved, so they are buffered in FIFO order within the promise. Once the promise is resolved, these messages are forwarded, in order, to its resolution.

8.2 Pipelining

Since an object can eventual-send to the promises resulting from previous eventual-sends, functional composition is straightforward. If object *L* in VatL executes

```
def r3 := x <- a() <- c(y <- b())
```

or equivalently

```
def r1 := x <- a()
def r2 := y <- b()
def r3 := r1 <- c(r2)
```

and `x` and `y` are on VatR, then all three requests are serialized and streamed out to VatR immediately and the turn in VatL continues without blocking. By contrast, in a conventional RPC system, the calling thread would only proceed after multiple network round trips.

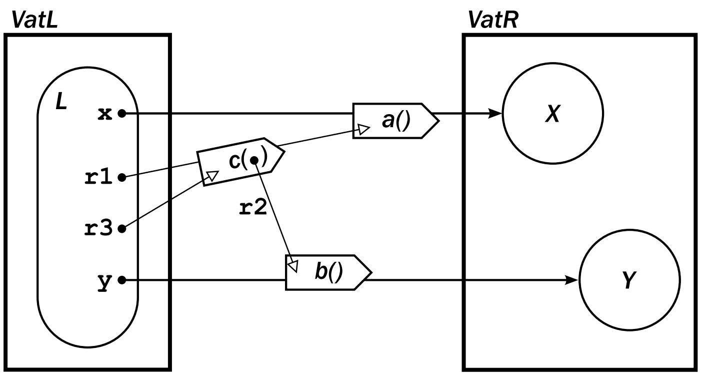

Fig. 4. The three messages in `def r3 := x <- a() <- c(y <- b())` are streamed out together, with no round trip. Each message box "rides" on the reference it is sent on. References `x` and `y` are shown with solid arrowheads, indicating that their target is known. The others are *promises*, whose open arrowhead represents their *resolvers*, which provide the right to choose their promises' value.

Figure 4 depicts an unresolved reference as an arrow stretching between its promise-end, the tail held by `r1`, and its resolver, the open arrowhead within the pending delivery sent to VatR. Messages sent on a reference always flow towards its destination and so "move" as close to the arrowhead as possible. While the pending delivery for `a()` is in transit to VatR, so is the resolver for `r1`, so we send the `c(r2)` message there as well. As VatR unserializes these three requests, it queues the first two in its local to-do list, since their target is known and local. It sends the third, `c(r2)`, on a local promise that will be resolved by the outcome of `a()`, carrying as an argument a local promise for the outcome of `b()`.

If the resolution of `r1` is local to VatR, then as soon as `a()` is done, `c(r2)` is immediately queued on VatR's to-do list and may well be serviced before VatL learns of `r1`'s resolution. If `r1` is on VatL, then `c(r2)` is streamed back towards VatL just behind the message informing VatL of `r1`'s resolution. If `r1` is on yet a third vat, then `c(r2)` is forwarded to that vat.

Across geographic distances, latency is already the dominant performance consideration. As hardware improves, processing will become faster and cheaper, buffers larger, and bandwidth greater, with limits still many orders of magnitude away. But latency will remain limited by the speed of light. Pipes between fixed endpoints can be made wider but not shorter. Promise pipelining reduces the impact of latency on remote communication. Performance analysis of this type of protocol can be found in Bogle's "Batched Futures" [BL94]; the promise pipelining protocol is approximately a symmetric generalization of it.

8.3 Datalock

Promise chaining allows some plans, like `c(r2)`, to be postponed pending the resolution of previous plans. We introduce other ways to postpone plans below.

Using the primitives introduced so far, however, it is possible to create circular data dependencies which, like deadlock, are a form of lost-progress bug. We call this kind of bug, *datalock*. For example, the `epimenides` function below returns a promise for the boolean opposite of `flag`.

```
var flag := true
def epimenides() { return flag <- not() }
```

If `flag` were assigned to the result of invoking `epimenides` eventually, datalock would occur.

```
flag := epimenides <- run()
```

In the current turn, a pending-delivery of `epimenides <- run()` is queued, and a promise for its result is immediately assigned to `flag`. In a later turn when `epimenides` is invoked, it eventual-sends a message to the promise in `flag`, and then resolves the `flag` promise to the new promise for the `not()` sent to that *same* `flag` promise. The datalock is created, not because a promise is resolved to another promise (which is acceptable and common), but because computing the eventual resolution of `flag` requires already knowing it.

Although the E model trades one form of lost-progress bug for another, it is still more reliable. As above, datalock bugs primarily represent circular dependencies in the computation, which manifest reproducibly like normal program bugs. This avoids the significant non-determinism, non-reproducibility, and resulting debugging difficulty of deadlock bugs. Anecdotally, in many years of programming in E and E-like languages and a body of experience spread over perhaps 60 programmers and two substantial distributed systems, we know of only two datalock bugs. Perhaps others went undetected, but these projects did not spend the agonizing time chasing deadlock bugs that projects of their nature normally must spend. Further analysis is needed to understand why datalock bugs seem to be so rare.

8.4 Explicit Promises

Besides the implicit creation of promise-resolver pairs by eventual-sending, E provides a primitive to create these pairs explicitly. In the following code

```
def [p, r] := Ref.promise()
```

`p` and `r` are bound to the promise and resolver of a new promise/resolver pair. Explicit promise creation gives us yet greater flexibility to postpone plans until other conditions occur. The promise, `p`, can be handed out and used just as any other eventual reference. All messages eventually-sent to `p` are queued in the promise. An object with access to `r` can wait until some condition occurs before resolving `p` and allowing these pending messages to proceed, as a later example will demonstrate.

8.5 Broken Promise Contagion

Because eventual-sends are executed in a later turn, an exception raised by one can no longer signal an exception and abort the plan of its "caller". Instead, the vat executing the turn for the eventual send catches any exception that terminates that turn and *breaks* the promise by resolving the promise to a *broken reference* containing that exception. Any immediate-call or eventual-send to a broken reference breaks the result with the broken reference's exception. Specifically, an immediate-call to a broken reference would throw the exception, terminating control flow. An eventual-send to a broken reference would break the eventual-send's promise with the broken reference's exception. As with the original exception, this would not terminate control flow, but does affect plans dependent on the resulting value.

E's split between control-flow exceptions and data-flow exceptions was inspired by signaling and non-signaling NaNs in floating point. Like non-signaling NaNs, broken promise contagion does not hinder pipelining. Following sections discuss how additional sources of failure in distributed systems cause broken references, and how E handles them while preserving defensive consistency.

9 Partial Failure

Not all exceptional conditions are caused by program behavior. Networks suffer outages, partitioning one part of the network from another. Machines fail: sometimes in a transient fashion, rolling back to a previous stable state; sometimes permanently, making the objects they host forever inaccessible. From a machine not able to reach a remote object, it is generally impossible to tell which failure is occurring or which messages were lost.

Distributed programs need to be able to react to these conditions so that surviving components can continue to provide valuable and correct—though possibly degraded—service while other components are inaccessible. If these components may change state while out of contact, they must recover distributed consistency when they reconnect. There is no single best strategy for maintaining consistency in the face of partitions and merges; the appropriate strategy will depend on the semantics of the components. A general purpose framework should provide simple mechanisms adequate to express a great variety of strategies. Group membership and similar systems provide one form of such a general framework, with strengths and weaknesses in comparison with E. Here, we explain E's framework. We provide a brief comparison with mechanisms like group membership in the "Related Work" section below.

E's support for partial failure starts by extending the semantics of our reference states. Figure 5 shows the full state transition diagram among these states.

We have added the possibility of a vat-crossing reference—a remote promise or a far reference—getting broken by a partition. A partition between a pair of

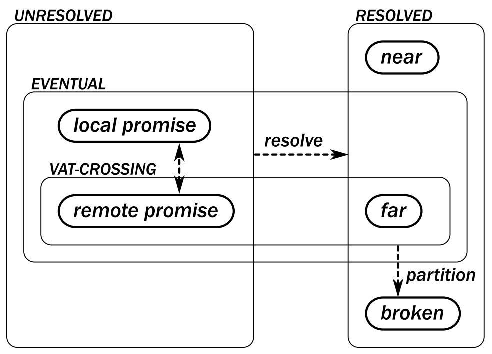

Fig. 5. A resolved reference's target is known. Near references are resolved and local; they carry both immediate-calls and eventual-sends. Promises and vat-crossing references are eventual; they carry only eventual-sends. Broken references carry neither. Promises may *resolve* to near, far or broken. *Partition* may break vat-crossing references.

vats eventually breaks all references that cross between these vats, creating eventual common knowledge of the loss of connection. A partition simultaneously breaks all references crossing in a given direction between two vats. The sender of messages that were still in transit cannot know which were actually received and which were lost. Later messages will only be delivered by a reference if all earlier messages sent on that same reference were already delivered. This fail-stop FIFO delivery order relieves the sender from needing to wait for earlier messages to be acknowledged before sending later dependent messages.[9]

On our state-transition diagram (a Harel statechart), we see that "near" and "broken" are terminal states. Even after a partition heals, all references broken by that partition stay broken.

In our listener example, if a partition separates the account's vat from the spreadsheet's vat, the statusHolder's reference to the spreadsheet's listener will eventually be broken with a partition-exception. Of the `statusChanged` messages sent by the statusHolder, this reference will deliver them reliably in FIFO order until it fails. Once it fails to deliver a message, it will never deliver any further messages and will eventually become visibly broken.

An essential consequence of these semantics is that defensive consistency is preserved across partition and reconnect. A defensively consistent program that makes no provisions for partition remains defensively consistent. In the earlier statusHolder example, `statusChanged` notifications sent to broken listener references (e.g., broken because the connection to its subscriber vat was severed) are harmlessly discarded.

[9] The message delivery order E enforces is stronger than FIFO and weaker than Causal [TMK^{+}87], but FIFO is adequate for all points we make in this paper.

9.1 Handling Failure

To explicitly manage failure of a reference, an object registers a handler to be eventually notified when that reference becomes broken. For the statusHolder to clean up broken listener references, it must register a handler on each one.

```
to addListener(newListener) {
    myListeners.push(newListener)
    newListener <- statusChanged(myStatus)
    def handler() { remove(myListeners, newListener) }
    newListener <- __whenBroken(handler)
}
```

The `__whenBroken` message is one of a handful of universally understood messages that all objects respond to by default.[10] Of these, the following messages are for interacting with a reference itself, as distinct from interacting only with the object designated by a reference.

`__whenBroken(`*`handler`*`)` When sent on a reference, this message registers its argument, `handler`, to be notified when this reference breaks.

`__whenMoreResolved(`*`handler`*`)` When sent on a reference, this message is normally used so that one can react when the reference is first resolved. We explain this in the later "When-Catch" section below.

`__reactToLostClient(`*`exception`*`)` When a vat-crossing reference breaks, it sends this message to its target object, to notify it that some of its clients may no longer be able to reach it.

Near references and local promises make no special case for these messages—they merely deliver them to their targets. Objects by default respond to a `__whenBroken` message by ignoring it, because they are not broken. So, in our single-vat scenario, when all these references are near, the additional code above has no effect. A broken reference, on the other hand, responds by eventual-sending a notification to the handler, as if by the following code:

```
to __whenBroken(handler) { handler <- run() }
```

When a local promise gets broken, all its messages are forwarded to the broken reference; when the `__whenBroken` message arrives, the broken reference will notify the handler.

A vat-crossing reference notifies these handlers if it becomes broken, whether by partition or resolution. In order to be able to send these notifications during partition, a vat-crossing reference registers the handler argument of a `__whenBroken` message at the tail end of the reference, *within the sending vat.* If the sending vat is told that one of these references has resolved, it re-sends equivalent `__whenBroken` messages to this resolution. If the sending vat decides that a partition has occurred (perhaps because the internal keep-alive timeout

[10] In Java, the methods defined in `java.lang.Object` are similarly universal.

has been exceeded), it breaks all outgoing references and notifies all registered handlers.

For all the reasons previously explained, the handler behavior built into E's references only eventual-sends notifications to handlers. Until the above handler reacts, the statusHolder will continue to harmlessly use the broken reference to the spreadsheet's listener. Contingency concerns can thus be handled separately from normal operation.

But what of the spreadsheet? We have ensured that it will receive `statusChanged` notifications in order, and that it will not miss any in the middle of a sequence. But, during a partition, its display may become arbitrarily stale. Technically, this introduces no new consistency hazards because the data may be stale anyway due to notification latencies. Nonetheless, the spreadsheet may wish to provide a visual indication that the displayed value may now be more stale than usual, since it is now out of contact with the authoritative source. To make this convenient, when a reference is broken by partition, it eventual-sends a `__reactToLostClient` message to its target, notifying it that at least one of its clients may no longer be able to send messages to it. By default, objects ignore `__reactToLostClient` messages. The spreadsheet could override the default behavior:

```
    to __reactToLostClient(exception) {  ...update display...  }
```

Thus, when a vat-crossing reference is severed by partition, notifications are eventually-sent to handlers at both ends of the reference. This explains how connectivity is safely severed by partition and how objects on either side can react if they wish. Objects also need to regain connectivity following a partition. For this purpose, we introduce *offline capabilities.*

9.2 Offline Capabilities

An offline capability in E has two forms: a "captp://..." URI string and an encapsulated `SturdyRef` object. Both contain the same information: the fingerprint of the public key of the vat hosting its target object, a list of TCP/IP location hints to seed the search for a vat that can authenticate against this fingerprint, and a so-called *swiss-number*, a large unguessable random number which the hosting vat associates with the target [Clo04a]. Like the popular myth of how Swiss bank account numbers work, one demonstrates knowledge of this secret to gain access to the object it designates. Like an object reference, if you do not know an unguessable secret, you can only come to know it if someone who knows it and can talk to you chooses to tell it to you. An offline capability is a form of "password capability"—it contains the cryptographic information needed both to authenticate the target and to authorize access to the target [Don76].

Both forms of offline capability are pass-by-copy and can be passed between vats even when the vat of the target object is inaccessible. Offline capabilities do not directly convey messages to their target. To establish or reestablish access to the target, one makes a new reference from an offline capability. Doing so initiates a new attempt to connect to the target vat and immediately returns a promise for the resulting inter-vat reference. If the connection attempt fails, this promise is eventually broken.

Typically, most inter-vat connectivity is only by references. When these break, applications on either end should not try to recover the detailed state of all the plans in progress between these vats. Instead, they should typically spawn a new fresh structure from the small number of offline capabilities from which this complex structure was originally spawned. As part of this respawning process, the two sides may need to explicitly reconcile in order to reestablish distributed consistency.

In our listener example, the statusHolder should not hold offline capabilities to listeners and should not try to reconnect to them. This would put the burden on the wrong party. A better design would have a listener hold an offline capability to the statusHolder. The listener's `__reactToLostClient` method would be enhanced to attempt to reconnect to the statusHolder and to resubscribe the listener on the promise for the reconnected statusHolder.

But perhaps the spreadsheet application originally encountered this statusHolder by navigating from an earlier object representing a collection of accounts, creating and subscribing a spreadsheet cell for each. While the vats were out of contact, not only may this statusHolder have changed, the collection may have changed so that this statusHolder is no longer relevant. In this case, a better design would be for the spreadsheet to maintain an offline capability only to the collection as a whole. When reconciling, it should navigate afresh, in order to find the statusHolders to which it should now subscribe.

The separation of references from offline capabilities encourages programming patterns that separate reconciliation concerns from normal operations.

9.3 Persistence

For an object that is designated only by references, the hosting vat can tell when it is no longer reachable and can garbage-collect it.[11] Once one makes an offline capability to a given object, its hosting vat can no longer determine when it is unreachable. Instead, this vat must retain the association between this object and its swiss-number until its obligation to honor this offline capability expires.

The operations for making an offline capability provide three options for ending this obligation: It can expire at a chosen future date, giving the association a *time-to-live*. It can expire when explicitly cancelled, making the association *revocable*. And it can expire when the hosting vat crashes, making the association *transient*. Here, we examine only this last option. An association which is not transient is *durable*.

A vat can be either ephemeral or persistent. An ephemeral vat exists only until it terminates or crashes; so for these, the last option above is irrelevant. A persistent vat periodically *checkpoints*, saving its persistent state to non-volatile

[11] E's distributed garbage collection protocol does not currently collect unreachable inter-vat references cycles. See [Bej96] for a GC algorithm able to collect such cycles among mutually suspicious machines.

storage. A vat checkpoints only between turns when its stack is empty. A crash terminates a vat-incarnation, rolling it back to its last checkpoint. Reviving the vat from checkpoint creates a new incarnation of the same vat. A persistent vat lives through a sequence of incarnations. With the possibility of crash admitted into E's computational model, we can allow programs to cause crashes, so they can preemptively terminate a vat or abort an incarnation.

The persistent state of a vat is determined by traversal from persistent roots. This state includes the vat's public/private key pair, so later incarnations can authenticate. It also includes all unexpired durable swiss-number associations and state reached by traversal from there. As this traversal proceeds, when it reaches an offline capability, the offline capability itself is saved but is not traversed to its target. When the traversal reaches a vat-crossing reference, a broken reference is saved instead and the reference is again not traversed. Should this vat be revived from this checkpoint, old vat-crossing references will be revived as broken references. A crash partitions a vat from all others. Following a revival, only offline capabilities in either direction enable it to become reconnected.

10 The When-Catch Expression

The `__whenMoreResolved` message can be used to be register for notification when a reference resolves. Typically this message is used indrectly through the "when-catch" syntax. A when-catch expression takes a promise, a "when" block to execute if the promise resolves to a value, and a "catch" block to execute if the promise is broken. This is illustrated by the following example.

```
def asyncAnd(answers) {
    var countDown := answers.size()
    if (countDown == 0) { return true }
    def [result, resolver] := Ref.promise()
    for answer in answers {
        when (answer) -> {
            if (answer) {
                countDown -= 1
                if (countDown == 0) {
                    resolver.resolve(true)
                }
            } else {
                resolver.resolve(false)
            }
        } catch exception {
            resolver.smash(exception)
        }
    }
    return result
}
```

The asyncAnd takes a list of promises for booleans. It immediately returns a reference representing the conjunction, which must eventually be true if all elements of the list become true, or false or broken if any of them become false or broken. Using when-catch, asyncAnd can test these as they become available, so it can report a result as soon as it has enough information.

If the list is empty, the conjunction is true right away. Otherwise, countDown remembers how many true answers are needed before asyncAnd can conclude that the conjunction is true. The "when-catch" expression is used to register a handler on each reference in the list. The behavior of the handler is expressed in two parts: the block after the "->" handles the normal case, and the catch-clause handles the exceptional case. Once answer resolves, if it is near or far, the normal-case code is run. If it is broken, the catch-clause is run. Here, if the normal case runs, answer is expected to be a boolean. By using a "when-catch", the if is postponed until asyncAnd has gathered enough information to know which way it should branch.

Once asyncAnd registers all these handlers, it immediately returns result, a promise for the conjunction of these answers. If they all resolve to true, asyncAnd *reveals* that the result is true, i.e., it eventually resolves the already-returned promise to true. If it is notified that any resolve to false, asyncAnd reveals false immediately. If any resolve to broken, asyncAnd reveals a reference broken by the same exception. Asking a resolver to resolve an already-resolved promise has no effect, so if one of the answers is false and another is broken, the above asyncAnd code may reveal either false or broken, depending on which handler happens to be notified first.

The following snippet illustrates using asyncAnd and when-catch to combine independent validity checks in a toy application to resells goods from a supplier.

```
def allOk := asyncAnd([inventory <- isAvailable(partNo),
                       creditBureau <- verifyCredit(buyerData),
                       shipper <- canDeliver(...)])
when (allOk) -> {
    if (allOk) {
        def receipt := supplier <- buy(partNo, payment)
        when (receipt) -> {
```

Promise-chaining postpones plans efficiently by data-flow; the "when-catch" postpones plans until the data needed for control-flow is available.

11 From Objects to Actors and Back Again

Here we present a brief history of E's concurrency-control architecture. In this section, the term "we" indicates that one or both of this paper's first two authors participated in a project involving other people. All implied credit should be understood as shared with these others.

Objects. The nature of computation provided within a single von Neumann machine is quite different than the nature of computation provided by networks of such machines. Distributed programs must deal with both. To reduce cases, it would seem attractive to create an abstraction layer that can make these seem more similar. Distributed Shared Memory systems try to make the network seem more like a von Neumann machine. Object-oriented programming started by trying to make a single computer seem more like a network.

> ... Smalltalk is a recursion on the notion of computer itself. Instead of dividing "computer stuff" into things each less strong than the whole—like data structures, procedures, and functions which are the usual paraphernalia of programming languages—each Smalltalk object is a recursion on the entire possibilities of the computer. Thus its semantics are a bit like having thousands and thousands of computers all hooked together by a very fast network.
>
> —Alan Kay [Kay93]

Smalltalk imported only the aspects of networks that made it easier to program a single machine—its purpose was not to achieve network transparency. Problems that could be avoided within a single machine—like inherent asynchrony, large latencies, and partial failures—were avoided. The sequential subset of E has much in common with the early Smalltalk: Smalltalk's object references are like E's near references and Smalltalk's message passing is like E's immediate-call operator.

Actors. Inspired by the early Smalltalk, Hewitt created the Actors paradigm [HBS73], whose goals include full network transparency within all the constraints imposed by decentralization and mutual suspicion [Hew85]. Although the stated goals require the handling of partial failure, the actual Actors model assumes this issue away and instead guarantees that all sent messages are eventually delivered. The asynchronous-only subset of E is an Actors language: Actors' references are like E's eventual references, and Actors' message passing is much like E's eventual-send operator. Actors provide both data-flow postponement of plans by futures (like E's promises without pipelining or contagion) and control-flow postponement by continuations (similar in effect to E's when-catch).

The price of this uniformity is that all programs had to work in the face of network problems. There was only one case to solve, but it was the hard case.

Vulcan. Inspired by Shapiro and Takeuchi [ST83], the Vulcan project [KTMB87] merged aspects of Actors and concurrent logic/constraint programming [Sha83, Sar93]. The pleasant properties of concurrent logic variables (much like futures or promises) taught us to emphasize data-flow postponement and de-emphasize control-flow postponement.

Vulcan was built on a concurrent logic base, and inherited from it the so-called "merge problem" [SM87] absent from pure Actors languages: Clients can only share access to a stateful object by explicit pre-arrangement, so the equivalent of object references were not usefully first-class. To address this problem, we created the "Channels" abstraction, which also provides useful ordering properties [TMK$^+$87].

Joule. The Joule language [TMHK95] is a capability-secure, massively-concurrent, distributed language that is one of the primary precursors to E. Joule merges insights from the Vulcan project with the remaining virtues of Actors. Joule channels are similar to E's promises generalized to provide multicasting. Joule tanks are the unit of separate failure, persistence, migration, and resource management, and inspired E vats. E vats further define the unit of sequentiality; E's event-loop approach achieves much of Joule's power with a more familiar and easy to use computational model. Joule's resource management is based on abstractions from KeyKOS [Har85]. E vats do not yet address this issue.

Promise pipelining in Udanax Gold. This was a pre-web hypertext system with a rich interaction protocol between clients and servers. To deal with network latencies, in the 1989 timeframe, we independently reinvented an asymmetric form of promise pipelining as part of our protocol design [Mil92]. This was the first attempt to adapt Joule channels to an object-based client-server environment (it did not support peer-to-peer).

Original-E. The language now known as Original-E was the result of adding the concepts from Joule to the sequential, capability-secure subset of Java. Original-E was the first to successfully mix sequential immediate-call programming with asynchronous eventual-send programming. Original-E cryptographically secured the Joule-like network extension—something that had been planned for but not actually realized in prior systems. Electric Communities created Original-E, and used it to build Habitats—a graphical, decentralized, secure, social virtual reality system.

From Original-E to E. In Original-E, the co-existence of sequential and asynchronous programming was still rough. E brought the invention of the distinct reference states and the transitions among them explained in this paper. With these rules, E bridges the gap between the network-as-metaphor view of the early Smalltalk and the network-transparency ambitions of Actors. In E, the local case is strictly easier than the network case, so the guarantees provided by near references are a strict superset of the guarantees provided by other reference states. When programming for known-local objects, a programmer can do it the easy way. Otherwise, the programmer must address the inherent problems of networks. Once the programmer has done so, the same code will painlessly also handle the local case without requiring any further case analysis.

12 Related Work

Promises and Batched Futures at MIT. The promise pipelining technique was first invented by Liskov and Shrira [LS88]. These ideas were then significantly improved by Bogle [BL94]. Like the Udanax Gold system mentioned above, these are asymmetric client-server systems. In other ways, the techniques used in Bogle's protocol resembles quite closely some of the techniques used in E's protocol.

Group Membership. There is an extensive body of work on group membership systems [BJ87, Ami95] and (broadly speaking) similar systems such as Paxos [Lam98]. These systems provide a different form of general-purpose framework for dealing with partial failure: they support closer approximations of common knowledge than does E, but at the price of weaker support for defensive consistency and scalability. These frameworks better support the tightly-coupled composition of separate plan-strands into a virtual single overall plan. E's mechanisms better support the loosely-coupled composition of networks of independent but cooperative plans.

For example, when a set of distributed components form an application that provides a single logical service to all their collective clients, and when multiple separated components may each change state while out of contact with the others, we have a *partition-aware application* [OBDMS98, SM03], providing a form of fault-tolerant replication. The clients of such an application see a close approximation of a single stateful object that is highly available under partition. Some mechanisms like group membership shine at supporting this model under mutually reliant and even Byzantine conditions [CL02].

E itself provides nothing comparable. The patterns of fault-tolerant replication we have built to date are all forms of primary-copy replication, with a single stationary authoritative host. E supports these patterns quite well, and they compose well with simple E objects that are unaware they are interacting with a replica. An area of future research is to see how well partition-aware applications can be programmed in E and how well they can compose with others.

Croquet and TeaTime. The Croquet project has many of the same goals as the Habitats project referred to above: to create a graphical, decentralized, secure, user-extensible, social virtual reality system spread across mutually suspicious machines. Regarding E, the salient differences are that Croquet is built on Smalltalk extended onto the network by TeaTime, which is based on Namos [Ree78] and Paxos [Lam98], in order to replicate state among multiple authoritative hosts. Unlike Habitats, Croquet is user-extensible, but is not yet secure. It will be interesting to see how they alter Paxos to work between mutually suspicious machines.

12.1 Work Influenced by E's Concurrency Control

The Web-Calculus. The Web-Calculus [Clo04b] brings to web URLs the following simultaneous properties:

- The cryptographic capability properties of E's offline capabilities—both authenticating the target and authorizing access to it.
- Promise pipelining of eventually-POSTed requests with results.
- The properties recommended by the REST model of web programming [Fie00]. REST attributes the success of the web largely to certain loose-coupling properties of "http://..." URLs, which are well beyond the scope of this paper. See [Fie00, Clo04b] for more.

As a language-neutral protocol compatible and composable with existing web standards, the Web-Calculus is well-positioned to achieve widespread adoption. We expect to build a bridge between E's references and Web-Calculus URLs.

Oz-E. Like Vulcan, the Oz language [VH04] descends from both Actors and concurrent logic/constraint programming. Unlike these parents, Oz supports shared-state concurrency, though Oz programming practice discourages its use. Oz-E [SV05a] is a capability-based successor to Oz designed to support both local and distributed defensive consistency. For the reasons explained in the "Defensive Correctness" section above, Oz-E suppresses Oz's shared-state concurrency.

Twisted Python. This is a library and a set of conventions for distributed programming in Python, based on E's model of communicating event-loops, promise pipelining, and cryptographic capability security [Lef].

13 Discussion and Conclusions

Electric Communities open-sourced E in 1998. Since then, a lively open source community has continued development of E at http://www.erights.org/. Seven companies and two universities have used E—to teach secure and distributed programming, to rapidly prototype distributed architectures, and to build several distributed systems.

Despite these successful trials, we do not yet consider E ready for production use—the current E implementation is a slow interpreter written in Java. Two compiler-based implementations are in progress: Kevin Reid is building an E on Common Lisp [Rei05], and Dean Tribble is building an E on Squeak (an open-source Smalltalk). Several of E's libraries, currently implemented in Java, are being rewritten in E to help port E onto other language platforms. Separately, Fred Spiessens continues to make progress on formalizing the reasoning about *authority* on which E's security is based [SV05b].

Throughout, our engineering premise is that lambda abstraction and object programming, by their impressive plan coordination successes in the small, have the seeds for coordinating plans in the large. As Alan Kay has urged [Kay98], our emphasis is less on the objects and more on the interstitial fabric which connects them: the dynamic reference graph carrying the messages by which their plans interact.

Encapsulation separates objects so their plans can avoid disrupting each other's assumptions. Objects compose plans by message passing while respecting each other's separation. However, when client objects request service from provider objects, their continued proper functioning is often vulnerable to their provider's misbehavior. When providers are also vulnerable to their clients, corruption is potentially contagious over the reachable graph in both directions, severely limiting the scale of systems we can compose.

Reduced vulnerability helps contain corruption. In this paper, we draw attention to a specific composable standard of robustness: when a provider is *defensively consistent*, none of its clients can corrupt it or cause it to give incorrect service to any of its well-behaved clients, thus protecting its clients from each other. When a system is composed of defensively consistent abstractions, to a good approximation, corruption is contagious only upstream. (Further vulnerability reduction beyond this standard is, of course, valuable and often needed.)

Under shared-state concurrency—conventional multi-threading—we have shown by example that defensive consistency is unreasonably difficult. We have explained how an alternate concurrency-control discipline, communicating event-loops, supports creating defensively consistent objects in the face of concurrency and distribution. Our enhanced reference graph consists of references in different states, where their message delivery abilities depends on their state. Only *eventual references* convey messages between event-loops, and deliver messages only in separately scheduled turns, providing temporal separation of plans. *Promises* pipeline messages towards their likely destinations, compensating for latency. *Broken references* safely abstract partition, and *offline capabilities* abstract the ability to reconnect.

We have used small examples in this paper to illustrate principles with which several projects have built large robust distributed systems.

Acknowledgements

For various helpful suggestions, we thank Darius Bacon, Dan Bornstein, John Corbett, Bill Frantz, Ian Grigg, Jim Hopwood, Piotr Kaminski, Alan Karp, Matej Kosik, Jon Leonard, Kevin Reid, Michael Sperber, Fred Spiessens, Terry Stanley, Marc Stiegler, Bill Tulloh, Bryce “Zooko” Wilcox-O’Hearn, Steve Witham, and the e-lang and cap-talk communities. We thank Terry Stanley for suggesting the listener pattern and purchase-order examples.

We are especially grateful to Ka-Ping Yee and David Hopwood for a wide variety of assistance. They reviewed numerous drafts, contributed extensive and deep technical feedback, clarifying rephrasings, crisp illustrations, and moral support.

References

[Ami95] Yair Amir. *Replication Using Group Communication Over a Partitioned Network*. PhD thesis, 1995.

[Bej96] Arturo Bejar. The Electric Communities distributed garbage collector, 1996.
`http://www.crockford.com/ec/dgc.html`.

[BH93] Per Brinch Hansen. Monitors and concurrent Pascal: a personal history. In *HOPL-II: The second ACM SIGPLAN conference on History of programming languages*, pages 1–35, New York, NY, USA, 1993. ACM Press.

[BJ87] Ken Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *SOSP ’87: Proceedings of the eleventh ACM Symposium on Operating systems principles*, pages 123–138, New York, NY, USA, 1987. ACM Press.

[BL94] Phillip Bogle and Barbara Liskov. Reducing cross domain call overhead using batched futures. In *OOPSLA '94: Proceedings of the ninth annual conference on Object-oriented programming systems, language, and applications*, pages 341–354, New York, NY, USA, 1994. ACM Press.

[Boe05] Hans-J. Boehm. Threads cannot be implemented as a library. *SIGPLAN Not.*, 40(6):261–268, 2005.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[CL02] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002.

[Clo04a] Tyler Close. Waterken YURL, 2004. `www.waterken.com/dev/YURL/httpsy/`.

[Clo04b] Tyler Close. web-calculus, 2004. `www.waterken.com/dev/Web/`.

[DH65] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. Technical Report MIT/LCS/TR-23, M.I.T. Laboratory for Computer Science, 1965.

[Don76] Jed Donnelley. A Distributed Capability Computing System, 1976.

[Eng97] Robert Englander. *Developing Java Beans.* O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, 1997.

[Fie00] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures.* PhD thesis, 2000. Chair-Richard N. Taylor.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnon, and John Vlissides. *Design Patterns, elements of reusable object-oriented software.* Addison Wesley, 1994.

[GK76] Adele Goldberg and Alan C. Kay. Smalltalk-72 instruction manual. March 1976. Xerox Palo Alto Research Center.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, Reading, MA, 1983.

[Har85] Norman Hardy. KeyKOS architecture. *SIGOPS Oper. Syst. Rev.*, 19(4):8–25, 1985.

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In Nils J. Nilsson, editor, *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, pages 235–245, Standford, CA, August 1973. William Kaufmann.

[Hew85] Carl Hewitt. The challenge of open systems: current logic programming methods may be insufficient for developing the intelligent systems of the future. *BYTE*, 10(4):223–242, 1985.

[Hoa65] C.A.R Hoare. Record handling, in Algol Bulletin 21.3.6, 1965.

[Hoa74] C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17(10):549–557, 1974.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[IBM68] IBM Corporation. *IBM System/360 Principles of Operation.* IBM Corporation, San Jose, CA, USA, eighth edition, 1968.

[Kay93] Alan C. Kay. The early history of Smalltalk. *SIGPLAN not.*, 28(3):69–95, 1993.

[Kay98] Alan Kay. prototypes vs classes, 1998. `lists.squeakfoundation.org/pipermail/-squeak-dev/1998-October/017019.html`.

[KTMB87] Kenneth M. Kahn, Eric Dean Tribble, Mark S. Miller, and Daniel G. Bobrow. Vulcan: Logical concurrent objects. In *Research Directions in Object-Oriented Programming*, pages 75–112. 1987.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[Lef] Glyph Lefkowitz. Generalization of deferred execution in Python. `python.org/pycon/papers/deferex/`.

[LS88] Barbara Liskov and Lubia Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 260–267, New York, NY, USA, 1988. ACM Press.

[MD88] Mark Miller and K. Eric Drexler. Markets and computation: Agoric open systems. In Bernardo Huberman, editor, *The Ecology of Computation*, pages 133–176. North-Holland, 1988.

[Mil92] Mark S. Miller. Transcript of talk: The promise system, 1992. `sunless-sea.net/Transcripts/promise.html`.

[MMF00] Mark S. Miller, Chip Morningstar, and Bill Frantz. Capability-based financial instruments. In *Financial Cryptography*, pages 349–378, 2000.

[MS03] Mark S. Miller and Jonathan S. Shapiro. Paradigm Regained: Abstraction mechanisms for access control. In *ASIAN*, pages 224–242, 2003.

[MTS04] Mark S. Miller, Bill Tulloh, and Jonathan S. Shapiro. The Structure of Authority: Why security is not a separable concern. In *MOZ*, pages 2–20, 2004.

[MYS03] Mark S. Miller, Ka-Ping Yee, and Jonathan Shapiro. Capability Myths Demolished, 2003.

[OBDMS98] Özalp Babaoğlu, Renzo Davoli, Alberto Montresor, and Roberto Segala. System support for partition-aware network applications. *SIGOPS Oper. Syst. Rev.*, 32(1):41–56, 1998.

[Ree78] David Patrick Reed. Naming and synchronization in a decentralized computer system., January 01 1978.

[Rei05] Kevin Reid. E on Common Lisp, 2005. `homepage.mac.com/kpreid/elang/e-on-cl/`.

[Sar93] Vijay A. Saraswat. *Concurrent Constraint Programming.* MIT Press, Cambridge, MA, 1993.

[Sha83] Ehud Shapiro. A subset of Concurrent Prolog and its interpreter. Technical Report TR-003, Institute for New Generation Computer Technology (ICOT), January 1983.

[SKYM04] Marc Stiegler, Alan H. Karp, Ka-Ping Yee, and Mark S. Miller. Polaris: Virus safe computing for Windows XP. Technical Report HPL-2004-221, Hewlett Packard Laboratories, 2004.

[SM87] E. Shapiro and C. Mierowsky. Fair, biased, and self-balancing merge operators: Their specification and implementation in Concurrent Prolog. In Ehud Shapiro, editor, *Concurrent Prolog: Collected Papers (Volume I)*, pages 392–413. MIT Press, London, 1987.

[SM03] Jeremy Sussman and Keith Marzullo. The bancomat problem: an example of resource allocation in a partitionable asynchronous system. *Theor. Comput. Sci.*, 291(1):103–131, 2003.

[SS75] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer system. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[SS04] Paritosh Shroff and Scott F. Smith. Type inference for first-class messages with match-functions, 2004.

[SSF99] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: a fast capability system. In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 170–185, New York, NY, USA, 1999. ACM Press.

[ST83] Ehud Y. Shapiro and Akikazu Takeuchi. Object oriented programming in concurrent prolog. *New Generation Comput.*, 1(1):25–48, 1983.

[Sti04] Marc Stiegler. The E language in a walnut, 2004. `www.skyhunter.com/marcs/ewalnut.html`.

[SV05a] Fred Spiessens and Peter Van Roy. The Oz-E project: Design guidelines for a secure multiparadigm programming language. In *Multiparadigm Programming in Mozart/Oz: Extended Proceedings of the Second International Conference MOZ 2004*, volume 3389 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.

[SV05b] Fred Spiessens and Peter Van Roy. A practical formal model for safety analysis in Capability-Based systems, 2005. To be published in Lecture Notes in Computer Science (Springer-Verlag). Presentation available at `www.info.ucl.ac.be/people/fsp/auredsysfinal.mov`.

[TM02] Bill Tulloh and Mark S. Miller. Institutions as abstraction boundaries. In Jack High, editor, *Social Learning: Essays in Honor of Don Lavoie*. 2002.

[TMHK95] E. Dean Tribble, Mark S. Miller, Norm Hardy, and David Krieger. Joule: Distributed application foundations. Technical Report ADd03.4P, Agorics Inc., Los Altos, December 1995. `www.agorics.com/Library/joule.html`.

[TMK$^+$87] Eric Dean Tribble, Mark S. Miller, Kenneth M. Kahn, Daniel G. Bobrow, Curtis Abbott, and Ehud Y. Shapiro. Channels: A generalization of streams. In *ICLP*, pages 839–857, 1987.

[vH45] Friedrich von Hayek. The uses of knowledge in society. *American Economic Review*, 35:519–530, September 1945.

[VH04] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, March 2004.

[WO01] Bryce "Zooko" Wilcox-O'Hearn. Deadlock-free, 2001. `www.eros-os.org/pipermail/e-lang/2001-July/005410.html`.

Links to on-line versions of many of these references are available at `http://www.erights.org/talks/promises/paper/references.html`.