Robust Composition:

Towards a Unified Approach to Access Control and Concurrency Control

by

Mark Samuel Miller

A dissertation submitted to Johns Hopkins University in conformity with the requirements for the degree of Doctor of Philosophy.

Baltimore, Maryland

May, 2006

Abstract

When separately written programs are composed so that they may cooperate, they may instead destructively interfere in unanticipated ways. These hazards limit the scale and functionality of the software systems we can successfully compose. This dissertation presents a framework for enabling those interactions between components needed for the cooperation we intend, while minimizing the hazards of destructive interference.

Great progress on the composition problem has been made within the object paradigm, chiefly in the context of sequential, single-machine programming among benign components. We show how to extend this success to support robust composition of concurrent and potentially malicious components distributed over potentially malicious machines. We present E, a distributed, persistent, secure programming language, and CapDesk, a virus-safe desktop built in E, as embodiments of the techniques we explain.

Advisor: Jonathan S. Shapiro, Ph.D.
Readers: Scott Smith, Ph.D., Yair Amir, Ph.D.

This dissertation is dedicated to the number “3469” and the letter “E”.

Acknowledgements

Jonathan Shapiro, my advisor, for encouraging me to continue this work in an academic setting, and for providing insight, encouragement, and support way beyond the call of any duty.

Terry Stanley, for her patient support, encouragement, and enthusiam for this project.

My parents, Ann and Bernard Miller. Knowing the naches they would feel, helped motivate me to complete this dissertation ("Naches" is approximately "reflected pride").

Hewlett-Packard Laboratories for supporting portions of this research, and Alan Karp for helping to arrange this.

Combex and Henry Boreen for investing in these ideas. I still hope to see this investment pay off.

The Defense Advanced Research Projects Agency for sponsoring the security review of E, CapDesk, and the DarpaBrowser [WT02].

Lauren Williams for rescue from crisis. I don't know what would have happened without your help.

The software systems explained in this dissertation are the results of collaborative efforts starting at Electric Communities, Inc. and continuing with the e-lang online community. Since the contributors are many and changing, they are accurately documented only by navigation from `erights.org`. Here, I'd like to single out and express my gratitude specifically to E's most creative and prolific user, Marc Stiegler. He is the creator and primary developer of the systems documented in Part IV: CapDesk, Polaris, and the DarpaBrowser. Without his contributions, the value of E would have remained inaccessibly abstract.

This dissertation borrows liberally from several of my previous papers [MMF00, MYS03, MS03, MTS04, MTS05]. Without further attribution, I'd like to especially thank my co-authors on these papers: Bill Frantz, Chip Morningstar, Jonathan Shapiro, E. Dean Tribble, Bill Tulloh, and Ka-Ping Yee. I cannot hope to enumerate all their contributions to the ideas presented here.

I'd also like to thank again all those who contributed to these prior papers: Yair Amir, Paul Baclace, Darius Bacon, Howie Baetjer, Hans Boehm, Dan Bornstein, Per Brand, Marc "Lucky Green" Briceno, Michael Butler, Tyler Close, John Corbett, M. Scott Doerrie, Jed Donnelley, K. Eric Drexler, Ian Grigg, Norm Hardy, Chris Hibbert, Jack High, Tad Hogg, David Hopwood, Jim Hopwood, Ted Kaehler, Ken Kahn, Piotr

Kaminski, Alan Karp, Terence Kelly, Lorens Kockum, Matej Kosik, Kevin Lacobie, Charles Landau, Jon Leonard, Mark Lillibridge, Brian Marick, Patrick McGeer, Eric Messick, Greg Nelson, Eric Northup, Constantine Plotnikov, Jonathan Rees, Kevin Reid, Matthew Roller, Vijay Saraswat, Christian Scheideler, Scott Smith, Michael Sperber, Fred Spiessens, Swaroop Sridhar, Terry Stanley, Marc Stiegler, Nick Szabo, Kazunori Ueda, David Wagner, Bryce "Zooko" Wilcox-O'Hearn, Steve Witham, and the e-lang and cap-talk communities.

Thanks 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. Ka-Ping Yee contributed Figures 14.2 (p. 107), 14.3 (p. 108), 16.1 (p. 118), and 17.1 (p. 124) with input from the e-lang community. Thanks to Terry Stanley for suggesting the listener pattern and purchase-order examples. Thanks to Marc Stiegler for the membrane example shown in Figure 9.3 (p. 71). Thanks to Darius Bacon for the promise pipelining example shown in Figure 16.1 (p. 118). Thanks to Mark Seaborn for suggesting that the when-catch expression evaluate to a promise for its handler's result, as explained in Section 18.1. Thanks to the Internet Assigned Numbers Authority, for choosing the perfect port number for Pluribus on their own. Thanks to Norm Hardy for bringing to my attention the relationship between knowledge and authority in computational systems.

For helpful suggestions regarding the dissertation itself, I thank Yair Amir, Tyler Close, M. Scott Doerrie, K. Eric Drexler, Bill Frantz, Norm Hardy, Chris Hibbert, Ken Kahn, Alan Karp, Patrick McGeer, Chip Morningstar, Eric Northup, Jonathan Shapiro, Matthew Roller, Scott Smith, Mark Smotherman, Swaroop Sridhar, Terry Stanley, Marc Stiegler, E. Dean Tribble, Bill Tulloh, and Lauren Williams.

I am eternally grateful to the board of Electric Communities for open sourcing E when their business plans changed.

Thanks to Kevin Reid and E. Dean Tribble for keeping the E development process alive and well while I spent time on this dissertation.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

When separately written programs are composed so that they may cooperate, they may instead destructively interfere in unanticipated ways. These hazards limit the scale and functionality of the software systems we can successfully compose. This dissertation presents a framework—a computational model and a set of design rules—for enabling those interactions between components needed for the cooperation we intend, while minimizing the hazards of destructive interference.

Most of the progress to date on the composition problem has been made in the context of sequential, single-machine programming among benign components. Within this limited context, object programming supports composition well. This dissertation explains and builds on this success, showing how to extend the object paradigm to support robust composition of concurrent and potentially malicious components distributed over potentially malicious machines. We present E, a distributed, persistent, secure programming language, and CapDesk, a virus-safe desktop built in E, as embodiments of the techniques we explain.

As Alan Kay has suggested [Kay98], our explanation of the power of object programming will focus not on the objects themselves, but on the reference graph that connects them. In the object model of computation, an object can affect the world outside itself only by sending messages to objects it holds references to.[1] The references that an object may come to hold thereby limit what effects it may cause. Our extensions to the object paradigm leverage this observation. References become the sole conveyers of (overt) inter-object causality, yielding the object-capability model of access control (Chapter 9), able to support certain patterns of composition among potentially malicious objects. We extend the reference graph cryptographically between potentially mutually malicious machines, yielding a distributed cryptographic capability system (Chapter 7).

A particularly vexing set of problems in distributed systems are the issues of partial failure (spontaneous disconnection and crashes). The most novel contribution of this dis-

[1] To explain call-return control flow purely in terms of *sending* messages, we often speak as if all programs are transformed to continuation-passing style before execution. This explains a call as a send carrying a (normally hidden) extra continuation argument reifying the "rest of the computation" to happen after this call returns. This corresponds approximately to the implementation concept of pushing a return address on the stack. The callee's returning is explained as sending the returned value to this continuation. This is discussed further in Section 18.2.

Those familiar with ML or Algol68 may find our use of the term "reference" confusing. By "reference" we mean "object reference" or "protected pointer," *i.e.*, the arrows that one draws when diagramming a data structure to show which objects "point at" which other objects. Our "references" have nothing to do with enabling mutability.

sertation is the definition of a state transition semantics for distributed references that supports deferred communication, failure notification, and reconnection while preserving useful limits on causal transmission. We define a set of reference states, *i.e.*, states that a reference may be in, and a set of associated transition rules, where the causal transmission properties provided by a reference depend on its state (Chapter 17). The resulting access-control and concurrency-control discipline helps us cope with the following pressing problems:

- Excessive authority which invites abuse (such as viruses and spyware),
- Inconsistency caused by interleaving (concurrency),
- Deadlock (though other forms of lost progress hazards remain),
- Inter-machine latency, and
- Partial failure (disconnects and crashes).

Some prior means of addressing these problems are similar to those presented in this dissertation. However, these prior solutions have not been composed successfully into a framework for simultaneously addressing these problems. Our comparative success at realizing an integrated solution is due to two observations:

1. Both access control and concurrency control are about enabling the causality needed for the inter-object cooperation we intend, while seeking to prevent those interactions which might cause destructive interference. In access control, we seek to distribute those access rights needed for the job at hand, while limiting the distribution of access rights that would enable mischief. In concurrency control, we seek to enable those interleavings needed for continued progress, while preventing those interleavings that would cause inconsistency.

2. References are already the natural means for enabling inter-object causality, so a natural and powerful means of limiting inter-object causality is to restrict the causal transmission properties provided by references.

We show how the consistent application of these principles, simultaneously, at multiple scales of composition, results in a multiplicative reduction in overall systemic vulnerability to plan interference.

1.1 Organization of this Dissertation

Part I explains the *Software Composition Problem*, and how object programming already helps address it under local, sequential, and benign conditions. We introduce those compositional forms of component robustness needed to extend the object paradigm beyond these limits. We introduce a subset of the E language, and a simplified form of Pluribus, E's distributed object protocol. The rest of the dissertation uses E examples to illustrate how to achieve some of these forms of robustness.

Part II, *Access Control*, explores the consequences of limiting inter-object (overt) causality to occur solely by means of messages sent on references. By explaining the role of *access* *abstractions*, we demonstrate that the resulting access control model is more expressive than much of the prior literature would suggest. In this part, we extend our attention to potentially malicious components, though still in a sequential and local context.

Part III, *Concurrency Control*, extends our ambitions to distributed systems. Separate machines proceed concurrently, interact across barriers of large latencies and partial failure, and encounter each other's misbehavior. Each of these successive problems motivates a further elaboration of our reference states and transition rules, until we have the complete picture, shown in Figure 17.1 (p. 124).

The above parts provide a micro analysis of compositional robustness, in which small code examples illustrate the interaction of individual objects, references, and messages. By themselves they demonstrate only increased robustness "in the small."

Part IV, *Emergent Robustness*, takes us on a macro tour through CapDesk, a proof of concept system built in E. It explains how the potential damage caused by bugs or malice are often kept from propagating along the reference graph. By examining patterns of susceptibility to plan interference across different scales of composition, we come to understand the degree of robustness practically achievable "in the large," as well as the remaining limits to this robustness.

Part V discusses *Related Work*, including how these reference-state rules bridge the gap between the network-as-metaphor view of the early Smalltalk and the network-transparency ambitions of Actors. In addition, many chapters end with notes on related work specific to that chapter.

Chapter 2

Approach and Contributions

This chapter provides a preview of topics developed in the rest of the dissertation, with forward references to chapters where each is explained. At the end of those chapters are summaries of related work. Further related work appears in Part V.

2.1 Unattenuated Composition

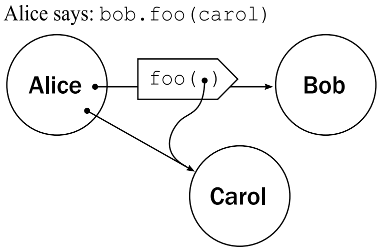


Figure 2.1: Unattenuated Composition.

In an object system (Figure 2.1), when object Alice says `bob.foo(carol)`, she invokes object Bob, passing as argument a reference to object Carol. By passing this reference, Alice composes Bob with Carol, so that their interaction will serve some purpose of Alice's. The argument reference enables Bob to interact with Carol.

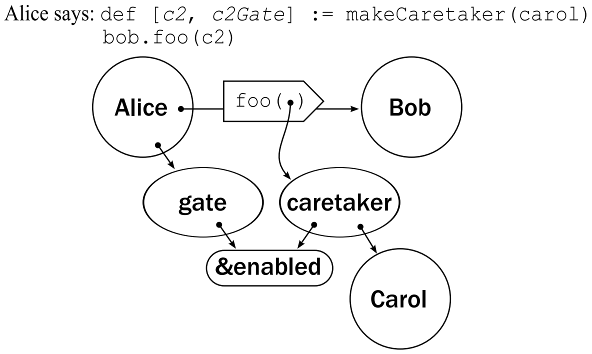
By restricting inter-object causality to flow only by messages sent on references, Bob's authority is limited according to the references he comes to hold. If Bob cannot interact with Carol unless he holds a reference to Carol, then the reference graph from the programming language literature *is* the access graph from the access control literature. When references can only be transmitted and acquired by the rules of the object programming model, this results in the object-capability model of secure computation. This model has been represented by concrete designs for over forty years, but previous attempts to state the model abstractly have left out crucial elements. A contribution of Chapter 9 is to present a single model, abstract enough to describe both prior object-capability languages and operating systems, and concrete enough to describe authority-manipulating behavior.

Alice's purpose requires Bob and Carol to interact in certain ways. Even within the object-capability restrictions, this reference provides Bob unattenuated authority to access

Carol: It gives Bob a perpetual and unconditional ability to invoke Carol's public operations. This may allow interactions well beyond those that serve Alice's purposes, including interactions harmful to Alice's interests. If Alice could enable just those interactions needed for her purposes, then, if things go awry, the damage that follows might be usefully isolated and limited.

This dissertation explores several mechanisms by which the authority Alice provides to Bob may be attenuated. Some of these mechanisms simply restate established practice of object-capability access control. A contribution of Chapter 3 and Part III is to view concurrency control issues in terms of attenuating authority as well, and to provide a unified architecture for attenuating authority, in which both access control and concurrency control concerns may be addressed together.

2.2 Attenuating Authority



Alice says:
```
def [c2, c2Gate] := makeCaretaker(carol)
bob.foo(c2)
```

Figure 2.2: Attenuating Authority.

In practice, programmers control access partially by manipulating the access graph, and partially by writing programs whose behavior attenuates the authority that flows through them. In Figure 2.2, Alice attenuates Bob's authority to Carol by interposing an access abstraction. In this case, the access abstraction consists of objects which forward some messages from Bob to Carol, where Alice controls this message-forwarding behavior. The access abstractions presented in Part II are idealizations drawn from existing prior systems. We are unaware of previous systematic presentations on the topic of access abstraction mechanisms and patterns.

The prior access control literature does not provide a satisfying account of how the behavior of such unprivileged programs contributes to the expression of access control policy. A contribution of Chapter 8 is to distinguish between "permission" and "authority," to develop a taxonomy of computable bounds on eventual permission and authority, and to explain why "partially behavioral bounds on eventual authority"—the "BA" of Table 8.1 (p. 60)—is needed to reason about many simple access abstractions.

2.3 Distributed Access Control

E objects are aggregated into persistent process-like units called *vats* (Figure 2.3). Inter-vat messages are conveyed by E's cryptographic distributed capability protocol, Pluribus, which

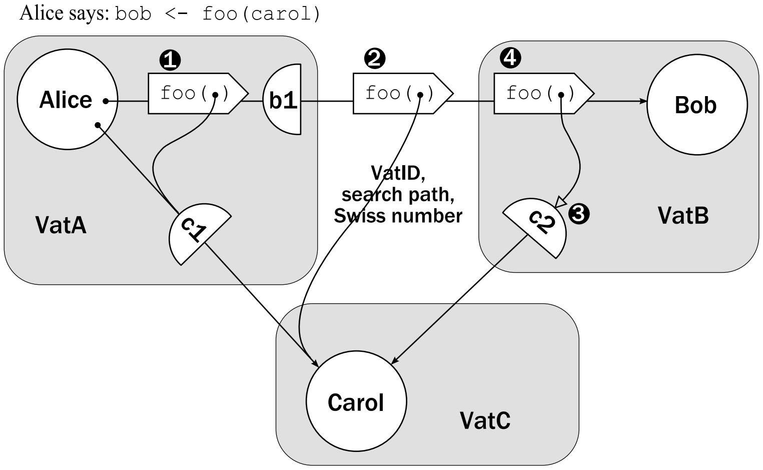


Figure 2.3: Distributed Access Control.

ensures that distributed object references are unforgeable and unspoofable.

Chapter 7 presents a simplified form of Pluribus. None of the access control properties provided by Pluribus are novel. Protocols like Pluribus transparently extend the reference graph across machines, while cryptographically enforcing *some* of the properties of the object-capability access control model. Section 11.5 explains several aspects of the object-capability model that cannot be enforced between mutually suspicious machines on open networks. Our overall system consists of object-capability islands sending messages to each other over a cryptographic capability sea. A contribution of this section is to introduce a unified account of access control in such a mixed system.

E has two forms of invocation. The "`.`" in our first two diagrams is the conventional "immediate-call" operator. The "`<-`" in Figure 2.3 is the "eventual-send" operator. Their differences bring us to concurrency control.

2.4 Distributed Concurrency Control

E's concurrency control is based on communicating event loops. Each vat has a heap of objects, a call-return stack, a queue of pending deliveries, and a single thread of control. An immediate-call pushes a new stack frame, transferring control there. An eventual-send enqueues a new pending delivery—a pair of a message and the target object to deliver it to—on the pending delivery queue of the vat hosting the target. A vat's thread is a loop, dequeueing the next pending delivery and invoking the target with the message. Each of these invocations spawns a *turn*, which runs to completion as a conventional sequential program, changing vat state and enqueuing new pending deliveries. All user-level code runs in these turns, including all state access and modification.

A key contribution of this dissertation is a system of reference states, transition rules, and message delivery properties associated with each state, as represented by Figure 2.4 and presented in Part III. This system isolates turns to support deadlock-free consistency (Chapters 14), reifies delayed outcomes to support dataflow patterns (Chapters 16), and propagates delayed problem reports to support application-level handling of and recovery from distributed failures (Chapter 17).

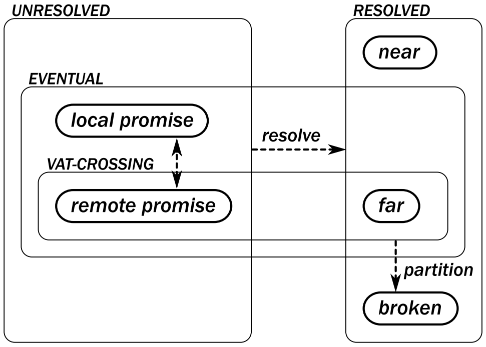


Figure 2.4: Distributed Concurrency Control.

A near reference is the form of reference familiar from conventional sequential non-distributed object programming. A near reference is a *local* reference to a *resolved* target. In other words, a near reference starts and ends in the same vat, and it knows which target object it should deliver messages to. A near reference conveys both immediate-calls and eventual-sends, providing full authority to invoke its target. Only near references convey immediate-calls, so objects in one vat may not immediate-call objects in another vat. Thus, each turn implicitly has mutually exclusive access to all state to which it has synchronous access. No explicit locks are needed or allowed.

A far reference is a vat-crossing reference to a resolved target. So long as its target's vat is reachable, a far reference conveys eventual-sends to its target. A far reference thereby provides only the authority to enqueue pending deliveries in its target's vat's queue, to be delivered to its target in separate isolated turns. An eventual-send is non-blocking—the sender cannot block its vat's thread waiting for a response. Thus, conventional deadlocks cannot occur.

A promise is a reference to the result of an eventual-send, *i.e.*, an eventually-send immediately returns a promise for the result of the turn in which this message will be delivered. Until this turn occurs, the value of this result remains *unresolved*, and messages eventually-sent to the promise are buffered in the promise. A promise can also be created explicitly, providing its creator the authority to determine its resolution. The creator may thereby control the buffering and release of messages eventually-sent by a sender to that promise. Promises are a form of delayed reference, supporting some of the concurrency control patterns associated with dataflow, actors, and concurrent logic programming.

A broken reference occurs as a result of a problem. If a turn terminates by throwing an exception, that breaks the promise for the turn's result. A partial failure—a network disconnect or crash—will break vat-crossing references. Once a reference is broken, it stays broken. Inter-vat connectivity can be regained using off-line capabilities.

An off-line capability (unshown) provides *persistent* access to the object it designates—it allows one to obtain a new reference to this object. One form of off-line capability, a URI string, can be transferred over out-of-band media in order to securely bootstrap initial connectivity.

For most purposes, programmers need be concerned about only two cases. When a reference is statically known to be near, their code may immediate-call it without concern for partial failure. When they don't statically know whether a reference is near or not, their code should eventual-send to it, and cope if a broken reference reports that the message may not have been delivered. Since near references provide a strict superset of the guarantees provided by the other reference states, code prepared for the vat-crossing case will be compatible with the local case without further case analysis. The "`<-`" of the eventual-send expression, and the "`->`" of the when-catch syntax explained in Chapter 18, alert programmers of the locations in their code where interleavings and partial failure concerns arise.

2.5 Promise Pipelining

Machines grow faster and memories grow larger. But the speed of light is constant and New York is not getting any closer to Tokyo. As hardware continues to improve, the latency barrier between distant machines will increasingly dominate the performance of distributed computation. When distributed computational steps require unnecessary round trips, compositions of these steps can cause unnecessary cascading sequences of round trips.

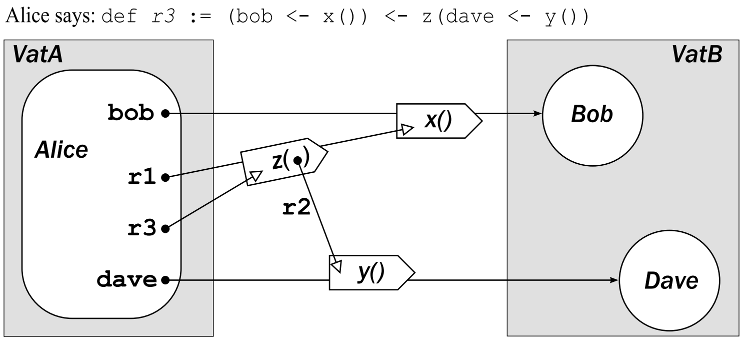


Figure 2.5: Promise Pipelining.

Related prior work shows how, in asymmetric client-server systems, distributed functional composition can use promise pipelining (Figure 2.5) to avoid cascading round trips. In Chapter 16, we show how to extend this technique to symmetric peer-to-peer systems.

2.6 Delivering Messages in E-ORDER

Various distributed systems provide useful constraints on message delivery orders, reducing the cases programmers must face. In Chapter 19, we show how FIFO and weaker orders fail to enforce a needed access control restriction: FIFO allows Bob to access Carol earlier than Alice expects. By the conventional taxonomy, CAUSAL order is the next stronger order. If enforced, it would safeguard Alice's expectations. However, CAUSAL order is too strong to be enforceable, purely by cryptographic means, between mutually suspicious machines.

A contribution of Chapter 19 is to present E-ORDER, defined as a further attenuation of the authority provided by a passed reference. When Alice passes Bob a reference to

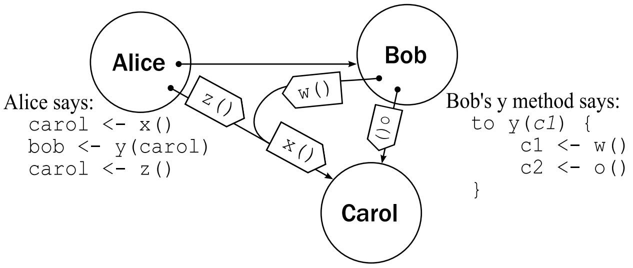


Figure 2.6: Delivering Messages in E-ORDER.

Carol, this reference provides Bob only the authority to cause messages to be delivered to Carol *after* prior messages already sent by Alice on this reference have been delivered. E-ORDER is sufficiently strong to enforce the restriction Alice needs, and sufficiently weak to be cryptographically enforceable among mutually suspicious machines.

2.7 Emergent Robustness

Part V takes us on a tour of some applications built on these foundations, demonstrating how these principles enhance overall robustness. This part explains the synergy that results from attenuating authority simultaneously at multiple layers of abstraction. Figure 2.7 sketches a visualization of the attack surface of the system we describe. The filled-in areas represent the vulnerabilities produced by the distribution of authority. Each level is an expansion of part of the level above. By attenuating authority at multiple scales simultaneously, overall vulnerability resembles the surface area of a fractal that has been recursively hollowed out.

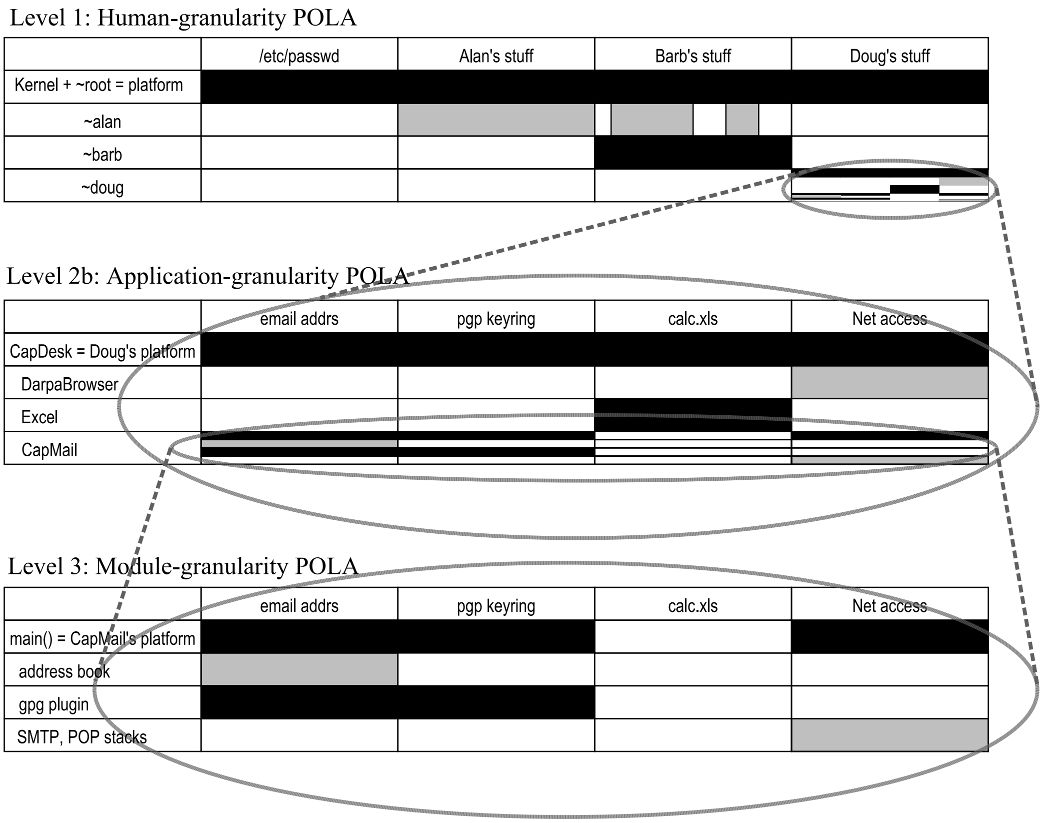


Figure 2.7: Emergent Robustness.

Part I

The Software Composition Problem

Chapter 3

Fragile Composition

Why is it currently so hard to build large-scale robust software systems? Within today's dominant paradigms of composition, programmers are faced with a dilemma, which we illustrate as the diagonal on Figure 3.1. At upper left, the price of enabling cooperative interaction is that destructive interference is enabled as well. In reaction to the resulting hazards, various mechanisms have been introduced into programming languages and operating systems to express and enforce perimeters that more safely separate components. Naively applied, they also prevent cooperative interaction. Manifestations of this dilemma appear in both access control (Figure 3.2) and concurrency control (Figures 3.3 and 13.3). Strategies for dealing with these are the subjects of Part II and Part III of this dissertation.

As language and system designers, our goal is to help programmers move into the upper right corner, where the interactions needed for cooperation are enabled while minimizing the extent to which problematic interactions are thereby also enabled.

By *enabling interaction*, we mean to examine which subjects can cause what effects on what objects when. Conventional access control reasoning examines which subjects can cause what effects on what objects, but is mostly unconcerned with temporal issues of when these effects may be caused. (More precisely: is unconcerned with the interleaving constraints needed to maintain consistency.) Conventional concurrency control reasoning is concerned with temporal issues (again, in terms of controlling interleaving) of what effects might be caused on what objects when, but is mostly unconcerned with what subjects might

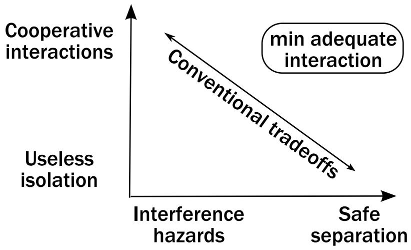


Figure 3.1: Purposes and Hazards. In conventional systems, the price of enabling components to cooperate is that hazardous interactions are enabled as well. To compose effectively, we must find ways to enable those interactions needed for the composition's purposes while minimizing the degree to which hazardous interactions are thereby also enabled.

be able to cause these effects. In this way, these two forms of reasoning are orthogonal projections of richer underlying issues of *interaction control*.

3.1 Excess Authority: The Gateway to Abuse

Software systems today are vulnerable to attack. This widespread vulnerability can be traced in large part to the excess authority we routinely grant programs [Kar03]. In these systems, virtually every program a user launches is granted the user's full authority, even a simple game program like Solitaire. While users need broad authority to accomplish their various goals, this authority greatly exceeds what any particular program needs to accomplish its task. All widely-deployed operating systems today—including Windows, UNIX variants, Macintosh, and PalmOS—work on this principle.

When you run Solitaire, it only needs adequate CPU time and memory space, the authority to draw in its window, to receive the UI events you direct at it, and to write into a file you specify in order to save your score. The *Principle of Least Authority* (POLA)[1] recommends that you grant each program only the authority it needs to do its job [MS03]. If you had granted Solitaire only this limited authority, a corrupted Solitaire might be annoying, but not a threat; it may prevent you from playing the game or lie about your score. Instead, under conventional systems, it runs with all of your authority. It can delete any file you can. It can scan your email for interesting tidbits and sell them on eBay to the highest bidder. It can install a back door and use your computer to forward spam. While Solitaire itself probably doesn't abuse its excess authority, it could. If an exploitable bug in Solitaire enables an attacker to gain control of it, the attacker can do anything the user running Solitaire is authorized to do.

If Solitaire only needs such limited authority, why does it get all of your authority? Well, what other choice do you have? Figure 3.2 shows your choices. On the one hand, you can run Solitaire as an application. Running it as an application allows you to use all the rich functionality and integration that current application frameworks have been built to support, but at the price of trusting it with all your authority. On the other hand, you can run it as an applet, granting it virtually no authority, but then it becomes isolated and mostly useless. A Solitaire applet could not even save its score into a file you specify.

Sandboxing provides a middle ground between granting a program the user's full authority and granting it no authority. Some approaches to sandboxing [GMPS97, Gon99, PS01] enable you to configure a static set of authorities (as might be represented in a policy file) to be granted to the program when it is launched. The problem is that you often do not know in advance what authorities the program actually needs: the least authority needed by the program changes as execution progresses [Sch03].

In order to successfully practice POLA, we need to take a different approach. Rather than trading security for functionality, we need to limit potential abuse without interfering with potential use. How far out might we move on the horizontal axis without loss of functionality or usability? Least authority, by definition, includes adequate authority to get the job done. Providing authority that is adequate means providing it in the right amount and at the right time. The key to putting POLA into practice lies in the dynamic allocation of authority; we must provide the right amount of authority just-in-time, not

[1] POLA is related to Saltzer and Schroeder's *Principle of Least Privilege* [SS75]. However, it is not clear precisely what they meant by "privilege." Section 8.1 explains what we mean by "authority."

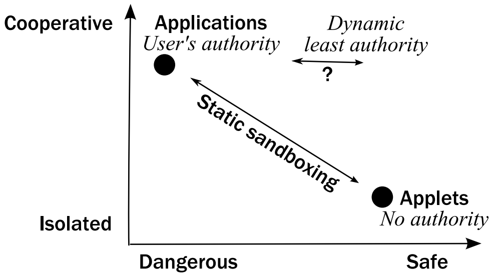


Figure 3.2: Functionality *vs.* Security? Conventional access control systems force a tradeoff between statically granting a program enough authority to do anything it *might* need to do versus denying it authority it could use to cause harm. By granting authority dynamically as part of the request, we can provide just that authority needed for that request. Since "least authority" includes adequate authority, least authority is as high on the vertical axis as needed for the requested functionality. The horizontal line with the question mark represents the design question: How much safety may we achieve without loss of functionality?

excess authority just-in-case.

3.2 How Much Authority is Adequate?

How do we know how much authority a program actually needs? Surprisingly, the answer depends on architectural choices not normally thought to be related to security—the logic of designation. Consider two Unix shell commands for copying a file. In the following example, they both perform the same task, copying the file `foo.txt` into `bar.txt`, yet they follow very different logics of designation in order to do so. The result is that the least authority each needs to perform this task differs significantly.

Consider how `cp` performs its task:

```
$ cp foo.txt bar.txt
```

Your shell passes to the `cp` program the two strings `"foo.txt"` and `"bar.txt"`. The `cp` program uses these strings to determine which files it should copy.

By contrast consider how `cat` performs its task:

```
$ cat < foo.txt > bar.txt
```

Your shell uses these strings to determine which files you mean to designate. Once these names are resolved, your shell passes direct access to the files to `cat`, as open file descriptors. The `cat` program uses these descriptors to perform the copy.

Now consider the least authority that each one needs to perform its task.

With `cp`, you tell it which files to copy by passing it strings. By these strings, you mean particular files in your file system, to be resolved using your namespace of files. In order for `cp` to open the files you name, it must already have the authority to use your namespace, and it must already have the authority to read and write any file you might name. Given this way of using names, cp's *least authority* still includes all of your authority to the file system. The least authority it needs is so broad as to make achieving either security or reliability hopeless.

With cat, you tell it which files to copy by passing it the desired (read or write) access to those two specific files. Like the cp example, you still use names in your namespace to say which files you wish to have cat copy, but these names get evaluated in your namespace prior to being passed to cat. By passing cat file descriptors rather than strings to convert to descriptors, we reduce the authority it needs to do its job. Its least authority is what you'd expect—the right to read your foo.txt and the right to write your bar.txt. It needs no further access to your file system.

Currently under Unix, both cp and cat, like Solitaire, run with all your authority. But the least authority they require to copy a file differs substantially. Today's widely deployed systems use both styles of designation. They grant permission to open named files on a per-user basis, creating dangerous pools of excess authority. These same systems dynamically grant an individual process access to a resolved file descriptor on a per-invocation basis. Ironically, only their support for the "cp" style is explained *as* their access control system. Shells supporting narrow least authority [Sea05] differ from conventional systems more by the elimination of the "cp" style than by the elaboration of the "cat" style.

Following Carl Ellison's usage [Ell96], we refer to the "cp" style of designation as "name-centric" and the "cat" style as "key-centric." In *name-centric* systems, *names* are generally assumed to be human readable, like the filenames in our cp example or DNS names in Ellison's. Name-centric systems pass names to communicate designations, requiring a shared namespace. In *key-centric* systems, *keys* are opaque and unambiguous, like the file descriptors in our cat example or cryptographic keys in Ellison's. Key-centric systems pass keys to communicate designations. In key-centric systems, names are bound to keys in namespaces local to each participant, such as the user's file system in our example. We return to Ellison's distinction in the related work section at the end of this chapter.

Programming languages based on lexical naming [Chu41] and references (including object [GK76, HBS73], lambda [KCR98, Mil84], and concurrent logic [Sha83, Sar93, RH04] languages) generally follow the key-centric style, combining designation and access as suggested by the cat example above. Names are evaluated in their caller's namespace to a value, such as a reference to an object, which is then passed as an argument to the callee. The callee doesn't know or care what the caller's name for this reference is. By passing this reference as a particular argument of a particular call, the caller both tells the callee what object it wishes the callee to interact with as part of this request, and provides the callee the ability to interact with this object. The changes to these languages needed to support least authority consist more of eliminating their support for the name-centric designation rather than elaborating their support for the key-centric designation. Besides E, in related work (Part V), we visit several other examples of such languages.

3.3 Shared-State Concurrency is Difficult

The diagonal on Figure 3.3 reflects common anecdotal experience when programming in the dominant concurrency control discipline: shared-state concurrency [RH04] (also described as shared memory multi-threading with fine grained locking). Correct programs must both avoid deadlock and preserve their own consistency. Within the shared-state concurrency

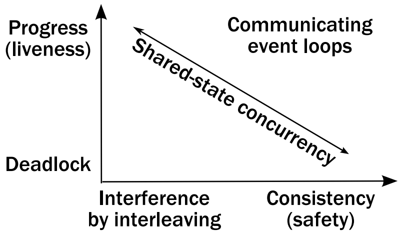


Figure 3.3: Progress *vs.* Consistency? In the presence of concurrency, correct programs must prevent those interleavings which threaten consistency while allowing all those needed for progress. In practice, the conventional approach of making components “thread-safe” forces a tradeoff between confidence that consistency is maintained versus confidence that progress will continue. The event-loop style allows both requirements to be met with confidence.

control discipline, it is certainly possible to write such thread-safe programs, and there are many beautiful small examples of such programs. However, the more common experience, especially when composing large systems, is that this discipline forces programmers to make a tradeoff between bad choices [RH04, Lee06, Ous96].

For a programmer to be confident that a program’s consistency is preserved, the program must lock often enough to exclude all interleavings which threaten consistency. It is often difficult to be confident that such programs are deadlock-free. In a program stingy enough with locking to be confident that it is deadlock-free, it is often difficult to be confident in its continued consistency. Errors in this style of programming have provided fertile ground for demonstrating the value of static checking [EA03]. To understand this dilemma, Part III first presents the `StatusHolder`, a composable abstraction that is properly considered trivial among sequential object programmers. It then attempts, by successive refinement, to develop a thread-safe `StatusHolder` that is robustly composable under shared-state concurrency. Figure 13.3 (p. 101) shows the results of our exploration.

We will argue that an alternative discipline, communicating event loops [HBS73, Agh86, DZK$^+$02, ZYD$^+$03], provides a framework in which programmers can straightforwardly write programs that are both consistency-preserving and deadlock-free. Although well known and at least as old as shared-state concurrency, communicating event-loops have received a minute fraction of the linguistic support which has been invested in shared-state concurrency. We hope the present work helps repair this imbalance.

3.4 Why a Unified Approach?

Even if access control and concurrency control are distinct projections of more general issues of interaction control, by itself this does not establish the need to address the more general issues in a unified way. Indeed, if all interaction control sub-problems can be cast either as pure access control problems or pure concurrency control problems, then keeping these two forms of reasoning separate would constitute a better separation of concerns.

We close Part III on page 137 with a counter-example to such separation: the rationale for E’s default distributed message delivery order, which we call E-ORDER. There is a conventional spectrum of distributed message delivery order guarantees. This spectrum includes the consecutive sequence: UNORDERED $\leq$ FIFO $\leq$ CAUSAL $\leq$ AGREED [Ami95, Bir05]. Unfortunately, CAUSAL and stronger orders are too strong to be enforced among mutually defensive machines; FIFO and weaker orders allow a race condition which violates a desirable access control property. E-ORDER, which is stronger than FIFO and weaker than CAUSAL (FIFO $\leq$ E-ORDER $\leq$ CAUSAL), satisfies all our constraints simultaneously. We arrived at this design choice only by thinking of access control issues and concurrency control issues together, in terms of using references to control the propagation of causality through the reference graph.

3.5 Notes on Related Work on Designation

This distinction between the two styles of designation explained above has intriguing parallels in the philosophy literature contrasting two perspectives on naming. In *Naming and Necessity* [Kri80], Saul Kripke reacts against Bertrand Russell's suggestion that names be regarded as "compact descriptions," arguing instead that names should be regarded as "rigid designators." Russell's "compact descriptions" are similar in some ways to the widely shared names of name-centric systems. Kripke's "rigid designators" are similar in some ways to the keys of key-centric systems. In other ways, Kripke's rigid designators are like the local names which are bound to keys. Kripke's causal account of how designations come to be shared is based on acts of pointing and communication, which have intriguing analogs to our notions of object creation, invocation, and argument passing.

Although it has long been understood *that* capabilities bundle designation with permission [Lev84], Norm Hardy's *The Confused Deputy* [Har88a] was the first to explain *why* these need to be bundled, and to point out other dangers that follow from their separation—dangers beyond the excess authority hazards explained above.

Carl Ellison's *Establishing Identity Without Certification Authorities* [Ell96] contrasts these two naming philosophies as applied to the Internet as a whole. He shows how the Domain Name System and conventional uses of X509 certificates are name-centric: These systems attempt to create an Internet-wide shared hierarchical namespace of human meaningful names, like a conventional file system writ large. He contrasts this with key-centric systems like SDSI/SPKI [EFL$^+$99], in which opaque cryptographic keys are the only globally meaningful designators, and all use of human readable names is rooted in the local naming environment of the entity employing the name. SDSI/SPKI is an offline certificate-based system. As with file systems or DNS names, human readable names are actually naming *paths*, for traversing paths through a graph starting at some root. However, in key-centric systems like SDSI/SPKI, the root at which this traversal starts is each participant's own local namespace. The Client Utility architecture covered in Related Work Section 25.6 included an online protocol that also used locally rooted path-based names. In these key-centric systems, as in our `cat` example, humans use human readable names to securely designate.

Marc Stiegler's *An Introduction to Petname Systems* [Sti05] explains phishing as the inverse problem: How may designations be communicated *to* humans as human readable names, in a manner resistant to both forgery and mimicry? Our vulnerabilities to phishing are due to the name-centric nature of the Domain Name System. Stiegler explains Tyler Close's realization that *petname systems* extend key-centric systems in a phishing resistant fashion. The local namespace of most key-centric systems provides a many-to-one mapping of names to keys. The inverse problem requires mapping keys back to names. The local namespaces of petname systems provide this backwards mapping. Close's "Petname tool" [Clo04a] and Ka-Ping Yee's "Passpet" are browser extensions for phishing resistant browsing. Stiegler's own CapDesk [WT02, SM02] and Polaris [SKYM04], which we explain briefly in Part IV, use petnames for secure window labeling of confined applications.

Global namespaces create intractable political problems. Froomkin's *Toward a Critical Theory of Cyberspace* [Fro03] examines some of the politics surrounding ICANN. In a world using key-centric rather than name-centric systems, these intractable political problems would be replaced with tractable technical problems.

Chapter 4

Programs as Plans

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. But you do not need to anticipate every possible contingency. If someone does something you did not expect, you will probably be better able to figure out how to cope at that time anyway.

When programmers write programs, they express plans for machines to execute. To formulate such plans, 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, corrupting its continued execution. Such corrupted executions likely violate assumptions on which other programs depend, potentially spreading corruption throughout a system. To program successfully, programmers use abstraction and modularity mechanisms to limit (usually implicitly) which assumptions must be made, and to structure these assumptions so they are more likely to mesh without conflict. Beyond these assumptions, correct programs must handle all remaining relevant contingencies. The case analysis burden this requires must be kept reasonable, or robust programming becomes impractical.

4.1 Using Objects to Organize Assumptions

We describe the tenets of conventional object programming practice in terms of *decomposition*, *encapsulation*, *abstraction*, and *composition*. We describe how programmers use each of these steps to organize assumptions.

4.1.1 Decomposition

When faced with the need to write a program providing complex functionality, programmers must subdivide the problem into manageable units. Emerging from the spaghetti-code software crisis of the 1960s [Dij72], programmers learned to practice hierarchical decomposition [Wir71, Mil76], dividing each task into subtasks. To oversimplify a bit, each node in this decomposition tree was represented by a procedure which achieves its task partially by contracting out subtasks to the procedures it calls. Programmers would seek task divisions where the responsibilities of each procedure was clear, simple and separate.

With hierarchical decomposition, programmers could organize assumptions about which procedure was supposed to achieve which purpose. But there remained the issue of what resources each could employ in order to accomplish its purpose. Different procedures might manipulate common data in conflicting ways—the manipulations performed by one might disrupt the assumptions of another. So long as access to common data was undisciplined, assumptions about how the data was to be manipulated were spread throughout the code, and could easily conflict.

4.1.2 Encapsulation

To avoid conflicting assumptions about data manipulation, programmers learned to package code and data together into modules and abstract data types [Par72, LZ74, LSA77]. For uniformity, let us refer to instances of abstract data types as *objects*. The code of each object still manipulates data, but the data it manipulates is now private to that object. When writing the code of such an object—when formulating a plan for how it will use its data to accomplish its purpose—the programmer may now assume that this data is not also being manipulated by other potentially conflicting plans. This discipline enables programmers to create systems in which a massive number of plans can make use of a massive number of resources without needing to resolve a massive number of conflicting assumptions. Each object is responsible for performing a specialized job; the data required to perform the job is encapsulated within the object [WBM03].

4.1.3 Abstraction

When programmers carve the functionality of a system into subtasks using only hierarchical decomposition, each provider serves only the specific concrete purpose needed to contribute to its one client. Instead, programmers learned to create opportunities for reuse and polymorphism [DN66, Mey87, Mey88, GHJV94, WBM03].[1] For *reuse*, a provider serves an abstract purpose (the classic example is a LIFO stack) that multiple clients can employ for multiple concrete purposes (such as parsing or reverse Polish arithmetic). The abstract purpose is represented by an *interface* or *type*, in which the message names (`push` and `pop`) indicate the abstract purpose they serve. For *polymorphism*, multiple concrete providers can implement the same abstract service in different concrete ways (indexing into an array or consing onto a linked list).

An interface designed to serve the needs of only one client will not help reuse. An interface that exposes implementation details will not help polymorphism. A well designed interface serves as an *abstraction boundary*, simultaneously abstracting over the multiple concrete reasons why a client may wish to employ this service and the multiple concrete means by which a provider may implement this service. The interface represents the relatively thin and stable assumptions by which clients and providers coordinate their plans. It provides the "what" (stack) that insulates the multiple "why"s of client plans (parsing, reverse Polish arithmetic) from the multiple "how"s of provider plans (array, cons), and vice versa, so they are freer to evolve separately without disrupting each other's assumptions.

[1] In this dissertation, we use the term "polymorphism" as it is used in the object programming literature: for late binding of message to method, allowing for runtime substitutability of different implementations of the same interface. For the concept ML programmers call "polymorphism" we use the term "parameterized types."

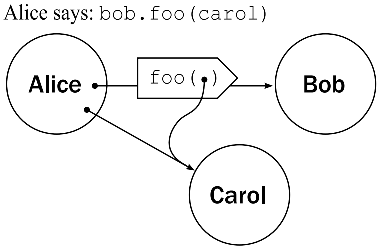


Figure 4.1: Composition Creates New Relationships. When Alice says `bob.foo(carol)`, she is Bob's client, Bob is her provider, and Carol is an argument parameterizing Alice's request. By passing this reference, Alice also composes Bob and Carol, enabling Bob to be Carol's client. Alice makes this request so that Bob, in interacting with Carol to serve some purpose of Bob's, will thereby serve some other purpose of Alice's.

Once programmers have designed the abstraction boundaries needed to carve up the required functionality, their remaining planning problems are comparatively well separated. Since abstract interfaces stand between client objects and provider objects, these objects also stand between abstract interfaces. As provider, each object's plan must be designed to contribute not to its client's concrete plans, but to the abstraction of these plans represented by the interface it is charged with implementing. As client, each object's plan should be based not on the concrete behavior of its providers, but only on the abstraction of their behavior represented by the provider interfaces it assumes.

4.1.4 Composition

Abstraction boundaries help separate concerns, giving composition its power.

The use of abstraction boundaries explained above motivates only relatively static client-provider subcontracting networks. In a static reference graph, the payoff from reuse would still be clear: Parsing is very different from reverse Polish arithmetic, so their reuse of common stack code provides a real design benefit. But polymorphism would be uninteresting: Since our example stack implementations all implement the same interface contract, they are semantically identical and differ only in efficiency. If all polymorphism were like our stacks, polymorphism would be a detail rather than a major tool of good object design. If the choice of provider were based only on efficiency, this decision would best be made at design time anyway, which would not require any runtime late binding mechanism. To understand the power of polymorphism, we must see beyond static "client-provider" reasoning when describing how objects use each other's services.

Figure 4.1 depicts the basic step of object computation: the message send [GK76, HBS73, Hew77]. (To encourage anthropomorphism, we use human names for objects.) When focusing on the `foo` message as requesting some service Bob provides, we may speak of Bob as a provider, Alice as Bob's client, and Carol as an argument, contributing meaning to the `foo` request. By making this request, Alice also gives Bob access to Carol—the ability to request whatever service Carol provides. When focusing on this aspect, we may speak of Carol as provider, Bob as Carol's client, and Alice as their introducer. In this latter perspective, Alice uses the `foo` message to compose Bob and Carol—to set up and give meaning to the client/provider relationship between Bob and Carol. This relationship has one meaning to Bob and a different meaning to Alice. To make this concrete, let us examine some examples.

Higher Order Functions Alice asks Bob to sort a list, and Carol is the comparison routine Alice wants Bob to use. As Carol's client, Bob is perfectly happy to use whatever comparison Carol represents. Alice chooses a comparison routine based on how she wishes the list to be sorted.

Redirecting Output Alice asks Bob to print himself onto the output stream she provides. As client of this stream, Bob cares only that it provides the operations he uses to render himself. Only Alice cares where it goes, or if it computes a hash of the text rather than going anywhere.

Visiting an Expression Tree Bob is the abstract syntax tree of an expression. Alice asks Bob to send his components to Carol as visitor [GHJV94]. Only Alice cares whether Carol reacts by evaluating Bob as an arithmetic expression or by pretty printing Bob.

Regarding Bob's purposes as client, so long as Carol implements the interface Bob assumes of an argument of a `foo` message, Bob typically does not care which Carol is chosen. He will simply proceed to make use of Carol's services in the same way regardless. The choice of Carol typically doesn't affect Bob, it affects Alice. Alice chooses a Carol so that Bob's use of Carol's services for Bob's purposes serves some other purpose of Alice's [Tri93]. The abstraction boundary between Bob and Carol gives Alice this flexibility. The abstraction boundary between Alice and Bob insulates the design of Bob's purposes from needing to know about Alice's purposes.

Above, we speak as if the notion of "program as plan" applies only at the granularity of individual objects. But such microscopic plans are rarely interesting. To understand patterns of composition, for purposes of description, we aggregate objects into *composites*. This aggregation is subjective: different aggregations are appropriate for different descriptive purposes. The interactions between disjoint composites are much like the interactions among individual objects and are subject to many of the same design rules.

Section 6.2 gives a simple example of a composite. The example plans in parts II and III consist of small object patterns. Part IV examines plan coordination across several scales of composition.

4.2 Notes on Related Work

Programs were viewed as plans quite early. The first high level *non*-von Neumann programming language is Konrad Zuse's Plankalkül language [Zus59], invented circa 1946, to run on one of the world's earliest computers, Zuse's Z3 [Zus41]. Plankalkül means "plan calculator."

Herbert Simon's *The Architecture of Complexity* [Sim62] explains how "nearly decomposable" hierarchies organize complexity across a broad range of natural and artificial systems. *Modularity in Development and Evolution* [SW04] gives a good overview of how modularity in biology contributes to evolvability. The "Structured Programming" movement in software engineering [Wir71, Mil76] applies similar insights for similar purposes:

by minimizing the dependencies between modules, each module is freer to "evolve" without disrupting the functioning of other modules.

David Parnas' *On the Criteria to be Used in Decomposing Systems into Modules* [Par72] explains why systems decomposed along data manipulation lines often hide complexity better than systems decomposed along chronological plan-sequence lines. The extensive literature on abstract data types (introduced by Liskov [LZ74]) shows how programming languages have leveraged these observations.

Of the extensive literature on object-oriented design principles, Bertrand Meyer's *Object-Oriented Software Construction* [Mey88] and Wirfs-Brock and McKean's *Object Design—Roles, Responsibilities and Collaborations* [WBM03] are particulary insightful. *Design Patterns, Elements Of Reusable Object-Oriented Software* [GHJV94] spawned a further extensive literature capturing, classifying, and analyzing particular arrangements making use of these object design principles.

Abadi and Lamport's *Conjoining Specifications* [AL95] distinguishes the correctness issues which arise when *decomposing* a system into specialized components *vs.* those which arise when *composing* a system from reusable components.

Our explanation of object programming owes a debt to the plan-coordination literature in economics. Our explanation of encapsulation derives in part from Friedrich Hayek's explanation of the plan-separation function of property rights [Hay37, Hay45, MD88]. Division of the society's resources among the active entities enables the decentralized formulation of plans, based on mostly non-conflicting assumptions about what resources will be at each plan's disposal.

Our explanation of abstraction boundaries derives in part from Ludwig Lachman's explanations of the plan coordinating function of institutions [Lac56, TM06]. The concept of "post office" stands between all the reasons why people may want to mail a letter and all the means by which delivery services may convey letters to their destinations. The abstraction boundary both separates these plans from one another's details, and it provides for interactions so they can usefully coordinate.

Our explanation of composition derives in part from Mark Granovetter's visual explanations of social connectivity dynamics [Gra73, MMF00], where new relationships form as people introduce people they know to each other. With Granovetter's permission, we refer to Figure 4.1 and similar diagrams as "Granovetter diagrams."

Chapter 5

Forms of Robustness

Under what conditions will a correct program behave correctly? A program is a mathematical object; its correctness is a mathematical question. As engineers, we care about what transpires when a program is run. We must examine the bridge from abstract correctness to desired behavior. If this bridge rests on flimsy assumptions, correctness is inadequate for robustness. This chapter distinguishes forms of correctness based on the structure of these assumptions, and examines how each contributes to robustness.

Although this chapter adapts some concepts from the formal correctness literature, we are concerned here more with conventional software engineering quality issues. A program is correct when it "meets its specification," *i.e.*, when it does what it is supposed to do. Formal correctness demands formal specification. However, even after decades of research and development on formal specification languages, the practical reality today is that most specifications are informal, and many are not even written down [DLP79]. Nevertheless, programmers informally reason about the correctness of their programs all the time, and with some success [Gut04]. The formal correctness literature helps explain how programmers do so, and how they may do better: by applying informal (and psychologically plausible) adaptations of these formal techiniques. We close this chapter (in Section 5.8) with some thoughts on related areas in formal correctness.

5.1 Vulnerability Relationships

What does the following C program do? Is it correct? What would this mean?

```
static int count = 0;
int incr() { return count += 1; }
```

When we say that a program P is correct, we normally mean that we have a specification in mind (whether written down or not), 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 correct behavior is implicitly allowed to depend on its machine's correct behavior. If P's correct behavior is allowed to depend on another component R's correct behavior, we will say that P *relies upon* R [Tin92, Sta85, AL93, AL95, CC96]. We will refer to the set of all elements on which P relies as P's *reliance set*.

5.2 Platform Risk

Systems are built in many layers of abstraction. For purposes of analysis, at any moment we must pick a frame of reference—a boundary between a *platform* (such as a language or operating system kernel) enforcing rules of permissible action, and the set of all possible programs running on that platform, assumed able to act only in permitted ways [And72, NBF$^+$80, SS87]. By *program* we refer only to code running on that platform, whose behavior is controlled by the platform's rules. We implicitly allow all programs running on a platform to assume their platform is correct—their platform is in their reliance set [Tin92]. For `incr`, the natural frame of reference is to choose the C language as the platform, requiring correct behavior only if the C implementation behaves correctly. The relevant universe of programs consists of that code whose interactions with `incr` are governed by the rules of C, which is to say, all other C programs which might be linked together and run in the same address space as `incr`.

Given a set of objects, anything within the reliance sets of all of them is a *central point of failure* for that set. A platform is a central point of failure for the set of all possible programs running on that platform.[1]

In a traditional timesharing context, or in a conventional centrally-administered system of accounts within a company, the platform includes the operating system kernel, the administrator accounts, and the administrators. The platform provides the mechanisms used to limit the authority of the other players, so all the authority it manages is vulnerable to the corruption or confusion of the platform itself. The platform is, therefore, a central point of failure for all the systems running on that platform. While much can be done to reduce the likelihood of an exploitable flaw in the platform—primarily by making it smaller and cleaner—ultimately, any centralized system will continue to have this Achilles heel of potential full vulnerability.

As we will see, distributed systems can support full decentralization. Each portion rests on a platform to which it is fully vulnerable, but different portions rest on different platforms. A fully decentralized system may have no central points of failure, such as a common administrator.

5.3 Conventional Correctness

Besides the platform, what else can cause `incr` to behave incorrectly?

What if other C code in the same address space overwrites `count`'s storage or `incr`'s instructions? What if a concurrently executing thread overwrites `incr`'s stack frame? Such behavior will cause `incr` to behave incorrectly, but this does not mean `incr` itself is incorrect. Because one piece of C code can do nothing to defend itself against such "friendly fire," we must allow all C code the assumption that all other C code in its address space is correct (or at least meets some broad correctness criteria).

The semantics of such fragility is often captured by introducing an *undefined* state. For

[1] As commonly used, the term "Trusted Computing Base" (TCB) sometimes means "reliance set," sometimes "platform," and sometimes the set of central points of failure for all the objects running on a given platform, *i.e.*, the intersection of their reliance sets. "Rely" is defined in terms of the objective situation (P is vulnerable to R), and so avoids confusions engendered by the word "trust."

While our focus here is on correctness and consistency, a similar "reliance" analysis could be applied to other program properties, such as promptness [Har85].

```
public class Counter {
    private int count = 0;
    public int incr() {
        return count += 1;
    }
}
```

Figure 5.1: A Cooperatively Correct Counter in Java. If the purpose of a counter is to report to its clients how many times it has been invoked, then the hazards presented by an instance of this Counter class are 1) that its count may overflow, or 2) that it may be called from multiple threads. If all clients of a counter carefully avoid these hazards, then that counter will correctly serve its purpose for each of them.

example, if any C code stores through a bad pointer, the effects are undefined, which is to say, the language specification then allows anything to happen. Corruption potentially spreads through the whole program instantly, totally, and undetectably. If `incr` misbehaves as a result, we would say the bug is in the other C code, not in `incr` or in the C language implementation. Because there is no isolation within the universe defined by the C language, no robustness is possible. Everything is fully vulnerable to accident or malice by anything else within its universe. Each is a central point of failure for all. Correct code can only be expected to behave correctly when all other code in its address space is also correct.

5.4 Cooperative Correctness

Memory-safe languages do better. The corresponding Java code in Figure 5.1 isn't vulnerable to the above non-local problems. However, in Java, as in C, addition may overflow. We could make it correct by revising `incr`'s specification to reflect what it actually does in these languages: modular addition adjusted for two's complement. Such a specification accurately describes what `incr` does, but not what it is for. To understand how programmers will use it, we examine *purposes* and *hazards*. The programmer of `incr`, merely by the names chosen, is clearly suggesting the purpose of counting how many times `incr` has been called. Of course, no code running on finite hardware can serve this purpose in all cases. Our understanding of `incr` as serving a purpose must therefore be qualified by its hazards, the cases under which we may no longer assume it will serve this purpose, such as if it is called too many times, or if it is called from two threads that might interleave.[2]

In this case, these hazards define preconditions [Hoa69, Mey92], since `incr`'s clients can choose not to call it too many times, and to always call it from the same thread. Both preconditions are an obligation that all of its clients must obey in order for `incr` to correctly serve its purpose for any of them. We say `incr` is *cooperatively correct* since it must serve each of its clients correctly only under the assumption that all of its clients obey `incr`'s preconditions. The continued proper functioning of any of `incr`'s clients is vulnerable to `incr`'s misbehavior. Since this misbehavior can be induced by any of `incr`'s other clients,

[2] Of course, different programmers may use `incr` for different purposes, and different purposes will imply different hazards.

they are also vulnerable to each other's misbehavior.

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 successfully compose. Fortunately, memory-safe languages with encapsulation make practical a higher standard of robustness.

5.5 Defensive Correctness

If a user browsing pages from a webserver were able to cause it to display incorrect pages 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 continues to provide correct behavior to well behaved clients despite arbitrary behavior on the part of its other clients. Before this definition can be useful, we need to pin down what we mean by "arbitrary" behavior.

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 is allowed this assumption. 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 is cooperatively correct and 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. By not relying on its clients, R enables them to avoid relying on each other.

5.6 Defensive Consistency

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 properties are feasible at different granularities. Many conventional operating systems attempt to provide support for protecting users from each other's misbehavior. 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." As with our earlier C example, they are each a central point of fail-

```
public class Counter {
    private int count = 0;
    public synchronized int incr() {
        if ((count + 1) < 0) {
            throw ...;
        }
        return count += 1;
    }
}
```

Figure 5.2: A Defensively Consistent Counter in Java. Instances of this counter either correctly serve their purpose—reporting how many times `incr` was called—or they fail safe.

ure for all.[3] Some operating system designs [DH65] support process-granularity defensive consistency. Others, by providing principled controls over computational resource rights [Har85, SSF99], support process-granularity resistance to denial of service attacks. Among machines distributed over today's Internet, cryptographic protocols help support defensive consistency, but defensive progress, and hence defensive correctness, remains infeasible.

Because they enforce some isolation of access to memory, memory-safe languages can aspire to object-granularity defensive consistency. In E, as in most of these, fundamental operations implicitly allocate memory, rendering object-granularity defensive progress unattainable. For example, any "proof" that an ML function terminates with a correct answer is subject to the implicit caveat "given enough memory," which is a condition that any other ML function in that same address space can cause to be false.

Similarly, within E's architecture, object-granularity defensive correctness is impossible. E objects are aggregated into process-like units called *vats*, explained in Chapter 7 and Section 14.1. Like a process, a vat is the minimum granularity to which resource controls could be practically enforced. With respect to progress, all objects within 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.

When a system is composed of defensively consistent abstractions, to a good approximation, corruption is contagious only upstream, from providers to clients [Tin92].

5.7 A Practical Standard for Defensive Programming

Programmers use programming standards as conventions to organize background assumptions—those things we generally assume to be true, and endeavor to make true, unless stated otherwise. Programming standards should compose well: It should be easier to compose systems that meet a given standard when relying on subsystems built to meet this same standard. This is true for each of the forms of robustness above. For example,

[3] In Part IV we explain Polaris, an unconventional way to use conventional operating systems to provide greater security.

it is easier to compose defensively correct systems if one can build on defensively correct components.

By convention, unless stated otherwise, E and its libraries are engineered to achieve and support the following forms of robustness. The examples and mechanisms presented in this dissertation should be judged by this standard as well.

Object-granularity defensive consistency both within a vat and when interacting with potentially misbehaving remote vats.

Least authority The disciplines of defensive consistency and POLA are mutually supportive. Reducing the authority held by an object's clients reduces the number of actions it may take, and thereby helps reduce the number of cases that the object must protect against in order to remain consistent. A concrete example appears in Chapter 15. Defensive consistency itself helps reduce authority, since otherwise, each of an object's clients has the authority to corrupt any of the object's other clients.

Defensive progress up to resource exhaustion, where we include non-termination, such as an infinite loop, as a form of resource exhaustion. Protocols that achieve only defensive progress up to resource exhaustion are normally regarded as satisfying a meaningful liveness requirement. Whether this standard is *usefully* stricter than cooperative progress we leave to the judgement of the reader.

Fail safe When a component is unable to fulfill its purpose, it must prevent normal progress on control-flow paths that assumed its success.

Fail stop When possible, a failed component should stop and notify interested parties that it has failed, so they can attempt to cope. The halting problem prevents us from doing this in general. Although timeouts can make fail-safe into fail-stop, they must be used rarely and with care, as they also non-deterministically cause some slow successes to fail-stop.

Figure 5.2 (p. 33) shows one way to write a `Counter` class in Java that lives up to this standard, but only if we regard integer overflow as a kind of resource exhaustion. In order to regard it as defensively consistent, we regard exceptional control flow and process termination as forms of non-progress rather than incorrect service. This perspective places a consistency burden on callers: If they make a call while their own state invariants are violated, they must ensure that exceptions thrown by the callee leave reachable state in a consistent form. They can do so either by abandoning or repairing bad state.

Much bad state will automatically be abandoned by becoming unreachable as the exception propagates. For the remainder, the caller either can repair bad state (for example, using a `try/finally` block) or, (when this is too dangerous or difficult) can convert exceptional control flow into process termination, forcibly abandoning bad state. In E, the appropriate unit of termination is the vat incarnation rather than the process. As we will see in Section 17.4, terminating an incarnation of a persistent vat is similar to a transaction abort—it causes the vat to roll back to a previous assumed-good state.

Full defensive correctness is infeasible in E even between directly communicating vats. An area for future research is to investigate whether per-client intermediate vats can guard an encapsulated shared vat in order to practically resist denial of service attacks [Sch06].

Fault tolerant systems are often built from fail-stop components combined with redundancy or restart [Gra86, Arm03]. "Tweak Islands" [SRRK05], covered in Related Work Section 26.8, combines an E-like model with fault tolerant replication in order to provide high availability.

For some systems, continued progress, even at the cost of inconsistency, is more important than maintaining consistency at the cost of progress. We may loosely characterize these two needs as *best-effort* vs. *fail-safe*. For yet others, the price of full correctness is worth paying, as neither consistency nor progress may be sacrificed. E is designed to support fail-safe engineering. For these other purposes, E is inappropriate.

5.8 Notes on Related Work

Cliff Jones' *The Early Search for Tractable Ways of Reasoning about Programs* [Jon03] is a clear and readable history of formal verification, mostly of sequential imperative programs, up to 1990. This history makes clear the seminal contribution of Hoare's *An Axiomatic Basis for Computer Programming* [Hoa69]. In the Floyd-Hoare proof technique, program statements are annotated with preconditions and postconditions. The basic proof step is to show that, if the preconditions hold before the statement is executed, then the postconditions will hold after the statement is executed. Compositions are correct when the postconditions of all immediate antecedents of a statement imply that statement's preconditions.

In this sense, Hoare's preconditions formalize the assumptions each statement may rely on, and its postconditions formalize the specification it must meet under the assumption that its preconditions were met. Hoare's compositional rule formalizes conventional correctness (or, at best, cooperative correctness): when analyzing the correctness of a particular statement, one assumes that all antecedent statements met their specifications—that if their preconditions were met, then they met their postconditions.

Dijkstra's *A Discipline of Programming* [Dij76], by introducing "weakest preconditions," extended Floyd-Hoare logic to deal with liveness as well as safety issues. In the correctness literature, a thrown exception is generally treated as a safety issue, rather than a loss of liveness. Therefore, this dissertation uses the terminology of consistency *vs.* progress, so that we may regard a thrown exception as a loss of progress without causing confusion.

The precondition/postcondition notion was extended by Stark to "rely/guarantee," for reasoning about shared-state concurrency [Sta85]. We borrow the term "rely" from him. It was separately extended by Jones [Jon83] and Abadi and Lamport [AL93, AL95] to "assumption/commitment," for reasoning about message-passing concurrency. These two extensions were generalized and unified by Cau and Collette into a compositional technique for reasoning about both kinds of concurrency [CC96]. An area of future work is to explore how these various extensions relate to the concurrency work presented in this dissertation.

Abadi and Lamport's *Composing Specifications* [AL93] explicitly states the assumption we have termed "conventional correctness," to whit: "The fundamental problem of composing specifications is to prove that a composite system satisfies its specification if all its components satisfy their specification." The paper proceeds to show how to handle the resulting circular proof obligations. Although defensive consistency helps decouple specifications, these techniques are still relevant, so long as there may be reliance cycles.

Mario Tinto's *The Design and Evaluation of INFOSEC Systems: The Computer Secu-*

rity Contribution to the Composition Discussion [Tin92] introduced the concept we have termed "rely" as "depends on." Tinto's admonition to avoid "circular dependencies," *i.e.*, circular reliance relationships, implies but is stronger than our notion of defensive consistency. The two are alike only when the client-provider graph is acyclic. Tinto's "primitivity" corresponds to our notion of platform risk, which we both assume is acyclic. Acyclic reliance relationships would avoid the need to resolve cyclic proof obligations.

Both Hoare and Dijkstra show preconditions and postconditions applied mainly at statement granularity. Liskov and Guttag's *Abstraction and Specification in Program Developement* [LG86] shows the synergy between this form of reasoning and the abstraction provided by abstract data types. Bertrand Meyer's *Applying "Design by Contract"* [Mey88, Mey92] shows the further synergy when applied to object interfaces designed for reuse and polymorphism. The preconditions, postconditions, and invariants of Meyer's contracts are predicates written in the programming language, to be checked at runtime, sacrificing static checkability to gain expressive power. Soft types [CF91] and higher-order contracts [FF02] extend this technique.

When *all* the preconditions of an interface contract can be successfully checked at runtime, defensive consistency can often be achieved simply by considering these checks to be non-optional parts of the program's logic for doing *input validation*, rejecting (with a thrown exception) all inputs that otherwise would have caused a precondition violation. If clients cannot cause a precondition violation, then the provider's obligations to meet its postconditions is not conditional on its client's behavior. Note that the resulting robustness exceeds defensive consistency, since a defensively consistent provider may give incorrect service to ill behaved clients.

The "arbitrary behaviors" we are concerned about, since they include both accident and malice, are conventionally classified as "Byzantine faults" [LSP82]. However, we avoid that terminology because it is invariably associated with the problem of Byzantine fault tolerance: How to use redundant, unreliable, and possibly malicious replicas to do a single well-characterized job reliably [LSP82, CL02]. If all one's eggs are in one basket, one should employ multiple guards to watch that basket and each other, so that an adequate number of honest guards can *mask* the misbehaviors of the other guards. This reduces the likelihood that *any* eggs will be lost. By contrast, the forms of defensiveness explained in this dissertation help *isolate* Byzantine faults, reducing the likelihood that *all* eggs will be lost. The two approaches are complementary. Ideally, robust systems should employ both. We return to this topic briefly in Related Work Sections 25.1, 25.2, and 26.8.

Defensive correctness and defensive consistency have been practiced within several communities, such as Key Logic, Inc., where much of our modern understanding of these practices originates. However, so far as we are aware, these communities explained only the designs of systems built according to these design rules [Har85, Lan92, Raj89, Key86], while these design rules themselves remained unwritten lore.

We chose the term "defensive" as homage to the widespread notion of "defensive programming." Defensive programming includes design heuristics for minimizing dependencies and for using assertions to check assumptions at runtime. However, we have been unable to find any crisp statement of design rules for characterizing kinds of defensive programming. Even without crisp design rules, pervasive practice of such ad hoc defensiveness helps systems *fail fast* following an accidental inconsistency. Jim Gray's *Why Do Computers Stop and What Can Be Done About It?* [Gra86] explains how to leverage such fail-fast components for robustness against software bugs. However, it is unclear how much protection such ad hoc defensiveness provides against malicious inconsistencies.

To give a more plausible account of how programmers informally reason about conformance to informal (and often inarticulate) specifications, we have divided specifications into purposes and hazards. Our notion of purpose derives from Daniel Dennett's notion of *intent* in *The Intentional Stance* [Den87]. As Dennett emphasizes, an artifact's purpose is not an objective property, but rather, a descriptive stance employed by observers for their own purposes, and so on. Sex provides an example of how hazards depend on purposes: When the purpose of sex is pleasure, pregnancy may be regarded as a hazard. To that person's genes, the purpose of sex is reproduction and condoms are a hazard. Their plans conflict, not because of inconsistent assumptions, but because of conflicting intents.

Chapter 6

A Taste of E

Like most "new" languages, little about E is actually novel. Most of its elements are drawn from prior language designs. The specific novel contribution of this dissertation is the state transition semantics explained in Part III and shown in Figure 17.1 (p. 124). The value of this contribution lies in its integration with other language features, so it is important to present a cohesive framework. In this dissertation, E serves as that framework.

This chapter briefly explains a subset of E as a conventional sequential object language. This dissertation will introduce other aspects of E as they become relevant. For a more complete explanation of E, see [Sti04].

6.1 From Functions to Objects

Object computation can be understood as the sum of three elements [GK76, HBS73]:

Objects == Lambda Abstraction + Message Dispatch + Local Side Effects

The remaining feature often thought to be defining of object-oriented programming is inheritance [Weg87]. Though we do not view inheritance as a fundamental ingredient of object computation, its widespread use in object-oriented programming practice motivates its inclusion in E. However, E's reconciliation of inheritance with capability security principles [Mil04] is beyond our present scope.

6.1.1 Lambda Abstraction

The call-by-value lambda calculus [Plo75] is a pure theory of nested function definition and application. Figure 6.1 shows nested function definition, which E shares with all lexically scoped lambda languages including ALGOL60 [NBB$^+$63], Scheme [KCR98], and ML [Mil84]. The call to `makeAddr` returns a function, an instance of the `adder` function definition expression, that adds `3` to its argument. Church originally thought about this as substitution—return an `adder` function in which `x` has been replaced by `3` [Chu41]. Unfortunately, this simple perspective makes it awkward to explain side effects. An alternative perspective is to consider a function, such as that held in the `addThree` variable, to be a combination of *code* describing behavior (the static code for `adder`), and *state* (the runtime bindings for its free variables).[1] `x` in `adder` is a free variable in that `adder` uses `x`, but the

[1] Smalltalk refers to this pair as "behavior" and "state" [GK76]. Actors refers to this pair as "script" and "acquaintances" [HBS73].

```
def makeAddr(x) {
    def adder(y) {
        return x + y
    }
    return adder
}

? def addThree := makeAddr(3)
# value: <adder>

? addThree(5)
# value: 8
```

Figure 6.1: Lexically Nested Function Definition. `makeAddr` defines and returns an `addr` function, an instance of the `adder` code appearing in its function definition expression. The instance variables of each `adder` are the variables used freely in its code (here, `x`), which must therefore be bound in its creation context.

corresponding definition of `x` is from `adder`'s creation context. These are commonly referred to as *instance variables.*

6.1.2 Adding Message Dispatch

The most visible difference between a function and an object is that a function's behavior is written to satisfy just one kind of request, and all calls on that function are forms of that one request. By contrast, an object's behavior enables it to satisfy a variety of different requests (each with a separate *method*). A request to an object (a *message*) identifies which of these requests is being made. This is not a fundamental distinction—either functions or objects can be trivially built from the other in a variety of ways. As we will see, in E, objects are the more primitive notions, of which functions are defined as a degenerate case.

Figure 6.2 shows the `makePoint` function which makes and returns `point` objects. From a lambda calculus perspective, `makePoint` is like `makeAddr`—it is a lexically enclosing function that defines the variable bindings used by the object it both defines and returns. From an object perspective, `makePoint` is simultaneously like a class and constructor—both defining the instance variables for points, and creating, initializing, and returning individual points.

The returned points are clearly object-like rather than function-like. Each point's behavior contains three methods: `getX`, `getY`, and `add`. Every request to a point names which of these services it is requesting.

Some shorthands in this code need a brief explanation.

- "`a + b`" is merely syntactic shorthand for "`a.add(b)`", and similarly for other expression operators. As in Smalltalk, all values, including integers, are objects. Addition happens by asking one integer to add another integer. (As E's robustness convention demands, this operation either returns the correct sum or visibly fails due to resource exhaustion.)

```
def makePoint(x,y) {
    def point {
        to getX()     { return x }
        to getY()     { return y }
        to add(other) {
            return makePoint(x + other.getX(), y + other.getY())
        }
    }
    return point
}

? def p := makePoint(3,5)
# value: <point>

? p.getX()
# value: 3

? (p + makePoint(4,8)).getX()
# value: 7
```

Figure 6.2: Objects as Closures. The expression defining point is an object definition expression. This is like a lambda expression and a variable definition. It evaluates to a closure, and binds the point variable to that closure. This closure has a single implicit parameter: a message consisting of a message name and a list of arguments. The closure implicitly dispatches on the message name and the number of arguments to select a method.

```
def makeAddr {
    to run(x) {
        def adder {
            to run(y) {
                return x.add(y)
            }
        }
        return adder
    }
}
```

Figure 6.3: Expansion to Kernel-E. Expanding the `makeAddr` function to Kernel-E reveals that E's "functions" are really objects with a single `run` method. In E, all values are ultimately objects, and all invocation is by message passing. So-called functions are just a syntactic shorthand.

- Functions are simply one-method objects where the method is named "`run`". The previous `makeAddr` is therefore just syntactic shorthand for the code shown in Figure 6.3.
- Likewise, the function call syntax "`makeAddr(3)`" is just shorthand for the method call "`makeAddr.run(3)`".

Once all the syntactic shorthands of an E program have been expanded away, we have a *Kernel-E* program, expressed in the subset of the E language without such shorthands.

6.1.3 Adding Side Effects

Starting with lambda calculus (or with lambda plus message dispatch), there are many ways to add side effects. The approach used by E, Scheme, ML and many other lambda languages is to introduce assignment. In E, variables are non-assignable (or "`final`") by default. For a variable to be assignable, it must be declared with "`var`" as shown in Figure 6.4.

Under the covers, each assignable variable is actually a distinct primitive variable-object, referred to as a *Slot*. By declaring `var` *`count`*, we introduce a binding from the name `"&count"` to a new mutable Slot object holding the current count. Writing a use-occurrence of `count` in an expression is like writing `(&count).getValue()`, asking the Slot for its current value. Writing `count := expr` is like writing `(&count).setValue(expr)`, asking the Slot to remember the argument as its new current value. Slots serve the same purpose in E as do so-called "Ref"s in Algol-68 and ML.

6.2 Composites and Facets

Because assignable variables are reified as Slot objects, they can be easily shared, as shown in Figure 6.5.

Each time `makeCounterPair` is called, it defines a new Slot and two objects that use this Slot. It then returns a list of the latter two objects, one that will increment and return the value of this variable and one that will decrement and return it. This is a trivial example

```
def makeCounter() {
    var count := 0
    def counter {
        to incr() { return count += 1 }
    }
    return counter
}

? def carol := makeCounter()
# value: <counter>

? carol.incr()
# value: 1
```

Figure 6.4: A Counter in E. Variables in E are, by default, unassignable. The `var` keyword makes `count` assignable, which is to say, it binds a new mutable Slot object to `"&count"`.

```
def makeCounterPair() {
    var count := 0
    def upCounter {
        to incr() { return count += 1 }
    }
    def downCounter {
        to decr() { return count -= 1 }
    }
    return [upCounter, downCounter]
}

? def [u, d] := makeCounterPair()
# value: [<upCounter>, <downCounter>]

? u.incr()
# value: 1

? u.incr()
# value: 2

? d.incr()
# problem: <NoSuchMethodException: <a downCounter>.incr/0>

? d.decr()
# value: 1
```

Figure 6.5: Two Counting Facets Sharing a Slot. `makeCounterPair` makes a Slot, and it makes and returns two objects sharing access to that Slot. These three objects form a *lexical composite*. The returned objects are facets of this composite. The Slot is encapsulated within this composite.

of a useful technique—defining several objects in the same scope, each providing different operations for manipulating a common state held in that scope.

For compactness of description, we often aggregate a set of objects into a *composite*. The three objects created together by each call to `makeCounterPair` form a *lexical composite*. Like an individual object, a composite is a combination of state and behavior. Like an individual object, the state consists of all of the variables within the composite. The behavior consists of all of the code within the composite, but here we have an important difference.

The behavior elicited by a message to the composite depends both on the message and on which object of the composite receives the message. Objects which may be referred to from outside the composite, like `upCounter` and `downCounter`—are *facets* of the composite. In this case, the Slot bound to `&count` need not be considered a facet since we can tell that no reference to it can escape from the composite. This demonstrates how the reference passing rules of objects, combined with encapsulation of object state, supports encapsulation of objects within composites.

The aggregation of a network of objects into a composite is purely subjective—it allows us to hide detail when we wish. The technique works because the possible interactions among disjoint composites obey the same rules as the possible interactions among individual objects.

6.3 Soft Type Checking

Explicit type systems help ensure compatibility between the behavior a client assumes and the behavior its provider implements. Static type systems help catch such assumption mismatches during development, but with two costs in expressive power: 1) Dynamically safe code that can't be statically checked is rejected. 2) The only kinds of mismatches that can be caught are those that can be checked for statically.

By default, E provides only implicit types and dynamic type safety, which catch interface assumption mismatches at the last moment: when a message is not understood by a recipient. This still fails safe with no loss of expressive power of the first kind. E does provide optional explicit soft types [CF91], trademarks [Mor73a, Mor73b], and auditors [YM03] for catching assumption mismatches earlier, though still at runtime. Variable declarations and method returns can be annotated by a colon followed by an expression which evaluates to a "guard." At runtime, a *guard* determines what values may pass, with no loss of expressive power of the second kind.

Object definition expressions can be annotated by an `implements` keyword followed by a list of expressions which evaluate to "auditors." At runtime, each *auditor* is given the abstract syntax tree of this object definition expression, to determine whether its instances would indeed implement the property the auditor represents. If the auditor approves, the new instances carry its trademark. This forms the basis for a dynamically extensible code verification system. This dissertation employs only two kinds of auditors, stamps and `Data`, which we explain here. Further explanation of the auditing framework is beyond the scope of this dissertation.

The `interface...guards...` expression defines a (guard, auditor) pair representing a new trademark. The auditor of this pair is a *stamp*, a rubber stamping auditor which approves any object definition expression it is asked to audit, thereby stamping these instances

```
interface TPoint guards TPointStamp {
    to getX() :int
    to getY() :int
    to add(other :TPoint) :TPoint
}

def makeTPoint(x :int, y :(0..x)) :TPoint {
    def tPoint implements TPointStamp {
        to getX() :int { return x }
        to getY() :int { return y }
        to add(other :TPoint) :TPoint {
            return makeTPoint(x + other.getX(), y + other.getY())
        }
    }
    return tPoint
}

? def p :TPoint := makePoint(3,5)
# problem: Not audited by TPoint

? def p :TPoint := makeTPoint(3,5)
# problem: 5 is not in the region 0..!4

? def p :TPoint := makeTPoint(3,2)
# value: <tPoint>
```

Figure 6.6: Soft Type Checking. Optional soft type checking can help ensure compatibility between the behavior a client assumes and the one its provider implements. The guard (after the colon) is asked to approve a value. Disapprovals occur at runtime, which is later than one might like, but the checks may thus use the full expressive power of the programming language.

with its trademark. The corresponding guard allows only objects carrying this trademark to pass. For example, Figure 6.6 shows `makeTPoint`, a variant of `makePoint` which uses guards to ensure that its `x` is an integer and its `y` is between 0 and `x`. It stamps all the points it makes with `TPointStamp` so that they pass the `TPoint` guard.

"`Data`" is both a guard and an auditor. As a guard, the values it passes convey only information, not authority of any kind. Among the conditions an object must satisfy to be considered data, it must be transitively immutable. As an auditor, `Data` only approves object definition expressions whose instances obey these constraints.[2] An identical copy of a data object is indistinguishable from the original. When passed as an argument in remote messages, data objects are passed by copy.

Soft type checking need not be expensive. Notice that compilers may transparently optimize `2 + 3` by performing the addition at compile time. Likewise, when a compiler can determine, by static type inference, that a guard will always pass, it may generate code without an unnecessary runtime check. A deterministic auditing check which uses only static information, like the abstract syntax tree of its object definition expression, can be memoized, so it only needs to occur once per expression. Current E implementations do not yet implement these optimizations.

6.4 Notes on Related Work

Reynold's Gedanken [Rey70], a dynamically-typed variant of Algol, seems to be the first language with first-class indefinite extent lexical closures. Gedanken introduced the trademarking technique explained above [Mor73a, Mor73b]. Gedanken was pathbreaking regarding many of the issues covered in this dissertation, and is covered further in Related Work Section 24.1.

E's technique for defining objects by nested lambda instantiation + message dispatch dates back at least to Hewitt's 1973 Actors languages [HBS73], which combined Gedanken's lexical closures with Smalltalk's method dispatch. Actors is covered further in Chapter 23. This notion of object is also hinted at in Hoare's note on "Record Handling" [Hoa65]. Shroff and Smith's *Type Inference for First-Class Messages with Match-Functions* [SS04] explains how to statically type objects defined by this technique. We have not yet applied this or any other static type analysis techniques to E programs.

Many prior systems have provided functions as degenerate objects. In Smalltalk [GK76], blocks are a lightweight syntax for defining function-like one-method objects, in which the default method name is elided. The T language [RA82] is a dialect of Scheme in which all closures are objects, and function-invocation syntax invokes one of an object's methods. A C++ value type can overload "`operator()`" in order to turn function application into method invocation [ES90].

Imperative programming language semantics often describe assignable locations as discrete location objects [KCR98], but this notion is not usually made accessible to the programmer. The first programmer-accessible reification of locations seems to be the so-called "ref" type constructor of Algol 68 [vWMPK69]. This notion is widely familiar today as the "ref" of ML [Mil84].

Any imperative language with lexically nested object definitions, including Smalltalk's

[2] Only the auditing framework in "E-on-CL," the Common LISP implementation of E [Rei05], is currently able to support auditors such as `Data`.

blocks, T [RA82], Beta [Mad00], Java's inner classes, and Emerald [HRB^{+}87] covered briefly in Related Work Section 24.6, supports the easy definition of multiple facets sharing state.

Soft types [CF91] and higher order contracts [FF02] have been explored largely in the context of the Scheme language.

The assurance provided by E's `Data` auditor—that any object branded with the `Data` trademark is transitively immutable—is similar to Anita Jones' notion of "memoryless procedures" [Jon73].

Chapter 7

A Taste of Pluribus

7.1 Pointer Safety

Within a single address space, references are typically implemented as memory addresses. So-called "dynamic type safety" consists largely of *pointer safety*, enforced by safe-language techniques made popular by Java, but going back to LISP 1.5. Safe pointers are unforgeable. Distributed object protocols [Don76, HRB$^+$87, Vin97, WRW96, BNOW94, vDABW96, LS83] stretch the reference graph between address spaces. Can pointer safety be preserved over the network?

A reference provides a kind of communication channel between objects, conveying messages from client to provider. We can explain the attributes of pointer safety in terms of security properties often desired of other communication channels: integrity (no tampering), confidentiality (no spying), unforgeability (authorization), and unspoofability (authentication). Java references have all these attributes; C++ pointers have none.

Integrity The message Alice sent on an Alice-to-Bob reference must be the message Bob receives. In an object context, we generalize this requirement to include both the data and the references carried by the message. Both must arrive unaltered.

Confidentiality Only Bob receives the message that Alice sent.

Unforgeability Even if Bob knows Carol's memory address, Bob cannot synthesize a new reference to Carol. Without a genuine reference to Carol, Bob cannot deliver messages to Carol.

Unspoofability Even if Charlie knows Carol's memory address, he cannot cause Bob's reference-to-Carol to deliver messages to him rather than Carol.

This chapter explains how Pluribus, E's distributed object protocol, enforces pointer safety among objects residing on mutually defensive platforms. (See [Don76, TMvR86, SJR86, vDABW96, Clo04c] for similar systems.) Unsurprisingly, since pointer safety resembles the attributes of so-called "secure socket layer" protocols like SSL and TLS [DA99], we build on such a protocol and let it do most of the hard work. Because of differences in our authentication requirements, we avoid SSL's vulnerability to certificate authorities. 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. Like SSL, Pluribus does not protect against denial of service or traffic analysis.

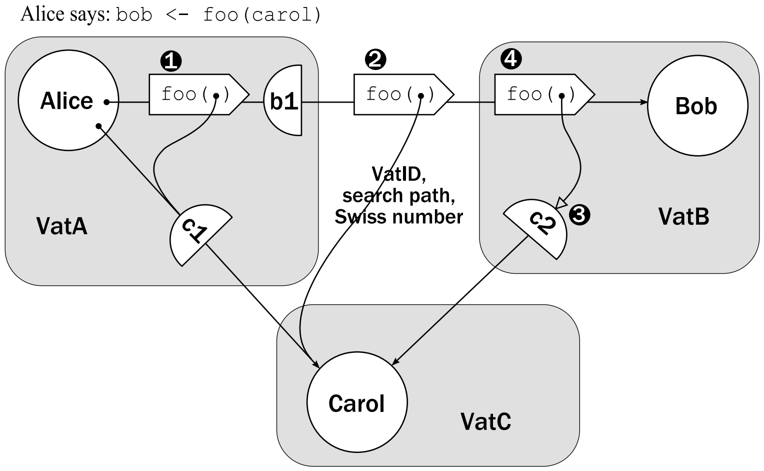


Figure 7.1: Distributed Introduction. At the distributed-object level of abstraction, Alice has a remote reference to Bob. At a lower level of abstraction, Alice has a reference to b1, VatA's local proxy for Bob. When Alice passes Bob a reference to Carol ❶, she is actually passing b1 a reference to c1, which b1 serializes into the network message ❷. VatB unserializes to obtain its own c2 proxy for Carol ❸, which it includes in the message sent to Bob ❹.

We present here a simplified form of Pluribus which ignores concurrency issues. Pointer equality places further requirements on unforgeability and unspoofability which we disregard here, but return to briefly in Section 19.5.

7.2 Distributed Objects

Objects are aggregated into units called *vats*. Each E object exists in exactly one vat. We speak of an object being *hosted* by its vat. A vat typically hosts many objects. A vat is a platform for the objects it hosts, and so is a central point of failure for them. Similarly, each vat runs on one machine platform at a time, but a machine may host many vats. A good first approximation is to think of a vat as a process full of objects—an address space full of objects plus a thread of control. Unlike a typical operating system process, a vat persists (that is, its state is saved to persistent storage periodically in case its process crashes). A process running a vat is an *incarnation* of a vat. The vat maintains its identity and state as it passes serially through a sequence of incarnations. We revisit persistence in Section 17.4.

To enable objects in separate vats to send messages to each other, we must bridge from the world of address-space-based programming language technology to the world of network communication protocols. Our first step is conventional: each vat contains a communication system allowing it to make connections to, and accept connections from, other vats. Each vat's communication system contains *proxy* objects [Sha86] shown as half circles in Figure 7.1. When an object in a vat holds a remote reference to an object in a different vat, it actually holds a reference to a local proxy, which is the local representative of the remote object. When proxy (b1) is sent a local message ❶, b1 serializes the message arguments (c1) into a packet which it sends out as a network message ❷. When VatB receives the network message, it unserializes it into a message local to VatB, initiating a handshake with remote vats (VatC) as necessary to create the needed proxies (c2) ❸. The unserialized message is finally delivered to Bob ❹.

The description so far applies equally well to many distributed object systems, such as CORBA [Vin97] and RMI [WRW96], that provide pointer safety only among mutually reliant processes on mutually reliant machines. This is true even when these systems use SSL as their transport layer. An SSL-like transport helps, but a bit more is needed.

7.3 Distributed Pointer Safety

On creation, each vat generates a public/private key pair. The fingerprint of the vat's public key is its vat identity, or *VatID*. What does the VatID identify? The VatID can only be said to designate any vat which knows and uses the corresponding private key apparently according to the protocol.

We care first about inductive correctness. Assuming secure references from Alice to Carol and from Alice to Bob, how does Pluribus deliver to Bob a similarly secure reference from Bob to Carol? Alice can talk to Carol only because VatA can talk to VatC. If there are any inter-vat references between objects on VatA and VatC, then there must be exactly one bidirectional SSL-like data pipe between these vats. All references between this pair of vats are multiplexed over this pipe. Our initial state implies the existence of an SSL-like pipe between VatA and VatC and one between VatA and VatB.

When VatC first exported across the vat boundary a reference to Carol, VatC generated an unguessable randomly chosen number to represent Carol. We call this a *Swiss number*, since it has the knowledge-grants-access logic popularly attributed to Swiss bank account numbers. Like a safe pointer, 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. (The "sparse capabilities" of Amoeba [TMvR86] are based on this same observation.) When VatA first received this reference, VatA thereby came to know VatC's VatID, VatC's search path, and Carol's Swiss number. A *search path* is a list of TCP/IP location hints to seed the search for a vat that can authenticate against this fingerprint [Clo04b].

When Alice sends Bob a reference to Carol, VatA tells VatB this same triple: VatC's VatID, VatC's search path, and Carol's Swiss number. In order to obtain its own remote reference to Carol, VatB uses the search path to contact candidate VatCs. It asks each alleged VatC for its public key which it verifies against VatC's VatID. The handshake logic proceeds along the lines of SSL: VatC proves her knowledge of the corresponding private key, then Diffie-Hellman key agreement leads to a shared session key for the duration of the inter-vat connection.[1] *Only* once an authenticated, secure data pipe is set up between them does VatB reveal Carol's Swiss number to VatC. Later, when Bob sends a message to Carol over this reference, c2 serializes it and sends it to VatC, asking VatC to deliver it to the object associated with Carol's Swiss number.

This covers the inductive case, where the security of new online connections leverages the presumed security of existing online connections. What of the base case?

[1] VatB does not delay the delivery of the `foo` message to Bob until VatC is found. Rather, it first makes c2 as a *remote promise* for Carol, and proceeds to send the message to Bob with c2 as the argument. Should a valid VatC be found, this promise will resolve to a *far reference*. Promises and far references are explained in Chapter 16.

7.4 Bootstrapping Initial Connectivity

A newly launched VatC has no online connections. An (adequately authorized) object on VatC with access to Carol can ask VatC for a "`captp://...`" URI string encoding the same triple used above: VatC's VatID, VatC's search path, and Carol's Swiss number. It can then write this string on out-of-band media such as the screen or a file. This string then needs to be conveyed securely to some location where an (adequately authorized) object in VatA can read it, such as Alice.

For example, the human operator of VatC might read the URI from his screen over the phone to the human operator of VatA, who might type it into a prompt feeding Alice. Or perhaps the string is sent by PGP email. For initial connectivity, the responsibility for conveying these strings securely must reside with external agents. Pluribus, like all cryptographic systems, must bootstrap their secure deployment by conveying cryptographic information over prior media.

Once Alice has this URI string, she can request a corresponding online reference, setting in motion the same search and handshake logic explained above. These URI strings are a form of "offline capability." Provided that they are conveyed securely, they satisfy all the pointer safety requirements. Section 17.3 explains how offline capabilities can also be used to reestablish connectivity following a partition.

7.5 No Central Points of Failure

A reference is an arrow, and an arrow has two ends. There is an imposter problem in each direction. The VatID ensures that the vat Bob is speaking to is the one hosting the object Alice meant to introduce him to—making spoofing infeasible. The Swiss number ensures that the entity permitted to speak to Carol is the one Alice chose to enable to do so—making forging infeasible. Since Bob does not care what name Alice uses for Carol, pointer safety requires no external naming authorities. By using key-centric designation, we avoid widely shared namespaces of name-centric systems, and the centralized vulnerabilities they create [Ell96, EFL$^+$99, Clo04a, Sti05, Fro03].

Pluribus enforces inter-vat pointer safety so that objects can rely on it between vats and therefore between machines. Even if VatB is run on a corrupted operating system or tampered hardware, or if it runs its objects in an unsafe language like C++, Alice could still view it as a set of colluding objects running on a properly functioning vat. If Betty, another C++ object in VatB, steals Bob's reference to Carol, from Alice's perspective this is equivalent to Bob colluding with Betty.

A misbehaving vat can misuse any of the remote references given to any of the objects it hosts, but no more. This is also true of a colluding set of objects running on a correct vat. If Alice's programmer wishes to defend against VatB's possible misbehavior, she can model VatB as a monolithic composite, and all its externally accessible objects (such as Bob and Betty) as facets of this composite. Alice's programmer can, without loss of generality, reason as if she is suspicious only of objects.

Alternatively, if Alice relies on Bob, then Alice also relies on VatB since Bob relies on VatB. Anything which relies on Alice therefore also relies on VatA and VatB. Such multi-vat vulnerability relationships will be many and varied. Pluribus supports mutual defensiveness between vats, not because we think inter-vat vulnerabilities can always be avoided, but in order to enable distributed application designers to choose their vulnerabilities.

Different participants in the reference graph will aggregate it differently according to their own subjective ignorance or suspicions, as we have seen, or merely their lack of interest in making finer distinctions. The reference graph supports the economy of aggregation and the necessary subjectivity in deciding how to aggregate.

7.6 Notes on Related Work

Programming Languages for Distributed Computing Systems [BST89] is a survey touching on over two hundred distributed programming languages, including many distributed object languages. One of the earliest of these is also the most closely related: Hewitt's Actors, covered in Chapter 23, provides for transparently distributed asynchronous object computation.

The Actors "worker" is analogous to our "vat," and provides a communicating event-loop concurrency control model with many of the properties we explain in Part III. The Actors model provides the object-capability security we explain in Part II. The stated goals of Actors [Hew85] would seem to demand that Actors be distributed by a cryptographic protocol with the security properties of Pluribus. However, we are unaware of any distributed Actors design or implementation that has done so.

The cryptographic distributed capability protocols related to Pluribus include DCCS, explained in Related Work Section 25.3, Amoeba (Section 25.4), Sansom's protocol for securely distributing Mach (Section 25.5), DEC SRC's Secure Network Objects (Section 24.7), HP's Client Utility (Section 25.6), The Web Calculus (Section 26.1), and the Twisted Python Perspective Broker (Section 26.2).

Part II

Access Control

Chapter 8

Bounding Access Rights

Access control systems must be evaluated in part on how well they enable one to distribute the access rights needed for cooperation, while simultaneously limiting the propagation of rights which would create vulnerabilities. Analysis to date assumes access is controlled only by manipulating a system's protection state—the topology of the permission arcs that form the access graph. Because of the limitations of this analysis, capability systems have been "proven" unable to enforce some basic policies: selective revocation, confinement, and the *-properties (explained in subsequent chapters).

In practice, programmers build *access abstractions*—programs that help control access, extending the kinds of access control that can be expressed. Working in Dennis and van Horn's original capability model, we show how abstractions were used in actual capability systems to enforce the above policies. These simple, often tractable programs limit the rights of arbitrarily complex, untrusted programs. When analysis includes the possibility of access abstractions, the original capability model is shown to be stronger than is commonly supposed.

Whether to enable cooperation or to limit vulnerability, we care about *authority* rather than *permissions*. Permissions determine what actions an individual program may perform on objects it can directly access. Authority describes the effects that a program may cause on objects it can access, either directly by permission, or indirectly by permitted interactions with other programs. To understand authority, we must reason about the interaction of program behavior and the arrangement of permissions. Dennis and van Horn's 1965 paper, *Programming Semantics for Multiprogrammed Computations* [DH65] clearly suggests both the need and a basis for a unified semantic view of permissions and program behavior.

However, the security literature has reasoned about bounds on authority based only on the potential evolution of the topology of the access graph. This assumes all programs are hostile. While conservatively safe, this approach omits consideration of security-enforcing programs. In practice, programmers express and implement security policy partially by manipulating permissions and partially by writing programs. The omission of this practice from theoretical analysis has resulted in false negatives—mistaken infeasibility results—diverting attention from the possibility that an effective access control model has existed since 1965.

In this part, we offer a new look at the original capability model proposed by Dennis and van Horn [DH65]—here called *object-capabilities*. Our emphasis—which was also their emphasis—is on expressing policy by using abstraction to extend the expressiveness of object-capabilities. Using abstraction, object-capability practitioners have solved prob-

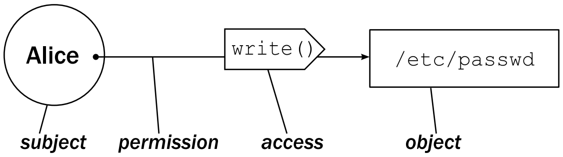


Figure 8.1: Access Diagrams Depict Protection State. Here, Alice has direct access to `/etc/passwd`, so she has *permission* to invoke any of its operations. She accesses the object, invoking its `write()` operation.

lems like selective revocation (withdrawing previously granted rights), overt[1] confinement (cooperatively isolating an untrusted subsystem), and the *-properties (enabling one-way communication between clearance levels). We show the logic of these solutions, using only functionality available in Dennis and van Horn's 1965 Supervisor [DH65], hereafter referred to as "DVH." In the process, we show that many policies that have been "proven" impossible are in fact straightforward.

8.1 Permission and Authority

When discussing any access control model, we must carefully distinguish between *permission* and *authority.* Our distinction between these is adapted from Bishop and Snyders distinction between de jure and de facto information transfer [BS79]. However, we are concerned here with leakage of access rights rather than leakage of information.

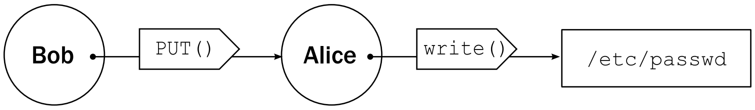
A direct access right to an *object* gives a *subject* the *permission* to invoke the behavior of that object [Lam74]. In Figure 8.1 Alice has direct access to `/etc/passwd`, so she has permission to invoke any of its operations. She accesses the object, invoking its `write()` operation.

By *subject* we mean the finest-grain unit of computation on a given system that may be given distinct direct access rights. Depending on the system, this could be anything from: all processes run by a given user account, all processes running a given program, an individual process, all instances of a given class, or an individual instance. To encourage anthropomorphism we use human names for subjects.

By *object*, we mean the finest-grain unit to which separate direct access rights may be provided, such as a file, a memory page, or another subject, depending on the system. Without loss of generality, we model restricted access to an object, such as read-only access to `/etc/passwd`, as simple access to another object whose behavior embodies the restriction, such as access to the read-only facet of `/etc/passwd` which responds only to queries.

Alice can directly write `/etc/passwd` by calling `write()` when the system's protection state says she has adequate permission. In Figure 8.2, Bob, who does not have permission, can indirectly write `/etc/passwd` so long as Alice writes there any text Bob requests her to. When Alice and Bob arrange this relying only on the "legal" overt rules of the system, we say Alice is providing Bob with an indirect access right to write `/etc/passwd`, that she is acting as his proxy, and that Bob thereby has *authority* to write it. Bob's authority derives

[1] Semantic models, specifications, and correct programs deal only in *overt* causation. Since this dissertation examines only models, not implementations, we ignore covert and side channels. In this dissertation, except where noted, the "overt" qualifier should be assumed.



	Bob	Alice	/etc/passwd
Bob	self	permission	authority
Alice		self	permission

Figure 8.2: Authority is the Ability to Cause Effects. If 1) Bob has permission to talk to Alice, 2) Alice has permission to write `/etc/passwd`, and 3) Alice chooses to write there any text Bob asks her to, then Bob has *authority* to write `/etc/passwd`.

from the structure of permissions (Alice's write permission, Bob's permission to talk to Alice), and from the behavior of subjects and objects on permitted causal pathways (Alice's proxying behavior). The thin black arrows in our access diagrams depict permissions. The filled-in portion of access matrices depict authority.

The *protection state* of a system is the structure of permissions at some instant in time, *i.e.*, the topology of the access graph [Lam74]. Whether Bob currently has permission to access `/etc/passwd` depends, by definition, only on the current topology of permissions, *i.e.*, on whether this permission is in the set of *current permissions* (CP), shown in Table 8.1. Whether Bob eventually gains permission (EP) depends on this topology and on the state and behavior of all subjects and objects that might cause Bob to be granted permission. We cannot generally predict if Bob will gain this permission, but conservative bounds can give us a reliable "no" or "maybe."

From a given system's *update rules*—rules governing permission to alter permissions—one might be able to calculate a bound on possible future permission topologies by reasoning only from the current topology.[2] This corresponds to Bishop and Snyder's "potential de jure analysis" [BS79], and gives us a *topology-only bound on permissions* (TP). With more knowledge, one can set tighter bounds. By taking into account the state and behavior of some subjects and objects in our reliance set, we may calculate a tighter *partially behavioral bound on permissions* (BP).

Bob's eventual authority to write `/etc/passwd` (EA) depends on the topology of permissions, and on the state and behavior of all subjects and objects on permitted causal pathways between Bob and `/etc/passwd`. One can derive a bound on possible overt causality by reasoning only from the current topology of permissions. This corresponds to Bishop and Snyder's "potential de facto analysis" [BS79], and gives us a *topology-only bound on authority* (TA). Likewise, by taking into account some of the state and behavior in our reliance set, we may calculate a tighter *partially behavioral bound on authority* (BA).

Systems have many levels of abstraction. At any moment our frame of reference is a boundary between a platform that creates rules and the subjects hosted on that platform, restricted to play by those rules. By definition, a platform manipulates only permissions. Subjects extend the expressiveness of a platform by building abstractions whose behaviors

[2] The Harrison Ruzzo Ullman paper [HRU75] is often misunderstood to say this calculation is never decidable. HRU actually says it is possible (indeed, depressingly easy) to design a set of update rules which are undecidable. At least three protection systems have been shown to be decidably safe [JLS76, SW00, MPSV00].

CP	Current Permissions	trivial	unsafe
EP	Eventual Permissions	intractable	unsafe
BP	Behavior-based bound on EP	interesting	unsafe
TP	Topology-based bound on EP	easy	unsafe
EA	Eventual Authority	intractable	safe
BA	Behavior-based bound on EA	interesting	safe
TA	Topology-based bound on EA	easy	safe

$$CP \subseteq EP \subseteq BP \subseteq TP$$
$$EP \subseteq EA$$
$$BP \subseteq BA$$
$$TP \subseteq TA$$
$$EA \subseteq BA \subseteq TA$$

Table 8.1: Bounds on Access Rights. Let us say a policy under consideration should be adopted if it would never allow Bob to access Carol. Given the current set of permission arcs, CP, it is trivial to check whether Bob may directly access Carol. But permissions-based analyses are unsafe, since they won't reveal the possibility of Bob obtaining indirect access. To decide, we'd need to know if Bob would eventually gain authority to access Carol, *i.e.*, whether Bob→Carol would be in the resulting EA.

It is intractible to calculate EA itself since it depends on program behavior. Fortunately, the absence of Bob→Carol from a tractable superset of EA, such as TA or BA, implies its absence from EA. If we consult a superset that is too large, such as TA, we'll reject as dangerous too many policies that are actually acceptable. BA is both safe and useful: It will never say a policy is acceptable when it is not, and when a policy is acceptable, it will often let us know. Among these choices, BA is the smallest safe tractable set.

express further kinds of limits on the authority provided to others. Taking this behavior into account, one can calculate usefully tighter bounds on authority. As our description ascends levels of abstraction [NBF$^+$80], the authority manipulated by the extensions of one level becomes the permissions manipulated by the primitives of the next higher platform. Permission is relative to a frame of reference. Authority is invariant. Section 11.3 and Part IV present examples of such shifts in descriptive level.

It is unclear whether Saltzer and Schroeder's *Principle of Least Privilege* [SS75] is best interpreted as "least permission" or "least authority." As we will see, there is an enormous difference between the two.

8.2 Notes on Related Work

Saltzer and Schroeder's *The Protection of Information in Computer Systems* [SS75] is a wonder of clear writing and careful analysis. The design rules and criteria they propose for the engineering of secure systems remain the yardstick of reference when discussing pros and cons of security architectures.

Lampson's *Protection* [Lam74] introduced the access matrix as a means of comparing and contrasting access control systems. Lampson's paper is often misinterpreted as claiming that the only difference between Access Control Lists and Capabilities is whether the access matrix is organized by rows or columns. Lampson's paper actually points out several ways in which these systems have different protection properties. Lampson's model was subsequently elaborated upon in Graham and Denning's *Protection — Principles and Practice* [GD72].

Harrison, Ruzzo, and Ullman's *On Protection in Operating Systems* [HRU75] is not in any way specific to operating systems, but rather applies to any access control system that can be described in terms of Lampson's access matrix. It emphasizes the importance of an access control system's *update rules*, the rules governing permission to change the permissions in the matrix. This paper shows that *some* update rules make the calculation of (in our terms) a topology-only bound on potential permission (TP) undecidable.

Bishop and Snyder's *The Transfer of Information and Authority in a Protection System* [BS79] was the first to see beyond permission-based analysis. Their "potential de facto" analysis showed that overt information transfer could be tractably bounded in capability systems. This approach inspired our notion of authority.

Fred Spiessens, Yves Jaradin, and Peter Van Roy have developed the SCOLL tool [SMRS04, SR05a], covered in Related Work Section 26.4, which builds on and extends Bishop and Snyder's "take grant" framework to derive partially behavioral bounds on potential authority (BA).

Chapter 9

The Object-Capability Paradigm

Matter tells space how to curve.
Space tells matter how to move.
—John Archibald Wheeler

In the object model of computation [GK76, HBS73], there is no distinction between subjects and objects. A non-primitive object, or *instance*, is a combination of code and state, where *state* is a mutable collection of *references* to objects. The computational system is the dynamic reference graph of objects. Objects—behaving according to their code—interact by sending messages on references. Messages carry references as arguments, thereby changing the connectivity of the reference graph.

The *object-capability* model uses the reference graph as the access graph, requiring that objects can interact *only* by sending messages on references. To get from objects to object-capabilities we need merely prohibit certain primitive abilities which are not part of the object model anyway, but which the object model by itself doesn't require us to prohibit—such as forged pointers, direct access to another's private state, and mutable static state [KM88, Ree96, MMF00]. For example, C++ [ES90], with its ability to cast integers into pointers, is still within the object model but not the object-capability model. Smalltalk and Java fall outside the object-capability model because their mutable static variables enable objects to interact outside the reference graph.

Such mutable static state can be modeled as instance state, and thereby explained in terms of the model presented in this chapter. In Chapter 10 we introduce an additional criterion, *loader isolation*, and explain how Java violates it. *Confinement*, explained in Chapter 11, relies on loader isolation, and so cannot be realized in these languages. The Related Work Section 26.5 on Joe-E explains restrictions adequate to turn Java into an object-capability language, such as ensuring that all static variables are `final` (not assignable) and initialized to hold only transitively immutable data. Except where stated otherwise, further expository use of Java will assume this restriction.

Whereas the functionality of an object program depends only on the abilities provided by its underlying system, the security of an object-capability program depends on underlying inabilities as well. In a graph of defensive objects, one object's correctness depends not only on what the rules of the game say it can do, but also on what the rules say its potential adversaries cannot do.

Model Term	Capability OS Terms	Object Language Terms
instance	process, domain	instance, closure
code	non-kernel program + literal data	lambda expression, class file, method table
state	address space + c-list (capability list)	environment, instance variable frame
index	virtual memory address, c-list index	lexical name, variable offset, argument position
object creation, instantiation	`fork`, facet creation	instantiation, construction, lambda evaluation
loader	domain creator, `exec`	`eval`, ClassLoader

Table 9.1: Capability / OS / Object corresponding concepts. The first column lists the names used here for various concepts common to all object-capability systems. The second column lists names commonly used for these concepts in an operating system context, especially by capability-based operating systems. The third column lists names commonly used for these concepts in a programming language context.

9.1 The Object-Capability Model

The following model is an idealization of various object languages and object-capability operating systems. All its access control abilities are present in DVH (Dennis and van Horn's Supervisor) and many other object-capability systems [Lev84].[1] Object-capability systems differ regarding concurrency control, storage management, equality, typing, and the primitiveness of messages, so we avoid these issues in our model. Our model does assume reusable references, so it may not fit object-capability systems based on concurrent logic/constraint programming [MBTL87, KM88, RH04]. However, our examples may easily be adapted to any object-capability system despite these differences. Table 9.1 shows how the terms used here correspond to terms commonly used to describe programming languages and operating systems.

The static state of the reference graph is composed of the following elements.

- An *object* is either a *primitive* or an *instance*. Later, we explain three kinds of primitives: *data*, *devices*, and *loaders*. Data is immutable.

- An *instance* is a combination of *code* and *state*. We say it is an *instance of* the behavior described by its code. For example, in an operating system context, we say a process is an instance of its program. In a lexically scoped lambda language, a closure is an instance of its lambda expression.[2]

[1] Our object-capability model is essentially the untyped call-by-value lambda calculus with applicative-order local side effects and a restricted form of `eval`—the model Actors and Scheme are based on. This correspondence of objects, lambda calculus, and capabilities was noticed several times by 1973 [GK76, HBS73, Mor73a], and investigated explicitly in [TMHK95, Ree96].

[2] To describe Java in these terms, we would say a class is an instance of its classfile, where the class' initial state is provided by its ClassLoader. Given the Joe-E restrictions, we can also say that a Java object is an instance of its class.

- An instance's *state* is a mutable map from *indexes* to *references*. (Alternatively, as in E, we can consider an instance's state to be an immutable map from indexes to references, and provide for mutability by introducing primitively mutable *Slot* objects. Either model can be easily expressed in the other.) The state of a process is its address space + c-list, providing a mapping from addresses to data and from c-list indexes to references.[3] The state of a closure is its captured environment, providing a mapping from the names of its instance variables to their values.

- A *reference* provides access to an object, indivisibly combining designation of the object, the permission to access it, and the means to access it. Our access diagrams become Granovetter diagrams, where the permission arrows now depict references.

- A *capability* is a reference to non-data.

- *Code* is some form of data (such as source text, an abstract syntax tree, or compiled instructions) used to describe an instance's behavior to a loader, as explained in Chapter 10. Code also contains literal data.

- Code describes how a *receiving instance* (or "self") reacts to an incoming message.

- While an instance is reacting, its *addressable references* are those in the incoming message, in the receiving instance's state, and in the literal data of the receiving instance's code. The *directly accessible objects* are those designated by addressable references. When Carol is directly accessible to Alice, we may also say that Alice *refers to*, *holds a reference to*, or *has direct access to* Carol.

- An *index* is some form of data used by code to indicate which addressable reference to use, or where in the receiving instance's state to store an addressable reference. Depending on the system, an index into state may be an instance variable name or offset, a virtual memory address, or a c-list index. An index into a message may be an argument position, argument keyword, or parameter name.

We distinguish three kinds of primitive objects.

1. *Data* objects, such as the number 7. Access to these are knowledge limited rather than permission limited. If Alice can figure out which integer she wants, whether 7 or your private key, she can have it. Data provides only information, not access. Because data is immutable, we need not distinguish between a reference to data and the data itself. (In an operating system context, we model user-mode compute instructions as data operations.)

2. *Devices*. For purposes of analysis we divide the world into a computational system containing all objects of potential interest, and an external world. On the boundary are primitive devices, causally connected to the external world by unexplained means. A non-device object can only affect the external world by sending a message to an accessible output device. A non-device object can only be affected by the external world by receiving a message from an input device that has access to it.

[3] A c-list index is analogous to a Unix file descriptor number. A c-list is a kernel-protected data structure analogous to the Unix kernel's per-process mapping from file descriptor numbers to file descriptors.

3. A *loader* makes new instances. Any object system must provide an object creation primitive. We distinguish two forms: In *closed creation*, such as nested lambda evaluation, the code of the new instance is already included in the code of the program creating this instance. A loader provides *open creation*, making new instances from explicitly provided code and state, thereby extending an already-running system with new behaviors. Chapter 10 explains how either closed creation or open creation can be built from the other. Chapter 11 uses a loader to implement confinement. The remainder of this chapter uses only closed creation.

9.2 Reference Graph Dynamics

Mutation (assignment), object creation, and message passing dynamically change the reference graph's connectivity. Let us examine all the ways in which Bob can come to hold a reference to Carol. By mutation, Bob can drop references, or can change the index `=>` reference map in his state; but mutation by itself does not enable him to acquire a reference. This leaves object creation and message passing.

9.2.1 Connectivity by Initial Conditions

For purposes of analysis, there is always a first instant of time. Bob might already refer to Carol when our universe of discourse comes into existence.

9.2.2 Connectivity by Parenthood

Bob says: `def` ***`carol`*** `{...}`

If Bob already exists and Carol does not, then, if Bob creates Carol, at that moment Bob is the only object that refers to Carol. From there, other objects can come to refer to Carol only by inductive application of these connectivity rules. Parenthood may occur by normal object instantiation, such as calling a constructor or evaluating a lambda expression, or by loading code as explained in Chapter 10.

9.2.3 Connectivity by Endowment

Alice says: `def` ***`bob`*** `{...carol...}`

If Carol already exists and Bob does not, then, if there exists an Alice that already refers to Carol, Alice can create Bob such that Bob is born already endowed with a reference to Carol. Bob might be instantiated by lambda evaluation, in which case a variable `carol` which is free within Bob might be bound to Carol within Bob's creation context, as supplied by Alice. Or Alice might instantiate Bob by calling a constructor, passing Carol as an argument.[4]

[4] The example of Smalltalk seems to argue that "by endowment" is not fundamental. Instead, Smalltalk objects are initialized "by introduction" after they are created, by an explicit initialization message. The price of this convention is that any of an object's clients could re-initialize it. To write defensively consistent objects in this style, every initialization method would have to check, in order to prevent re-initialization. "By endowment" then reappears to compactly describe the level of abstraction where we can assume such checking is pervasive.

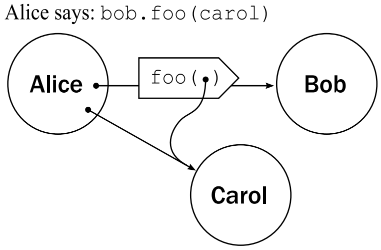


Figure 9.1: Introduction by Message Passing. In an object system, when Alice passes to Bob the message `foo(carol)`, she is both telling Bob about Carol and giving Bob the ability to access Carol. In an object-capability system, if Bob and Carol already exist and Bob does not already have access to Carol, the only way he can obtain such access is if an introducer, such as Alice, passes a message to him, thereby granting him that access. To be an introducer, Alice must already have access to Bob and Carol.

9.2.4 Connectivity by Introduction

Alice says: `bob.foo(carol)`

If Bob and Carol already exist, and Bob does not already refer to Carol, then the only way Bob can come to refer to Carol is if there exists an Alice that

- already refers to Carol,
- already refers to Bob, and
- decides to share with Bob her reference to Carol.

In object terms, if Alice has variables in her scope, `bob` and `carol`, that hold references to Bob and Carol, then Alice may send a message to Bob containing a copy of her reference to Carol as an argument. Unlike the `cp` example, and like the `cat` example, Alice does not communicate the string `"carol"` to Bob. Bob does not know or care what name Alice's code uses to refer to Carol.

In the initial conditions of Figure 9.1, Bob and Carol are *directly accessible* to Alice. When Alice sends Bob the message `foo(carol)`, she is both accessing Bob and permitting Bob to access Carol.

Alice can cause effects on the world outside herself only by sending messages to objects directly accessible to her (Bob), where she may include, at distinct argument indexes, references to objects directly accessible to her (Carol). We model a call-return pattern as an object creation and two messages [Hew77, Ste78], as explained further in Section 18.2). For example, Alice gains information from Bob by first creating a continuation—a facet of herself to receive this information, and then causing Bob (with a call carrying this continuation) to cause her to be informed (with a return invoking this continuation).

Bob is affected by the world outside himself only by the arrival of messages sent by those with access to him. On arrival, the arguments of the message (Carol) become directly accessible to Bob. Within the limits set by these rules, and by what Bob may feasibly know or compute, Bob reacts to an incoming message only according to his code. All computation happens only in reaction to messages.

9.2.5 Only Connectivity Begets Connectivity

By these rules, *only connectivity begets connectivity*—all access must derive from previous access. Two disjoint subgraphs cannot become connected, as no one can introduce them. More interestingly, if two subgraphs are almost disjoint, they can only interact or become further connected according to the decisions of those objects that bridge these two subgraphs. Topology-based analysis of bounds on permission (TP) proceeds by graph reachability arguments. Overt causation, carried only by messages, flows only along permitted pathways, so we may again use reachability arguments to reason about bounds on authority and causality (TA). The transparency of garbage collection relies on such arguments.

The object-capability model recognizes the security properties latent in the object model. All the restrictions above are consistent with good object programming practice even when security is of no concern.

9.3 Selective Revocation: Redell's Caretaker Pattern

Redell's caretaker [Red74] is our first example of an access abstraction. We present it to demonstrate how capability programmers write programs that attenuate authority. Previous formal analyses have led some researchers to conclude that revocation is impossible in capability systems [CDM01]. On examination, we see that permissions-based analyses would indeed arrive at this conclusion. The caretaker is but one example of this gap between the authority-based intuitions of capability practitioners and the permissions-based analyses in the formal security literature.

Capabilities do not directly provide for revocation. When Alice says `bob.foo(carol)`, she gives Bob unconditional, full, and perpetual access to Carol. Given the purpose of Alice's message to Bob, such access may dangerously exceed least authority. In order to practice POLA, Alice might need to somehow restrict the rights she grants to Bob. For example, she might want to ensure she can revoke access at a later time. But in a capability system, references themselves are the only representation of permission, and they provide only unconditional, full, perpetual access to the objects they designate.

What is Alice to do? She can use (a slight simplification of) Redell's caretaker pattern for revoking access. Rather than saying `bob.foo(carol)`, Alice can use the `makeCaretaker` function from Figure 9.2 to say instead:

```
def [carol2, carol2Gate] := makeCaretaker(carol)
bob.foo(carol2)
```

The caretaker `carol2` transparently forwards messages it receives to `target`'s value, but only if `enabled`'s current value is true. The gate `carol2Gate` changes what `enabled`'s current value is. Alice can later revoke the effect of her grant to Bob by saying `carol2Gate.disable()`.

Within the scope where `target` and `enabled` are defined, `makeCaretaker` defines two objects, `caretaker` and `gate`, and returns them to its caller in a two element list. Alice receives this pair, defines `carol2` to be the new caretaker, and defines `carol2Gate` to be the corresponding gate. Both objects use `enabled` freely, so they both share access to the same assignable `enabled` variable (which is therefore a separate Slot bound to `"&enabled"`).

What happens when Bob invokes `carol2`, thinking he is invoking the kind of thing Carol is? An object definition contains "`to`" clauses defining methods followed by an optional "`match`" clause defining a matcher. If an incoming message (say `x.add(3)`) doesn't match

```
def makeCaretaker(target) {
    var enabled := true
    def caretaker {
        match [verb, args] {
            if (enabled) {
                E.call(target, verb, args)
            } else {
                throw("disabled")
            }
        }
    }
    def gate {
        to enable()  { enabled := true  }
        to disable() { enabled := false }
    }
    return [caretaker, gate]
}
```

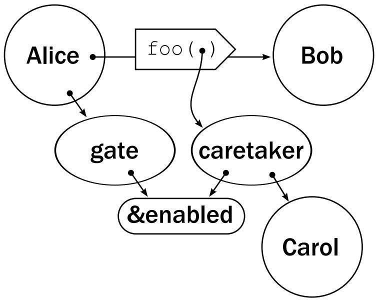


Figure 9.2: Redell's Caretaker Pattern. The code above shows, in E, a slightly simplified form of Redell's caretaker pattern. `makeCaretaker` makes three objects, a mutable Slot representing the current value of the `enabled` variable, a `caretaker` which conditionally forwards messages to `target`'s value, and a `gate` which will set `enabled` to `false`. It returns the caretaker and gate.

any of the methods, it is given to the matcher. The `verb` parameter is bound to the message name (`"add"`) and the `args` to the argument list (`[3]`). This allows messages to be received generically without prior knowledge of their API, much like Smalltalk's `doesNotUnderstand:` or Java's `Proxy`. Messages are sent generically using `E.call(...)`, much like Smalltalk's `perform:`, Java's "reflection," or Scheme's `apply`.

This caretaker provides a temporal restriction of authority. Similar patterns provide other restrictions, such as filtering forwarders that let only certain messages through.

Alice can use the caretaker pattern to protect against Bob's misbehavior, but only by relying on aspects of Carol's behavior: for example, that Carol does not make herself directly accessible to her clients. Alternatively, Alice could use the membrane pattern shown in Figure 9.3 to protect against both Bob and Carol, separating them even if they both strive to remain connected.

9.4 Analysis and Blind Spots

Given Redell's existence proof in 1974, what are we to make of subsequent arguments that revocation is impossible in capability systems? Of those who made this impossibility claim, as far as we are aware, none pointed to a flaw in Redell's reasoning. The key is the difference between permission and authority analysis. If we examine only permissions, we find Bob was never given permission to access Carol, so there was no access to Carol to be revoked! Bob was given permission to access `carol2`, and he still has it. No permissions were revoked.

Karger and Herbert propose [KH84] to give a security officer a list of all subjects who are, in our terms, permitted to access Carol. This list will not include Bob's access to Carol, since this indirect access is represented only by the system's protection state taken together with the behavior of objects playing by the rules. Within their system, Alice, by restricting the authority given to Bob, as she should, has inadvertently thwarted the security officer's ability to get a meaningful answer to his query. This leads to the following disconcerting observation:

> To render a permission-only analysis useless, a threat model need not include either malice or accident; it need only include subjects following security best practices.

Even in systems not designed to support access abstraction, many simple patterns happen naturally. Under Unix, Alice might provide a filtering forwarder as a process reading a socket Bob can write. The forwarder process would access Carol using Alice's permissions. Therefore, these blind spots are applicable to any system, if we mistakenly use permission alone to reason about what subjects may do.

A topology-only bound on permission (TP) or authority (TA) would include the possibility of the caretaker giving Bob direct access to Carol—precisely what the caretaker was constructed not to do. Only by reasoning about behaviors can Alice see that the caretaker is a "smart reference." Just as `makePoint` extends our vocabulary of data types, raising the abstraction level at which we express solutions, so does `makeCaretaker` extend our vocabulary for expressing access control. Alice (or her programmer) should use topology-only analysis for reasoning about objects outside her reliance set. But Alice also relies on some objects, like the caretaker, because she has some confidence she understands their actual behavior.

```
def makeMembrane(target) {
    var enabled := true
    def wrap(wrapped) {
        if (Ref.isData(wrapped)) {
            # Data provides only irrevocable knowledge, so don’t
            # bother wrapping it.
            return wrapped
        }
        def caretaker {
            match [verb, args] {
                if (enabled) {
                    def wrappedArgs := map(wrap, args)
                    wrap(E.call(wrapped, verb, wrappedArgs))
                } else {
                    throw("disabled")
                }
            }
        }
        return caretaker
    }
    def gate {
        to enable()  { enabled := true }
        to disable() { enabled := false }
    }
    return [wrap(target), gate]
}
```

Figure 9.3: Membranes Form Compartments. The simple caretaker pattern shown previously is only safe for Alice to use when she may rely on Carol not to provide Carol’s clients with direct access to herself. When Alice may not rely on Carol, she can use the membrane pattern. A membrane additionally wraps each capability (non-data reference) passing in either direction in a caretaker, where all these caretakers revoke together. By spreading in this way, the membrane remains interposed between Bob and Carol. The simplified membrane code shown here has the security properties we require, but is not yet practically efficient.

9.5 Access Abstraction

The object-capability model does not describe access control as a separate concern, to be bolted on to computation organized by other means. Rather it is a model of modular computation with no separate access control mechanisms. All its support for access control is well enough motivated by the pursuit of abstraction and modularity. Parnas' principle of information hiding [Par72] in effect says our abstractions should hand out information only on a need to know basis. POLA simply adds that authority should be handed out only on a need to do basis [Cro97]. Modularity and security each require both of these principles.

The object-capability paradigm, in the air by 1967 [WN79, Fab74], and well established by 1973 [Red74, HBS73, Mor73a, WCC$^+$74, WLH81], adds the observation that the abstraction mechanisms provided by the base model are not just for procedural, data, and control abstractions, but also for access abstractions, such as Redell's caretaker. (These are "communications abstractions" in [TMHK95].)

Access abstraction is pervasive in actual capability practice, including filtering forwarders, unprivileged transparent remote messaging systems [Don76, SJR86, vDABW96], reference monitors [Raj89], transfer, escrow, and trade of exclusive rights [MKH$^+$96, MMF00, Clo06], and recent patterns like the Powerbox [WT02, SM02]. Further, every non-security-oriented abstraction that usefully encapsulates its internal state provides, in effect, restricted authority to affect that internal state, as mediated by the logic of the abstraction. Defensively consistent abstractions guard the consistency of their state, providing integrity controls along the lines suggested by Clark and Wilson [CW87].

The platform is also a security-enforcing program, providing abstractions for controlling access. When all code is either within the platform or considered untrusted, one can only extend the expressiveness of a protection system by adding code to the platform, making everyone fully vulnerable to its possible misbehavior. By contrast, only Alice relies on the behavior of her caretaker, and only to limit authority flowing between Bob and Carol. The risks to everyone, even Alice, from its misbehavior are limited because the caretaker itself has limited authority. Alice can often bound her risk even from bugs in her own security-enforcing programs.

9.6 Notes on Related Work

Henry Levy's book *Capability-based Computer Systems* [Lev84] provides an overview of the early history of capability operating systems. (There is some coverage of hardware support for capabilities, but nothing on capability languages or cryptographic capability protocols.) While it is clear that these systems express similar perspectives on computation, prior attempts to capture this commonality have left out crucial elements. Lampson's characterization of capabilities in *Protection* [Lam74] does not include the requirement that Alice must have access to Bob in order to give Bob permission to access Carol. Although this property was present in DVH and many other systems, it was not universal. For example, Cal-TSS [LS76] does not impose this requirement. As we will see in Chapter 11, without this requirement, capabilities would be discretionary and unable to solve Lampson's confinement problem [Lam73].

Bishop and Snyder's "take grant" framework [BS79] does capture this requirement, but elides the issue of how objects name which capabilities they employ, grant, and receive, in order to keep them distinct. Eliding this issue was appropriate for their purpose: their conservative topology-only analysis assumed that any object might do anything it was permitted to do.

Hewitt's Actors model of computation [HBS73, Hew77], covered briefly in Chapter 23, does express a computational model of object-capability computation. Our connectivity rules above are essentially a restatement of the "locality laws" of Hewitt and Baker [HB78]. However, this expression of the capability model was coupled to the rest of Actors, which had many other elements. As a result, the rest of the capability community, and those seeking to understand the capability model, failed to be influenced by this crisp expression of the model. This chapter was written to help repair this state of affairs. In order to capture the lore among capability practitioners, we adapted the relevant aspects of Hewitt and Baker's model and explained how it relates to the elements of capability operating systems and languages.

Chapter 10

The Loader: Turning Code Into Behavior

Closed creation, such as nested lambda evaluation or constructor calls, naturally provides for the creation of new instances running old code. How does new code enter an already running system? The answers provided by different object-capability systems differ in many details, but an idealization of their underlying logic is captured by the *loader* abstraction we present here—essentially a restricted form of Scheme's `eval`. We define a safety criterion, *loader isolation*, which enables a loader's client to withhold permissions from the new instance. In Chapter 11, we rely on loader isolation to enforce confinement.

10.1 Closed Creation is Adequate

Our object-capability model defines an instance as combining code and state, where code is data describing the instance's behavior, and state provides a mutable map from indexes to references. We can model open creation as a *loader*, an object whose `load` method takes these two ingredients as arguments and returns the new instance they describe.

In programming language terms, this corresponds to a restricted form of Scheme's `eval` function, in which the first argument is a lambda expression and the second argument is a lexical environment providing mappings for all the free variables in the lambda expression. With only a closed creation primitive, one can write a loader as an interpreter for the language in which its code argument is expressed. In E, one might follow the pattern shown in Figure 10.1.

Unlike many interpretive definitions of `eval`, this one stops at object boundaries. When Alice's interpreter attempts to interpret a call to Bob, Bob is as encapsulated from Alice's interpreter as he would be from Alice, so Alice's interpreter uses `E.call(...)` to "reflectively" send the message to Bob. Likewise, Alice's interpreter is as encapsulated from her own clients as Alice would be, so Alice's interpreter uses `match` to "reflectively" accept any incoming message.

10.2 Open Creation is Adequate

Now that we understand what a loader does, we can use a loader to model closed creation primitives, such as nested lambda evaluation. Figure 10.2 shows how to transform nested

```
def loader {
    to load(code :Data, state) {
        def instance {
            match [verb, args] {
                ... # the interpreter goes here
            }
        }
        return instance
    }
}
```

Figure 10.1: Closed Creation is Adequate. We can use nested object definitions to create a new object whose code-following behavior is provided by an interpreter. The interpreter has access to the `verb` and `args` of the message to react to, and the `code` and `state` of the instance whose reaction it needs to emulate. A loader which follows this pattern and interprets using only these four variables provides *loader isolation*. The service it provides obeys the object-creation connectivity rules.

```
def makeCaretaker(target) {
    var enabled := true
    def caretaker := loader.load("def caretaker {...}",
                                 ["target"   => target,
                                  "&enabled" => &enabled,
                                  "E"        => E,
                                  "throw"    => throw])
    def gate := loader.load("def gate {...}",
                            ["&enabled" => &enabled,
                             "true"     => true,
                             "false"    => false])
    return [caretaker, gate]
}
```

Figure 10.2: Open Creation is Adequate. This transform of Figure 9.2 (p. 69) shows how a loader can model nested lambda evaluation. The loader makes a new instance behaving according to the code as endowed with the state. The state provides a mapping for all the code's instance variable names. The loader returns the only reference to the new instance. Transforming recursively would unnest all object definitions.

object definitions into loader calls. For each assignable instance variable, this transformation passes its Slot object, in order to preserve and make explicit the shared state relationships between objects in the same scope. Applying this transformation recursively would unnest all object definitions.

Of course, at least one of these forms of creation must be provided primitively. Many object-capability languages and operating systems actually provide primitives for both forms of creation. For example, in DVH, the open process creation primitive takes as arguments an address space and a c-list. The address space contains both code and that portion of state mapping from addresses to data. The c-list is the remaining portion of state mapping from c-list indexes to capabilities. DVH also provides a closed creation primitive by which an already-running process can create a "protected procedure," a new capability representing a new entry point into that process. All the code and state used by the protected procedure is already contained in the code and state of its parent process. These new entry points are facets of their process. Other object-capability operating systems [Har85, SSF99] provide analogous pairs of open process-creation and closed facet-creation primitives.

10.3 Loader Isolation

For efficiency, primitively provided loaders hand the code to some lower level interpreter for faster execution. For example, an operating system's loader loads machine instructions, and arranges for these to be interpreted directly by the hardware. A loader must ensure that the behavior of the new instance is limited by the rules of our model, no matter what the code might attempt to express to the contrary. Operating systems ensure this using hardware protection. Programming languages might use code verification and/or trusted compilers. The behavior of a primitively loaded instance must not exceed what would have been possible of an interpretive instance created by the loader in Figure 10.1.

If a loader obeys *loader isolation*, it is an object creation primitive satisfying the connectivity constraints of Section 9.2. The "by parenthood" constraint implies that the loader returns to its caller the *only* reference to the new instance. The "by endowment" constraint implies that the state argument provides *all* the instance's initial state. In programming language terms, state provides mappings for *all* the variable names used freely in the code. All linking happens only by virtue of these mappings. When a loader obeys loader isolation, Alice can use it to load code she doesn't understand and is unwilling to rely on. Alice is assured that the new instance only has that access Alice grants. Alice can withhold any permission from the new instance merely by not providing it.

A common way in which systems violate loader isolation is by honoring *magic names.* By *magic name*, we mean a name used in the code which is magically mapped to some source of authority not appearing in the state. For the interpretive loader of Figure 10.1 to violate loader isolation, it must use these magic names freely itself, in order to behave as an instance having access to these values. When a loader honors magic names, Alice has no ability to withhold this access from the new instance. This violation will be more or less problematic according to the authority accessible by magic. If this authority cannot be withheld, what sort of mischief does it enable?

The E loader does honor a few magic names, but the associated values provide no authority. In Figure 10.2, the loader calls need not actually include "`E`", "`throw`", "`true`", or "`false`", as the loader will resolve these magic names to their well known values, none of which provide authority. Although this violates our model and should be fixed, it is mostly harmless. The E loader also considers itself mostly harmless, and provides magic access to itself.[1]

To model Java's ClassLoader in these terms, we would say that the ClassLoader class corresponds to the "loader" of our model, and that loading proceeds in two steps. First, Alice creates a new ClassLoader, providing "state" as logic mapping import names to other classes. Alice then uses this ClassLoader to load classfiles. Each resulting class is an instance of this classfile as endowed by its ClassLoader's state. By considering a Java class to correspond to an "instance" in our model, we would then model Java's mutable static variables as instance variables of this class object. Modeled in this way, mutable static variables by themselves do not disqualify Java from being considered an object-capability language.

We can now say why Java is not an object-capability language.[2] The import mapping that Alice provides cannot override some magic system class names, and some of these system classes provide authority. Some of the accessible values are not harmless. (See Related Work Section 24.5 on the J-Kernel for an extended discussion.) Similar remarks apply to Scheme, Oz, ML, and Smalltalk. Efforts to make object-capability variants of these languages are covered in related work sections on the J-Kernel (Section 24.5), Joe-E (Section 26.5), W7 (Section 24.4), Oz-E (Section 26.3), Emily (Section 26.6), and Tweak Islands (Section 26.8). These efforts can be understood partially in terms of how they achieve loader isolation on these platforms.

10.4 Notes on Related Work

The LISP *1.5 Programmer's Manual* [M+62] presents a LISP meta-interpreter as mutually recursive `eval` and `apply` functions, where `eval` takes an S-Expression representing an expression to be evaluated, and an association list representing an environment.

Reflection and Semantics in LISP [Smi84] explains how meta-interpreters differ from each other according to which features of the language they "absorb" and which they "reify." For example, the LISP 1.5 meta-interpreter *absorbs* sequential and stacking execution, because the sequencing and stacking of the language being interpreted is implicitly provided by the sequencing and stacking of the language doing the interpreting. By contrast, the LISP 1.5 meta-interpreter *reifies* scoping, since it explicitly represents binding environments and variable lookup.

Structure and Interpretation of Computer Programs [AS86] presents several meta-interpreters for Scheme, a lexically-scoped LISP-like language with true encapsulated lexical closures. The first of these uses mutually recursive `eval` and `apply` functions in the spirit of the LISP 1.5 meta-interpreter. Successive meta-interpreters reify successively more aspects of the language, making the execution model yet more explicit.

The interpretive loader shown in Figure 10.1 derives from the meta-interpreters shown

[1] The definition of loader isolation presented here is also too strict to allow E's trademarking, explained in Section 6.3, but we believe this conflict is again mostly harmless for reasons beyond the scope of this dissertation. Although we use both loader isolation and trademarking, this dissertation does not use trademarking within loaded code, and so does not depend on the resolution to this conflict.

[2] This observation does not constitute a criticism of Java's designers. Java's security architecture [Gon99], by design, does not attempt to provide object-capability security. Rather, it is to their credit that Java comes so tantalizingly close to satisfying these goals anyway.

in *Meta Interpreters for Real* [SS86]. These meta-interpreters effectively absorb `apply`, so that interpreted code can interact with non-interpreted code, with neither being aware of whether the other is being interpreted.

Whereas LISP's `eval` and DVH's process creation primitive take the actual code as arguments, other loaders generally take a name to be looked up to find this code. Unix `exec` is an example of an operating system loader that takes a filename as argument, and obtains the code by reading the file. Programming language loaders allow the delayed linking and loading of modules. *Program Fragments, Linking, and Modularization* [Car97] analyzes the semantics of linking, and the conditions under which the linking of separately compiled modules should be expected to be type safe.

Generally, name-based loaders are name-centric [Ell96], *i.e.*, they assume a globally shared namespace. For `exec`, this is the file system. Programming languages, even those otherwise careful to avoid global variable names, generally assume a global namespace of modules. Java may be the first programming language with a name-based loader to demonstrate that all global namespaces can be avoided [LY96, QGC00]. A Java module is a class, whose identity is uniquely defined by a pair of a first-class anonymous ClassLoader object and a fully qualified name. The fully qualified name functions as a key-centric name path [Ell96, EFL$^+$99], whose traversal is rooted in the local namespace of its ClassLoader. A Java programmer can define new ClassLoaders, and each ClassLoader can define its own behavior for obtaining the actual code from a fully qualified class name. The J-Kernel covered in Related Work Section 24.5 makes use of this flexibility.

Popek and Goldberg's *Formal Requirements for Virtualizable Third Generation Architectures* [PG74] explains the conditions needed for a hardware architecture to be cleanly virtualizable. First, they divide the instruction set into *privileged* and *non-privileged* instructions. For an instruction to be considered privileged, it must trap if executed in user mode, so that it can be emulated by a virtual machine monitor. Then they separately divide instructions into *innocuous* and *sensitive.* Sensitive instructions are further divided into *control sensitive* and *behavior sensitive*, though an instruction can be sensitive in both ways. Control sensitive instructions can cause an effect outside the program's addressable space—its address space and its normal register set. Behavior sensitive instructions are those which can be affected by state outside the program's addressable space, *i.e.*, it enables the program to sense external state, such as an instruction for reading the clock. An architecture is considered to be cleanly virtualizable if all sensitive instructions are privileged, *i.e.*, if all non-privileged instructions are innocuous. An example which makes their distinctions clear is an instruction which does something when executed in privileged mode, but acts as a noop, rather than trapping, when executed in user mode. Since it doesn't trap, it is a non-privileged instruction. Since its behavior depends on the privilege bit, it is a behavior sensitive instruction. A machine with such an instruction is not cleanly virtualizable.

Their notion of cleanly virtualizable corresponds to our notion of loader isolation. Innocuous instructions only allow local computation and manipulation of one's own state. Sensitive instructions provide authority. The privilege bit allows a virtual machine monitor or operating system kernel to deny to the loaded code the authority that privileged instructions would provide. The monitor or kernel can provide what authority it chooses according to how it reacts to trapped privileged instructions. Non-privileged sensitive instructions are magic names violating loader isolation. The authority they provide may only be denied by filtering, translating, or interpreting loaded code, in order to prevent or control the execution of such instructions.

Given a cleanly virtualizable machine, a virtual machine monitor responds to traps by emulating an isolated copy of the physical machine. By contrast, an operating system kernel responds to certain traps as system calls, creating a virtual machine whose privileged instructions are system calls, which are generally quite different from the privileged instructions of the physical machine. Here again, loader isolation issues arise. Unix's many system calls provide magic names, by which the loaded program has causal connectivity beyond that provided by `exec`'s caller. By contrast, the system calls provided by KeyKOS [Har85], EROS [SSF99], and Coyotos [SDN$^+$04] are like E's `E.call(...)` and `E.send(...)`. They enable the program to invoke objects it already has direct access to (by virtue of capabilities held in protected memory as that program's c-list), but provide no other authority. Such system calls do not violate loader isolation. A program only able to execute such system calls and innocuous instructions may still only affect the world outside itself according to the capabilities it acquires.

Chapter 11

Confinement

> ... a program can create a controlled environment within which another, possibly untrustworthy program, can be run safely... call the first program a customer and the second a service. ... [the service] may leak, *i.e.*, transmit ... the input data which the customer gives it. ... We will call the problem of constraining a service [from leaking data] the confinement problem.
>
> —Lampson [Lam73]

Once upon a time, in the days before wireless, you (a human customer) could buy a box containing a calculator (the service) from a manufacturer you might not trust. Although you might worry whether the calculations were correct, you could at least enter your financial data confident that the calculator could not leak your secrets back to its manufacturer. How did the box solve the confinement problem? The air gap let you see that the box came with no strings attached. When the only causation to worry about would be carried by wires, the visible absence of wires emerging from the box—the isolation of the subgraph—is adequate evidence of confinement.

In this chapter, we use this same technique to achieve confinement, substituting references for wires. To solve confinement, assume that the manufacturer, Max, and customer, Cassie, have mutual access to a `[Factory, makeFactory]` pair created by the code in Figure 11.1. Assume that Cassie relies on this pair of objects to behave according to this code.[1]

The `interface...guards...` expression evaluates to a (trademark guard, stamp) pair representing a new trademark, similar in purpose to an interface type. This syntax also defines variables to hold these objects, here named `Factory` and `FactoryStamp`. We use the `FactoryStamp` to mark instances of `factory`, and nothing else, as carrying this trademark. We use the `Factory` guard in soft type declarations, like `:Factory`, to ensure that only objects carrying this trademark may pass. This block of code evaluates to a `[Factory, makeFactory]` pair. Only the `makeFactory` function of a pair can make objects, instances of `factory`, which will pass the `Factory` guard of that pair.[2]

[1] Given *mutual* reliance in this pair, our same logic solves an important mutual defensiveness problem. Max knows Cassie cannot “open the case”—cannot examine or modify his code.

[2] Such trademarking can be implemented in DVH and in our model of object-capability computation [Mor73a, MBTL87, TMHK95, Ree96], so object-capability systems which provide trademarking primitively [WLH81, Har85, SSF99, YM03] are still within our model.

```
def [Factory, makeFactory] := {
    interface Factory guards FactoryStamp {...}

    def makeFactory(code :Data) :Factory {
        def factory implements FactoryStamp {
            to new(state) {
                return loader.load(code, state)
            }
        }
        return factory
    }
    [Factory, makeFactory]
}
```

Figure 11.1: Factory-based Confinement. Creates a `Factory` guard and a `makeFactory` function which will make confined factories that pass this guard.

Max uses a `makeFactory` function to package his proprietary calculator program in a box he sends to Cassie.

```
def calculatorFactory := makeFactory("...code...")
cassie.acceptProduct(calculatorFactory)
```

In Figure 11.2 Cassie uses a `:Factory` declaration on the parameter of her `acceptProduct` method to ensure that she receives only an instance of the `factory` definition. Inspection of the `factory` code shows that a `factory`'s state contains only data and no other references—no access to the world outside itself. Cassie may therefore use the factory to make as many live calculators as she wants, confident that each calculator has only that access beyond itself that Cassie authorizes. They cannot even talk to each other unless Cassie allows them to.

With closed creation such as lambda evaluation, a new subject's code and state both come from the same parent. To solve the confinement problem, we combine code from Max with state from Cassie to give birth to a new calculator, and we enable Cassie to verify that she is the only state-providing parent. This state is an example of Lampson's "controlled environment." To Cassie, the calculator is a *controlled subject*—one Cassie knows is born into an environment controlled by her. By contrast, should Max introduce Cassie to an already instantiated calculation service, Cassie would not be able to tell whether it has prior connectivity. (Extending our analogy, if Max offers the calculation service from his web site, no air gap would be visible to Cassie.) The calculation service would be an *uncontrolled subject* to her.

We wish to reiterate that by "confinement," we refer to the overt subset of Lampson's problem, in which the customer accepts only code ("a program") from the manufacturer and instantiates it in a controlled environment. We do not propose to confine information or authority given to uncontrolled subjects. For those systems that claim, in effect, to confine uncontrolled subjects, we should carefully examine whether they confine authority or only permission.

11.1 A Non-Discretionary Model

Capabilities are normally thought to be discretionary, and to be unable to enforce confinement. Our confinement logic relies on the non-discretionary nature of object-capabilities. What does it mean for an access control system to be discretionary?

> "Our discussion ... rested on an unstated assumption: the principal that creates a file or other object in a computer system has unquestioned authority to authorize access to it by other principals. ... We may characterize this control pattern as *discretionary*." [emphasis in the original]
>
> —Saltzer and Schroeder [SS75]

Object-capability systems have no principals. A human user, together with his shell and "home directory" of references, participates, in effect, as just another subject. With the substitution of "subject" for "principal," we will use this classic definition of "discretionary."

By this definition, object-capabilities are not discretionary. In our model, in DVH, and in many actual capability system implementations, even if Alice creates Carol, Alice may still only authorize Bob to access Carol if Alice has authority to access Bob. If capabilities were discretionary, they would indeed be unable to enforce confinement. To illustrate the power of confinement, we use it below to enforce the *-properties.

11.2 The *-Properties

Briefly, the *-properties taken together allow subjects with lower (such as "secret") clearance to communicate to subjects with higher (such as "top secret") clearance, but prohibit communication in the reverse direction [PB96]. However, claims that capabilities cannot enforce the *-properties continue [KL86, Gon89, WBDF97, SJ03], citing [Boe84] as their support. To lay this matter to rest, we show how Cassie solves Boebert's challenge problem—how she provides a one way communication channel to subjects she doesn't trust, say Q and Bond, whom she considers to have secret and top secret clearance respectively. Can Cassie prevent Boebert's attack, in which Q and Bond use the rights Cassie provides to build a reverse channel?

Completing our earlier confinement example, Cassie accepts a calculator factory from Max using the method shown in Figure 11.2.

Cassie creates two calculators to serve as Q and Bond. She builds a data diode by defining a `writeDiode` function, a `readDiode` function, and an assignable `diode` variable they share. She gives Q and Bond access to each other only through the data diode. Applied to Cassie's arrangement, Boebert's attack starts by observing that Q can send a capability as an argument to `writeDiode`. A topology-only analysis of bounds on permissions (TP) or authority (TA) supports Boebert's case—the data diode might introduce this argument to Bond. Only by examining the behavior of the data diode (BA) can we see the tighter bounds it was built to enforce. It transmits data (here, integers) in only one direction and capabilities in neither. (Q cannot even read what he just wrote.) Cassie relies on the behavior of the factory and data diode abstractions to enforce the *-properties and prevent Boebert's attack. (See [MYS03, SMRS04, SR05a] for further details.)

```
to acceptProduct(calcFactory :Factory) {
    var diode :int := 0
    def writeDiode(val) { diode := val }
    def readDiode() { return diode }

    def q := calcFactory.new(["writeUp" => writeDiode, ...])
    def bond := calcFactory.new(["readDown" => readDiode, ...])
    ...
}
```

Figure 11.2: Cassie Checks Confinement. Cassie's `acceptProduct` method uses a `:Factory` guard for accepting only confined calculator factories from Max.

11.3 The Arena and Terms of Entry

Policies like the *-properties are generally assumed to govern a computer system as a whole, to be enforced in collaboration with a human sys-admin or security officer. In a capability system, this is a matter of initial conditions. If the owner of the system wishes such a policy to govern the entire system, she can run such code when the system is first generated, and when new users join. But what happens once the system is already running? Let us say Alice meets Bob, who is an uncontrolled subject to her. Alice can still enforce "additive" policies on Bob, e.g., she can give him revocable access to Carol, and then revoke it. But she cannot enforce a policy on Bob that requires removing prior rights from Bob, for that would violate Bob's security!

Instead, as we see in the example above, acting as Lampson's "customer," she sets up an arena—Lampson's "controlled environment"—with initial conditions she determines, governed by her rules, and over which she is the sys-admin. If her rules can be enforced on uncontrolled subjects, she can admit Bob onto her arena as a player. If her rules require the players not to have some rights, she must set terms of entry. "Please leave your cellphones at the door." A prospective participant (Max) provides a player (`calcFactory`) to represent his interests within the arena, where this player can pass the security check at the gate (here, `:Factory`). No rights were taken away from anyone; participation was voluntary.

The arena technique corresponds to meta-linguistic abstraction—an arena is a virtual machine built within a virtual machine [AS86, SS86]. The resulting system can be described according to either level of abstraction—by the rules of the base level object-capability platform or by the rules of the arena. The subjects built by the admitted factories are also subjects within the arena. At the base level, we would say Q has permission to send messages to `writeDiode` and authority to send integers to Bond. Shifting our frame of reference to the arena level of description, we would say a data diode is a primitive part of the arena's protection state, and say Q has permission to send integers to Bond. Any base level uncontrolled subjects admitted into the arena are devices of the arena—they have mysterious connections to the arena's external world.

When the only inputs to a problem are data (here, code), any system capable of universal computation can solve any solvable problem, so questions of absolute possibility become useless for comparisons. Conventional language comparisons face the same dilemma, and language designers have learned to ask instead an engineering question: Is this a good

```
def q {
    def gizmo(intFunc) { ... intFunc(i) ... }
    def [writeUp2, writeUp2Gate] := makeMembrane(writeUp)
    gizmo(writeUp2)
    ...
    if (...) { writeUp2Gate.disable() }
    ...
}
```

Figure 11.3: Unplanned Composition of Access Policies. Let us say that Max uses the above text as the source code provided to the `calcFactory` that will build Q. Q builds a gizmo to which he gives revocable access to the `writeUp` channel which Cassie provides to him. At some later time, Q can then revoke the gizmo's access to this channel. Q's policy—to limit `gizmo`'s access—depends on the membrane abstraction, by which `writeUp2` attenuates the authority provided by `writeUp`. Cassie's policy depends on the diode abstraction, by which `writeUp` attenuates the authority provided by the `diode` variable. Cassie and Max each express policies to defend their interests, and these compose correctly without preplanning.

machine on which to build other machines? How well did we do on Boebert's challenge? The code admitted was neither inspected nor transformed. Each arena level subject was also a base level subject. The behavior interposed by Cassie between the subjects was very thin. Mostly, we reused the security properties of the base level object-capability platform to build the security properties of our new arena level platform.

11.4 Composing Access Policies

When mutually defensive interests build a diversity of abstractions to express a diversity of co-existing policies, how do these extensions interact?

Let us say that Q builds a gizmo that might have bugs, so Q makes a membrane to give the gizmo revocable access to his `writeDiode` function, as shown in Figure 11.3. Q's policy relies on the behavior of his membrane but not necessarily on Cassie's `writeDiode` function. To Cassie, Q's gizmo and membrane are part of Q's subgraph and indistinguishable from Q. Cassie's policy relies on the behavior of her `writeDiode` function, but not on Q's membrane. They each do a partially behavioral analysis over the same graph, each from their own subjective perspective. This scenario shows how diverse expressions of policy often compose correctly even when none of the interested parties are aware this is happening.

11.5 The Limits of Decentralized Access Control

This dissertation mentions three forms of capability system: operating systems like DVH, programming languages like the non-distributed subset of E, and cryptographic capability protocols like Pluribus. The object-capability model we have presented applies only to the first two: operating systems and programming languages. Our model implicitly assumes a mutually relied upon platform, which is therefore a central point of failure. In the absence of this assumption, we are left only with cryptographic enforcement mechanisms. Cryptography by itself can only enforce weaker access control properties. In exchange, cryptography enables us to build systems with no central points of failure.

Tamper detecting hardware with remote attestation, such as NGSCB and TPM [AW04, Tru02], seems to enable forms of distributed security beyond what cryptography alone can provide. These mechanisms lower the risk of physically distributing a collection of mutually reliant machines. However, if these machines rely on each other's attestations, then they each remain a central point of failure for the collection as a whole. Regarding either robustness or security, such a collection of machines still forms a single logically centralized platform. Between platforms unwilling to rely on each other's attestations, we again are left only with cryptography. By *decentralized*, we mean a distributed system with no central points of failure.

11.5.1 Implications for Confinement

Pluribus provides cryptographically protected inter-vat references and messages. Of course, Pluribus cannot help any participant detect what protection mechanisms (languages, operating systems, etc.) are used internally by the platforms of the other participants, or what causal access these other participants may have to each other. Pluribus provides only the "by introduction" subset of the connectivity rules, since it deals only in messages. It cannot account for object creation, so it cannot enable any participant to detect an air gap between other participants. To each machine, all other machines are uncontrolled subjects.

Of course, any participant can claim their platform is running, for example, a trustworthy E system, and offer an alleged loader over the network. Other participants may choose which of these claims they are willing to rely on for what. In this way, the overall E system consists of alleged object-capability islands sending messages to each other over a cryptographic-capability sea.

For example, our discussion of confinement assumed that Max and Cassie could both rely on the same loader to provide loader isolation, and likewise with the `[Factory, makeFactory]` pair built on this loader. So long as they both rely on a common platform, we may as well bundle such useful services with the platform. But in a decentralized system, we need to be explicit about which platform's loader is used, and who is willing to rely on that platform for what. If Cassie is willing to rely on that loader's platform to provide loader isolation, and if Max is willing to submit his code to that loader, then they can still solve the confinement problem. But if Max wishes to keep his code secret, then he would only send his code to a platform he is willing to rely on to keep this secret. If there is no one platform that Cassie and Max are both willing to rely on, they can no longer arrange for Cassie to use Max's calculator code.

11.5.2 Implications for the *-Properties

The *-properties provide another illustration of cryptography's limits. If Q, Bond, and Cassie are three machines spread over the Internet, the only secure way for Cassie to distinguish Q from Bond is based on their knowledge of secrets. Therefore, if Q tells Bond everything he knows, including all his private keys, then Bond can impersonate Q in every way—Bond can be both Q and Bond. Knowledge consists only of bits. Between machines, the Internet provides only a bit channel anyway. Between mutually defensive machines, it is impossible to provide a bit channel that cannot transmit permissions, so Boebert's attack cannot be prevented—not by capabilities or by any other means. However, if Q and Bond are objects running on platforms Cassie is willing to rely on, then Cassie can arrange their separation in a manner she should also rely on.

Between mutually reliant hardware, on the one hand, and open networks protected only by cryptography, on the other hand, there exists an intermediate option: Since bits received on a particular physical wire must have been sent by a machine with access to that wire, this authenticates these bits as having come from such a machine, without needing to rely on that machine to behave correctly. The Client Utility [KGRB01, KK02] ties the bits that represent a capability to the channel over which the bits are received. If that channel is authenticated by means other than a secret, such as a physical wire, then bits cannot be used to transmit permission, since bits cannot transmit the ability to transmit on that wire.

11.5.3 Implications for Revocation

The caretaker provides another lesson on the limits of decentralized access control. If Alice, Bob, and Carol are on three separate machines, and Alice wishes to instantiate a caretaker to give Bob revocable access to Carol, Alice chooses where to instantiate this caretaker. By default, she instantiates it in her own vat, in which case she can be confident in her ability to revoke. But messages from Bob to Carol would then be routed through Alice's machine. Alternatively, she might use a loader exported by Carol's vat in order to instantiate the caretaker there. Alice already relies on Carol's vat to prevent unauthorized access to Carol by Bob, so Alice should also be willing to rely on Carol's vat to host this caretaker. However, if a network disconnect prevents Alice's `disable()` message from reaching Alice's `gate` on Carol's vat, then Bob's authority fails to be revoked.

The object-capability model was originally created for single machine systems, where a reference is a reliable conveyer of messages. Between machines separated by unreliable networks, a reference can be at best a fail-stop conveyer of messages. When these messages would exercise some authority, or when they would cause an increase in the authority of some other object, then the failure to deliver a message is still a fail safe deviation from the original object-capability model. However, when the message would cause a *decrease* in the authority of some other object, than the failure to deliver such a message fails *unsafe*.

In Section 17.1, we see that a network partition would cause a `__reactToLostClient(`*`exception`*`)` to be sent to Alice's `gate`, to notify it that a partition has occurred. To remain robust in the face of this hazard, Alice should change her `gate`'s code to function as a "dead man's switch" [Cud99], so that it can react to her absence:

```
def gate {
    to enable()               { enabled := true  }
    to disable()              { enabled := false }
    to __reactToLostClient(ex) { gate.disable() }
}
```

If Alice's and Carol's vats cannot communicate for longer than a built-in timeout period, Carol's vat will decide that a partition has occurred and notify Alice's `gate`, which will then auto-disable itself, preventing further access by Bob. Once the network heals, Section 17.1 explains how Alice can reestablish access to her `gate`. Alice can decide whether to re-enable

Bob's access at that time. With this pattern, the timeout built into Pluribus is also the limit on the length of time during which Bob may have inappropriate access. After this delay, this pattern fails safe.

11.6 Notes on Related Work

The Gedanken language covered in Related Work Section 24.1 invented the "trademarking" technique.

The `factory` mechanism presented in this chapter is a simplification of the mechanisms used to achieve confinement in actual object-capability systems [Key86, SSF99, SW00, WT02, YM03]. Among the simplifications is that it is overly conservative: The only objects admissable by Cassie's entrance check (`:Factory`) are instances of our `factory` code, which are therefore transitively immutable. The calculator objects made by these factories derive all their causal connectivity from Cassie.

The KeyKOS Factory [Key86] and the EROS Constructor [SSF99] both allow more connectivity while remaining safe. These systems primitively provide a notion of a transitively read-only capability ("sensory" and "weak" capabilities, respectively), where an object holding such a capability can observe state changes another can cause, but cannot (using this capability) cause such state changes. In these systems, Max could make a calculator factory whose calculators interpret a program he can write. This allows Max to upgrade the algorithms used by Cassie's calculators, while still providing Cassie the assurance that the calculators cannot communicate back to Max. The KeyKOS Factory would also allow Max to endow the factory with arbitrary capabilities as "holes," to provide to the new calculators in turn as part of their initial endownment. A factory would only pass Cassie's admission check if she approves of its holes. In KeyKOS and EROS, Cassie would test whether the object from Max is a Factory/Contructor by testing whether it carries the Factory *brand* [Har85]. Brands are essentially a rediscovery of Gedanken's trademarking, but with the restriction that an object can carry only one brand.

Verifying the EROS Confinement Mechanism by Shapiro and Weber [SW00] presents the first formal proof of overt confinement in such systems.

The "*-properties" above combine the "*-property" and the "simple security property" of the "Bell-La Padula" security model [PB96]. Earl Boebert's *On the Inability of an Unmodified Capability Machine to Enforce the *-property* [Boe84] presents an arrangement of capabilities set up as an example of how one might try to enforce the *-properties using capabilities. He then demonstrates how this arrangement allows an attack which violates the *-properties.

Responding to this challenge, Kain and Landwehr's *On Access Checking in Capability Systems* proposes several schemes to add identity-based access checks to capabilities to prevent this attack [KL86]. Li Gong's *A Secure Identity-based Capability System* [Gon89] explains a cryptographic protocol implementing distributed capabilities combined with such an identity-based access check. The *Secure Network Objects* [vDABW96] covered briefly in Related Work Section 24.7 incorporates an identity check into their cryptographic capability protocol. Boebert himself addressed these vulnerabilities with special hardware [Boe03]. Vijay Saraswat and Radha Jagadeesan propose topological rules for confining permissions in object-capability systems, in order to address the vulnerabilities Boebert points out [SJ03].

KeySAFE is a concrete and realistic design [Raj89] for enforcing the *-properties on KeyKOS, a pure object-capability operating system. The techniques presented in this chapter are an idealization of the relevant aspects of the KeySAFE design.

Chapter 12

Summary of Access Control

Just as we should not expect a base programming language to provide us all the data types we need for computation, we should not expect a base protection system to provide us all the elements we need to directly express access control policies. Both issues deserve the same kind of answer: We use the base to build abstractions, extending the vocabulary we use to express our solutions. In evaluating an access control model, one must examine how well it supports the extension of its own expressiveness by abstraction and composition.

Security in computational systems emerges from the interaction between primitive protection mechanisms and the behavior of security-enforcing programs. As we have shown, such programs are able to enforce restrictions on more general, untrusted programs by building on and abstracting more primitive protection mechanisms. To our knowledge, the object-capability model is the only protection model whose semantics can be readily expressed in programming language terms: approximately, lambda calculus with local side effects. This provides the necessary common semantic framework for reasoning about permission and program behavior together. Because security-enforcing programs are often simple, the required program analysis should frequently prove tractable, provided these programs are built on effective primitives.[1]

By recognizing how program behavior contributes to access control, we establish a semantic basis for extensible protection. Diverse and mutually defensive interests can each build abstractions to express their plans regarding new object types, new applications, new requirements, and each other, and these plans can co-exist and compose. This extensibility is well outside the scope of purely topological access graph analyses.

Analyses based on the potential evolution of access graphs are conservative approximations. A successful verification that demonstrates the enforcement of a policy using only the access graph (as in [SW00]) is robust, in the sense that it holds without regard to the behavior of programs. Verification failures are *not* robust—they may indicate a failure in the protection model, but they can also result from what might be called "failures of excess conservatism"—failures in which the policy is enforceable but the verification model has been simplified in a way that prevents successful verification.

We have shown by example how object-capability practitioners set tight bounds on

[1] Since these words were first written, Fred Spiessens, Yves Jaradin, and Peter Van Roy have built the SCOLL system, explained in Related Work Section 26.4. SCOLL derives partially behavioral bounds on authority (BA) along the lines we have suggested. SCOLL has indeed found it tractable to reason about the bounds on authority created by several security-enforcing programs including the caretaker and *-properties patterns we have presented.

authority by building abstractions and reasoning about their behavior, using conceptual tools similar to those used by object programmers to reason about any abstraction. We have shown, using only techniques easily implementable in Dennis and van Horn's 1965 Supervisor [DH65], how actual object-capability systems have used abstraction to solve problems that analyses using only protection state have "proven" impossible for capabilities.

The object-capability paradigm, with its pervasive, fine-grained, and extensible support for the principle of least authority, enables mutually defensive plans to cooperate more intimately while being less vulnerable to each other.

Part III

Concurrency Control

Overview of Concurrency Control

Access control by itself cannot explain how to maintain consistency in the face of concurrency. Traditional concurrency control is concerned with consistency, but only under cooperative assumptions. For defensive consistency, we must take a fresh look. We illustrate many of our points with a simple example, a "statusHolder" object implementing the listener pattern. We use the listener pattern as an example of coordinating loosely coupled plans. This part proceeds as follows.

Interleaving Hazards introduces the statusHolder and examines its hazards, first, in a sequential environment. It then presents several attempts at building a conventionally thread-safe statusHolder in Java and the ways each suffers from plan interference.

Two Ways to Postpone Plans shows a statusHolder written in E and explains E's eventual-send operator, first, in the context of a single thread of control. It then explains how the statusHolder with this operator handles concurrency and distribution under cooperative 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, 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 reestablishing distributed consistency.

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

Delivering Messages in E-ORDER explains a message delivery order strong enough to enforce a useful access control restriction among objects, but weak enough to be enforceable among the machines hosting these objects.

Chapter 13

Interleaving Hazards

13.1 Sequential Interleaving Hazards

Throughout this part, we will examine different forms of the listener pattern [Eng97]. The Java code shown in Figure 13.1 is representative of the basic sequential listener pattern.[1] 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.

Even under sequential and cooperative conditions, this pattern creates plan interference hazards:

Aborting the wrong plan: A listener that throws an exception 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 depends on how updating the `myListeners` collection affects an iteration in progress.

[1] 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);
        }
    }
}
```

Figure 13.1: The Sequential Listener Pattern in Java. Subscribers subscribe listeners by calling `addListener`. Publishers publish a new status by calling `setStatus`, causing all subscribed listeners to be notified of the new status.

```
acctMgr.withdraw(100)
  statusHolder.setStatus(3900)
    myStatus := 3900
    financeListener.statusChanged(3900)
      acctMgr.deposit(1000)
        statusHolder.setStatus(4900)
          myStatus := 4900
          financeListener.statusChanged(4900)
          cellViewer.statusChanged(4900)
            ... display "$4900" in spreadsheet cell ...
    cellViewer.statusChanged(3900)
      ... display "$3900" in spreadsheet cell ...
```

Figure 13.2: Anatomy of a Nested Publication Bug. An account manager uses a StatusHolder to publish the account balance. The subscribers are: 1) a finance application, which reacts to the balance falling below $4000 by depositing $1000, and 2) a spreadsheet, which will display each successive balance in a spreadsheet cell. When $100 is withdrawn, each is notified of the new $3900 balance. In reaction, the finance listener deposits $1000, causing each to be notified of the $4900 balance. $4900 is adequate, so the finance application ignores the nested notification. The spreadsheet displays $4900. The outer notification of the finance listener completes, so the spreadsheet is notified of the $3900 balance, which it displays. The last assignment to `myStatus` was 4900, but the last update of the spreadsheet displays “$3900.”

Nested publication: Similarly, a listener may cause a publisher to publish a new status, possibly unknowingly due to aliasing. In the example shown in Figure 13.2, 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 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 cooperative conditions.

13.2 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 [RH04], 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 nondeterministic 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, Han93] 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 [Rep99], and the synchronous π-calculus [Mil99] are loosely in the rendezvous tradition. With minor adjustments, the following comments apply to both.

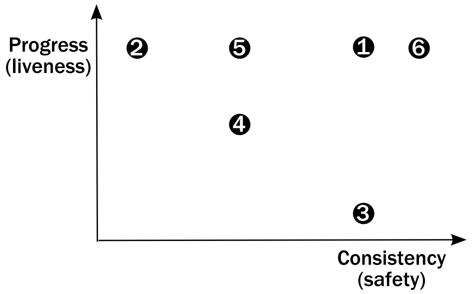


Figure 13.3: Thread-safety is Surprisingly Hard. A correct program must both remain consistent and continue to make progress. The sequence above represents our search for a statusHolder which supports both well: ❶ The sequential statusHolder. ❷ The sequential statusHolder in a multi-threaded environment. ❸ The fully synchronized statusHolder. ❹ Placing the for-loop outside the synchronized block. ❺ Spawning a new thread per listener notification. ❻ Using communicating event-loops. ❻ is safer than ❶ since it also avoids the sequential interleaving hazards.

13.3 Preserving Consistency

On Figure 13.3, ❶ represents the sequential statusHolder of Figure 13.1 executing in a sequential environment. If we place our sequential statusHolder into a concurrent environment ❷, publishers or subscribers may call it from different threads. The resulting interleaving of operations might, for example, mutate the `myListeners` list while the for-loop is in progress.

Adding the "`synchronized`" keyword to all methods would cause the statusHolder to resemble a monitor ❸. 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.

13.4 Avoiding Deadlock

To avoid this problem, Englander recommends [Eng97] changing the `setStatus` method to clone the listeners list within the synchronized block, and then to exit the block before entering the for-loop (Figure 13.3, ❹), as shown in Figure 13.4. 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

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

Figure 13.4: A First Attempt at Deadlock Avoidance. A recommended technique for avoiding deadlock when using the Listener pattern is to copy the listeners list within the synchronized block, and then to notify these listeners after execution has exited the synchronized block.

publisher may hold other locks.

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.

13.5 Race Conditions

The approach in Figure 13.4 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 (Figure 13.3, ❺). 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 hazardous 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 [RH04, Lee06, Ous96].

> 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]

13.6 Notes on Related Work

Edward A. Lee’s *The Problem with Threads* [Lee06] explains the same hazards explained in this chapter, and then presents a variety of computational models which are free of these hazards.

John Ousterhout’s talk *Why Threads are a Bad Idea (For Most Purposes)* [Ous96] explains several of the hazards of threads, and explains why event loops better serve many of the purposes for which threads are currently employed.

There is an ongoing controvery over whether threads or events are more suitable for high performance. *SEDA: An Architecture for Well-Conditioned, Scalable Internet Services* [WCB01] presents a high performance architecture using events. Eric Brewer, one of the co-authors of this paper, then went on to co-author *Why Events Are a Bad Idea (for High-Concurrency Servers)* [vBCB03] which argues the opposite case. However, the focus of both of these papers is performance rather than correctness or robustness.

Chapter 14

Two Ways to Postpone Plans

Figure 14.1 shows the same statusHolder as defined in E. 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`.

The E code for statusHolder in Figure 14.1 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: Postpone the rest of X, work on Y until complete, then go back to X.

Eventually: Postpone Y by putting it on a "to-do" list and work on it after X is complete.

The "immediate" option corresponds to conventional, sequential call-return control flow

```
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
}
```

Figure 14.1: The Sequential Listener Pattern in E. The `statusHolder` above still has the same sequential hazards as its Java counterpart shown in Figure 13.1 (p. 98).

(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 publisher'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.

14.1 The Vat

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.

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 Chapter 16.

The combination of a stack, a pending delivery queue, and the heap of objects they operate on is called a *vat*, illustrated in Figure 14.2. 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

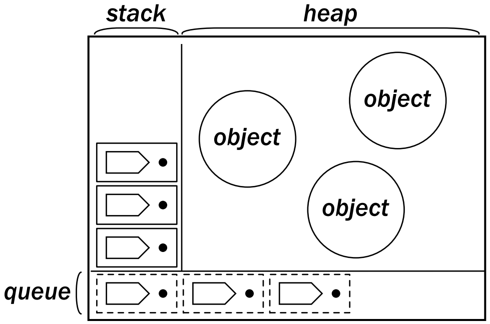


Figure 14.2: A Vat's Thread Services a Stack and a Queue. An E vat consists of a heap of objects and a thread of control. The stack and queue together record the postponed plans the thread needs to process. An immediate-call pushes a new frame on top of the stack, representing the delivery of a message (*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.

failure, resource control, preemptive termination/deallocation, and defense from denial of service. We will return to some of these topics in subsequent chapters.

14.2 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.*[1] As we have seen, near references carry both immediate-calls and eventual-sends. Only *eventual references* may cross vat boundaries, so the spreadsheet (in VatS) holds an eventual reference to the statusHolder (in VatA), which in turn holds an eventual reference to the spreadsheet's listener (in VatS). 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 14.3 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 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.

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

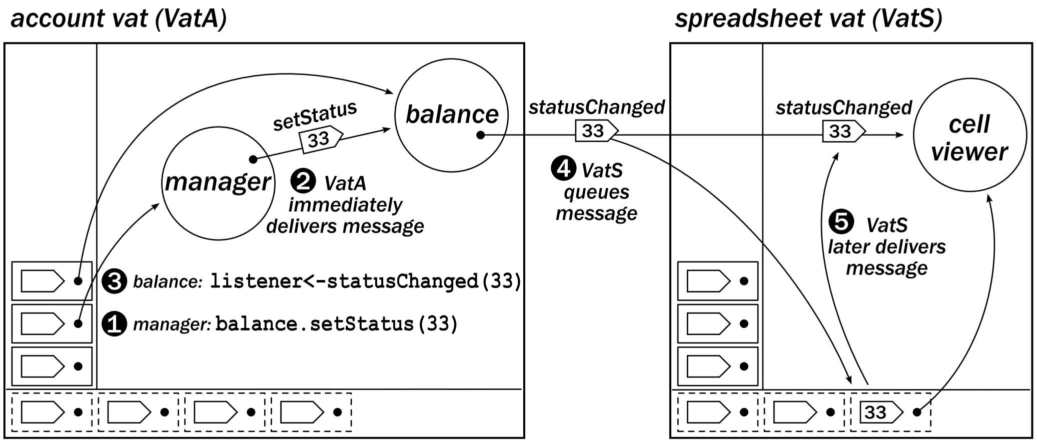

account vat (VatA)
spreadsheet vat (VatS)
setStatus
33
balance
manager
❷ VatA immediately delivers message
statusChanged
33
❹ VatS queues message
statusChanged
33
cell viewer
❺ VatS later delivers message
❸ balance: listener<-statusChanged(33)
❶ manager: balance.setStatus(33)
33

Figure 14.3: An Eventually-Sent Message is Queued in its Target's Vat. If the account manager and the spreadsheet are in separate vats, when the account manager ❶ tells the statusHolder that represents its balance to immediately update, this ❷ transfers control to the statusHolder, which ❸ notes that its listeners should eventually be notified. The message is ❹ sent to the spreadsheet's vat, which queues it on arrival and eventually ❺ delivers it to the listener, which updates the display of the spreadsheet cell.

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 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.[2] Because E has no explicit locking constructs, computation within a turn can never block—it can only

[2] 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 dissertation, 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.

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 Section 16.3 on "Datalock."

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, the spreadsheet could, in a single turn, immediate-call `getStatus`, initialize the spreadsheet cell with `getStatus`'s result, and then immediate-call `addListener` to subscribe the cell, so that it will see *exactly* the updates to its initial value. But when the spreadsheet can only eventual-send the `getStatus` and `addListener` messages, they may be delivered to 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)
}
```

14.3 Issues with Event-loops

This architecture imposes some strong constraints on programming (e.g., no threads or co-routines), which can impede certain useful patterns of plan cooperation. In particular, recursive algorithms, such as recursive-descent parsers, must either a) happen entirely within a single turn, b) be redesigned (e.g., as a table-driven parser), c) be transformed into an analog of continuation-passing style shown in Section 18.2, or d) 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.

Symmetric multiprocessors, multi-cores, and hyperthreading provide hardware support for cache coherency between processing units. Although it may still make sense to run a separate vat on each processing unit, to make efficient use of this hardware, we would need to avoid the expense of serializing inter-vat messages. Can multiple vats on tightly coupled processing units gain the performance advantages of shared memory without losing the robustness advantages of large-grain isolated units of operation?

For this purpose, E implementations allow several vats to run in a single address space. Between vats within the same address space, E provides a special purpose comm system, the BootCommSystem, similar in semantics to Pluribus, but implemented by thread-safe pointer manipulations rather than serialization. The BootCommSystem's important optimization is simply that it passes data—any message argument accepted by the `Data` guard—by pointer sharing rather than copying. Since data is transitively immutable, and since a copy of data is indistinguishable from the original, this optimization is semantically transparent. It enables large immutable structures to be cheaply shared.

Nevertheless, the event-loop approach may be unsuitable for some problems with fine-grained parallelism that cannot easily be adapted to a message-passing hardware architecture. For example, we do not see how to adapt the optimization above to allow fine-grained sharing of updates to commonly addressable data.

14.4 Notes on Related Work

Lauer and Needham's *On the Duality of Operating System Structures* [LN79] contrasts "message-oriented systems" with "procedure-oriented systems." *Message-oriented systems* consist of separate process, not sharing any memory, and communicating only by means of messages. The example model presented in their paper uses asynchronous messages. *Procedure-oriented systems* consist of concurrently executing processes with shared access to memory, using locking to exclude each other, in order to preserve the consistency of this memory. The example model presented in their paper uses monitor locks [Hoa74]. Their "procedure-oriented systems" corresponds to the term "shared-state concurrency" as used by Roy and Haridi [RH04] and this dissertation.

Lauer and Needham's "message-oriented systems" does not directly correspond to Roy and Haridi's "message-passing concurrency." Lauer and Needham's "message-oriented systems" includes an "`AwaitReply`" operation by which a process as a whole can block, waiting for a reply from a previously sent message. Although less obviously a blocking construct, their "`WaitForMessage`" operation takes as a parameter the list of ports to wait on (much like Unix `select`). While the process is blocked waiting for a message to arrive on some of these ports, it is unresponsive to messages on the other ports. By contrast, Roy and Haridi's "message-passing concurrency" includes no separate blocking constructs. A process is "blocked" only when it is idle—when its incoming message queue is empty. To avoid confusion, this dissertation uses "event loop" for systems using non-blocking message-passing concurrency.

Lauer and Needham explain various comparative engineering strengths and weaknesses of message-oriented systems and procedure-oriented systems, but show (given some reasonable assumptions) that they are approximately equivalent. Further, they explain how programs written in either style may be transformed to the other. This equivalence depends on the ability of message-oriented programs to block their process awaiting some inputs, leaving it insensitive to other inputs.

Examples of message-oriented systems with blocking constructs are Hoare's *Communicating Sequential Processes* (CSP) [Hoa78], Milner's "Communicating Concurrent Systems" (CCS) [Mil83, Mil89], the "Synchronous π Calculus" [Mil99], and many systems derived from them, such as Concurrent ML [Rep99].

Examples of event loop systems, *i.e.*, non-blocking message-oriented systems, include Actors [HBS73, Hew77], libasync [DZK$^+$02, ZYD$^+$03], and Twisted Python [Lef03] covered in Related Work Section 26.2. Although derived from Actors, the Erlang language [Arm03] covered in Related Work Section 24.2 is a message-passing system in Lauer and Needham's sense: a typical programming pattern is for a process to block waiting for a reply, and to remain unresponsive to further requests while blocked.

As with the distinction between procedure-oriented systems and message-oriented systems, the distinction between (blocking) message-oriented systems and (non-blocking) event loop systems is not always clear. In some ways, the distinction depends on whether certain structures are best regarded as suspended computations or as objects awaiting invocation. Transforming code to continuation-passing style [Hew77, Ste78] converts suspended computations into objects. E's use of this technique, explained in Section 18.2 is still non-blocking because other objects remain responsive while continuation-like objects await their invocations.

In other ways, this distinction depends on whether we regard a change of state in reaction to a message as having processed the message or as having buffered the message. The later Actors work on "receptionists" [Agh86] makes it convenient to buffer messages from some sources while awaiting a response from other sources. Conditional buffering patterns can simulate locking, and thus re-introduce the hazards of "simulated" deadlock.

Due to these issues, the Asynchronous π Calculus [HT91] and Concurrent logic/constraint programming languages [Sha83, ST83, Sar93] can be classified as either blocking or non-blocking message-based systems, depending on the level of abstraction at which they are described.

Chapter 15

Protection from Misbehavior

15.1 Can't Just Avoid Threads by Convention

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`. Under cooperative assumptions, it is indeed adequate to avoid shared-state concurrency merely by careful convention.

Such avoidance-by-convention is even adequate for inter-vat defensive consistency. Even if remote clients of the statusHolder spawn threads in their own vat, they can still only eventual-send messages to the statusHolder. These messages are queued and eventually delivered in the statusHolder's vat thread, insulating it from concern with the internal concurrency decisions of remote vats. For this case, statusHolder relies on its platform only to prohibit inter-vat immediate calls.

However, for object-granularity defensive consistency, it is inadequate to avoid threads merely by convention. A defensively consistent statusHolder must assume that its local clients might spawn threads if they can, and might then immediate-call it from one of these threads. As we have shown, defensive consistency in the face of multi-threading is unreasonably difficult. To relieve statusHolder's programmer of this burden, statusHolder's reliance set must prevent local clients from spawning threads.

E simply prevents all spawning of threads within the same vat. Each local E vat implementation is the platform for the objects it hosts, and therefore part of their reliance set. These objects can validly rely on their local E platform to prevent multi-threading.

15.2 Reify Distinctions in Authority as Distinct Objects

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 problem brings us back to access control. The statusHolder, by bundling two

```
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]
}
```

Figure 15.1: Reify Distinctions in Authority as Distinct Objects. The earlier `statusHolder` provides both the authority to be informed about the current status (whether by query or notification) and the authority to update the status. This encourages patterns which enable supposed subscribers to publish and supposed publishers to subscribe. By representing each coherent bundle of authority with a separate object providing only that authority, we can easily enable supposed publishers only to publish and enable supposed subscribers only to subscribe.

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, as shown by the definition of makeStatusPair in Figure 15.1. The account manager can use makeStatusPair as follows:

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

The call to makeStatusPair on the right side makes four objects—a Slot 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.

This example shows how POLA helps support defensive consistency. We wish to provide objects the authority needed to carry out their proper duties—publishers gotta publish—but little more. 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.

15.3 Notes on Related Work

Roy and Haridi's textbook *Concepts, Techniques, and Models of Computer Programming* [RH04] compare and contrast several models of concurrency, including shared-state concurrency, message passing concurrency (much like our communicating event loops), and declarative concurrency. These are all presented using the Oz language. We have borrowed much terminology and conceptual framing from their book. The Related Work Section 26.3 explains how the derived Oz-E project will attempt to retain both message passing and declarative concurrency while still suppressing shared state concurrency, in order to support defensive consistency.

The phrase "reify distinctions in authority as distinct objects" derives from E. Dean Tribble [Tri98]. Our understanding of this principle largely derives from studying the interfaces designed at Key Logic, Inc. [Key81].

Chapter 16

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 chapter discusses the handling of return results and exceptions produced by eventual-sends.

16.1 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.

16.2 Pipelining

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

```
def r3 := (bob <- x()) <- z(dave <- y())
```

or equivalently

```
def r1 := bob <- x()
def r2 := dave <- y()
def r3 := r1 <- z(r2)
```

Alice says: `def r3 := (bob <- x()) <- z(dave <- y())`

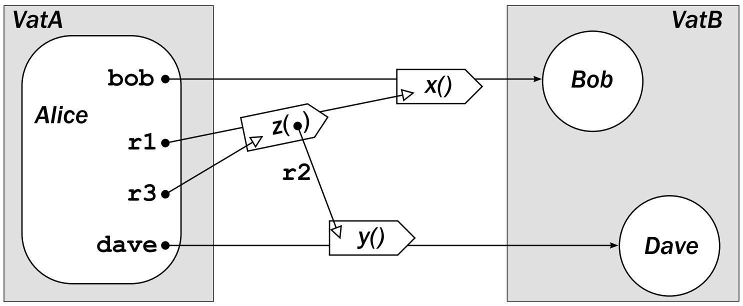


Figure 16.1: Promise Pipelining. The three messages in `def` *r3* `:= (bob <- x()) <- z(dave <- y())` are streamed out together, with no round trip. Each message box "rides" on the reference it is sent on. References `bob` and `dave` are shown with solid arrowheads, indicating that their target is known. The others are *promises*, whose open arrowheads represent their *resolvers*, which provide the right to choose their promises' value.

and `bob` and `dave` refer to objects on VatB, then all three requests are serialized and streamed out to VatB immediately and the turn in VatA continues without blocking. By contrast, in a conventional RPC system, the calling thread would only proceed after multiple network round trips.

Figure 16.1 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 VatB. 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 `x()` is in transit to VatB, so is the resolver for `r1`, so we send the `z(r2)` message there as well. As VatB 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, `z(r2)`, on a local promise that will be resolved by the outcome of `x()`, carrying as an argument a local promise for the outcome of `y()`.

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

16.3 Datalock

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

```
? var flag := true
# value: true

? def epimenides() { return flag <- not() }
# value: <epimenides>

? flag := epimenides <- run()
# value: <Promise>
```

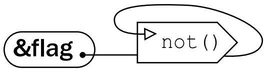


Figure 16.2: Datalock. These three expressions together assign to `flag` a promise for the result of sending `not()` to the resolution of this very promise. This creates the illustrated circular dependency, in which the `not()` message is forever buffered in the promise which only `not()`'s delivery could resolve.

primitives introduced so far, 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 defined in Figure 16.2 returns a promise for the boolean opposite of `flag`. If `flag` is assigned to the result of invoking `epimenides` eventually, datalock occurs: the promise stored in `flag` will never resolve, and any messages eventually-sent to it will never be delivered.

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 `not()` message to the promise in `flag`, and then resolves the `flag` promise to the new promise for the result of 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 robust. 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.

16.4 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 the asyncAnd example in Figure 18.1 (p. 130) will demonstrate.

Like the caretaker of Section 9.3, promises provide temporal control of authority, but in the opposite direction. The caretaker's *control facet* (the gate) can be used to terminate the authority provided by its *use facet* (the caretaker). By contrast, the promise's control facet (the resolver) can be used to delay the granting of the authority provided by its use facet (the promise).

16.5 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 dataflow exceptions was inspired by signaling and non-signaling NaNs in floating point [Zus41, IoEE85, Kah96]. 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.

16.6 Notes on Related Work

The notion of a delayed reference in software seems to originate with the dataflow architectures of Richard Karp and Raymond Miller [KM66] and Rodriguez [Rod67]. Bredt's survey [Bre73] covers this and related early work. In our terms, pure dataflow models provide for parallelism but not for concurrency control. These pure models are constrained to provide the confluence property: the value computed is insensitive to the order in which intermediate computations happen. Confluent models show how to use concurrency for speedup while keeping the semantics concurrency-free. These notions migrated into pure LISP as "promises" through the works of Friedman and Wise, for both lazy evaluation [FW76a] and eager evaluation [FW76b].

Baker and Hewitt's "futures"' [BH77] adapt pure LISP's promises to the Actors context, which can express concurrency control. Futures are references to values that have not been computed yet, where a separate "process" is spawned to compute this value. Actors futures were adapted by Halstead for Multilisp, a parallel Scheme [Hal85]. All these forms of delayed reference made no provision for representing exceptions. An attempt to use a delayed reference before it was computed would block, rather than pipeline. Delayed references were spawned only by lazily or eagerly evaluating an expression. There was no resolver-like reification of the right to determine what value the delayed reference resolves to. Multilisp uses shared-state concurrency, and so uses locking to prevent conflicts on shared data structures.

Concurrent logic [Sha83] and constraint [Sar93] programming languages use logic variables as delayed references. In these languages, an unannotated logic variable provides both a delayed right to use a future value, and the right to determine (by unification) what this future value is. Different languages of this ilk provide various annotations to distinguish these two roles. Concurrent Prolog's read-only annotation (indicated by "?") creates a read-only capability to the logic variable [Sha83]. A unification which would equate it with a value (a non-variable) would block waiting for it to be bound. Vijay Saraswat's ask/tell framework [Sar93] distinguishes unifications which "ask" what the current bindings entail *vs.* unifications which "tell" the binding what constraints they must now entail. Both of these efforts inspired the stricter separation of these rights in the Joule language [TMHK95], covered in Related Work Section 23.4. E derives its separation of promise from resolver from Joule's channels.

The promise pipelining technique was first invented by Liskov and Shrira [LS88] and independently re-invented as part of the Udanax Gold system mentioned in Related Work Section 23.5. These ideas were then significantly improved by Bogle [BL94]. These are asymmetric client-server systems. In other ways, the techniques used in Bogle's protocol quite closely resemble some of the techniques used in Pluribus.

In 1941, Konrad Zuse introduced a NaN-like *undefined* value as part of the floating point pipeline of one of the world's earliest computers, the Z3 [Zus41]. Zuse was awarded a patent on pipelining in 1949 [Zus49]. According to the CDC 6600 manual [Con67], its NaN-like "indefinite" value was optionally contagious through further operations. However, Kahan's *Lecture Notes on the Status of the IEEE Standard 754 for Binary Floating Point Arithmetic* [Kah96], which credits Zuse's "undefined" as the source of the NaN idea, claims that Zuse's "undefined" and the CDC 6600's "indefinite" values were not non-signalling. Kahan credits the non-signalling NaN to the IEEE floating point standard [IoEE85].

Chapter 17

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 that is unable 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 [BJ87, Ami95, Lam98] 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 Section 25.1.

E's support for partial failure starts by extending the semantics of our reference states. Figure 17.1 shows the full state transition diagram among these states. This diagram uses David Harel's statechart notation [Har88b], which combines node-arc diagrams and so-called Venn diagrams (both of which, Harel points out, were invented by Euler). On our diagram, each bold oval represents a singleton set, containing only its named state. Each rounded rectangle represents the set of states it spatially contains, providing a taxonomy. A transition arrow from set X to set Y says that any of the states in set X may make the indicated transition to any of the states in set Y. For example, given our taxonomy, when we say "An unresolved reference may *resolve* to a resolved reference." this is shorthand for "A local promise or a remote promise may *resolve* to a near, far, or broken reference."

In addition to the concepts developed in previous chapters, 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 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

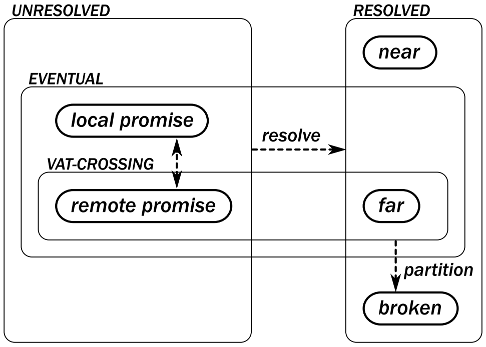


Figure 17.1: Reference States and Transitions. A resolved reference's target is known. Near references are resolved and local; they carry both immediate-calls and eventual-sends. Promises and vat-crossing references are eventual; they carry only eventual-sends. Broken references carry neither. Promises may *resolve* to near, far or broken. *Partitions* break vat-crossing references.

relieves the sender from needing to wait for earlier messages to be acknowledged before sending later dependent messages.[1]

On our state-transition diagram, 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.

A 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. A `statusChanged` notification sent to a broken listener reference is harmlessly discarded.

17.1 Handling Loss of a Provider

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)
}
```

[1] The message delivery order E enforces, E-ORDER, is explained in Chapter 19. E-ORDER is stronger than fail-stop FIFO, but fail-stop FIFO is adequate for all points we make in this chapter.

The `__whenBroken` message is one of a handful of universally understood messages that all objects respond to by default.[2] 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 Chapter 18.

`__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. In addition to the explanation here, see also the example in Section 11.5.3.

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(thisBrokenRef) }
```

When a local promise gets broken, all its messages are forwarded to the broken reference; when the `__whenBroken` message is delivered, 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 has been exceeded), it breaks all outgoing references and notifies all registered handlers.

For all the reasons previously explained, the notification 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 separated from normal operation.

17.2 Handling Loss of a Client

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

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

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 revisit the *offline capabilities* introduced in Section 7.4.

17.3 Offline Capabilities

An offline capability in E has two forms: a "`captp://...`" URI string and an encapsulated `SturdyRef` object. Both are pass-by-copy and can be passed between vats even when the vat of the designated 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.

In E, 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.

17.4 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.[3] Once one makes an offline capability to a given

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

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 incarnation crashes, making the association *transient.* An association which is not transient is *durable.* Here, we examine only the transient *vs.* durable option.

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

17.5 Notes on Related Work

Atkinson and Buneman's *Types and Persistence in Database Programming Languages* [AB87] surveys early work in persistent programming languages. The first orthogonally persistent languages are Smalltalk-80 [GR83] and PS-Algol [ACC82].

KeyKOS [Har85] is an early orthogonally persistent operating systems. KeyKOS is not distributed. Each checkpoint is a globally consistent snapshot of the entire system image. Nevertheless, after a crash, it still must cope with consistency issues regarding devices. E's separation of disconnection concerns from reconnection concerns derives directly from KeyKOS's handling of access to devices [Key81, Lan92].

KeyKOS provides access to devices by *device capabilities* and *device creator capabilities.* A user-level device driver interacts with a device using only a device capability. A revived user-level device driver blindly continues executing the plan it was in the midst of when last checkpointed, assuming that the device's state is as it was then. However, this is harmless. On revival from a checkpoint, all device capabilities are revived in a permanently invalidated state, just as a partition in E permanently breaks all vat-crossing references. When the device driver notices that its device capability is invalid, it asks the device creator capability to give it a new device capability, much like an E program will ask an off-line capability for a new reference. The plan for reconnection must determine the device's current state, in order to reestablish consistency between the device and its driver. The plan for using the device via the device capability may then safely assume that the device's state follows from its own manipulations. Its ability to use that device capability lasts only so long as these assumptions remain valid.

Chapter 18

The When-Catch Expression

The __whenMoreResolved message can be used to 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 is fulfilled (becomes near or far), and a "catch" block to execute if the promise is broken. This is illustrated by the asyncAnd example in Figure 18.1.

The asyncAnd function 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.

Promise pipelining postpones plans efficiently, in a dataflow-like manner, delaying message delivery until a message's recipient is known. The when-catch delays plans until the information needed for control flow is available.

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 eventually resolves the already-returned promise to true. If it is notified that any resolve to false, asyncAnd resolves this promise to false immediately. If any resolve to broken, asyncAnd breaks this promise with 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 asyncAnd code may resolve the promise to be either false or broken, depending on which handler happens to be notified first.

The snippet shown in Figure 18.2 illustrates using asyncAnd and when-catch to combine independent validity checks in a toy application to resell goods from a supplier.

```
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) {
                if ((countDown -= 1) == 0) { resolver.resolve(true) }
            } else {
                resolver.resolve(false)
            }
        } catch ex { resolver.smash(ex) }
    }
    return result
}
```

Figure 18.1: Eventual Conjunction. The `asynchAnd` function eventually resolves the returned promise to be the conjunction of a list of promises for booleans. Should all resolve to true, the returned promise will resolve to true. If any resolve to false or broken, the returned promise will resolve to false or likewise broken without waiting for further resolutions.

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

Figure 18.2: Using Eventual Control Flow. In this toy purchase order example, we send out several queries and then use `asyncAnd` so that we only buy the goods once we're satisfied with the answers.

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

Figure 18.3: The Default Joiner. The when-catch expression provides a syntactic shorthand for using this joiner, which postpones plans until either all its inputs are fulfilled or any of them break.

18.1 Eventual Control Flow

Concurrency control in E is often expressed by creating and using eventual-control-flow abstractions. The `asyncAnd` is an example of a user-defined *joiner*—an abstraction for waiting until several causal inputs are ready before signalling an output.

Eventual-sends delay delivery until a message's recipient is known, but, by itself, it does not provide full dataflow postponement; it does not also delay until arguments are resolved. For example, if `partNo` is an unresolved promise, then

```
def receipt := supplier <- buy(partNo, payment)
```

may deliver the `buy` message to the supplier before `partNo` is resolved. If the supplier requires a resolved `partNo` at the time of purchase, it may reject this `buy` request, and `receipt` would become broken. To support stateful programming with minimal plan disruption, this is the right default, as full dataflow delays conflict with the need to deliver successive messages sent on the same reference in fail-stop FIFO order.

Fortunately, either the purchaser or supplier can use a when-catch expression to express such additional delays. To make this easier, the value of a when-catch expression is a promise for the value that the handler-body will evaluate to. Further, if the catch-clause is left out, it defaults to re-throwing the exception, thereby breaking this promise.

```
def receipt := when (partNo) -> { supplier <- buy(partNo, payment) }
```

is equivalent to

```
def receipt := when (partNo) -> {
    supplier <- buy(partNo, payment)
} catch ex { throw(ex) }
```

is equivalent to

```
def [receipt, r] := Ref.promise()
when (partNo) -> {
    r.resolve(supplier <- buy(partNo, payment))
} catch ex { r.smash(ex) }
```

Of course, sometimes we wish to delay an action until several relevant promises resolve. Figures 18.1 and 18.2 show how to define and use joiner abstractions for this purpose. Figure 18.3 shows E's default joiner abstraction: Given a list of promises, if they all become fulfilled (near or far), then `promiseAllFulfilled` will resolve its returned promise to this same list. If any of the list's elements become broken, then `promiseAllFulfilled` will break its returned promise with the same exception. The when-catch expression provides syntactic support for this joiner: If multiple expressions appear in the head of a when-catch expression, these turn into a call to `promiseAllFulfilled` with a list of these expressions as its argument.

```
def receipt := when (partNo, payment) -> {
    supplier <- buy(partNo, payment)
}
```

is equivalent to

```
def receipt := when (promiseAllFulfilled([partNo, payment])) -> {
    supplier <- buy(partNo, payment)
}
```

18.2 Manual Continuation-Passing Style

Section 14.3 mentions a common programming pattern that is particularly challenging for non-blocking event-loops: a recursive descent parser that might block while waiting on input, such as a command-line interpreter waiting for the user. There, we mention four possible approaches. Here, we expand on the third of these approaches: manually transforming the program into an analog of continuation-passing style.

Conventional systems have the option of running this parser code in a thread, and running other threads while this one is blocked. If the threading system is preemptive, we'd have all the problems of shared-state concurrency previously explained. What about non-preemptive threading systems, in which each thread operates as a co-routine?

An oft-cited expressiveness advantage of such co-routining is that it allows a plan to be suspended mid-flight while it is blocked waiting on some external input [AHT$^+$02], exactly as we seem to need here. The typical *implementation* of a suspended co-routine is as an inactive call-stack. To explain the *semantics* of setting a stack aside and resuming it later, one imagines that the program is instead transformed into continuation-passing style [Hew77, Ste78], where each call site is transformed to pass an extra continuation argument representing the "rest of the computation." (The continuation concept in the semantics mirrors the implementation's concept of the saved return address and frame pointer.) In this transformation, the remaining work that would be done by the caller, and its caller, etc., once the callee returns, is reified as a newly created continuation object—typically a function of one parameter representing the value to be returned.

For example, let us say `getint()` might block waiting on input. We could then explain

```
def foo() { return bar(getint(), y()) }
```

by transforming it into continuation-passing style:

```
def foo(c1) {
    getint(def c2(i) { y(def c3(j) { bar(i, j, c1) }) })
}
```

With this version, `getint` can suspend the rest of the parsing activity by simply storing its continuation argument, and then enable other co-routines to be scheduled in the meantime. When `getint` is ready to allow the parse to resume, it simply invokes the previously stored continuation, rescheduling this co-routine. Even in a sequential system that provides no built-in co-routining, programmers can still achieve the effect by manually transforming code as above; but the results are typically as unmaintainable as the above example would suggest [AHT+02, vBCB03]. Such violent transformations to code structure, in order to accomodate the absence of built-in co-routining, have been termed *stack ripping* by Adya *et al.* [AHT+02].

To account for exceptions, a continuation can be a two-method object rather than a function. Say the two methods are `resolve(result)` and `smash(ex)`. We would then explain a return as the callee invoking its continuation's `resolve` method. We would explain throwing an exception as invoking the continuation's `smash` method. All control-flow then becomes patterns of object creation and one-way message sending. If written out manually, such explicit exception handling makes stack ripping that much worse.

E does not provide transparent co-routining within a vat, since this would disrupt a caller's plan assumptions. Between a call and a return, if the callee could suspend and resume, any co-routine which may have been scheduled in the meantime could affect vat state. If this could happen during any call, then programmers would again need to face the hazards of plan interleaving throughout their programs [DZK+02].

Notice that co-routine interleaving would be much less hazardous if the return points of all calls to possibly-suspending procedures had to be specially marked in the source text, and if the return points of all calls to procedures containing such marks therefore also had to be specially marked [AHT+02].

```
def foo() { return bar(getint()↗, y()) }
...foo()↗...
```

The programmer could then recognize these marks as the points in the program when other vat-turns might interleave [AHT+02]. Without such a mark-propagation rule, functional composition would hide these interleaving points from callers. The "`->`" symbol following the "`when`" keyword effectively serves as such a mark.

```
def foo() { return when (def i := getint()) -> { bar(i, y()) } }
...when (foo()) -> {...}
```

The handler it registers, to be called back when the promise is resolved, reifies the desired portion of the rest of the current procedure into an explicit object. Instead of a possibly-suspending procedure which eventually returns a result, we have a promise-returning procedure which eventually resolves this promise to that result. An eventually-sent message implicitly carries a resolver, serving as a continuation, to which the outcome (result or thrown exception) of delivering the message will be reported. Together, promises, resolvers, eventual-sends, and the when-catch expression give the programmer the ability to spread a plan over mutiple vat turns, a bit less conveniently but more flexibly, while still remaining explicit *at every level of the call stack* about where interleaving may and may not occur.

18.3 Notes on Related Work

Carl Hewitt's *Viewing Control Structures as Patterns of Passing Messages* [Hew77] introduced the idea of reified continuations into programming languages, and the notion that call-return patterns could be transparently re-written into continuation-passing style. Hewitt's treatment was in an event-based massively concurrent system. In a sequential context, Guy Steele's "Rabbit" compiler for the Scheme language [Ste78] was the first to use this transformation in a compiler, in order to reduce cases and generate higher performance code.

Adya *et al.*'s *Cooperative Task Management Without Manual Stack Management* [AHT+02] expands the dichotomy between "multi-threaded" and "event-driven" programming into a finer grained taxonomy involving five distinct dimensions. By "multi-threaded," they mean what is here termed "shared-state concurrency." The two dimensions the paper focuses on are the two most relevant to our present discussion: *task management*, which they divide into *cooperative*, and *preemptive*, and *stack management*, which they divide into *manual* and *automatic*. Multi-threaded programs use preemptive task management and automatic stack management. Event-driven programs use cooperative task management and manual stack management. Non-preemptive co-routine scheduling systems, which they advocate, use cooperative task management and automatic stack management.

Their paper explains the danger of hiding interleaving points from callers, and proposes a static marking rule similar in effect to the "↗" annotation presented above. The paper states explicitly that it is concerned with "stack management problems in conventional languages without elegant closures." Although it is not clear how the authors would regard E, we find their taxonomy useful, and regard E as event-driven: with cooperative task management and manual stack management. Although E's when-catch prevents stack ripping, combining much of the syntactic convenience of co-routines with an explicit marking rule as their paper suggests, it also reifies these delayed results as first class promises, which can be stored in data structures, passed in messages, or eventually sent to, providing a level of expressive power well beyond that available by co-routine scheduling.

Regarding the remaining dimensions of their taxonomy, E provides both *synchronous I/O* and *asynchronous I/O* operations, though ideally we should remove synchronous I/O from E. E provides no explicit *conflict management*, *i.e.*, no explicit locking constructs, since, as they observe, cooperative task management implicitly provides programmer-defined large-grain atomicity. Promises do serve as a form of conflict management, as they can buffer messages until an object holding the resolver decides that it is now safe to release them. Finally, E *explicitly partitions* all mutable state into separate vats, so that no two threads of execution (vats) ever have synchronous access (near references) to shared state.

The "libasync" and "libasync-mp" libraries [DZK+02] are event-driven libraries in C++. Although C++ does not provide closures, these libraries use C++ templates to alleviate the inconvenience of stack ripping. In these libraries, an asynchronous call takes an explicit continuation-like callback as an argument, rather than returning a promise. Unlike E's multi-vat approach, the libasync-mp library extends libasync to make efficient use of shared memory multiprocessors. Rather than explicitly partition objects into vats, libasync-mp aggregates events into colors. No two events of the same color will execute concurrently, but an event of one color may execute on one processor while an event of a different color executes on a different processor. In the absence of explicit coloring by the programmer, all events carry the same default color. However, it seems that if the programmer does use colors to obtain a speedup, it is the programmer's responsibility to ensure, without resort to locking constructs, that differently colored events do not interfere.

In the Joule language [TMHK95], covered in Related Work Section 23.4, one receives a response from a message by passing in a *distributor* object as an explicit callback and holding onto the corresponding *acceptor*. Joule's acceptor/distributor pair is much like E's promise/resolver pair. Joule provides a syntactic shorthand where the last argument can be elided when the call expression is used in a syntactic context needing a value. This case expands to explicit creation of an acceptor/distributor pair, adds the distributor to the argument list, and uses the acceptor as the value of the expression as a whole. The Oz language [RH04] provides a similar shorthand for passing a logic variable as the last argument. Although distinguished by convention, these last arguments are not otherwise special.

The E eventual send also implicitly generates a promise/resolver pair to represent the return result, with the resolver passed as a hidden argument in the message, and the promise serving as the value of the send expression. However, in E, this is not just a shorthand; the hidden resolver argument is indeed special in two ways. First, when the eventual send in question is on a remote reference, the serialized message encodes a resolver to be created in the *receiving vat*, so that messages sent to its promise in the sending vat are pipelined to that destination. Second, if a partition prevents the message from being delivered or acknowledged, the promise for its result is automatically broken. By contrast, if a resolver is passed as a normal message argument, it remains where it is created, and only a remote reference to that resolver is actually passed. Partition would cause a `__reactToLostClient` message to be delivered to that resolver, which resolvers by default ignore.

Twisted Python [Lef03], covered in Related Work Section 26.2, is another event-driven language, with cooperative task management and manual stack management. It makes heavy use of Python's support for closures to alleviate the inconvenience of stack ripping. Its primary plan postponement abstraction manages callbacks by chaining them into multi-turn sequences. Each callback in a chain consumes a value produced by its predecessor, and each produces a value to be consumed by its successor. This approach resembles a nested sequence of when-catch blocks without the syntactic nesting cost.

Chapter 19

Delivering Messages in E-ORDER

19.1 E-ORDER Includes Fail-Stop FIFO

Among messages successively sent on a single reference using the eventual send operator, E guarantees fully order-preserving delivery, or *fail-stop* FIFO. All messages are delivered in the order sent unless and until a partition breaks the reference. Once the reference breaks, no further messages are delivered. Therefore, if a particular message does get delivered, E guarantees that all messages sent earlier on the same reference were already delivered.

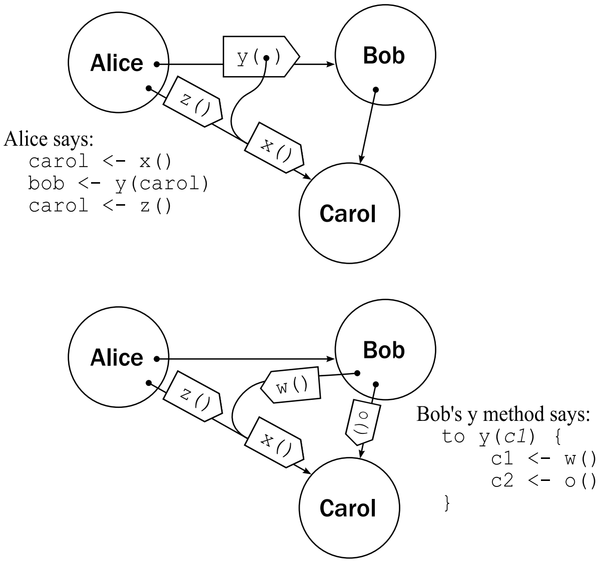
As a result, once Alice has sent the message `x()` to Carol, Alice should think of the reference she is now holding as a reference to "post-x() Carol," *i.e.*, the Carol that has already seen `x()`. This is true even though Carol has not yet seen `x()` and may never see `x()`. Although Carol hasn't seen `x()` yet, using this reference Alice no longer has any ability to deliver a message to Carol before Carol sees `x()`.

(Note: If Alice and Carol are in the same Vat and Alice has a near reference to Carol, Alice can still immediately call Carol and deliver a message before messages which were eventually sent earlier.)

19.2 FIFO is Too Weak

The above ordering guarantee is quite useful for Alice—it allows the Alice-to-Carol protocol to be naively stateful. However, fail-stop FIFO by itself is too weak. As explained above, after Alice sends `x()` on her reference to Carol, this reference represents, to her, only the ability to talk to the Carol that has already received `x()`. In the simplified Pluribus protocol explained in Chapter 7, if Alice sends that reference to Bob in message `y(carol)`, and Bob uses that reference to send message `w()` to Carol, the happenstance of variable network delays may result in `w()` being delivered before `x()`. The reference as handed to Bob gave Bob a dangerous possibility—of delivering a message `w()` ahead of `x()`—that was beyond Alice's notion of the reference's meaning. Why is this possibility dangerous?

As an example, suppose that Carol is a collection, the `x()` is an update message that deletes an entry from the collection, and that `w()` is a query message. Alice knows that if she sends queries following an update, then the queries, if they are delivered, will only be delivered after the updates. However, if she sends Bob `y(carol)` in order to delegate some of this querying activity to Bob, within a system providing only fail-stop FIFO, Bob's query may be delivered before Alice's update, and retrieve from Carol the entry Alice assumed



Alice says:

```
carol <- x()
bob <- y(carol)
carol <- z()
```

Bob's y method says:

```
to y(c1) {
     c1 <- w()
     c2 <- o()
}
```

Figure 19.1: Forks in E-ORDER. By placing messages on references in the order they are sent, and by placing reference-forks according to when the reference was eventually-sent as an argument, we obtain a Hasse diagram [Ski90] of the resulting constraints on message delivery order. Since both `o()` and `x()` are next to Carol, either may be delivered next. Once a message is delivered, we erase it and simplify the diagram accordingly. Once `x()` is delivered, then both `z()` and `w()` would be next to Carol, and so would be eligible to be delivered.

was inaccessible. Even under cooperative assumptions, this is dangerous.

19.3 Forks in E-ORDER

Instead, when a reference is included as an argument of an eventually sent message (as Alice's reference to Carol is included in message `y(carol)`), we say the reference is *forked*. Bob does not get the reference Alice sent, but a fork of this reference. When a fork occurs between two sending events, we diagram the fork-position on the reference between these two messages. In Figure 19.1, Alice sends, in sequence, `x()`, `y(carol)`, and `z()`. Bob reacts to the `y(carol)` message by eventual-sending `w()` to the arriving argument, and by eventual-sending `o()` on a reference he already had to Carol.

To enforce E-ORDER, VatA must not communicate to VatB the permission to access Carol herself. VatA must communicate only a permission to communicate to a post-`x()` Carol, i,e., VatA must send a cryptographic encoding of a post-`x()` fork of the Carol reference that provides the recipient no feasible way to access Carol too early. (The actual protocol is an adaptation of the introduction-by-shortening protocol originally proposed for the Client Utility Architecture [KGRB01].)

By following these diagramming rules, we visualize (approximately as a Hasse diagram [Ski90, HP97]) the partial ordering guarantees E provides. The tree of messages connected by a reference topology is the partial order itself.

As of any diagrammed state, the messages that may be delivered to Carol are those that have no messages ahead of them in the Hasse diagram. Here, these are only messages `x()` and `o()`. Once `x()` is delivered, we erase it and simplify the diagram. This results in both `w()` and `z()` becoming adjacent to Carol, and so becoming candidates to be delivered next. Either choice is consistent with the specification.

With these rules, the reference Bob receives from Alice has no more power in Bob's hands than it had in Alice's. The assumptions Alice needs to make for herself, for the sake of her own sanity, are assumptions that remain valid as she delegates to Bob.

Note that the Hasse-diagram topology is in the specification only, not in the implementation. The implementation is free to deliver messages to Carol in any full order consistent with the specified partial order. It can therefore collapse partial orders to full orders whenever convenient, consistent with these constraints.

19.4 CAUSAL Order is Too Strong

CAUSAL order would provide all the guarantees listed above and more. In our example scenario, it would also guarantee that `o()` is delivered to Carol only after `x()`, since Bob sent `o()` in reaction to `y()`, and `y()` was causally after `x()`. If E enforced CAUSAL or stronger orders [BJ87, Ami95], programmers would only need to handle a smaller number of cases. E doesn't provide CAUSAL order because we don't know how to enforce it among mutually defensive machines.

In the example, `VatB` already had access to Carol by virtue of the reference held in Bob's `c2` variable. A message eventually-sent by `VatB` on this reference already had the possibility of being delivered to Carol ahead of Alice's `x()` message. Enforcing CAUSAL order would require, in the case where `VatB` sends `o()` on `c2` *in reaction to* the arrival of `y()`, that `o()` must then be delivered only after `x()`. In order to enforce CAUSAL order on a possibly misbehaving `VatB`, somehow, the arrival of `y(carol)` from `VatA` would have to preclude this previously present possibility. In the absence of mutually-reliant hardware [ST94], this seems difficult.

By contrast, E-ORDER only requires restricting the new possibilities the newly arriving reference-to-Carol provides to `VatB`, rather than the removal of previously present possibilities.

19.5 Joins in E-ORDER

E has no eventual equality primitive. Rather, it has an immediate equality primitive, "`==`", which can only be applied to resolved references. Using this primitive, eventual equality can be programmed in E as yet another joiner pattern, as shown in Figure 19.2. Given two references, the `join` function gives us a promise for the one object they (hopefully) both designate. If both references do eventually come to designate the same object, `join` fulfills this promise, creating a join in the message-ordering constraints.

Given `a` and `b`, then `def c := join(a, b)` defines `c` as a promise for the one object they both designate. We may immediately start sending messages on `c`, confident that these messages will only get delivered if this promise is fulfilled. If either `a` or `b` eventually break, or if then both eventually resolve to refer to different objects, then the promise for their join will eventually be broken. In this case, all messages sent to it will be discarded using the

```
def join(left, right) {
    return when (left,right) -> {
        if (left == right) { left } else { throw("unequal") }
    }
}
```

Figure 19.2: Eventual Equality as Join. The `join` function returns a promise for the object that both `left` and `right` will designate. If `left` and `right` both eventually designate the same object, then the returned promise will become a resolved reference to that object. If not, then this promise will be broken by an alleged explanation of what went wrong. Messages sent on this promise will not be delivered until after all messages that had been sent on `left` and `right` prior to the `join` have already been delivered to this object.

usual broken promise contagion rules. Because `join` returns the promise for the result of a when-catch, all messages sent to this promise are buffered locally, in the vat where `join` was called, rather than being pipelined to their expected destination. This avoids speculatively revealing messages to a vat before determining whether that vat should have access to that message.

The `c` promise is a fork of `a` and a fork of `b`. Given the sequence of actions shown in Figure 19.3 if `a` and `b` are independent references to the same object, and assuming no partition occurs, then:

- `u()` and `v()` may be delivered in any order.
- `w()` may only be delivered after `u()` is delivered.
- `x()` may only be delivered after `u()` is delivered, as implied by "`c` is a fork of `a` after `u()`".
- `x()` may only be delivered after `v()` is delivered, as implied by "`c` is a fork of `b` after `v()`".
- `y()` may only be delivered after `v()` is delivered.
- `w()`, `x()`, and `y()` may be delivered in any order.
- All these messages are delivered at most once.

If `a` and `b` do not designate the same object, then all the above statements hold except that `x()` must not be delivered. In addition, `c` must eventually resolve to broken.

Should a partition occur, all the above statements continue to hold anyway, but not in the obvious way. For example, should `v()` be lost in a partition, never to be delivered, then `x()` and `y()` must never be delivered, and both `b` and `c` must eventually become broken, as implied by "`c` is a fork of `b` after `v()`".

```
a <- u()
b <- v()
def c := join(a, b)
a <- w()
c <- x()
b <- y()
```

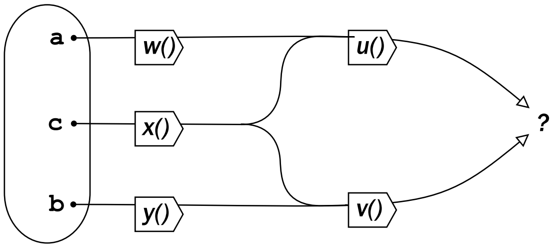


Figure 19.3: Joins in E-ORDER. A `join` returns a single reference that is a join of the forks of each of its arguments. A message sent on the joined reference may only be delivered after all messages that were already sent ahead of these fork points. Once `u()` and `v()` are delivered and erased, if they have the same target, then the diagram simplifies so that `x()` would be eligible to be delivered next. If they have different targets, then `x()` will never be eligible.

19.6 Fairness

Our ordering constraints above define when a message is eligible to be delivered next. Of all the messages eligible to be delivered to any of the objects hosted by a given vat, E-ORDER allows any to be delivered next. However, once a message is eligible, it must eventually be delivered or dropped. It may only be dropped as a result of a partition, whose consequences have already been explained. By *eventually*, we do not mean that there must be a future time at which this delivery will occur, since an infinite turn can prevent that. Rather, we define E-ORDER to preclude the possibility that an infinite number of other eligible messages will be chosen ahead of a message that is already eligible. This definition of fairness is an adaptation of Clinger's definition in terms of unbounded non-determinism [Cli81].

19.7 Notes on Related Work

Lamports's *happened before* relationship in *Time, Clocks, and the Ordering of Events in a Distributed System* [Lam78] and Hewitt and Baker's *ordering laws* in *Actors and Continuous Functionals* [HB78] (further debugged and refined by Clinger's *Foundations of Actor Semantics* [Cli81]) represent contemporaneous discoveries of similar adaptations of special relativity's partial causal order of events [Sch62] to describe the order of events in distributed systems. For both, the overall partial order comes from combining two weaker orders, and then taking the transitive closure of the combination. The role played by Hewitt and Baker's "actor" is the same as Lamport's "process" and E's "vat." Using Hewitt and Baker's terminology, the two orders are the "activation order" and the "arrival order."

The *arrival order* is the linear sequence of events within an actor (process/vat). Each event depends on the actor's current state, and leaves the actor in a new state, so each event *happens before* all succeeding events in the same actor. The *activation order* says that the event from which actor Alice sent message `foo` *happens before* the event in which that message is delivered to actor Bob. The closure of these two orders taken together is the partial causal order of events in a distributed system. When all inter-actor causality of possible interest is modeled as occuring by such messages, then one event can only influence another if the first happens before the second.

This partial causal order is descriptive: it describes any physically realizable system of computation. In the taxonomy of distributed system message delivery orders, in the absence of further constraints, this would be termed UNORDERED, UNRELIABLE. It is UNORDERED because the only ordering constraint between when a message is sent and when it is delivered is that it cannot be delivered before it is sent—*happened before* cycles are assumed impossible. It is UNRELIABLE because sent messages may not be delivered. However, it is assumed that delivered messages were sent. (Here, we will further assume that a sent message is delivered at most once.)

CAUSAL and stronger orders are usually explained in the context of reliable muticast and group communications systems. Ken Birman's *Reliable Distributed Systems* [Bir05] examines in detail the field of Group Membership protocols and the hierarchy of message delivery orders these systems generally provide. A more formal survey can be found in *Group Communication Specifications: A Comprehensive Study* [VCKD99]. Défago and Urbán's *Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey* presents a taxonomy of systems providing AGREED and stronger orders in the context of multicast communication systems, including but not limited to group communications systems.

Although usually presented in the context of multicast, these various message delivery orders can be used to describe unicast systems, such as that presented in this dissertation. In the unicast context, CAUSAL order guarantees that if the event sending `m1` happens before the event sending `m2`, and if both are delivered, then the event in which `m1` is delivered happens before the event in which `m2` is delivered. When `m1` and `m2` are delivered to the same actor, this is directly prescriptive: it says that `m1` must actually arrive first at the recipient. This arrival event must appear first in the recipient's arrival order. When they are delivered to different actors, this constraint is only indirectly prescriptive: it adds an ordering constraint to the happens before relationship, which must remain acyclic as we take the closure.

Protocols for CAUSAL and stronger orders are cooperative. Participating machines are mutually reliant on each other to operate according to the protocol. Smith and Tygar's *Security and Privacy for Partial Order Time* [ST94] examines the security hazards that result when dishonest machines undetectably violate CAUSAL order by violating the protocol designed to provide CAUSAL order. They examine the issues involved in designing a purely cryptographic protocol for enforcing CAUSAL order, including their own previous attempts. They conclude that tamper-resistant mutually-reliant hardware is needed, and then show how to use such hardware to enforce CAUSAL order among machines constrained to communicate only through such hardware.

Conventional FIFO order can be defined in terms of pairs of actors (processes/vats): If Alice sends `m1` and then `m2` to Bob, so that the sending of `m1` happens before the sending of `m2` *in Alice's local arrival order*, and if both are delivered to Bob, then the delivery of `m1` must happen before the delivery of `m2`. Otherwise, no additional constraints are imposed.

This construction is stronger than FIFO order as provided by TCP [Pos81], since TCP only provides FIFO order separately within each communications channel. Rather than requiring `m1` and `m2` merely to be sent by the same Actor, TCP additionally requires that they be sent on the same channel. We can visualize the channel as carrying a linear chain of messages which have been sent but not yet delivered. Sending adds to the tail end of the chain. Only the message at the head is eligible to be delivered next. If an actor receives from multiple chains, then, except for possible fairness issues [Cli81], the order in which their messages are interleaved is not further constrained.

Once we introduce channels, we can then define *fail-stop* as the guarantee that a message sent later on a channel will only be delivered if all messages sent earlier on the same channel will eventually be delivered. Without perfect foresight, the only way to ensure that they will be delivered is if they were delivered, so fail-stop will normally imply at least FIFO order. Fail-stop channels are like degenerate two-party group views [Ami95], and channel failure reports are like group-view membership messages.

An end of a TCP channel is not normally considered to be transmissable between machines, so the issue of multiple machines sending on the same channel does not arise. For references, this issue does arise. Shapiro and Takeuchi's *Object Oriented Programming in Concurrent Prolog* [ST83] was the first to explore the tree-order subset of E-ORDER. An object reference to Carol was modeled as an incomplete list of logic variables. Alice sent `x()` to Carol by unifying the tail of the list with a pair of the `x()` message and a new unbound variable representing the new list tail. The sender would then hold onto this new list tail *as* the new reference to post-`x()` Carol. However, this replacement was manual and error prone: only the individual logic variables were first class, not the references defined by this pattern of replacement. Alice and Bob could not usefully share a logic variable as a means to communicate with Carol, since whoever uses it first uses it up, preventing its use by the other.

Using Shapiro and Mierowsky's *Fair, Biased, and Self-Balancing Merge Operators: Their Specification and Implementation in Concurrent Prolog* [SM87], when Alice wishes to send Bob a reference to Carol, she sends a `merge()` message to Carol, corresponding to the fork points shown in Figure 19.1. She then sends the new variable representing this merged-in reference to Bob. However, these references were still not first-class, as they could only be shared by explicit merging. Tribble *et al.*'s *Channels: A Generalization of Streams* [TMK$^+$87], explained briefly in Related Work Section 23.3, repaired these difficulties, and removed the need for the manual tail-replacement and merge messages.

Process calculi such as the π calculus [Mil99] provide elegant frameworks for specifying and reasoning about many of the properties of message-passing systems. However, so far as we are aware, none has succeeded in specifying fairness. Benjamin Pierce's "Pict" language [PT00] does provide fairness, but the specification of fairness is not captured in the π calculus description of the language.

Concurrent Prolog [SM87], Actors [Cli81], and E-ORDER all specify fairness. Fairness is necessary for E programs to satisfy the liveness requirement we state in Section 5.7 as "Defensive progress up to resource exhaustion." The qualifier "up to resource exhaustion" acknowledges that fairness does not prevent denial of service attacks. Fairness helps guarantee only that other clients cannot slow service to a given client down to zero. These other clients can still cause service to get arbitrarily close to zero, preventing progress for all practical purposes. Christian Scheideler's *Towards a Paradigm for Robust Distributed Algorithms and Data Structures* [Sch06], covered in Related Work Section 26.7, presents an approach for resisting denial of service attacks within E-like systems.

Part IV

Emergent Robustness

Chapter 20

Composing Complex Systems

In this part, we examine the practice of least authority at four major layers of abstraction—from humans in an organization down to individual objects within a programming language. We explain the special role of languages—such as E—which support object-granularity least authority and defensive consistency.

In order to build systems that are both functional and robust, we first provide programmers with foundations that combine designation with permission. We then need to provide the tools, practices, and design patterns that enable them to align knowledge and authority.

20.1 The Fractal Locality of Knowledge

We can identify two main places where acts of designation occur: users designate actions through the user interface, and objects designate actions by sending requests to other objects. In both places, developers already have extensive experience with supporting acts of designation. User-interface designers have developed a rich set of user-interface widgets and practices to support designation by users [Yee04]. Likewise, programmers have developed a rich tool set of languages, patterns, and practices to support designation between objects.

What the object model and object-capability model have in common is a logic that explains how computational decisions dynamically determine the structure of knowledge in our systems—the topology of the "knows-about" relationship. The division of knowledge into separate objects that cooperate through sending requests creates a natural sparseness of knowledge within a system. The object-capability model recognizes that this same sparseness of knowledge, created in pursuit of good modular design, can be harnessed to protect objects from one another. Objects that do not know about one another directly or indirectly, and consequently have no way to interact with each other, cannot cause each other harm. By combining designation with permission, the logic of the object-capability model explains how computational decisions dynamically determine the structure of authority in our systems—the topology of the "access to" relationship.

Herbert Simon argues that a hierarchic nesting of subsystems is common across many types of complex systems [Sim62]. "Hierarchy," he argues, "is one of the central structural schemes that the architecture of complexity uses." For example, in biology, organisms are composed of organs, which are composed of tissues, which are composed of cells. This chapter describes software systems in terms of four major layers of abstraction: at the organizational level systems are composed of users; at the user level, systems are composed of applications; at the application level, systems are composed of modules; at the module level, systems are composed of objects.

As Simon notes, the nesting of subsystems helps bring about a sparseness of knowledge between subsystems. Each subsystem operates (nearly) independently of the detailed processes going on within other subsystems; components within each level communicate much more frequently than they do across levels. For example, my liver and my kidneys in some sense know about each other; they use chemical signals to communicate with one another. Similarly, you and I may know about each other, using verbal signals to communicate and collaborate with one another. On the other hand we would be quite surprised to see my liver talk to your kidneys.

While the nesting of subsystems into layers is quite common in complex systems, it provides a rather static view of the knowledge relationship between layers. In contrast, within layers we see a much more dynamic process. Within layers of abstraction, computation is largely organized as a dynamic subcontracting network. Subcontracting organizes requests for services among clients and providers. Abstraction boundaries between clients and providers help to further reduce the knows-about relationship within systems; they enable separation of concerns at the local level [TM06]. Abstraction boundaries allow the concerns of the client (why request a particular service) to be separated from the concerns of the provider (how the service will be implemented). Abstraction boundaries, by hiding implementation details, allow clients to ignore distractions and focus on their remaining concerns. Similarly, abstraction boundaries protect clients from unwanted details; by denying the provider authority that is not needed to do its job, the client does not need to worry as much about the provider's intent. Even if the intent is to cause harm, the scope of harm is limited.

Friedrich Hayek has argued that the division of knowledge and authority through dynamic subcontracting relationships is common across many types of complex systems [Hay37, Hay45, Hay64]. In particular, Hayek has argued that the system of specialization and exchange that generates the division of labor in the economy is best understood as creating a division of knowledge where clients and providers coordinate their plans based on local knowledge. Diverse plans, Hayek argues, can be coordinated only based on local knowledge; no one entity possesses the knowledge needed to coordinate all agents' plans. Similarly no one entity has the knowledge required to allocate authority within computer systems according to the principle of least authority. To do this effectively, the entity would need to understand the duties of every single abstraction of the system, at every level of composition. Without understanding the duties of each component, it is impossible to understand what would be the least authority needed for it to carry out these duties. "Least" and "duties" can only be understood locally.

Chapter 21

The Fractal Nature of Authority

The access matrix model [Lam74, GD72] has proven to be one of the most durable abstractions for reasoning about access control in computational systems. An access matrix, such as Level 1 of Figure 21.2, provides a snapshot of the access relationships of a particular system. It shows the rights (the filled-in cells) that active entities (or "subjects," assigned to the rows) have with respect to protected resources (or "objects," assigned to the columns). While not designed for reasoning about authority, we adapt the access matrix model to show how the consistent application of POLA across levels can significantly reduce the ability of attackers to exploit vulnerabilities. We show how POLA applied at the four major layers of abstraction—from humans in an organization down to individual objects within a programming language—can achieve a multiplicative reduction in a system's overall vulnerability.

The access matrix is normally used to depict only current permissions. Since we wish to reason about our overall exposure to attack, we use access matrices to depict potential authority. When we wish to speak specifically about the structure of permissions, we will instead use access diagrams in which permissions are shown as arcs of the graph.

Howard, Pincus and Wing [HPW03] have introduced the notion of an attack surface as a way to measure, in a qualitative manner, the relative security of various computer systems. This multi-dimensional metric attempts to capture the notion that system security depends not only on the number of specific bugs found, but also on a system's "process and data resources" and the actions that can be executed on these resources. These resources can serve as either targets or enablers depending on the nature of the attack. Attackers gain control over the resources through communication channels and protocols; access rights place constraints on which resources can be accessed over these channels.

They define the attack surface of a system to be the sum of the system's attack opportunities. An attack is a means of exploiting a vulnerability. Attack opportunities are exploitable vulnerabilities in the system weighted by some notion of how exploitable the vulnerability is. By treating exploitability not just as a measure of the likelihood of a particular exploit occurring, but as a measure of the extent of damage that can occur from a successful attack, we can gain insight into the role least authority can play in reducing a system's attack surface.

We can use the area of the cells within the access matrix to visualize, in abstract, the attack surface of a system. Imagine that the heights of the rows were resized to be proportional to the likelihood that each actor could be corrupted or confused into enabling an attack, as shown in Figure 21.1. Imagine that the widths of the columns were resized to

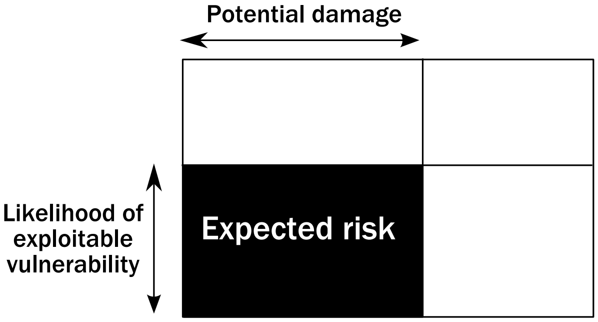


Figure 21.1: Attack Surface Area Measures Risk. In an access matrix, each row represents an active entity ("subject") that is given rights, shown as filled-in cells, to access various resources ("objects" or "assets"), assigned to columns. To measure risk, make each row height be the probability that its subject is corruptible, each column width the damage that could be caused by abusing that resource, and fill in a cell when its subject has authority over its resource.

be proportional to the damage an attacker with authority to that asset could cause. Our overall attack surface may, therefore, be approximated as the overall filled-in area of the access matrix. We do not show such resizing, as the knowledge needed to quantify these issues is largely inaccessible.

By taking this perspective and combining it with Simon's insight that complex systems are typically organized into nested layers of abstractions, we can now show how applying POLA to each level can multiplicatively reduce the attack surface of a system. We show how the same nesting of levels of abstraction, used to organize system functionality, can be used to organize the authority needed to provide that functionality.

We now take a tour through four major levels of composition of an example system:

1. Among the people within an organization

2. Among the applications launched by a person from their desktop

3. Among the modules within an application

4. Among individual language-level objects

Within this structure, we show how to practice POLA painlessly at each level, and how these separate practices compose to reduce the overall attack surface.

Some common themes will emerge in different guises at each level:

- The relatively static nesting of subsystems

- The dynamic subcontracting networks within each subsystem

- The co-existence of legacy and non-legacy components

- The limits placed on POLA by platform risk and by legacy code

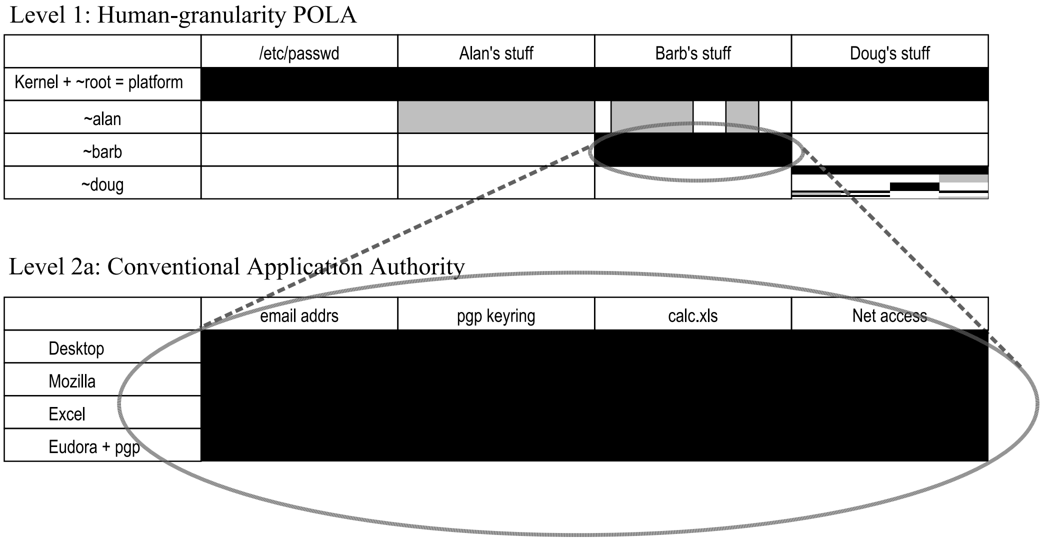


Figure 21.2: Barb's Situation. Conventional access control systems are only built to restrict access at one level of composition, such as between accounts under an operating system. The large black rectangle in Level 2a is not a printing error. Since Barb runs programs under her account in a conventional manner, all the assets entrusted to Barb are at risk to the misbehavior of any of her programs. All are central points of failure for Barb. The distinction between gray and black areas will be explained in Section 22.4.

21.1 Human-Granularity POLA in an Organization

When an organization is small, when there is little at stake, or when all an organization's employees are perfectly incorruptible and non-confusable, the internal distribution of excess authority creates few vulnerabilities. Otherwise, organizations practice separation of responsibilities, need to know, and POLA to limit their exposure.

The box labeled "Level 1" in Figure 21.2 uses the access matrix to visualize how conventional operating systems support POLA within a human organization. Alan (the "~alan" account) is given authority to access all of Alan's stuff, and likewise with Barb and Doug. In addition, because Barb and Alan are collaborating, Barb gives Alan authority to access some of her stuff. The organization should give Alan those authorities needed for him to carry out his responsibilities. This can happen in both a hierarchical manner (an administrator determining which of the organization's assets are included in "Alan's stuff") and a decentralized manner (by Barb, when she needs to collaborate with Alan on something) [AB95]. If an attacker confuses Alan into revealing his password, the assets the attacker can then abuse are limited to those entrusted to Alan. While better training or screening may reduce the likelihood of an attack succeeding, limits on available authority reduce the damage a successful attack can cause.

To the traditional access matrix visualization, we have added a row representing the platform, and a column, labeled `/etc/passwd`, which stands for resources which are effectively part of the platform.

21.2 Application-Granularity POLA on the Desktop

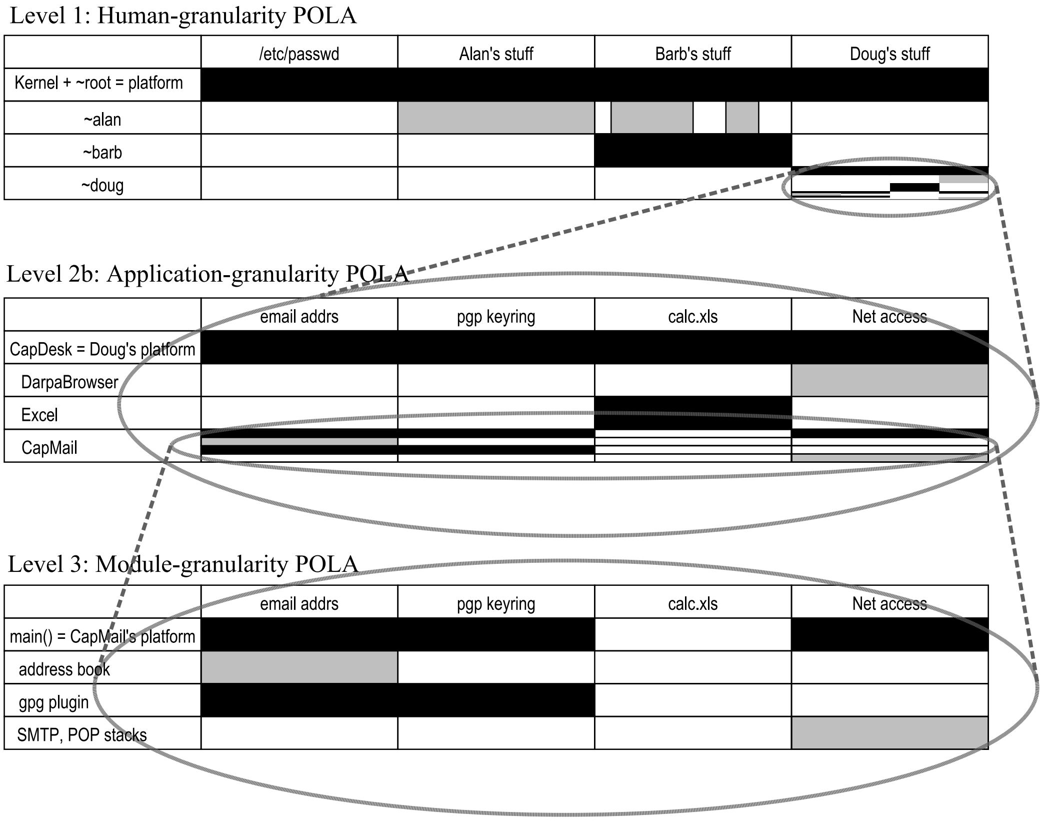
With the exception of such platform risk, organizations have wrestled with these issues since long before computers. Operating system support for access control evolved largely in order to provide support for the resulting organizational practices [MS88]. Unfortunately, conventional support for these practices was based on a simplifying assumption that left us exposed to viruses, worms, Trojan horses, spyware, and the litany of problems that currently infest our networks. The simplifying assumption? When Barb runs a program to accomplish some goal, such as by launching `calc.xls`, an Excel spreadsheet, conventional systems assume the program is a perfectly faithful extension of Barb's intent. But Barb didn't write Excel or `calc.xls`.

Zooming in on Level 1 brings us to Level 2a (Figure 21.2), showing the conventional distribution of authority among the programs Barb runs—they are all given all of Barb's authority. If Excel is corruptible or confusable—if it contains a bug allowing an attacker to subvert its logic for the attacker's purposes, then anything Excel may do, the attacker can do. The attacker can abuse all of Barb's authority—sending itself to her friends and deleting her files—even if her operating system, her administrator, and Barb herself are operating flawlessly. Since all the assets entrusted to Barb are exposed to exploitable flaws in any program she runs, all her programs are central points of failure for her, and for all assets entrusted to her. If Barb enables macros, even her documents, like `calc.xls`, would be a central point of failure for her. How can Barb reduce her exposure to the programs she runs?

Good organizational principles apply at many scales of organization. If the limited distribution of authority we saw in Level 1 is a good idea, can we adopt it at this level as well?

Level 2b (Figure 21.3) is at the same "scale" as Level 2a, but depicts Doug's situation rather than Barb's. Like Barb, Doug launches various applications interactively from his desktop. Unlike Barb, let us say Doug runs his desktop and these applications in such a way as to reduce his exposure to their misbehavior. One possibility would be that Doug runs a non-conventional operating system that supports finer-grained POLA [DH65, Har85, SSF99]. In this chapter, we explore a surprising alternative—the use of language-based security mechanisms such as those provided by E. We will explain how Doug uses CapDesk and Polaris to reduce his exposure while still running on a conventional operating system. But first, it behooves us to be clear about the limits of this approach. In our story, we combine the functionality of CapDesk and Polaris, though they are not yet actually integrated. (Integrating CapDesk's protection with that provided by an appropriate secure operating system would yield yet further reductions in exposure, but these are beyond the scope of this dissertation.)

CapDesk [SM02] is a capability-secure distributed desktop written in E, for running *caplets*—applications written in E to be run under CapDesk. CapDesk is the user's graphical shell, turning a user-interface action into a request to a caplet, carrying a reference saying what object the caplet should operate on. As with the `cat` example in Section 3.2, these requests also convey the permission to operate on this object. For legacy applications like Excel, CapDesk delegates their launching to Polaris [SKYM04]. Like `cp`, Excel needs to run with all the authority of its user's account. Polaris creates and administers separate user accounts for this purpose, each of which starts with little authority. CapDesk has Polaris launch Excel in one of these accounts, and dynamically grant to this account the needed



Level 1: Human-granularity POLA

	/etc/passwd	Alan's stuff	Barb's stuff	Doug's stuff
Kernel + ~root = platform	■	■	■	■
~alan		▨	▨ ▨	
~barb			■	
~doug				■ ▨

Level 2b: Application-granularity POLA

	email addrs	pgp keyring	calc.xls	Net access
CapDesk = Doug's platform	■	■	■	■
DarpaBrowser				▨
Excel			■	
CapMail	■ ▨	■		■ ▨

Level 3: Module-granularity POLA

	email addrs	pgp keyring	calc.xls	Net access
main() = CapMail's platform	■	■		■
address book	▨			
gpg plugin	■	■		
SMTP, POP stacks				▨

Figure 21.3: Doug's Situation. Doug uses CapDesk and Polaris to reduce the damage his programs may cause to the assets entrusted to him. Polaris restricts authority one more level to protect against legacy applications. CapDesk, together with caplets written in E, enables restrictions on authority at every level of composition down to individual objects.

portion of the actual user's authority.

CapDesk is the program Doug uses to subdivide his authority among these applications. To do this job, CapDesk's least authority is all of Doug's authority. Doug launches CapDesk as a conventional application in his account, thereby granting it all of his authority. Doug is no less exposed to a flaw in CapDesk than Barb is to a flaw in each application she runs. CapDesk is part of Doug's platform, and is therefore a central point of failure for Doug; but the programs launched by CapDesk are not.

CapDesk does not affect Doug's vulnerability to Barb. Doug is no more or less exposed to an action taken by Barb, or one of her applications, than he was before. If the base operating system does not protect his interests from actions taken in other accounts, then the whole system is a central point of failure for him. Without a base operating system that provides foundational protection, no significant reduction of exposure by other means is possible. So, let us assume that the base operating system *does* provide effective per-account protection. For any legacy programs that Doug installs or runs in the conventional manner—outside the CapDesk framework—Doug is no less exposed than he was before. All such programs remain central points of failure for him. If the "˜doug" account is corrupted by this route, again, CapDesk's protections are for naught.

However, if the integrity of "˜doug" survives these threats, Doug can protect the assets entrusted to him from the programs he runs by using CapDesk + Polaris to grant them least authority. This granting must be done in a usable fashion—unusable security won't be used, and security which isn't used doesn't protect anyone. As with `cat`, the key to usable POLA is to bundle authority with designation [Yee02, Yee04]. To use Excel to edit `calc.xls`, Doug must somehow designate this file as the one he wishes to edit. This may happen by double-clicking on the file, by selecting it in an open file dialog box, or by drag-and-drop. The least authority Excel needs includes the authority to edit this one file, but typically not any other interesting authorities. Polaris runs each legacy application in a separate account which initially has little authority. Doug's act of designation dynamically grants Excel's account the authority to edit this one file. Polaris users regularly run with macros enabled, since they no longer live in fear of their documents.

21.3 Module-Granularity POLA Within a Caplet

Were we to zoom into Doug's legacy Excel box, we'd find that there is no further reduction of authority within Excel. All the authority granted to Excel as a whole is accessible to all the modules of which Excel is built, and to the macros in the spreadsheets it runs. Each is a central point of failure for all. Should the math library's `sqrt` function wish to overwrite `calc.xls`, nothing will prevent it. At this next smaller scale we'd find the same full-authority picture depicted in Level 2a.

Caplets running under CapDesk do better. The DarpaBrowser is a web browser caplet, able to use a potentially malicious plug-in as a renderer. Although this is an actual example, the DarpaBrowser is "actual" only as a proof of concept whose security properties have been reviewed [WT02]—not yet as a practical browser. We will instead zoom in to the hypothetical email client caplet, CapMail. All the points we make about CapMail are also true for the DarpaBrowser, but the email client makes a better expository example. Of the programs regularly run by normal users—as opposed to system administrators or programmers—the email client is the worst case we've identified. Its least authority includes

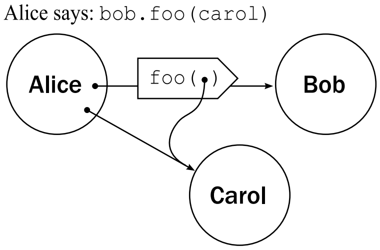


Figure 21.4: Level 4: Object-granularity POLA. In the object paradigm, objects only obtain access to each other dynamically, on an as-needed basis, by the passing of references. By removing other causal pathways, permission follows designation. By building frameworks and adopting programming styles at each level of composition that leverages these properties, authority structures can also be made to follow knowledge structures.

a dangerous combination of authorities. Doug would grant some of these authorities—like access to an SMTP server—by static configuration, rather than dynamically during each use.

When Doug decides to grant CapMail these authorities, he is deciding to rely on the authors of CapMail not to abuse them. However, the authors of CapMail didn't write every line of code in CapMail—they reused various reusable libraries written by others. CapMail should not grant its math library the authority needed to read your address book and send viruses to your friends.

Zooming in on the bottom row of Level 2b brings us to Level 3. A caplet has a startup module that is the equivalent of the C or Java programmer's `main()` function. CapDesk grants to this startup module all the authority it grants to CapMail as a whole. If CapMail is written well, this startup module should do essentially nothing but import the top level modules constituting the bulk of CapMail's logic, and grant each module only that portion of CapMail's authority that it needs during initialization. This startup module is CapMail's platform—its logic brings about this further subdivision of initial authority, so all the authority granted to CapMail as a whole is vulnerable to this one module.

When a CapMail user launches an executable caplet attachment, CapMail could ask CapDesk to launch it, in which case the attachment would only be given the authority the user grants by explicit actions. CapMail users would no longer need to fear executable attachments. (The DarpaBrowser already demonstrates equivalent functionality for downloaded caplets.)

21.4 Object-Granularity POLA

At Level 3, we again see the co-existence of boxes representing legacy and non-legacy. For legacy modules, we've been using a methodology we call "taming" to give us some confidence, under some circumstances, that a module doesn't exceed its proper authority [SM02, WT02]. Again, for these legacy boxes, we can achieve no further reduction of exposure within the box. Zooming in on a legacy box would again give a picture like Level 2a. Zooming in on a non-legacy box gives us our finest scale application of these principles—at the granularity of individual programming language objects (Figure 21.4). These are the indivisible particles, if you will, from whose logic our levels 2 and 3 were built.

21.5 Object-Capability Discipline

Good software engineering	**Capability discipline**
Responsibility driven design	Authority driven design
Omit needless coupling	Omit needless vulnerability
assert(..) preconditions	Validate inputs
Information hiding	Principle of Least Authority
Designation, need to know	Permission, need to do
Lexical naming	No global namespaces
Avoid global variables	Forbid mutable static state
Procedural, data, control, ...	... and access abstractions
Patterns and frameworks	Patterns of safe cooperation
Say what you mean	Mean only what you say

Table 21.1: Security as Extreme Modularity. On the left we list various well known tenets of good object-oriented software engineering practice. On the right, we list corresponding tenets of good capability discipline. In all cases, the capability-oriented tenet is an extreme form of its object-oriented counterpart. Anecdotal reports of those with experience at capability discipline is that they find it a useful practice even when security is of no concern, since it leads to more modular systems.

Knowing the rules of chess is distinct from knowing how to play chess. The practice of using these rules well to write robust and secure code is known as capability discipline. Some of the tenets of capability discipline are listed in Table 21.1. Capability discipline is mostly just an extreme form of good modular software engineering practice. Of the people who have learned capability discipline, several have independently noticed that they find themselves following capability discipline even when writing programs for which security is of no concern. They find that it consistently leads to more robust, modular, maintainable, and composable code.

This completes the reductionist portion of our tour. We have seen many issues reappear at each level of composition. Let us zoom back out and see what picture emerges.

21.6 Notes on Related Work

Howard, Pincus and Wing's *Measuring Relative Attack Surfaces* [HPW03] introduced the spatial "measure" of a system's vulnerability that inspired the analysis presented in this chapter.

Ka-Ping Yee's *User Interaction Design for Secure Systems* [Yee02] and *Aligning Security and Usability* [Yee04] presents many principles of secure user interface design. He explains how the user's acts of designation, such as selecting a filename in an open file dialog box, can also be used to convey narrow least authority.

A Capability Based Client: The DarpaBrowser by Marc Stiegler and Mark S. Miller [SM02] summarizes our work on E, CapDesk, and the DarpaBrowser. CapDesk and the DarpaBrowser implemented eight of Yee's ten principles of secure usability [Yee02].

David Wagner and E. Dean Tribble's *A Security Analysis of the Combex DarpaBrowser Architecture* [WT02] reports on the many vulnerabilities they found in our work, and explains the implications of these vulnerabilities.

Polaris: Virus Safe Computing for Windows XP [SKYM04] explains how some of the principles demonstrated by CapDesk are being applied to limit the authority of legacy applications running on a legacy operating system, without modifying either the applications or the operating system.

Chapter 22

Macro Patterns of Robustness

22.1 Nested Platforms Follow the Spawning Tree

The nesting of subsystems within each other corresponds to a spawning tree. The platform of each system creates the initial population of subsystems within it, and endows each with their initial portion of the authority granted to this system as a whole. The organization decides what Alan's responsibilities are, and its administrators configure Alan's initial authorities accordingly. Doug uses CapDesk to endow CapMail with access to his SMTP server by static configuration. CapMail's `main()` grants this access to its imported SMTP module. A lambda expression with a free variable "`carol`" evaluates to a closure whose binding for "`carol`" is provided by its creation context.

As we nest platforms, we accumulate the platform risk explained in Section 5.2 and in Shockley and Schell's *TCB Subsets for Incremental Evaluation* [SS87]. This layering has the hierarchic structure that Herbert Simon explains as common to many kinds of complex systems [Sim62]. Mostly static approaches to POLA, such as policy files, may succeed at mirroring this structure.

22.2 Subcontracting Forms Dynamic Networks of Authority

Among already instantiated components, we see a network of subcontracting relationships whose topology dynamically changes as components make requests of each other. Barb finds she needs to collaborate with Alan; or Doug selects `calc.xls` in an open file dialog box; or object A passes a reference to object C as an argument in a message to object B. In all these cases, by following capability discipline, the least authority a provider needs to perform a request can often be painlessly conveyed along with the designations such requests must already carry. The least adjustments needed to the topology of the access graph are often identical to the adjustments made anyway to the reference graph.

22.3 Legacy Limits POLA, But Can be Managed Incrementally

Among the subsystems within each system, we must engineer for a peaceful co-existence of legacy and non-legacy components. Only such co-existence enables non-legacy systems to be adopted incrementally. For legacy components, POLA can and must be practiced without modifying them—Polaris restricts the authority available to `calc.xls` without modifying the spreadsheet, Excel, or Windows XP. However, we can only impose POLA on the legacy component. We cannot enable that component to further practice POLA with the portion of its authority it grants to others, or to sub-components of itself. Following initial adoption, as we replace individual legacy components, we incrementally increase our safety.

22.4 Nested POLA Multiplicatively Reduces Attack Surface

The gray shown within the non-legacy boxes we did not zoom into—such as the "˜alan" row—represents our abstract claim that exposure was further reduced by practicing POLA within these boxes. This claim can now be explained by the fine structure shown in the non-legacy boxes we did zoom into—such as the "˜doug" box. Whatever fraction of the attack surface we removed at each level by practicing POLA, these effects compose to create a multiplicative reduction in our overall exposure. Secure languages used according to capability discipline can extend POLA to a much finer grain than is normally sought. By spanning a large enough range of scales, the remaining attack surface resembles the area of a fractal shape which has been recursively hollowed out [Man83]. Although we do not yet know how to quantify these issues, we hope any future quantitative analysis of what is practically achievable will take this structure into account.

22.5 Let "Knows About" Shape "Access To"

To build useful and usable systems, software engineers build sparse-but-capable dynamic structures of knowledge. The systems most successful at supporting these structures—such as object, lambda, and concurrent logic languages—exhibit a curious similarity in their logic of designation. Patterns of abstraction and modularity divide knowledge, and then use these designators to compose divided knowledge to useful effect. Software engineering discipline judges these patterns partially by their support for the principle of information hiding—by the sparseness of the knowledge structures they build from these designators.

To build useful, usable, and robust general purpose systems, we must leverage these impressive successes to provide correspondingly sparse-but-capable dynamic structures of authority. Only authority structures aligned with these knowledge structures can both provide authority needed for use while narrowly limiting the excess of authority available for abuse. To structure authority in this way, we need "merely" make a natural change to our foundations, and a corresponding natural change to our software engineering discipline.

Capability discipline judges patterns as well by their support for the principle of least authority—by the sparseness of the authority structures they build from these permissions. Not only is this change needed for safety, it also increases the modularity needed to provide ever greater functionality.

An object-capability language can extend this structuring of authority down to finer granularities, and therefore across more scales, than seems practical by other means. We have presented a proof-of-concept system—consisting of E, CapDesk, and Polaris—together with a geometric argument, that explains an integrated approach for using such foundations to build general purpose systems that are simultaneously safer, more functional, more modular, and more usable than is normally thought possible.

22.6 Notes on Related Work

The *Computer Security Technology Planning Study* volumes I and II [NBF$^+$80], also known as "The Anderson Report," explains why any security analysis should divide systems into the "reference validation mechanism," *i.e.*, the platform which enforces the system's access control rules, and those programs executing on the platform, only able to perform those accesses allowed by the reference validation mechanism. The report emphasizes that *all* programs exempt from these access checks must be considered part of the reference validation mechanism, and stresses the need to keep this set very small, because of the great risks that follow from flaws in the platform. The concept of "Trusted Computing Base" (TCB) from the Department of Defense's *Trusted Computer System Evaluation Criteria* [oD85], also known as "The Orange Book," derives from these observations.

The Anderson Report and the Orange Book phrase their distinctions abstractly enough that they can be applied to multiple levels of abstraction. In *A Provably Secure Operating System: The System, Its Applications, and Proofs* [NBF$^+$80], Neumann *et al.* explain their architecture in terms of carefully layered abstraction levels, analyzing this layering of mechanisms from multiple perspectives.

Shockley and Schell explain the notion of multiple layered platforms as "TCB subsets" [SS87], and the hierarchical accumulation of platform risk. Tinto divides reliance into "primitivity," reflecting this multi-level reliance on platform, and "dependence," reflecting reliance among the modules co-existing on the same platform [Tin92].

Scale-free Geometry in OO Programs [PNFB05] presents measurements of the distribution of incoming and outgoing degree (number of references) among objects in snapshots of Java heaps. These measurements indicate that object connectivity is statistically fractal. Their results are both stronger and weaker than needed to support the argument made in this chapter. These results are stronger than we need, because the argument made here does not require the sparseness of connectivity at different scales to be statistically uniform; it requires only the existence of significant sparseness across multiple scales, and that this spareness has the hierarchic nesting structure explained by Simon's *Architecture of Complexity* [Sim62]. These results are weaker than we need, because a snapshot of actual connectivity reflects only what would be current permissions (the "CP" in Table 8.1 (p. 60)) if object-capability programs have similar statistical properties, whereas we are concerned with the structure of potential authority.

Connectivity-based Garbage Collection [HDH03] examines statically calculated conservative bounds on potential connectivity (BP). They show that connected clusters of objects tend to become garbage together, and that even statically calculated conservative bounds on potential connectivity can provide adequate clustering information to aid garbage collection. This provides some support for the hypothesis that statically understandable bounds on potential authority (BA) are usefully clustered as well, although we should expect them to be less clustered than potential permission.

Part V

Related Work

Chapter 23

From Objects to Actors and Back Again

This chapter presents a brief history of E's concurrency-control architecture. Here, the term "we" indicates this dissertation's author participated in a project involving other people. Where this pronoun is used, all implied credit should be understood as shared with these others.

23.1 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 [GK76] 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.

23.2 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 (in our terms) mutual defensiveness [Hew85]. Actors supports this defensiveness explicitly by use of object-capabilities. The connectivity rules we present in Section 9.2 are essentially a restatement of Hewitt and Baker's "locality laws" [HB78].

Although Hewitt's 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 provides both dataflow 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.

23.3 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 dataflow 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 an explicitly pre-arranged *merge networks*. These merge networks provide ordering properties similar to the tree-order subset of E-ORDER, but the object references they provide were not first-class. To address this problem, we created the "Channels" abstraction [TMK$^+$87], which preserved these ordering properties while providing first-class object references.

23.4 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.

23.5 Promise Pipelining in Udanax Gold

Udanax Gold 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, SMTH91]. This was the first attempt to adapt Joule channels to an object-based client-server environment (it did not support peer-to-peer).

23.6 Original-E

The language now known as Original-E was the result of adding the concurrency concepts from Joule to a sequential, capability-secure subset of Java. This effort to identify such a subset of Java had much in common with the J-Kernel and Joe-E projects briefly explained in sections 24.5 and 26.5. 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, Inc. created Original-E, and used it to build EC Habitats—a graphical, decentralized, secure, social virtual reality system.

23.7 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 dissertation. 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.

Chapter 24

Related Languages

24.1 Gedanken

James Morris' 1973 paper, *Protection in Programming Languages* [Mor73a], presents Reynolds' Gedanken language [Rey70] as (in our terms) an object-capability language. Gedanken is a dynamically-typed Algol-60 derivative with first-class lexical closures. Morris explains the correspondence of Gedanken's concepts with those of Dennis and van Horn's Supervisor [DH65]. Curiously, Dennis and van Horn credit some elements of the Burroughs B5000 architecture as having inspired the capability concept. The B5000 was a language-directed architecture, designed to run Algol-60. Algol-60 is closely tied to the lambda calculus, and Gedanken even more so.

As far as we are aware, Morris' paper is the first to propose the two rights amplifications abstractions used by E: The trademarking explained in Section 6.3, and sealer-unsealer pairs [MMF00, Sti04]. Morris' *Types Are Not Sets* [Mor73b] explains the rationale for using trademarks as types in programming languages. Trademarking provides security properties similar to public key signatures. Sealer-unsealer pairs provide security properties similar to public key encryption [MBTL87]. Public key cryptography had not yet been publicly invented.

Actors derives partly from Morris' work on Gedanken.

24.2 Erlang

Joe Armstrong's *Making Reliable Distributed Systems in the Presence of Software Errors* [Arm03] explains the design and rationale for the Erlang language. Erlang has many similarities with E. Erlang is a dynamically type safe distributed language, inspired by Actors, and based on communicating asynchronous messages between encapsulated processes. It was designed for writing systems that remain reliable despite software errors. Erlang has been in commercial use at Ericsson as the basis for their highly reliable (five nines) telephone switching products. This represents an empirical demonstration of robustness *well* beyond anything we have accomplished. One particular notable accomplishment of Erlang—well beyond anything in E but needed for E to achieve its goals—is Erlang's support for robustly upgrading code in a running system.

Although Erlang was inspired by Actors, a common programming pattern in Erlang is for a process to block waiting for a reply to a previous request. While blocked, the process is not responsive to other requests. This is like the “message-oriented systems” of Lauer and Needham [LN79] rather than the non-blocking event loops of E.

Although Erlang is not engineered to provide robustness against malice, it seems to be an object-capability language. Erlang processes are encapsulated objects and processIds are unforgeable capabilities, providing the right to send messages to a process. The rules by which processIds are transmitted and acquired seem to be precisely object-capability rules. Most importantly, the language has no global scope. Each process has its own separate state.

Unfortunately, a process has only one processId, so the common convention is for making a request and receiving a reply is for the client to include its own processId in the message as a reply port. This gives the provider the authority, not only to reply to the client, but to invoke the client. It is not clear how to repair this convention within the overall Erlang framework.

24.3 Argus

Liskov and Scheifler’s Argus system [LS83] is a distributed programming language, based on Liskov’s earlier abstract data type language, CLU [LSA77], for writing reliable distributed programs. For purposes of this summary, we will ignore distinctions between abstract data types and objects.

The main new ideas Argus brings to CLU are “guardians” and “actions.” Like E’s vat, Argus objects are aggregated into *guardians.* Each guardian runs on one machine at a time. Again like E’s vats, guardians persist, so the long lived state of a distributed Argus system is the committed states of its guardians. Like E’s turns, computational steps are aggregated into *actions.* Each action is an atomic transaction, and computation as a whole is equivalent to a serializable sequence of actions. When one object invokes another, it can do so within the same action (like E’s immediate call), or it can cause the recipient to be invoked in a new action (like E’s eventual send). Like E’s persistence, the only states committed to stable storage are states between actions.

Unlike E’s turns, Argus actions could, and generally did, span guardians. The Argus infrastructure therefore primitively provided distributed atomic transactions, implemented by a two-phase commit protocol. This provides expressiveness and flexibility well beyond that provided by E’s vat-local turn mechanism: Coordinated state changes of distributed state could be expressed, and would either be committed as a whole or aborted as a whole. However, distributed atomicity necessarily comes at a price: If the network partitions during the protocol’s window of vulnerability, some guardians could be unable to determine whether an action committed or aborted until the partition healed. Any local state affected by such an action would be inaccessible during this period, and any other action that touches this state would get stuck as well. By contrast, since E provides only vat-local atomicity, distributed E applications remain non-blocking and responsive during partitions.

The Argus protocol design assumes mutually reliant machines. To extend an Argus-like system between mutually defensive machines would require a distributed atomic committment protocol resistent to malicious participants.

24.4 W7

The Scheme Language [KCR98], by virtue of its Actors ancestry, its close ties to the lambda calculus, and its minimalism, already comes very close to being an object-capability language. Jonathan Rees' thesis, *A Security Kernel Based on the Lambda-Calculus* [Ree96], explains well the rationale for the object-capability model of protection, and how closely it already corresponds to Scheme's computational model. Rees then explains W7, his Scheme derivative, which is a full object-capability language. The differences between Scheme and W7 are chiefly 1) eliminating global namespaces, 2) eliminating forms of `eval` which violate loader isolation, such as Scheme's one argument `load` function, and 3) adding Morris' trademarking and sealer-unsealer pairs.

24.5 J-Kernel

The Java language, by virtue of its memory safety, strong encapsulation, and load-time verification, is tantalizingly close to being an object-capability language. Unfortunately, Java provides objects with two forms of statically accessible authority: authority to affect the outside world (e.g., by static native methods that do I/O) and authority to affect other objects (e.g., by assigning to static variables or by mutating statically accessible objects). Original-E (Section 23.6) and Joe-E (Section 26.5) attempt to turn Java into an object-granularity realization of the object-capability model, where the "object" of the object-capability model corresponds one-to-one with the "object" of the programming language. They forbid statically accessible authority, including mutable static state.

The J-Kernel [vECC$^+$99] demonstrates another way to reconcile Java with the object-capability model. The J-Kernel's designers explicitly reject the goal of providing object-granularity protection. Rather, because they wish to bundle resource controls with capability protections, their protection model starts with a unit they call a "domain," which is similar in many ways to E's vat. By the J-Kernel's architecture, the "object" of the object-capability model corresponds, not to a Java object, but to a J-Kernel domain. A domain consists of a ClassLoader, all the classes loaded by that ClassLoader, and all the instances of those classes. Like E's vat, computation within a domain proceeds within threads local to that domain. Like E's BootCommSystem, explained briefly in Section 14.3, communication between domains proceeds by sending messages over a distributed object protocol customized to take advantage of the shared address space. The J-Kernel adapted RMI [WRW96], Java's distributed object protocol.

They consider only inter-domain references to be capabilities. Although the rules governing objects, references, and messages within a domain so closely resemble the rules of an object-capability system, they explicitly argue against making use of this fact. Legacy Java code violates object-capability design rules, so this stance allows J-Kernel domains to use some of this legacy code.

In Java itself, two classes that import `java.lang.System` could communicate by assigning to and reading `System`'s static `in`, `out`, and `err` variables. To provide approximately the loader isolation we explain in Section 10.3, the J-Kernel ClassLoader subclassed the Java ClassLoader class in order to prevent importation of dangerous system classes by magic name. With this hole closed, the "object creation" of the object-capability model corresponds properly to creating a new J-Kernel ClassLoader endowed with an import resolution table.

Unfortunately, this ClassLoader flexibility conflicted with Java's own security architecture. Starting with Java 1.2, ClassLoaders can no longer override the importing of system classes. The J-Kernel no longer runs. Short of inspecting and/or modifying classfiles prior to loading them, it seems it cannot be fixed.

24.6 Emerald

The Emerald programming language [HRB$^+$87] is a transparently distributed object language with many similarities to E. In Emerald, object references are capabilities. Emerald's predecessor, Eden, was also an object-capability language. Experience with Eden identified the inadequacy of permission bits as a technique for distinguishing different authorities to the same object [Bla85], and identified the need to support first-class facets. Like E, Emerald objects are instantiated by nested lambda instantiation, rather than classes or prototypes.

As far as we have been able to determine, Emerald's distributed object protocol assumes mutually reliant hardware, forming in our terms a single logical shared platform that is physically distributed. Emerald provides a *call-by-move* argument passing mode, which would seem problematic even under this mutual reliance assumption: If the network partitions during the protocol's inescapable window of vulnerability, during the partition, the object being moved would either exist on both sides, on neither side, or be indeterminate on at least one side.

24.7 Secure Network Objects

Emerald inspired the *Network Objects* work [BNOW94], done at Digital Equipment Corporation's System Research Center, providing mostly transparent distribution for Modula-3 objects. Like Emerald, Network Objects preserved pointer safety only between mutually reliant machines. However, it led to the system described in *Secure Network Objects* system [vDABW96], which provided distributed pointer safety between mutually defensive machines, using cryptographic techniques much like Pluribus. Even though Secure Network Objects paid a complexity price for working around the export controls of the time, the system is still quite simple, well crafted, and clearly explained.

Chapter 25

Other Related Work

25.1 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 forms 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 fault 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.

25.2 Croquet and TeaTime

The Croquet project [Ree05] has many of the same goals as the EC Habitats project mentioned in Section 23.6: to create a graphical, decentralized, secure, user-extensible, social virtual reality system spread across mutually defensive 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 mutually reliant machines. Neither EC Habitats nor Croquet have yet achieved secure user-extensible behavior. At Electric Communities, we withheld user-extensibility until we could support it securely; the company failed before then. As of this writing, Croquet is insecurely user-extensible. It will be interesting to see how they alter Paxos to work between mutually defensive machines.

As of this writing, Croquet is using the "Simplified TeaTime" of the Hedgehog architecture [SRRK05], perhaps as an interim measure. Hedgehog is based on Tweak Islands, which we cover briefly in Related Work Section 26.8.

25.3 DCCS

Jed Donnelley's "DCCS" [Don76] is the first to realize and explain how the capability model may be cryptographically stretched between mutually defensive machines in a decentralized manner, why this preserves many of the useful properties of single-machine capability operating systems, and why the network service that accomplishes this needs no special privileges. Although E and Pluribus were both engineered so that Pluribus could likewise be implemented by unprivileged E code, this has not yet happened.

Because of DCCS's early date—preceding the publication of public key cryptography—its use of cryptography was somewhat crude by modern standards. Donnelley's later work [Don79] repairs these problems, and is in some ways arguably more secure than Pluribus. As explained in Section 11.5, the security properties enforceable within a single machine (or mutually reliant machines) are stronger than the security properties enforceable between mutually defensive machines by cryptographic means. Like E, Donnelley's "DCCS" was a distributed capability system capable of these stronger properties within each machine.

25.4 Amoeba

The "sparse capabilities" of the influential Amoeba distributed operating system [TMvR86] used unguessable numbers to represent capabilities both between machines and within each machine. This limits the properties enforceable within each machine to those properties that are enforceable between mutually defensive machines.

25.5 Secure Distributed Mach

Mach is an object-capability operating system based on *ports* with separate send and receive rights. Sansom's *Extending a Capability Based System into a Network Environment* [SJR86] presents a cryptographic capability protocol for transparently stretching Mach's ports over the network. This work reiterates Donnelley's observation that the network-handling code needs no special privilege.

25.6 Client Utility

The Client Utility [KGRB01, KK02] was based on "split capabilities" [KGRB03]. Split capabilities have interesting similarities and differences from object-capabilities, which are well beyond the scope of this summary. The Client Utility resembles a distributed operating system more than a distributed programming language: its "objects," or *clients* were processes that could be expressed in any language. However, it ran portably on top of other operating systems, with the possibility of intercepting system calls when run on host operating systems where this was possible. Clients invoked each other's services by asynchronous messages. These messages would pass through the Client Utility "core," another process serving as the shared platform for a group of clients on the same machine. From a distribution and a security perspective, a core and its set of clients, called a "logical machine", is analogous to an E vat and the set of objects it hosts.

Like many distributed capability systems, starting with DCCS and including E, the Client Utility's semantics and security were transparently stretched over the network by proxies, which were further unprivileged domains, running in each logical machine, forwarding requests over the network to unprivileged clients providing network services to their logical machine.

In recent years, E has learned much from the Client Utility architecture. In particular, E's three-vat introduction protocol, which enforces the E-ORDER explained in Chapter 19, is an adaptation of the proposed (but unimplemented) Client Utility introduction protocol.

Chapter 26

Work Influenced by E

26.1 The Web-Calculus

The Web-Calculus [Clo04c] 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 dissertation. See [Fie00, Clo04c] 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.

26.2 Twisted Python

Twisted Python 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 [Lef03]. Like Actors, it considers control-flow postponement (like chaining `when` blocks in E) to be the more fundamental mechanism, and dataflow postponement (by promise pipelining) as secondary. Twisted Python provides a built-in joiner abstraction similar to the `promiseAllFulfilled` shown in Figure 18.3 (p. 131).

Twisted provides protection only at the granularity of communicating Python processes. As with the J-Kernel, because legacy Python code violates capability design rules, this large protection granularity allows programmers to use legacy code within each process.

26.3 Oz-E

Like Vulcan, the Oz language [RH04] descends from both Actors and concurrent logic/constraint programming. From its concurrent logic/constraint ancestry, Oz inherits logic variables which provide dataflow postponement much like E's promises. The primary form of parallelism in Oz is "declarative concurrency": If a computation uses no non-declarative primitives like assignment, then it can spawn parallel threads at fine grain, sharing an address space, without any visible effects of concurrency. By the terminology used in this dissertation, "declarative concurrency" is therefore not a form of concurrency at all, but rather a form of parallelism. It provides a convenient way to use parallel hardware to run a computation faster. But since the effects of declarative parallelism are not observable, such a computation as a whole is still a single unit of operation. So-called declarative concurrency cannot express the concurrency issues needed to interact with an ongoing concurrent world.

The other recommended form of parallelism in Oz is "message passing concurrency," which is approximately the same as our communicating event-loops. To support imperative programming within an event-loop, Oz provides Cells, much like E's Slots or ML's Refs. Unfortunately, by providing both shared-memory spawning of threads and mutable Cells, Oz thereby enables shared-state concurrency, though Oz programming practice discourages its use.

Oz-E [SR05b] is a successor to Oz designed to support the object-capability model at Oz's object-granularity. Like E, Oz-E is being designed to provide protection within a process by safe language techniques, and to provide protection among mutually defensive machines using a cryptographic capability protocol. In order to support defensive consistency, Oz-E will suppress Oz's shared-state concurrency. To support both declarative parallelism and message-passing concurrency while suppressing shared-state concurrency, Oz-E will tag each Cell with its creating thread. Attempting to access a Cell from any other thread will cause an exception to be thrown. Properly declaratively parallel programs will not encounter this error since they use no Cells. Proper message-passing concurrent programs will not encounter this error, since all Cells are accessed only from their event-loop's thread.

26.4 SCOLL

SCOLL [SMRS04, SR05a] is not a programming language, but rather a description language, based on Datalog, for statically reasoning about authority relationships in object-capability systems. In SCOLL, one describes a tractable abstraction of the behavior of various objects, such as the caretaker or data diodes we have presented in Part II. These abstractions must be safely conservative: all actions the object might take must fall within the described behavior, but the described behavior can include possibilities beyond what the object might actually do. SCOLL then calculates bounds on possible authority, based on a form of Bishop and Snyder's take-grant logic for reasoning about "potential de facto information transfer" [BS79] elaborated to take into account the described limits on object behaviors.

Patterns like our caretaker and data diode have been described in the SCOLL language, and SCOLL has computed the resulting authority relationships. The caretaker was particularly challenging for two reasons: 1) The `disable()` action causes a non-monotonic decrease in authority, whereas the take-grant framework naturally reasons only about the limits of monotonically increasing authority. 2) Unlike the membrane, the extent of authority revoked by the caretaker depends in subtle ways on Carol's behavior. If Carol returns Dave to Bob, then after Alice revokes Bob's access to Carol, Bob might still have access to

Dave, and thereby might still have all the authority such access implies.

We hope to see SCOLL grow into a static safety checker that can be incorporated into future object-capability languages, much as static type checkers are incorporated into many languages today. Type declarations are also conservative abstractions of actual behavior. Both kinds of static description provide programmers a useful design language prior to writing the program itself. Both kinds of static checkers ensure that certain problems—message-not-understood errors and excess authority errors, respectively—do not happen at runtime. The price of static checking is that some programs that are actually safe will be rejected, leading to a loss of expressive power. Whether this price is worth paying depends on the importance of preventing these errors. Message-not-understood errors not caught until runtime would still fail safe. Without some separate description of intended limits of authority, like SCOLL, excess authority errors are not detectable either statically or at runtime. Should they occur, they would not fail safe. The case for static authority checking would seem to be much stronger than the case for static type checking.

26.5 Joe-E

The Joe-E language [MW06] is a subset of the Java language which forms an object-granularity object-capability language. Of the features removed from Java to make Joe-E, like mutable static state, most were removed with no interesting loss of expressive power. For example, if a class would have had static state, one can write an inner class instead, in which this state is part of its creating object. Beyond prohibiting violations of object-capability rules, Joe-E also prohibits elements of Java that egregiously endanger capability discipline, like shared-state concurrency. Joe-E is currently a purely sequential language, though it would be straightforward to add a library supporting communicating event-loops.

Unlike the J-Kernel (Section 24.5), Joe-E does not seek compatibility with existing legacy Java code. It is not imagined that much existing Java code happens to fall within the Joe-E subset. Rather, Joe-E subsets Java in order to be compatible with legacy Java tools: integrated development environments, debuggers, profilers, refactoring browsers, compilers, virtual machines, etc. The Joe-E loader enforces the Joe-E restrictions merely by testing and refusing to load any Java code that falls outside the Joe-E subset. Any Java code that passes these tests is also Joe-E code, and is loaded as is. Because the Joe-E loader does no transformation on the code it accepts, any Joe-E program is a Java program with the same meaning. All existing Java tools apply to this code without modification.

The first attempt at a Joe-E loader, by Chip Morningstar, worked by subclassing Java's ClassLoader and applying further tests to classfiles. The Joe-E verification rules were thereby added to the JVM bytecode verification rules. Had this worked, we would have said that Joe-E defines a subset of the JVM rather than a subset of Java. The Java language has stricter rules than the JVM. Java compilers must prohibit many cases that JVM bytecode verifiers do not. The Joe-E architects, Adrian Mettler and David Wagner, realized that many of these cases must also be prohibited by Joe-E. Therefore, Joe-E code is submitted as Java source, not bytecode, and the Joe-E verifier builds on a Java compiler.

Joe-E does not provide a user-extensible auditing framework. Rather, the Joe-E library provides a few interface types which trigger auditing checks built into the Joe-E verifier. For example, the Joe-E `Incapable` type (meaning "without capabilities") is much like E's `Data` guard/auditor. If a Joe-E class declares that it implements `Incapable`, the Joe-E loader statically verifies that all its instance variables are `final` and can hold only `Incapable`s. All `Incapable`s are therefore transitively immutable.

The most novel aspect of the Joe-E project is the abstraction of authority the authors use to reason about security. In this dissertation, we define authority as the ability to overtly cause effects. Unfortunately, causation is surprisingly tricky to reason about. Let us say Alice builds a caretaker that will only pass messages that contain a secret number Bob does not know and cannot guess. In this situation, Bob does not have authority to access Carol. But if Bob's interaction with Dave gives Bob enough knowledge that he can feasibly guess the number, then Dave has granted Bob authority to access Carol.

The Joe-E programmer reasons about authority more simply, using the conservative assumption that anyone might already know anything—as if secrets were not possible. Therefore, in the initial conditions above, a Joe-E programmer would assume Bob already has this authority, since he might already know this number. Data gained from Dave cannot give Bob any authority beyond what we assume he already has. Curiously, this abstraction prevents reasoning about confidentiality or cryptography.

The Joe-E verifier also runs as an Eclipse plugin, marking Java code that violates Joe-E's additional rules. When hovering over these markups, an explanation of the violation pops up, helping people learn incrementally what object-capability programming is about.

26.6 Emily

The Emily language [Sti06] is an object-capability subset of OCaml. As with the W7 and Joe-E efforts, Emily differs from OCaml mainly by removing static sources of authority. Emily modules cannot import system modules that magically provide I/O calls, and Emily modules cannot define top level variables that hold mutable state. Beyond that, the Emily effort is in a very early and fluid stage, so it may be premature to go into much detail.

A discovery from the Emily exercise is the tension between object-capability programming and purely static type checking. For example, it seems impossible to translate the membrane of Figure 9.3 (p. 71) or the meta-interpretive loader of Figure 10.1 (p. 76) into OCaml or Emily. Although Java is mostly-statically typed, Java does not suffer from the same loss of expressive power. Java programs use so-called "reflection" to mix static typing and dynamic typing as needed. (Joe-E does not yet offer a form of reflection, but there is no reason in principle why this should be difficult.)

26.7 Subjects

Christian Scheideler's *Towards a Paradigm for Robust Distributed Algorithms and Data Structures* [Sch06] extends an E-like framework with explicit resource controls, to support the creation of highly decentralized applications that are robust against even distributed denial of service attacks. In this framework, a highly available service is represented by the equivalent of a routing proxy on an intermediate vat, whose clients know its stationary network location. It forwards messages to any one of a number of dynamically replicating service-providing vats, which accept traffic only from their proxies. When Alice composes Bob and Carol, Alice also decides how much bandwidth capacity she grants to Bob to communicate with Carol. When bandwidth is scarce, Carol's proxy will throttle any bandwidth usage by Bob exceeding his allotment.

26.8 Tweak Islands

Tweak is a derivative of Squeak (an open-source Smalltalk) designed to better support end-user programming by direct manipulation. Croquet uses Tweak to provide user-extensibility of Croquet objects. The distributed concurrency ideas in Tweak Islands [SRRK05] mix ideas from E and TeaTime [Ree05]. An Island is a vat-like aggregate of objects, where computation within an Island services a queue of pending deliveries sequentially and deterministically. Immediate calls happen only between objects within the same Island. Only FarRefs and promises can span between Islands. Promise pipelining and broken promise contagion are in development, but not yet available.

The unary "`future`" message is used like the E eventual send operator, "`<-`", and means approximately the same thing: this message should be delivered at some future event in the recipient's event sequence. Unlike E, Islands also provides a parameterized form of this operator, e.g., "`future: 100`", where the numeric argument says when, in the Island's future virtual time, this message should be delivered. Messages sent later can be delivered before messages sent earlier, simply by specifying an earlier delivery time. The pending delivery queue is therefore a priority queue sorted by requested delivery time. This notion of virtual time is provided in support of soft real time deadline scheduling, in order to maintain interactive responsiveness.

The only source of non-determinism is deciding on queue order. This corresponds to Actors' "arrival order non-determinism." Once queue order is decided, all computation within an Island proceeds deterministically from there. The most interesting aspect of Islands is that they use this determinism for fault-tolerant replication of an Island. Among all replicas of an Island, at any moment there is a master which resolves arrival-order races to decide on queue order. This master tells all replica Islands this queue order, so they can all locally compute the same future Island states and outgoing messages.

Chapter 27

Conclusions and Future Work

The dynamic reference graph is the fabric of object computation. Objects interact by sending messages on references. Messages carry argument references, composing recipients with arguments, enabling them to interact. An object composes other objects so that their interactions will serve its purposes. Whether through accident or malice, their interaction may cause damage instead. To compose robust systems, we must enable desired behavior while minimizing damage from misbehavior, so we design patterns of defensive consistency and dynamic least authority.

By restricting objects to interact *only* by sending messages on references, the reference graph becomes an access graph supporting fine-grained dynamic least authority. By stretching the graph cryptographically between machines, the reference graph provides decentralized access control with no central points of failure. By controlling *when* references deliver messages, the reference graph forms communicating event-loops, supporting defensive consistency without deadlocks. By allowing remote references to visibly break, event-loops can be distributed and handle disconnects and crashes.

Most viruses abuse authority that they need not have been given in the first place. By following these principles simultaneously at multiple scales of composition, from human organizations down to individual objects, we painlessly provide components the authority they need to use while limiting the authority available for abuse. Systems constructed by these principles would be mostly immune to viruses and other malware.

27.1 Contributions

This dissertation makes seven primary contributions:

- The *object-capability model*, a new model of secure computation that is abstract enough to describe both prior object-capability languages and operating systems, and concrete enough to describe authority-manipulating behavior.

- A novel approach to concurrency control in terms of attenuating authority, including a unified architecture in which both access control and concurrency control concerns may be addressed together.

- A clearly framed distinction between "permission" and "authority." We develop a taxonomy of computable bounds on eventual permission and authority, enabling us to reason about many simple access abstractions. Using this framework, we provide a unified account of access control and authority in distributed systems consisting of object-capability islands sending messages to each other over a cryptographic capability sea.

- A system of reference states, transition rules, and message delivery properties associated with each state, that reify partial failures in a manner that permits robust application-level recovery. In E, these are coupled to language constructs that enable programmers to anticipate and handle distributed plan failures at the syntactic locus where a dependency on remote behavior is introduced. These same syntactic constructs alert the programmer of the loci where interleavings of execution may occur.

- A generalization of prior work on promise pipelining from asymmetric client-server systems to symmetric peer-to-peer systems. Promises enable explicit expression of dataflow in distributed computation. Promise pipelining enables this computational structure to be exploited without cascading round trips.

- An adaptation of non-signaling errors to the propagation of broken references, allowing programmers to consistently *defer* handling of reference failure when direct defense is inappropriate, and to compose chains of data-dependent computation whose reference failure propagation has a sensible semantics. By providing a coherent error reporting behavior for promise pipelines, this mechanism provides a manageable foundation for recovery from delayed errors in distributed dataflow computations.

- Finally, we introduce E-ORDER, a message delivery ordering constraint that achieves an enforceable middle ground between CAUSAL and FIFO message orderings. The resulting ordering is strong enough to prevent adversaries from exploiting race conditions to obtain inappropriate access, but weak enough to be enforceable. We show how the delivery ordering may be defined in terms of an attenuation of the authority provided by passed references. This allows us to reason about the effects of messaging within our unified model of concurrency and access control.

Several programming systems described in Chapter 26 have adopted (in varying subsets) each of the contributions noted above by borrowing elements from the E system. Twisted Python, in particular, is reported to be in use in more than 50 real-world applications at the time of this writing.

27.2 Future Work

The work presented here suggests several directions for future work. One intriguing challenge is to determine how one might combine the high availability supported by group membership systems [BJ87, Ami95] with the support for mutual suspicion and scalability provided by the E model. The two models take very different approaches to security and robustness, but are potentially complementary.

The auditor framework sketched in Section 6.3 provides a user-extensible system for code verification, wherein different auditors can check and certify that a provider's code conforms to user-specified safety properties, without disclosing the code of the provider to the party checking the certification. It is an open research question to determine how this mechanism is most effectively applied, and to explore the limits of the properties that it is able to check.

Spiessens *et al.*'s SCOLL framework for static expression of authority constraints, and static checking of program behaviors against these constraints [SMRS04, SR05a], might usefully be integrated into E, perhaps in a fashion analogous to the integration of type declarations and checking in languages today. If SCOLL's logic can be expressed within E's auditing framework, this would allow user-level experimentation, extension, and coexistence of such checking frameworks outside the language definition.

27.3 Continuing Efforts

Any dissertation necessarily reflects the state of an ongoing project at a particular point in time. There is a vibrant community that has been building on and extending the E language and its runtime system. Interested readers may wish to explore the E web site, which can be found at `www.erights.org`.

Bibliography

[AB87] Malcolm P. Atkinson and Peter Buneman. Types and Persistence in Database Programming Languages. *ACM Computing Surveys*, 19(2):105–190, 1987.

[AB95] Marshall Abrams and David Bailey. Abstraction and Refinement of Layered Security Policy. In Marshall D. Abrams, Sushil Jajodia, and Harold J. Podell, editors, *Information Security: An Integrated Collection of Essays*, pages 126–136. IEEE Computer Society Press, 1995.

[ACC82] Malcolm P. Atkinson, Kenneth Chisholm, and W. Paul Cockshott. PS-Algol: an Algol with a Persistent Heap. *SIGPLAN Notices*, 17(7):24–31, 1982.

[Agh86] Gul A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. The MIT Press Series in Artificial Intelligence. MIT Press, Cambridge, Massachusetts, 1986.

[AHT$^+$02] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative Task Management Without Manual Stack Management. In *USENIX Annual Technical Conference, General Track*, pages 289–302, 2002.

[AL93] Martín Abadi and Leslie Lamport. Composing Specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, January 1993.

[AL95] Martín Abadi and Leslie Lamport. Conjoining Specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, May 1995.

[Ami95] Yair Amir. *Replication Using Group Communication Over a Partitioned Network*. PhD thesis, Institute of Computer Science, The Hebrew University of Jerusalem, 1995.

[And72] J. Anderson. Computer Security Technology Planning Study. Technical Report ESD-TR-73-51, AFSC, Hanscom AFB, Bedford, MA, October 1972. AD-758 206, ESD/AFSC.

[Arm03] Joe Armstrong. *Making Reliable Distributed Systems in the Presence of Software Errors*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, November 27 2003.

[AS86] H. Abelson and G. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1986.

[AW04] Martín Abadi and Ted Wobber. A Logical Account of NGSCB. In *Proc. Formal Techniques for Networked and Distributed Systems – Forte 2004*. Springer-Verlag, Berlin Germany, September 2004.

[Bej96] Arturo Bejar. The Electric Communities Distributed Garbage Collector, 1996.
www.crockford.com/ec/dgc.html.

[BH77] Henry G. Baker, Jr. and Carl E. Hewitt. The Incremental Garbage Collection of Processes. In *Proc. International Joint Conference on Artificial Intelligence*, pages 55–59, Cambridge, Massachusetts, August 1977. IJCAI.

[Bir05] Kenneth P. Birman. *Reliable Distributed Systems.* Springer Verlag, 2005.

[BJ87] Ken Birman and T. Joseph. Exploiting Virtual Synchrony in Distributed Systems. In *SOSP '87: Proc. 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: Proc. Ninth Annual Conference on Object-Oriented Programming Systems, Language, and Applications*, pages 341–354, New York, NY, USA, 1994. ACM Press.

[Bla85] Andrew P. Black. Supporting Distributed Applications: Experience with Eden. In *Proc. Tenth Annual Symposium on Operating Systems Principles*, pages 181–193. ACM, December 1985.

[BNOW94] Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network Objects. In *Proc. Fourteenth ACM Symposium on Operating Systems Principles*, pages 217–230, Asheville, NC, 1994. ACM Press.

[Boe84] W. E. Boebert. On the Inability of an Unmodified Capability Machine to Enforce the *-property. In *Proc. 7th DoD/NBS Computer Security Conference*, pages 291–293, Gaithersburg, MD, USA, September 1984. National Bureau of Standards.

[Boe03] Earl Boebert. Comments on Capability Myths Demolished, 2003.
www.eros-os.org/pipermail/cap-talk/2003-March/001133.html.

[Boe05] Hans-J. Boehm. Threads Cannot be Implemented as a Library. *SIGPLAN Notices*, 40(6):261–268, 2005.

[Bre73] T. H. Bredt. A Survey of Models for Parallel Computing. Technical Report CSL-TR-70-8, Stanford University Computer Systems Laboratory, Stanford, CA, December 1973.

[BS79] Matt Bishop and Lawrence Snyder. The Transfer of Information and Authority in a Protection System. In *Proc. 7th ACM Symposium on Operating Systems Principles*, pages 45–54, December 1979. Published as *Operating System Review*, Vol 13, No 4.

[BST89] H. E. Bal, J. G. Steiner, and A. S. Tanenbaum. Programming Languages for Distributed Computing Systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[Car97] Luca Cardelli. Program Fragments, Linking, and Modularization. In *The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 266–277, Paris, January15–17 1997.

[CC96] Antonio Cau and Pierre Collette. Parallel Composition of Assumption-Commitment Specifications: A Unifying Approach for Shared Variable and Distributed Message Passing Concurrency. *Acta Informatica*, 33(2):153–176, 1996.

[CDM01] A. Chander, D. Dean, and J. Mitchell. A State-Transition Model of Trust Management and Access Control. In *14th IEEE Computer Security Foundations Workshop*, Cape Breton, Nova Scotia, June 2001.

[CF91] R. Cartwright and M. Fagan. Soft Typing. In *Proc. SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 278–292, 1991.

[Chu41] Alonzo Church. *The Calculi of Lambda Conversion.* Number 6 in Annals of Mathematical Studies. Princeton University Press, 1941.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[CL02] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance and Proactive Recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, 2002.

[Cli81] W. D. Clinger. *Foundations of Actor Semantics.* PhD thesis, Dept. of Mathematics, Massachusetts Institute of Technology, Cambridge, MA, 1981.

[Clo04a] Tyler Close. Decentralized Identification, 2004.
`www.waterken.com/dev/YURL/Definition/`.

[Clo04b] Tyler Close. Waterken YURL, 2004.
`www.waterken.com/dev/YURL/httpsy/`.

[Clo04c] Tyler Close. Web-calculus, 2004.
`www.waterken.com/dev/Web/`.

[Clo06] Tyler Close. Credit Transfer Within Market-based Resource Allocation Infrastructure. Tech Report HPL-2006-5, Hewlett Packard Laboratories, 2006.

[Con67] Control Data Corporation. *6400/6500/6600 Computer Systems; Reference Manual.* Control Data Corp., Documentation Dept., Palo Alto, CA, USA, revised May 1967 edition, 1967.
`ed-thelen.org/comp-hist/CDC-6600-R-M.html`.

[Cro97] Douglas Crockford, 1997. Personal communication.

[Cud99] Brian Cudahy. *The Malbone Street Wreck.* Fordham University Press, New York, 1999.

[CW87] D. D. Clark and D. R. Wilson. A Comparison of Commercial and Military Computer Security Policies. In *Proc. 1987 IEEE Symposium on Security and Privacy*, pages 184–195, 1987.

[DA99] T. Dierks and C. Allen. The TLS protocol version 1.0, January 1999. Internet RFC 2246.

[Den87] Daniel C. Dennett. *The Intentional Stance.* MIT Press, Cambridge, Massachusetts, 1987.

[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.

[Dij72] Edsger W. Dijkstra. The Humble Programmer. *Communications of the ACM*, 15(10):859–866, 1972.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[DLP79] Richard A. DeMillo, Richard J. Lipton, and Alan J. Perlis. Social Processes and Proofs of Theorems and Programs. *Communications of the ACM*, 22(5):271–280, 1979.

[DN66] Ole-Johan Dahl and Kristen Nygaard. SIMULA - an ALGOL-based simulation language. *Communications of the ACM*, 9(9):671–678, 1966.

[Don76] Jed Donnelley. A Distributed Capability Computing System, 1976. `nersc.gov/~jed/papers/DCCS/`.

[Don79] James E. Donnelley. Components of a Network Operating System. *Computer Networks*, 3:389–399, 1979.

[DZK$^+$02] Frank Dabek, Nickolai Zeldovich, Frans Kaashoek, David Mazieres, and Robert Morris. Event-driven Programming for Robust Software. In *Proc. 10th ACM SIGOPS European Workshop*, September 2002.

[EA03] Dawson Engler and Ken Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Proc. 19th ACM Symposium on Operating Systems Principles*, pages 237–252, Bolton Landing, NY, October 2003.

[EFL$^+$99] Carl Ellison, Bill Frantz, Butler Lampson, Ronald Rivest, B. Thomas, and T. Ylonen. SPKI Certificate Theory (IETF RFC 2693), September 1999.

[Ell96] Carl Ellison. Establishing Identity Without Certification Authorities. In *Proc. Sixth USENIX Security Symposium*, pages 67–76, Berkeley, 1996. Usenix.

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

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[Fab74] R. S. Fabry. Capability-based Addressing. *Communications of the ACM*, 17(7):403–412, 1974.

[FF02] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. *SIGPLAN Notices*, 37(9):48–59, September 2002.

[Fie00] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, Dept. of Information and Computer Science, University of California at Irvine, 2000.

[Fro03] A. Michael Froomkin. Toward a Critical Theory of Cyberspace. *Harvard Law Review*, 116(3):750–871, 2003.
`www.icannwatch.org/article.pl?sid=03/01/16/057208`.

[FW76a] Daniel P. Friedman and D. S. Wise. Cons Should not Evaluate its Arguments. In S. Michaelson and Robin Milner, editors, *Automata, Languages and Programming*, pages 257–284. Edinburgh University Press, 1976.

[FW76b] Daniel P. Friedman and David S. Wise. The impact of Applicative Programming on Multiprocessing. In *Proc. 1976 International Conference on Parallel Processing (ICPP'76)*, pages 263–272, Walden Woods, Michigan, August 1976. IEEE.

[GD72] G. Scott Graham and Peter J. Denning. Protection — Principles and Practice. *Proc. Spring Joint Computer Conference*, 40:417–429, 1972.

[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 Kay. Smalltalk-72 Instruction Manual. Technical Report SSL 76-6, Learning Research Group, Xerox Palo Alto Research Center, 1976.

[GMPS97] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2. In *USENIX Symposium on Internet Technologies and Systems*, pages 103–112, Monterey, CA, 1997.

[Gon89] Li Gong. A Secure Identity-based Capability System. In *Proc. 1989 IEEE Symposium on Security and Privacy*, pages 56–65, 1989.

[Gon99] Li Gong. *Inside Java 2 Platform Security*. The Java Series. Addison Wesley, 1999.

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

[Gra73] Mark Granovetter. The Strength of Weak Ties. *American Journal of Sociology*, 78:1360–1380, 1973.

[Gra86] Jim Gray. Why Do Computers Stop and What Can Be Done About It? In *Proc. 5th Symposium on Reliability in Distributed Software and Database Systems*, Los Angeles, CA, January 1986.

[Gut04] Peter Gutmann. *Cryptographic Security Architecture: Design and Verification.* Springer Verlag, 2004.

[Hal85] Robert H. Halstead, Jr. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985.

[Han93] 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.

[Har85] Norman Hardy. KeyKOS Architecture. *SIGOPS Operating Systems Review*, 19(4):8–25, 1985.

[Har88a] Norm Hardy. The Confused Deputy. *Operating Systems Review*, October 1988.

[Har88b] David Harel. On Visual Formalisms. *Communications of the ACM*, 31(5):514–530, 1988.

[Hay37] Friedrich Hayek. Economics and Knowledge. *Economica IV*, pages 33–54, 1937.

[Hay45] Friedrich Hayek. The Uses of Knowledge in Society. *American Economic Review*, 35:519–530, September 1945.

[Hay64] Friedrich Hayek. The Theory of Complex Phenomena. In M Bunge, editor, *The Critical Approach to Science and Philosophy*, pages 332–349. Collier McMillan, London, 1964.

[HB78] Carl Hewitt and Henry Baker. Actors and Continuous Functionals. In E. J. Neuhold, editor, *Formal Description of Programming Concepts*, pages 367–390. North-Holland, Amsterdam, 1978.

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In Nils J. Nilsson, editor, *Proc. 3rd International Joint Conference on Artificial Intelligence*, pages 235–245, Standford, CA, August 1973. William Kaufmann.

[HDH03] Martin Hirzel, Amer Diwan, and Matthew Hertz. Connectivity-based Garbage Collection. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '03)*, pages 359–373, 2003.

[Hew77] Carl Hewitt. Viewing Control Structures as Patterns of Passing Messages. *Journal of Artificial Intelligence*, 8(3):323–364, 1977.

[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.

[Hoa69] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 583, October 1969.

[Hoa74] C. A. R. Hoare. Monitors: an Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, 1974.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[HP97] Lenwood S. Heath and Sriram V. Pemmaraju. Stack and Queue Layouts of Posets. *SIJDM: SIAM Journal on Discrete Mathematics*, 10(4):599–625, November 1997.

[HPW03] Michael Howard, Jon Pincus, and Jeannette Wing. Measuring Relative Attack Surfaces. Technical Report CMU-TR-03-169, Carnegie Mellon University, August 2003. Proc. 2003 Workshop on Advanced Developments in Software and Systems Security
`www.cs.cmu.edu/~wing/publications/Howard-Wing03.pdf`.

[HRB$^+$87] Norman C. Hutchinson, Rajindra K. Raj, Andrew P. Black, Henry M. Levy, and Eric Jul. The Emerald Programing Lanuage Report. Technical Report 87-10-07, Department of Computer Science, University of Washington, Seattle, October 1987.

[HRU75] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. On Protection in Operating Systems. In *The Symposium on Operating Systems Principles*, pages 14–24, 1975.

[HT91] Kohei Honda and Mario Tokoro. An Object Calculus for Asynchronous Communication. In *European Conference on Object Oriented Programming*, pages 133–147, 1991.

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

[IoEE85] American National Standards Institute, Institute of Electrical, and Electronic Engineers. IEEE Standard for Binary Floating-Point Arithmetic. *ANSI/IEEE Standard, Std 754-1985,* New York, 1985.

[JLS76] Anita K. Jones, Richard J. Lipton, and Lawrence Snyder. A Linear Time Algorithm for Deciding Security. In *Foundations of Computer Science*, pages 33–41, 1976.

[Jon73] Anita K. Jones. *Protection in Programmed Systems.* PhD thesis, Department of Computer Science, Carnegie-Mellon University, June 1973.

[Jon83] Cliff B. Jones. Specification and Design of (Parallel) Programs. In *IFIP Congress*, pages 321–332, 1983.

[Jon03] Cliff B. Jones. The Early Search for Tractable Ways of Reasoning about Programs. *IEEE Annals of the History of Computing*, 25(2):26–49, 2003.

[Kah96] William Kahan. Lecture Notes on the Status of the IEEE Standard 754 for Binary Floating Point Arithmetic.
`www.cs.berkeley.EDU/~wkahan/ieee754status/ieee754.ps`, 1996.

[Kar03] Alan H. Karp. Enforce POLA on Processes to Control Viruses. *Communications of the ACM*, 46(12):27–29, 2003.

[Kay93] Alan C. Kay. The Early History Of Smalltalk. *SIGPLAN Notices*, 28(3):69–95, 1993.

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

[KCR98] R. Kelsey, W. Clinger, and J. Rees. Revised^5 Report on the Algorithmic Language Scheme. *ACM Sigplan Notices*, pages 26–76, 1998.

[Key81] Key Logic, Inc. Gnosis Design Documentation, 1981.
`www.agorics.com/Library/KeyKos/Gnosis/keywelcome.html`.

[Key86] Key Logic, Inc. *U.S. Patent 4,584,639: Computer Security System.* U. S. Patent Office, 1986.

[KGRB01] Alan H. Karp, Rajiv Gupta, Guillermo Rozas, and Arindam Banerji. The Client Utility Architecture: The Precursor to E-Speak. Technical Report HPL-2001-136, Hewlett Packard Laboratories, June 09 2001.

[KGRB03] Alan H. Karp, R. Gupta, G. J. Rozas, and A. Banerji. Using Split Capabilities for Access Control. *IEEE Software*, 20(1):42–49, January 2003.

[KH84] P. A. Karger and A. J. Herbert. An Augmented Capability Architecture to Support Lattice Security and Traceability of Access. In *Proc. 1984 IEEE Symposium on Security and Privacy*, pages 2–12, Oakland, CA, April 1984. IEEE.

[KK02] Alan H. Karp and Vana Kalogeraki. The Client Utility as a Peer-to-Peer System. In *Revised Papers from the NETWORKING 2002 Workshops on Web Engineering and Peer-to-Peer Computing*, pages 260–273, London, UK, 2002. Springer-Verlag.

[KL86] Richard Y. Kain and Carl E. Landwehr. On Access Checking in Capability Systems. In *Proc. 1986 IEEE Symposium on Security and Privacy*, pages 95–100, 1986.

[KM66] Richard M. Karp and Raymond E. Miller. Properties of a Model for Parallel Computations: Determinacy, Termination, Queueing. *SIAM J. Appl. Math.*, 14(6):1390–1411, November 1966.

[KM88] Kenneth M. Kahn and Mark S. Miller. Language Design and Open Systems. In Bernardo A. Huberman, editor, *Ecology of Computation*. Elsevier Science Publishers, North-Holland, 1988.

[Kri80] Saul A. Kripke. *Naming and Necessity*. Library of Philosophy and Logic. Blackwell, Oxford, 1980.

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

[Lac56] Ludwig Lachmann. *Capital and its Structure*. Kansas City: Sheed Andrews and McNeel Inc., 1956.

[Lam73] Butler W. Lampson. A Note on the Confinement Problem. *Communications of the ACM*, 16(10):613–615, 1973.

[Lam74] Butler W. Lampson. Protection. *Operating Systems Review*, 8(1):18–24, January 1974.

[Lam78] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[Lam98] Leslie Lamport. The Part-time Parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[Lan92] Charles R. Landau. The Checkpoint Mechanism in KeyKOS. In *Proc. Second International Workshop on Object Orientation in Operating Systems*, pages 86–91. IEEE, September 1992.

[Lee06] Edward A. Lee. The Problem with Threads. Technical Report UCB/EECS-2006-1, EECS Department, University of California, Berkeley, January 10 2006.
`www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-1.html`.

[Lef03] Glyph Lefkowitz. Generalization of Deferred Execution in Python, 2003.
`python.org/pycon/papers/deferex/`.

[Lev84] Henry M. Levy. *Capability-based Computer Systems*. Digital Press, 1984.

[LG86] B. Liskov and J. Guttag. *Abstraction and Specification in Program Developement*. MIT Press, Cambridge, Mass., 1986.

[LN79] Hugh C. Lauer and Roger M. Needham. On the Duality of Operating System Structures. In *Proc. 2nd International Symposium on Operating Systems*, pages 3–19. IRIA, October 1979. ACM Operating System Review.

[LS76] Butler W. Lampson and Howard E. Sturgis. Reflections on an Operating System Design. *Communications of the ACM*, 19(4):251–265, May 1976.

[LS83] Barbara Liskov and Robert Scheifler. Guardians and Actions: Linguistic Support for Robust, Distributed Programs. *Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[LS88] Barbara Liskov and Lubia Shrira. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In *PLDI '88: Proc. ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 260–267, New York, NY, USA, 1988. ACM Press.

[LSA77] Barbara H. Liskov, A. Snyder, and Russell R. Atkinson. Abstraction Mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, August 1977.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[LY96] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley, Inc., Reading, MA, 1996.

[LZ74] Barbara Liskov and Stephen N. Zilles. Programming with Abstract Data Types. *SIGPLAN Notices*, 9(4):50–59, 1974.

[M$^+$62] John McCarthy et al. *LISP 1.5 Programmer's Manual.* MIT Press, Cambridge, Massachusetts, 1962.

[Mad00] Ole Lehrmann Madsen. Abstraction and Modularization in the BETA Programming Language. In *Proc. Joint Modular Languages Conference on Modular Programming Languages*, Lecture Notes in Computer Science, pages 211–237, 2000.

[Man83] B. B. Mandelbrot. *The Fractal Geometry of Nature.* W. H. Freeman and Company, New York, 1983.

[MBTL87] Mark S. Miller, Daniel G. Bobrow, E. Dean Tribble, and Jacob Levy. Logical Secrets. In *International Conference on Logic Programming*, pages 704–728, 1987.

[MD88] Mark S. 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.

[Mey87] Bertrand Meyer. Reusability: The Case for Object-Oriented Design. *IEEE Software*, 4(2):50–64, 1987.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall, 1988.

[Mey92] Bertrand Meyer. Applying "Design by Contract". *IEEE Computer*, 25(10):40–51, 1992.

[Mil76] Harlan D. Mills. Software Development. *IEEE Trans. Software Eng.*, 2(4):265–273, 1976.

[Mil83] Robin Milner. Calculi for Synchrony and Asynchrony. *Theoretical Computer Science*, pages 267–310, 1983.

[Mil84] Robin Milner. A Proposal for Standard ML. In *ACM Symposium on Lisp and Functional Programming*, pages 184–197, August 1984.

[Mil89] Robin Milner. *Communication and Concurrency.* International Series in Computer Science. Prentice Hall, New York, 1989.

[Mil92] Mark S. Miller. The Promise System, 1992.
`sunless-sea.net/Transcripts/promise.html`.

[Mil99] Robin Milner. *Communicating and Mobile Systems: The π Calculus.* Cambridge University Press, Cambridge, England, 1999.

[Mil04] Mark S. Miller. Inheritance as a Static Pattern of Message Passing, 2004.
`www.erights.org/elang/blocks/inheritance.html`.

[MKH$^+$96] Mark S. Miller, David Krieger, Norman Hardy, Chris Hibbert, and E. Dean Tribble. An Automatic Auction in ATM Network Bandwidth. In S. H. Clearwater, editor, *Market-based Control, A Paradigm for Distributed Resource Allocation.* World Scientific, Palo Alto, CA, 1996.

[MMF00] Mark S. Miller, Chip Morningstar, and Bill Frantz. Capability-based Financial Instruments. In *Proc. Financial Cryptography 2000*, pages 349–378, Anguila, BWI, 2000. Springer-Verlag.
`www.erights.org/elib/capability/ode/index.html`.

[Mor73a] James H. Morris, Jr. Protection in Programming Languages. *Communications of the ACM*, 16(1):15–21, 1973.

[Mor73b] James H. Morris, Jr. Types are Not Sets. In *Principles of Programming Languages*, pages 120–124, 1973.

[MPSV00] Rajeev Motwani, Rina Panigrahy, Vijay A. Saraswat, and Suresh Ventkatasubramanian. On the Decidability of Accessibility Problems (extended abstract). In *Proc. Thirty-Second Annual ACM Symposium on Theory of Computing (STOC '00)*, pages 306–315, New York, NY, USA, 2000. ACM Press.

[MS88] Jonathan D. Moffett and Morris Sloman. The Source of Authority for Commercial Access Control. *IEEE Computer*, 21(2):59–69, 1988.

[MS03] Mark S. Miller and Jonathan S. Shapiro. Paradigm Regained: Abstraction Mechanisms for Access Control. In *Proc. Eigth Asian Computing Science Conference*, pages 224–242, Tata Institute of Fundamental Research, Mumbai, India, 2003.

[MTS04] Mark S. Miller, Bill Tulloh, and Jonathan S. Shapiro. The Structure of Authority: Why Security Is Not a Separable Concern. In *Multiparadigm Programming in Mozart/Oz: Extended Proc. Second International Conference MOZ 2004*, pages 2–20, 2004.

[MTS05] Mark S. Miller, E. Dean Tribble, and Jonathan S. Shapiro. Concurrency Among Strangers: Programming in E as Plan Coordination. In *Trustworthy Global Computing, International Symposium, TGC 2005, Edinburgh, UK, April 7-9, 2005, Revised Selected Papers*, pages 195–229, 2005.

[MW06] Adrian Matthew Mettler and David Wagner. The Joe-E Language Specification (draft). Technical Report UCB/EECS-2006-26, EECS Department, University of California, Berkeley, March 17 2006. `www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-26.html`.

[MYS03] Mark S. Miller, Ka-Ping Yee, and Jonathan S. Shapiro. Capability Myths Demolished. Technical Report Report SRL2003-02, Systems Research Laboratory, Department of Computer Science, Johns Hopkins University, mar 2003.

[NBB$^+$63] Peter Naur, John W. Backus, Friedrich L. Bauer, Julien Green, C. Katz, John L. McCarthy, Alan J. Perlis, Heinz Rutishauser, Klaus Samelson, Bernard Vauquois, Joseph Henry Wegstein, Adriaan van Wijngaarden, and Michael Woodger. Revised Report on the Algorithmic Language ALGOL 60. *Communications of the ACM*, 6(1):1–17, 1963.

[NBF$^+$80] P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, and L. Robinson. A Provably Secure Operating System: The System, Its Applications, and Proofs. Technical Report Report CSL-116, Computer Science Laboratory, may 1980.

[OBDMS98] Özalp Babaoğlu, Renzo Davoli, Alberto Montresor, and Roberto Segala. System Support for Partition-aware Network Applications. *SIGOPS Operating Systems Review*, 32(1):41–56, 1998.

[oD85] U.S. Department of Defense. *Trusted Computer System Evaluation Criteria*, 1985.

[Ous96] John K. Ousterhout. *Why Threads are a Bad Idea (For Most Purposes)*, 1996. Invited talk, 1996 USENIX Technical Conference.

[Par72] David Lorge Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[PB96] Leonard J. La Padula and David Elliott Bell. Secure Computer Systems: A Mathematical Model. *Journal of Computer Security*, 4:239–263, 1996.

[PG74] Gerald J. Popek and Robert P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. *Communications of the ACM*, 17(7):412–421, 1974.

[Plo75] Gordon Plotkin. Call-by-Name, Call-by-Value, and the λ-Calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[PNFB05] Alex Potanin, James Noble, Marcus R. Frean, and Robert Biddle. Scale-free Geometry in OO Programs. *Communications of the ACM*, 48(5):99–103, 2005.

[Pos81] John Postel. Transmission Control Protocol. RFC 793, Internet Society (IETF), 1981.

[PS01] Vassilis Prevelakis and Diomidis Spinellis. Sandboxing Applications. In *USENIX 2001 Technical Conference Proceedings: FreeNIX Track*, pages 119–126, Berkeley, CA, June 2001. USENIX Association.

[PT00] Benjamin C. Pierce and David N. Turner. Pict: A Programming Language Based on the Pi-Calculus. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pages 455–494. MIT Press, 2000.

[QGC00] Zhenyu Qian, Allen Goldberg, and Alessandro Coglio. A Formal Specification of Java Class Loading. In *Proc. 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*, pages 325–336, New York, NY, USA, 2000. ACM Press.

[RA82] Jonathan A. Rees and Norman I. Adams. T: A Dialect of Lisp or, LAMBDA: The Ultimate Software Tool. In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 114–122, 1982.

[Raj89] Susan A. Rajunas. The KeyKOS/KeySAFE System Design. Technical Report SEC009-01, Key Logic, Inc., March 1989. www.cis.upenn.edu/~KeyKOS.

[Red74] David D. Redell. *Naming and Protection in Extensible Operating Systems.* PhD thesis, Department of Computer Science, University of California at Berkeley, November 1974.

[Ree78] David Patrick Reed. *Naming and Synchronization in a Decentralized Computer System.* PhD thesis, Dept. of Electrical Engineering, Massachusetts Institute of Technology, 1978.

[Ree96] Jonathan A. Rees. A Security Kernel Based on the Lambda-Calculus. Technical report, Massachusetts Institute of Technology, 1996.

[Ree05] David Patrick Reed. Designing Croquet's TeaTime: a Real-time, Temporal Environment for Active Object Cooperation. In *OOPSLA Companion*, page 7, 2005.

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

[Rep99] John H. Reppy. *Concurrent Programming in ML.* Cambridge University Press, 1999.

[Rey70] J. C. Reynolds. Gedanken: A Simple Typeless Language Based on the Principle of Completeness and the Reference Concept. *Communications of the ACM*, 13(5):308–318, May 1970.

[RH04] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming.* MIT Press, March 2004.

[Rod67] J. E. Rodriguez. *A Graph Model for Parallel Computation.* PhD thesis, Dept. of Electrical Engineering, Massachusetts Institute of Technology, Cambridge, MA, 1967.

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

[Sch62] Jacob T. Schwartz. *Relativity in Illustrations.* New York University Press, New York, 1962.

[Sch03] Fred B. Schneider. Least Privilege and More. *IEEE Security and Privacy*, 1(5):55–59, September/October 2003.

[Sch06] Christian Scheideler. Towards a Paradigm for Robust Distributed Algorithms and Data Structures. In *6th Int. Heinz Nixdorf Institute Symposium on New Trends in Parallel and Distributed Computing*, 2006.

[SDN$^+$04] Jonathan S. Shapiro, M. Scott Doerrie, Eric Northup, Swaroop Sridhar, and Mark S. Miller. Towards a Verified, General-Purpose Operating System Kernel. In G. Klein, editor, *Proc. NICTA Invitational Workshop on Operating System Verification*, pages 1–19, National ICT Australia, 2004.

[Sea05] Mark Seaborn. Plash: The Principle of Least Authority Shell, 2005.
`plash.beasts.org/`.

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

[Sha86] Marc Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle. In *International Conference on Distributed Computing Systems*, pages 198–204, Cambridge, Massachusetts, 1986.

[Sim62] Herbert A. Simon. The Architecture of Complexity. *Proc. American Philosophical Society*, 106:467–482, 1962.

[SJ03] Vijay A. Saraswat and Radha Jagadeesan. Static Support for Capability-based Programming in Java, March 16 2003.
`www.cse.psu.edu/~saraswat/neighborhood.pdf`.

[SJR86] R. D. Sansom, D. P. Julin, and R. F. Rashid. Extending a Capability Based System into a Network Environment. In *Proc. 1986 ACM SIGCOMM Conference*, pages 265–274, August 1986.

[Ski90] Steven Skiena. Hasse Diagrams. In *Implementing Discrete Mathematics: Combinatorics and Graph Theory With Mathematica*, pages 163, 169–170, 206–208. Addison-Wesley, Reading, MA., 1990.
See also `en.wikipedia.org/wiki/Hasse_diagram`.

[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] Ehud Y. 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.

[SM02] Marc Stiegler and Mark S. Miller. A Capability Based Client: The DarpaBrowser. Technical Report Focused Research Topic 5 / BAA-00-06-SNK, Combex, Inc., June 2002.
www.combex.com/papers/darpa-report/index.html.

[SM03] Jeremy Sussman and Keith Marzullo. The Bancomat Problem: an Example of Resource Allocation in a Partitionable Asynchronous System. *Theoretical Computer Science*, 291(1):103–131, 2003.

[Smi84] Brian Cantwell Smith. Reflection and Semantics in LISP. In *Conf. record, 11th annual ACM symp. on principles of programming languages*, pages 23–35, Salt Lake City, jan 1984.

[SMRS04] Fred Spiessens, Mark S. Miller, Peter Van Roy, and Jonathan S. Shapiro. Authority Reduction in Protection Systems.
www.info.ucl.ac.be/people/fsp/ARS.pdf, 2004.

[SMTH91] Jonathan S. Shapiro, Mark S. Miller, E. Dean Tribble, and Chris Hibbert. *The Xanadu Developer's Guide*. Palo Alto, CA, USA, 1991.

[SR05a] Fred Spiessens and Peter Van Roy. A Practical Formal Model for Safety Analysis in Capability-based Systems. In *Trustworthy Global Computing, International Symposium, TGC 2005, Edinburgh, UK, April 7-9, 2005, Revised Selected Papers*, Lecture Notes in Computer Science, pages 248–278. Springer-Verlag, 2005. Presentation available at www.info.ucl.ac.be/people/fsp/auredsysfinal.mov.

[SR05b] 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 Proc. Second International Conference MOZ 2004*, volume 3389 of *Lecture Notes in Computer Science*, pages 21–40. Springer-Verlag, 2005.

[SRRK05] David A. Smith, Andreas Raab, David Patrick Reed, and Alan Kay. Hedgehog Architecture, 2005.
croquetproject.org/Site PDFs/2005 Hedgehog Architecture.pdf.

[SS75] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. *Proc. IEEE*, 63(9):1278–1308, September 1975.

[SS86] M. Safra and Ehud Y. Shapiro. Meta Interpreters for Real. In H. J. Kugler, editor, *Information Processing '86*, pages 271–278. North-Holland, Amsterdam, 1986.

[SS87] William R. Shockley and Roger R. Schell. TCB Subsets for Incremental Evaluation. In *Proc. Third AIAA Conference on Computer Security*, pages 131–139, December 1987.

[SS04] Paritosh Shroff and Scott F. Smith. Type Inference for First-Class Messages with Match-Functions. In *International Workshops on Foundations of Object-Oriented Languages*, Venice, Italy, 2004.

[SSF99] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A Fast Capability System. In *SOSP '99: Proc. 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 Computing*, 1(1):25–48, 1983.

[ST94] S. W. Smith and J. D. Tygar. Security and Privacy for Partial Order Time. Technical Report CMU-CS-94-135, Dept. of Computer Science, Carnegie Mellon University, 1994.

[Sta85] E. W. Stark. A Proof Technique for Rely/Guarantee Properties. In S. N. Maheshwari, editor, *Foundations of Software Technology and Theoretical Computer Science: Proceedings of the 5th Conference*, volume 206, pages 369–391, New Delhi, 1985. Springer-Verlag.

[Ste78] G. L. Steele. Rabbit: A Compiler for Scheme. AI Technical Report 474, MIT Artificial Intelligence Laboratory, May 1978.

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

[Sti05] Marc Stiegler. An Introduction to Petname Systems. In *Advances in Financial Cryptography Volume 2*. Ian Grigg, 2005.
`www.financialcryptography.com/mt/archives/000499.html`.

[Sti06] Marc Stiegler. Emily, a High Performance Language for Secure Cooperation, 2006.
`www.skyhunter.com/marcs/emily.pdf`.

[SW00] Jonathan S. Shapiro and Samuel Weber. Verifying the EROS Confinement Mechanism. In *Proc. 2000 IEEE Symposium on Security and Privacy*, pages 166–176, 2000.

[SW04] Gerhard Schlosser and Gunter Wagner. *Modularity in Development and Evolution.* University of Chicago Press, 2004.

[Tin92] Mario Tinto. The Design and Evaluation of INFOSEC Systems: The Computer Security Contribution to the Composition Discussion, June 1992. C Technical Report 32-92, NSA.

[TM06] Bill Tulloh and Mark S. Miller. Institutions as Abstraction Boundaries. In Jack High, editor, *Humane Economics: Essays in Honor of Don Lavoie*, pages 136–188. Edward Elgar Publishing, Cheltenham, UK, 2006.

[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] E. Dean Tribble, Mark S. Miller, Kenneth M. Kahn, Daniel G. Bobrow, Curtis Abbott, and Ehud Y. Shapiro. Channels: A Generalization of Streams. In *International Conference on Logic Programming*, pages 839–857, 1987.

[TMvR86] A. S. Tanenbaum, S. J. Mullender, and R. van Renesse. Using Sparse Capabilities in a Distributed Operating System. In *Proc. 6th International Symposium on Distributed Computing Systems*, pages 558–563. IEEE, 1986.

[Tri93] E. Dean Tribble, 1993. Personal communication.

[Tri98] E. Dean Tribble. Re: Types, Makers, and Inheritance, October 1998.
`www.eros-os.org/pipermail/e-lang/1998-October/002024.html`.

[Tru02] Trusted Computing Platform Alliance. Trusted Platform Module Protection Profile, July 2002.

[vBCB03] J. Robert von Behren, Jeremy Condit, and Eric A. Brewer. Why Events Are a Bad Idea (for High-Concurrency Servers). In *Proceedings of HotOS IX: The 9th Workshop on Hot Topics in Operating Systems*, pages 19–24, 2003.

[VCKD99] R. Vitenberg, G. V. Chockler, I. Keidar, and Danny Dolev. Group Communication Specifications: A Comprehensive Study. Technical Report MIT-LCS-TR-790, Massachusetts Institute of Technology, 1999.

[vDABW96] Leendert van Doorn, Martín Abadi, M. Burrows, and Edward P. Wobber. Secure Network Objects. In *Proc. 1996 IEEE Symposium on Security and Privacy*, pages 211–221, 1996.

[vECC$^+$99] Thorsten von Eicken, Chi-Chao Chang, Grzegorz Czajkowski, Chris Hawblitzel, Deyu Hu, and Dan Spoonhower. J-Kernel: A Capability-based Operating System for Java. In *Secure Internet Programming*, pages 369–393, 1999.

[Vin97] S. Vinoski. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. *IEEE Communications Magazine*, 35(2):46–55, 1997.

[vWMPK69] Adriaan van Wijngaarden, B. J. Mailloux, J. E. L. Peck, and C. H. A. Koster. Report on the Algorithmic Language ALGOL 68. *Numerische Mathematik*, 14(2):79–218, 1969.

[WBDF97] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible Security Architectures for Java. In *Proc. 16th ACM Symposium on Operating Systems Principles*, pages 116–128, Saint-Malo, France, October 1997.

[WBM03] Rebbeca Wirfs-Brock and Alan McKean. *Object Design — Roles, Responsibilities and Collaborations.* Addison-Wesley, 2003.

[WCB01] Matt Welsh, David Culler, and Eric Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In Greg Ganger, editor, *Proc. 18th ACM Symposium on Operating Systems Principles*, volume 35, 5 of *ACM SIGOPS Operating Systems Review*, pages 230–243, New York, October 21–24 2001. ACM Press.

[WCC^{+}74] William A. Wulf, Ellis S. Cohen, William M. Corwin, Anita K. Jones, Roy Levin, C. Pierson, and Fred J. Pollack. HYDRA: The Kernel of a Multiprocessor Operating System. *Communications of the ACM*, 17(6):337–345, 1974.

[Weg87] Peter Wegner. Dimensions of Object-based Language Design. In *Proc. ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 168–182, December 1987.

[Wir71] Niklaus Wirth. Program Development by Stepwise Refinement. *Communications of the ACM*, 14(4):221–227, 1971.

[WLH81] William A. Wulf, Roy Levin, and Samuel P. Harbison. *HYDRA/C.mmp: An Experimental Computer System*. McGraw Hill, 1981.

[WN79] M. V. Wilkes and R. M. Needham. *The Cambridge CAP Computer and its Operating System*. Elsevier North Holland, 1979.

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

[WRW96] Ann Wollrath, Roger Riggs, and Jim Waldo. A Distributed Object Model for the Java System. In *2nd Conference on Object-Oriented Technologies & Systems (COOTS)*, pages 219–232. USENIX Association, July 24 1996.

[WT02] David Wagner and E. Dean Tribble. A Security Analysis of the Combex DarpaBrowser Architecture, March 2002.
`www.combex.com/papers/darpa-review/`.

[Yee02] Ka-Ping Yee. User Interaction Design for Secure Systems. In *ICICS: International Conference on Information and Communications Security*, pages 278–290, 2002.

[Yee04] Ka-Ping Yee. Aligning Security and Usability. *IEEE Security and Privacy*, 2(5):48–55, September/October 2004.

[YM03] Ka-Ping Yee and Mark S. Miller. Auditors: An Extensible, Dynamic Code Verification Mechanism, 2003.
`www.erights.org/elang/kernel/auditors/index.html`.

[Zus41] Konrad Zuse. Patentanmeldung Z-391, 1941.

[Zus49] Konrad Zuse. Rechenmaschine zur Durchfhrung arithmetischer Rechenoperationen (z.B. Multiplikationen), 1949.
See `www.epemag.com/zuse/part4c.htm`.

[Zus59] Konrad Zuse. Über den "Plankalkül". *Elektronische Rechenanlagen*, 1(2):68–71, 1959.

[ZYD$^+$03] Nickolai Zeldovich, Alexander Yip, Frank Dabek, Robert Morris, David Mazières, and M. Frans Kaashoek. Multiprocessor Support for Event-Driven Programs. In *USENIX 2003 Annual Technical Conference, General Track*, pages 239–252, 2003.

Vita

Education

Johns Hopkins University, Baltimore, Maryland. *Advisor: Jonathan S. Shapiro.* Ph.D., Computer Science, 2006.

Yale University, New Haven, Connecticut. B.S., Computer Science, 1980.

Employment

Open Source Coordinator, ERights.org *1998 to Present.* Leading the open source design and development of the E programming language.

Visiting Scientist, Virus-Safe Computing Initiative, Hewlett-Packard Laboratories, *2003 to Present.* Contributor to projects applying least authority (Polaris, CapDesk, DarpaBrowser), secure naming for phishing-proof browsing (Petname tool, Passpet), rights-transfer protocol design (IOU protocol), decentralized computational market infrastructure (DonutLab).

Chief Technology Officer, Combex, Inc. *2000 to 2003.* Founder. Contributor to CapDesk and the DarpaBrowser.

Director, Extropy Institute *1997 to 2003.* The Extropy Institute promotes understanding of both the promises and dangers of future technologies.

Chief Architect, E platform, Electric Communities, Inc. *1996 to 1998.* Led both the Original-E and E efforts. Brought Original-E to sufficient maturity to support Electric Communities Habitats, a secure decentralized virtual world. Chief architect of Habitat's security architecture. Contributor to Habitat's distributed object architecture.

Senior Software Architect, Agorics, Inc. *1994 to 1996.* Co-Founder. Technical lead of Sun Lab's WebMart framework for object-capability-oriented electronic commerce. Built and demonstrated at Interop an ATM Network Bandwidth Auction, six months after starting the Sun Labs contract. Contributor to the Joule language.

Co-Director, George Mason University Agorics Project, *1994 to 2001.* We explored the question: What can economics (especially plan-coordination economics) learn from software engineering?

Co-Architect, Project Xanadu, Xanadu Inc. *1989 to 1994.* Xanadu was a distributed hypertext publishing system that preceded and influenced the Web. Initiated the Trusty Scheme project, which built a W7-like secure programming language for enabling safe active content in hypertext pages.

Consultant, Xerox Palo Alto Research Center. *1985 to 1988.* Co-creator of the agoric paradigm of market-based computation. Co-founder of the Vulcan project.

Software engineer, Datapoint Technology Center, *1980 to 1985.* Created the first commercial distributed window system, VistaView.

Publications

Marc Stiegler, Alan H. Karp, Ka-Ping Yee, Tyler Close, and Mark S. Miller. "Polaris: Virus Safe Computing for Windows XP." *Communications of the ACM.* 49(9) pages 83–88, September 2006. `doi.acm.org/10.1145/1151030.1151033` `www.hpl.hp.com/techreports/2004/HPL-2004-221.html`.

Mark Samuel Miller. "Robust Composition: Towards a Unied Approach to Access Control and Concurrency Control." PhD thesis, Johns Hopkins University, Baltimore, Maryland, USA, May 2006. `www.erights.org/talks/thesis/`

Mark S. Miller, E. Dean Tribble, and Jonathan S. Shapiro. "Concurrency Among Strangers: Programming in E as Plan Coordination." *Proc. 2005 Symposium on Trustworthy Global Computing*, European Joint Conference on Theory and Practice of Software 2005. Appears in volume 3705 of Lecture Notes in Computer Science, pages 195–229. Springer, 2005. `www.erights.org/talks/promises/index.html`

Mark S. Miller, Bill Tulloh, and Jonathan S. Shapiro. "The Structure of Authority: Why Security Is Not a Separable Concern." *Proc 2nd International Conference on Multiparadigm Programming in Mozart/OZ* (MOZ/2004), Charleroi Belguim, pages 2–20, October 2004. `www.erights.org/talks/no-sep/`

Jonathan S. Shapiro, M. Scott Doerrie, Eric Northup, Swaroop Sridhar, and Mark S. Miller. "Towards a Verified, General-Purpose Operating System Kernel," *NICTA Invitational Workshop on Operating System Verifica-*

tion, University of New South Wales, Sydney, Australia, October 2004. www.coyotos.org/docs/osverify-2004/osverify-2004.html

Fred Spiessens, Mark S. Miller, Peter Van Roy, and Jonathan S. Shapiro. "Authority Reduction in Protection Systems." 2004. www.info.ucl.ac.be/people/fsp/ARS.pdf

Mark S. Miller and Jonathan S. Shapiro. "Paradigm Regained: Abstraction Mechanisms for Access Control." *Eighth Asian Computing Science Conference* (ASIAN '03), Tata Institute of Fundamental Research, Mumbai India, pages 224–242, December 1013 2003. www.erights.org/talks/asian03/

Mark S. Miller, Ka-Ping Yee, and Jonathan S. Shapiro. "Capability Myths Demolished," Report SRL2003-02, Systems Research Laboratory, Department of Computer Science, Johns Hopkins University. 2003. srl.cs.jhu.edu/pubs/SRL2003-02.pdf

Mark S. Miller and Marc Stiegler. "The Digital Path: Smart Contracting and the Third World," in *Austrian Perspectives on the Internet Economy*, 2003 (Routledge). www.erights.org/talks/pisa/paper/, www.cypherpunks.to/erights/talks/pisa/paper/digital-path.pdf

Bill Tulloh and Mark S. Miller. "Institutions as Abstraction Boundaries." In Jack High, editor, *Social Learning: Essays in Honor of Don Lavoie.* 2002. www.erights.org/talks/categories/

Marc Stiegler and Mark S. Miller. "A Capability Based Client: The DarpaBrowser." Technical Report Focused Research Topic 5 / BAA-00-06- SNK, Combex, Inc., June 2002. www.combex.com/papers/darpa-report/index.html.

Mark S. Miller, Chip Morningstar, and Bill Frantz. "Capability-based Financial Instruments." *Proc. Financial Cryptography 2000*, pages 349–378, 2000. www.erights.org/elib/capability/ode/

Mark S. Miller. "Learning Curve." *Reason Magazine*, December 1996. reason.com/9612/bk.miller.shtml

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.

Mark S. Miller, David Krieger, Norman Hardy, Chris Hibbert, and E. Dean Tribble. "An Automatic Auction in ATM Network Bandwidth." In S. H. Clearwater, editor, *Market-based Control, A Paradigm for Distributed Resource Allocation.* World

Scientific, Palo Alto, CA, 1996.

Mark S. Miller, E. Dean Tribble, Ravi Pandya, and Marc Stiegler. “The Open Society and its Media.” In Markus Krummenacker, James Lewis, editor, *Prospects in Nanotechnology.* Proceedings of the 1992 First General Conference on Nanotechnology: Development, Applications, and Opportunities. Wiley, 1995. `www.caplet.com/papers/open-media.html`

Ted Kaehler, Hadon Nash, Mark S. Miller. “Betting, Bribery, and Bankruptcy—A Simulated Economy that Learns to Predict.” in *IEEE CompCon Proceedings*, pp. 357–361, 1989

Mark S. 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. `www.agorics.com/Library/agoricpapers.html`, `www.cypherpunks.to/erights/talks/agoric/mnc-aos.pdf`

Mark S. Miller and K. Eric Drexler. “Comparative Ecology: A Computational Perspective” In Bernardo Huberman, editor, *The Ecology of Computation*, North-Holland, 1988. `www.agorics.com/Library/agoricpapers.html`

K. Eric Drexler and Mark S. Miller. “Incentive Engineering for Computational Resource Management” In Bernardo Huberman, editor, *The Ecology of Computation*, North-Holland, 1988. `www.agorics.com/Library/agoricpapers.html`

Kenneth M. Kahn and Mark S. Miller. “Language Design and Open Systems.” In Bernardo A. Huberman, editor, *The Ecology of Computation.* Elsevier Science Publishers, North-Holland, 1988.

Mark S. Miller, Daniel G. Bobrow, E. Dean Tribble, and Jacob Levy. “Logical Secrets.” In *International Conference on Logic Programming*, pages 704–728, 1987.

Kenneth M. Kahn, E. Dean Tribble, Mark S. Miller, and Daniel G. Bobrow. “Vulcan: Logical Concurrent Objects.” In *Research Directions in Object-Oriented Programming*, pages 75–112. 1987.

David S. Fogelsong, Daniel G. Bobrow, Mark S. Miller. “Definition Groups: Making Sources into First-Class Objects.” *Research Directions in Object-Oriented Programming*, pages 129–146, 1987.

E. Dean Tribble, Mark S. Miller, Kenneth M. Kahn, Daniel G. Bobrow, Curtis Abbott, and Ehud Y. Shapiro. “Channels: A Generalization of Streams.” In *International Conference on Logic Programming*, pages 839–857, 1987.

Kenneth M. Kahn, E. Dean Tribble, Mark S. Miller, Daniel G. Bobrow. "Objects in Concurrent Logic Programming Languages." *Object Oriented Programming Systems, Languages, and Applications* (OOPSLA) 1986, pages 242–257.