Refinement for Fault-Tolerance: An Aircraft Hand-off Protocol

Keith Marzullo*
Fred B. Schneider**
Jon Dehn

TR 94-1417
April 1994

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

* This author is supported in part by the Defense Advanced Research Projects Agency under NASA Ames grant number NAG 2-593, Contract N00140-87-C-8904 and by AFOSR grant number F496209310242. The views, opinions, and findings contained in this report are those of the author and should not be construed as an official Department of Defense position, policy, or decision.
** This author is supported in part by the Office of Naval Research under contract N00014-91-J-1219, AFOSR under proposal 93NM312, the National Science Foundation under Grant CCR-8701103, and DARPA/NSF Grant CCR-9014363. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not reflect the views of these agencies.

Refinement for Fault-Tolerance: An Aircraft Hand-off Protocol

Keith Marzullo*
University of California at San Diego
Dept. of Computer Science & Engineering
La Jolla, CA 92093-0114

Fred B. Schneider†
Cornell University
Dept. of Computer Science
Ithaca, NY 14853-7501

Jon Dehn
IBM Corp.
9231 Corporate Blvd.
Rockville, MD 20850

30 January 1993

Abstract

Part of the Advanced Automation System (AAS) for air-traffic control is a protocol to permit flight hand-off from one air-traffic controller to another. The protocol must be fault-tolerant and, therefore, is subtle—an ideal candidate for the application of formal methods. This paper describes a formal method for deriving fault-tolerant protocols that is based on refinement and proof outlines. The AAS hand-off protocol was actually derived using this method; that derivation is given.

1 Introdution

The next-generation air traffic control system for the United States is currently being built under contract to the U.S. government by the IBM Federal Systems Company (recently acquired by Loral Corp.). *Advanced Automation*

*This author is supported in part by the Defense Advanced Research Projects Agency under NASA Ames grant number NAG 2-593, Contract N00140-87-C-8904 and by AFOSR grant number F496209310242. The views, opinions, and findings contained in this report are those of the author and should not be construed as an official Department of Defense position, policy, or decision.

†This author is supported in part by the Office of Naval Research under contract N00014-91-J-1219, AFOSR under proposal 93NM312, the National Science Foundation under Grant CCR-8701103, and DARPA/NSF Grant CCR-9014363. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not reflect the views of these agencies.

System (*AAS*) [1] is a large distributed system that must function correctly, even if hardware components fail.

Design errors in AAS software are avoided and eliminated by a host of methods. This paper discusses one of them—the formal derivation of a protocol from its specification—and how it was applied in the AAS protocol for transferring authority to control a flight from one air-traffic controller to another. The flight hand-off protocol we describe is the one actually used in the production AAS system (although the protocol there is programmed in Ada). And, the derivation we give is a description of how the protocol actually was first obtained.

The formal methods we use are not particularly esoteric nor sophisticated. The specification of the problem is simple, as is the characterization of hardware failures that it must tolerate. Because the hand-off protocol is short, computer-aided support was not necessary for the derivation. Deriving more complex protocols would certainly benefit from access to a theorem prover.

We proceed as follows. The next section gives a specification of the problem and the assumptions being made about the system. Section 3 describes the formal method we used. Finally, Section 4 contains our derivation of the hand-off protocol.

2 Specification and System Model

The air-traffic controller in charge of a flight at any time is determined by the location of the flight at that time. However, the *hand-off* of the flight from one controller to another is not automatic: some controller must issue a command requesting that the ownership of a flight be transferred from its current owner to a new controller. This message is sent to a process that is executing on behalf of the new controller. It is this process that starts the execution of the hand-off itself.

The hand-off protocol has the following requirements:

P1: No two controllers own the same flight at the same time.

P2: The interval during which no controller owns a flight is brief (approximately one second).

P3: A controller that does not own a flight knows which controller does own that flight.

The hand-off protocol is implemented on top of AAS system software that implements several strong properties about message delivery and execution time [1]. For our purposes, we simplify the system model somewhat and mention only those properties needed by our hand-off protocol.

The system is structured as a set of processes running on a collection of processors interconnected with redundant networks. The services provided by AAS system software include a point-to-point FIFO interprocess communication facility and a name service that allows for location-independent interprocess communication. AAS also supports the notion of a *resilient process* s comprising a *primary* process $s.p$ and a *backup* process $s.b$. The primary sends messages to the backup so that the backup's state stays consistent with the primary. This allows the backup to take over if the primary fails.

A resilient process is used to implement the services needed by an air-traffic controller, including screen management, display of radar information, and processing of flight information. We denote the primary process for a controller C as $C.p$ and its backup process as $C.b$. If C is the owner of a flight f, then $C.p$ can execute commands and send messages that affect the status of flight f; $C.b$, like all backup processes in AAS, only receives and records information from $C.p$ in order to take over if $C.p$ fails.

AAS implements a simple failure model for processes [3]:

S1: Processes can fail by crashing. A crashed process simply stops executing without otherwise taking any erroneous action.

S2: If a primary process crashes, then its backup process detects this and begins executing a user-specified routine.

Property *S2* is implemented by having a *failure detector* service. This service monitors each process and, upon detecting a failure, notifies any interested process.

If the hand-off protocol runs only for a brief interval of time, then it is safe to assume that no more than a single failure will occur during execution. So, we assume:

S3: In any execution of the hand-off protocol, at most one of the participating processes can crash.

S4: Messages in transit can be lost if the sender or receiver of the message crashes. Otherwise, messages are reliably delivered, without corruption, and in a timely fashion. No spurious messages are generated.

We also can assume that messages are not lost due to failure of network components such as controllers and repeaters. This is a reasonable assumption because the processors of AAS are interconnected with redundant networks and it is assumed that no more than one of the networks will fail.

In any long-running system in which processes can fail, there must be a mechanism for restarting processes and reintegrating them into the system. We ignore such issues here because that functionality is provided by AAS system software. Instead, we assume that at the beginning of a hand-off from A to B, all four processes $A.p, A.b, B.p, B.b$ are operational.

3 Fault-tolerance and Refinement

A protocol is a program that runs on a collection of one or more processors. We indicate that S is executed on processor p by writing:

$$\langle S \rangle \textbf{ at } p \tag{1}$$

Execution of (1) is the same as **skip** if p has failed and otherwise is the same as executing S as a single, indivisible action. This is exactly the behavior one would expect when trying to execute an atomic action S on a fail-stop processor.

Sequential composition is indicated by juxtaposition.

$$\langle S_1 \rangle \textbf{ at } p_1 \quad \langle S_2 \rangle \textbf{ at } p_2 \tag{2}$$

This statement is executed by first executing $\langle S_1 \rangle$ **at** p_1 and then executing $\langle S_2 \rangle$ **at** p_2. Notice that execution of $\langle S_2 \rangle$ **at** p_2 cannot assume that S_1 has actually been performed. If p_1 fails before execution of $\langle S_1 \rangle$ **at** p_1 completes, then the execution of $\langle S_1 \rangle$ **at** p_1 is equivalent to **skip**. Second, observe that an actual implementation of (2) when p_1 and p_2 are different will require some form of message-exchange in order to enforce the sequencing.

Finally, parallel composition is specified by:

$$\textbf{cobegin } \langle S_1 \rangle \textbf{ at } p_1 \parallel \langle S_2 \rangle \textbf{ at } p_2 \parallel \ldots \parallel \langle S_n \rangle \textbf{ at } p_n \textbf{ coend} \tag{3}$$

This statement completes when each component $\langle S_i \rangle$ **at** p_i has completed. Since some of these components may have been assigned to processors that fail, all that can be said when (3) completes is that a subset of the S_i have been performed. If, however, we also know the maximum number t of failures that can occur while (3) executes, then at least $n - t$ of the S_i will be performed.

Proof Outlines

We use proof outlines to reason about execution of a protocol. A *proof outline* is a program that has been annotated with assertions, each of which is enclosed in braces. A *precondition* appears before each atomic action, and a *postcondition* appears after each atomic action. Assertions are Boolean formulas involving the program variables. Here is an example of a proof outline.

$$
\begin{array}{ll}
 & \{x = 0 \wedge y = 0\} \\
\mathrm{X1}: & x := x + 1 \\
 & \{x = 1 \wedge y = 0\} \\
\mathrm{X2}: & y := y + 1 \\
 & \{x = 1 \wedge y = 1\}
\end{array}
$$

In this example, $x = 0 \wedge y = 0$, $x = 1 \wedge y = 0$, and $x = 1 \wedge y = 1$ are assertions. Assertion $x = 0 \wedge y = 0$ is the precondition of X1, denoted *pre*(X1), and assertion $x = 1 \wedge y = 0$ is the postcondition of X1, denoted *post*(X2). The postcondition of X1 is also the precondition of X2.

A proof outline is *valid* if its assertions are an accurate characterization of the program state as execution proceeds. More precisely, a proof outline is valid if the *proof outline invariant*

$$\bigwedge_{\mathrm{S}} ((\mathit{at}(\mathrm{S}) \Rightarrow \mathit{pre}(\mathrm{S})) \wedge (\mathit{after}(\mathrm{S}) \Rightarrow \mathit{post}(\mathrm{S})))$$

is not invalidated by execution of the program, where *at*(S) is a predicate that is *true* when the program counter is at statement S, and *after*(S) is a predicate this is *true* when the program counter is just after statement S.

The proof outline above is valid. For example, execution starting in a state where $x = 1 \wedge y = 0 \wedge \mathit{after}(\mathrm{X1})$ is *true* satisfies the proof outline invariant and, as execution proceeds, the invariant remains *true*. Notice, our definition of validity allows execution to begin anywhere—even in the middle of the program. Changing *post*(X1) (and *pre*(X2)) to $x = 1$ destroys the validity of the above proof outline. (Start execution in state $x = 1 \wedge y = 23 \wedge \mathit{after}(\mathrm{X1})$. The proof outline invariant will hold initially but is invalidated by execution of X2.)

A simple set of (syntactic) rules can be used to derive valid proof outlines. The first such programming logic was proposed in [2]. The logic that we use is a variant of that one, extended for concurrent programs [4].

Additional extensions are needed to derive a proof outline involving statements like (1). Here is a rule for (1); it uses the predicate $up(p)$ to assert that processor p has not failed.

$$\textbf{Action at Processor: } \frac{\{A\}\ S\ \{B\}}{\{A\}\ \langle S\rangle\ \textbf{at}\ p\ \{up(p) \Rightarrow B\}}$$

Since execution of $\langle S\rangle$ **at** p when p has crashed is equivalent to a **skip**, one might think that

$$\{A\}\ \langle S\rangle\ \textbf{at}\ p\ \{(up(p) \Rightarrow B) \wedge (\neg up(p) \Rightarrow A\} \tag{4}$$

should be valid if $\{A\}\ S\ \{B\}$ is. Proof outline (4), however, is not valid. Consider an execution that starts in a state satisfying A and suppose p has not crashed. According to the rule's hypothesis, execution of S would produce a state satisfying B. If process p then crashed, the state would satisfy $\neg up(p) \wedge B$. Unless B implies A, the postcondition of (4) no longer holds.

The problem with (4) is that the proof outline invariant is invalidated by a processor failure. The predicate $up(p)$ changing value from *true* to *false* causes the proof outline invariant to be falsified. We define a proof outline to be *fault-invariant* with respect to a class of failures if the proof outline invariant is not falsified by the occurence of any allowable subset of those failures.

For the hand-off protocol, we are concerned with tolerating a single processor failure. We, therefore, are concerned with proof outlines whose proof outline invariants are not falsified when $up(p)$ becomes *false* for a single processor (provided $up(p)$ is initially *true* for all processors). Checking that a proof outline is fault-invariant for this class of failures is simple:

Fault-Invariance: For each assertion A:
$$(A \wedge \bigwedge_{p} up(p)) \Rightarrow \bigwedge_{p'} A[up(p'){:=}\,false]$$

where $L[x{:=}\ e]$ stands for L with every free occurrence of x replaced by e.

4 Derivation of the Hand-off Protocol

Let $CTR(f)$ be the set of controllers that own flight f. Property *P1* can then be restated as

$P1'$: $|CTR(f)| \leq 1$.

Desired is a protocol Xfer(A, B) satisfying

$$\begin{array}{l} \{A \in CTR(f) \wedge P1'\} \\ \text{Xfer}(A, B) \\ \{B \in CTR(f) \wedge P1'\} \end{array}$$

such that $P1'$ holds throughout the execution of Xfer(A, B).

A simple implementation of this protocol would be to use a single variable $ctr(f)$ that contains the identity of the controller of flight f and to change $ctr(f)$ with an assignment statement:

$$\begin{array}{l} \{A \in ctr(f) \wedge P1'\} \\ ctr(f) := (ctr(f) - \{A\}) \cup \{B\} \\ \{B \in ctr(f) \wedge P1'\} \end{array}$$

This implementation is problematic because the variable $ctr(f)$ must reside at some site. Not only does this lead to a possible performance problem, but it makes determining the owner of f dependent on the availability of this site. Therefore, we represent $CTR(f)$ with a Boolean variable $C.ctr(f)$ at each site C, where

$$CTR(f) : \{C | C.ctr(f)\}.$$

By doing so, we now require at least two separate actions in order to implement Xfer(A, B)—one action that changes $A.ctr(f)$ and one action that changes $B.ctr(f)$. Using the Action at Processor Rule, we get:

$$\begin{array}{rl} & \{A \in CTR(f) \wedge P1'\} \\ \text{X1}: & \langle A.ctr(f) := \mathit{false} \rangle \textbf{ at } A \\ & \{(up(A) \Rightarrow ((A \notin CTR(f)) \wedge (CTR(f) = \emptyset))) \wedge P1'\} \end{array}$$

$$\begin{array}{rl} & \{CTR(f) = \emptyset\} \\ \text{X2}: & \langle B.ctr(f) := \mathit{true} \rangle \textbf{ at } B \\ & \{(up(B) \Rightarrow (B \in CTR(f))) \wedge P1')\} \end{array}$$

Note that pre(X2) must assert that $CTR(f) = \emptyset$ holds, since otherwise execution of X2 invalidates $P1'$.

The preconditions of X1 and X2 are mutually inconsistent, so these statements cannot be executed in parallel. Moreover, X2 cannot be run first because pre(X2), $CTR(f) = \emptyset$, does not hold in the initial state. Thus, X2 must execute after X1. Unfortunately, *post*(X1) does not imply *pre*(X2); if $up(A)$ does not hold, then we cannot assert that $CTR(f) = \emptyset$ is *true*. This should not be surprising: if A fails, then it cannot relinquish ownership.

One solution for this availability problem is to employ a resilient process. That is, each controller C will have a primary process $C.p$ and a backup process $C.b$ executed on processors that fail independently. Each process has its own copy of $C.ctr(f)$, and these copies will be used to represent $C.ctr(f)$ in a manner that tolerates a single processor failure:

$$C.ctr(f): \begin{cases} C.p.ctr(f) & \text{if } up(C.p) \\ C.b.ctr(f) & \text{if } \neg up(C.p) \end{cases}$$

Since we assume that there is at most one failure during execution of the protocol, the above definition never references the variable of a failed process. Replacing references to processor "A" in Statement X1 with "$A.p$" produces the following:

$$\begin{array}{rl} & \{A \in CTR(f) \wedge P1'\} \\ \text{X1a}: & \langle A.p.ctr(f) := \mathit{false} \rangle \ \textbf{at}\ A.p \\ & \{(up(A.p) \Rightarrow ((A \notin CTR(f)) \ \wedge (CTR(f) = \emptyset))) \wedge P1'\} \end{array}$$

This proof outline is not fault-invariant, however. If $A.p$ were to fail when the precondition holds, then the precondition might not continue to hold. In particular, if $A \in CTR(f)$ holds because $A.p.ctr(f)$ is *true* and $A.b.ctr(f)$ happens to be *false*, then when $A.p$ fails, $A \in CTR(f)$ would not hold. We need to assert that $A.p.ctr(f) = A.b.ctr(f)$ also holds whenever *pre*(X1a) does. We express this condition using the following definition:

Pr: $(up(A.p) \wedge up(A.b)) \ \Rightarrow (A.b.ctr(f) = A.p.ctr(f))$

Note that if one of $A.p$ or $A.b$ has failed then $A.p.ctr(f)$ and $A.b.ctr(f)$ need not be equal. Adding Pr to *pre*(X1a) gives the following proof outline, which is fault-invariant for a single failure:

$$\begin{array}{rl} & \{A \in CTR(f) \wedge P1' \wedge Pr\} \\ \text{X1a}: & \langle A.p.ctr(f) := \mathit{false} \rangle \ \textbf{at}\ A.p \\ & \{(up(A.p) \Rightarrow ((A \notin CTR(f)) \ \wedge (CTR(f) = \emptyset))) \wedge P1'\} \end{array}$$

We need more than just X1a to implement X1, however. X1a does not re-establish Pr, which must hold for subsequent ownership transfers. This suggests that $A.b.ctr(f)$ also be updated. Another problem with X1a is that $post$(X1a) still does not imply pre(X2): if $up(A.p)$ does not hold, then $CTR(f) = \emptyset$ need not hold.

An action whose postcondition implies

$$up(A.b) \Rightarrow (\neg A.b.ctr(f) \ \wedge (\neg up(A.p) \Rightarrow (CTR(f) = \emptyset))$$

suffices. By our assumption, $up(A.p) \vee up(A.b)$ holds, so this postcondition and $post$(X1a) will together allow us to conclude $CTR(f) = \emptyset$ holds, thereby establishing pre(X2). Here is an action that, when executed in a state satisfying pre(X1a), terminates with the above assertion holding:

$$\begin{array}{ll} & \{(A \in CTR(f)) \wedge P1' \wedge Pr\} \\ \text{X1b}: & \langle A.b.ctr(f) := \mathit{false} \rangle \ \textbf{at} \ A.b \\ & \{up(A.b) \Rightarrow (\neg A.b.ctr(f) \ \wedge (\neg up(A.p) \Rightarrow (CTR(f) = \emptyset)))\} \end{array}$$

One might think that since X1a and X1b have the same preconditions they could be run in parallel, and the design of the first half of the protocol would be complete. Unfortunately, we are not done yet.

The original protocol specification implicitly restricted permissable ownership transitions. Written as a regular expression, the allowed sequence of states is:

$$(CTR(f) = \{A\})^{+} \ (CTR(f) = \emptyset)^{*} \ (CTR(f) = \{B\})^{+} \tag{5}$$

That is, first A owns the flight, then no controller owns the flight for zero or more states, and finally B owns the flight. The proof outline above does not tell us anything about transitions; it only tells that $P1'$ holds throughout (because $P1'$ is implied by all assertions). We must strengthen the proof outline to deduce that only correct transitions occur.

A regular expression (like the one above) can be represented by a finite state machine that accepts all sentences described by the regular expression. Furthermore, a finite state machine is characterized by a next-state transition function. The following next-state transition function δ_{AB} characterizes the finite state machine for (5):

$$\delta_{AB} : \begin{cases} \{\{A\}, \emptyset, \{B\}\} & \text{if } A \in CTR(f) \\ \{\emptyset, \{B\}\} & \text{if } CTR(f) = \emptyset \\ \{\{B\}\} & \text{if } B \in CTR(f) \end{cases}$$

The value of δ_{AB} is the set of values of $CTR(f)$ that are next allowed for the protocol. For example, when $CTR(f) = \emptyset$ holds, δ_{AB} says that a transition to a state in which $CTR(f) = \emptyset$ holds or to a state in which $CTR(f) = \{B\}$ holds are the only permissible transitions. Note that since $P1'$ holds, we know that the three cases $A \in CTR(f), CTR(f) = \emptyset, B \in CTR(f)$ are mutually exclusive, so δ_{AB} always has a unique value.

We further define $\Theta\delta_{AB}$ to be the value of δ_{AB} in the previous state during the execution of the hand-off protocol, or $\{\{A\}, \emptyset, \{B\}\}$ if there is no previous state. Our hand-off protocol only will make permissable state transitions provided each assertion implies that $CTR(f) \in \Theta\delta_{AB}$; that is, provided the current owner of f is one of the owners that was acceptable as the "next owner" in the previous state of the system.

We therefore add conjunct $CTR(f) \in \Theta\delta_{AB}$ to the assertions in the proof outline and check to see if the stronger proof outline is valid. If it is valid, then we can move on to implementing X2, the second part of Xfer(A, B); otherwise, we will have reason to make further modifications.

Here is the (strengthened) proof outline with X1a and X1b running in parallel:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\{CTR(f) \in \Theta\delta_{AB} \land\ A \in CTR(f) \land P1' \land Pr\}} \\
\multicolumn{2}{l}{\textbf{cobegin}} \\
 & \{CTR(f) \in \Theta\delta_{AB} \land\ A \in CTR(f) \land P1' \land Pr\} \\
\mathrm{X1a}: & \langle A.p.ctr(f) := \mathit{false}\rangle\ \textbf{at}\ A.p \\
 & \{CTR(f) \in \Theta\delta_{AB} \land \\
 & \quad (up(A.p) \Rightarrow (A \notin CTR(f) \land (CTR(f) = \emptyset)) \land P1'\} \\
\| & \{CTR(f) \in \Theta\delta_{AB} \land\ (A \in CTR(f)) \land P1' \land Pr\} \\
\mathrm{X1b}: & \langle A.b.ctr(f) := \mathit{false}\rangle\ \textbf{at}\ A.b \\
 & \{CTR(f) \in \Theta\delta_{AB} \land \\
 & \quad (up(A.b) \Rightarrow (\neg A.b.ctr(f) \land (\neg up(A.p) \Rightarrow (CTR(f) = \emptyset))))\} \\
\multicolumn{2}{l}{\textbf{coend}} \\
\multicolumn{2}{l}{\{CTR(f) \in \Theta\delta_{AB} \land} \\
\multicolumn{2}{l}{\quad (up(A.p) \Rightarrow (A \notin CTR(f) \land (CTR(f) = \emptyset))) \land} \\
\multicolumn{2}{l}{\quad (up(A.b) \Rightarrow (\neg A.b.ctr(f) \land (\neg up(A.p) \Rightarrow (CTR(f) = \emptyset)))} \\
\multicolumn{2}{l}{\quad \land P1' \land Pr\}}
\end{array}
$$

Unfortunately, this proof outline is not fault-invariant. If $A.p$ fails in a state satisfying $\mathit{after}(\mathrm{X1a}) \land \mathit{at}(\mathrm{X1b})$ then the following holds before the failure:

$$
\begin{aligned}
&\mathit{after}(\mathrm{X1a}) \land \mathit{at}(\mathrm{X1b}) \land up(A.p) \\
&\quad \land up(A.b) \land (CTR(f) = \emptyset) \\
&\quad \land \neg A.p.ctr(f) \land A.b.ctr(f) \land \Theta\delta_{AB} = \{\emptyset, B\}
\end{aligned}
$$

After the failure, we have:

$$\begin{aligned}
&\mathit{after}(\mathrm{X1a}) \wedge \mathit{at}(\mathrm{X1b}) \wedge \neg \mathit{up}(A.p)\\
&\quad \wedge \mathit{up}(A.b) \wedge (A \in \mathit{CTR}(f))\\
&\quad \wedge \neg A.p.\mathit{ctr}(f) \wedge A.b.\mathit{ctr}(f) \wedge \Theta\delta_{AB} = \{\emptyset, B\}
\end{aligned}$$

So, $CTR(f) \in \Theta\delta_{AB}$ does not hold after the failure, and the first conjunct of *post*(X1a) is invalidated. One simple solution is to preclude states where *at*(X1b) $\wedge$ *after*(X1a) holds. This can be done by running the two actions in sequence—first X1b and then X1a. The result is described by the following proof outline:

$$\begin{array}{ll}
& \{CTR(f) \in \Theta\delta_{AB} \wedge P1' \wedge Pr \ \wedge A \in CTR(f)\}\\
\mathrm{X1b}: & \langle A.b.\mathit{ctr}(f) := \mathit{false}\rangle \ \textbf{at}\ A.b\\
& \{CTR(f) \in \Theta\delta_{AB} \wedge P1' \wedge A.p.\mathit{ctr}(f) \wedge\\
& \quad (\mathit{up}(A.b) \Rightarrow \neg A.b.\mathit{ctr}(f))\}\\
\mathrm{X1a}: & \langle A.p.\mathit{ctr}(f) := \mathit{false}\rangle \ \textbf{at}\ A.p\\
& \{CTR(f) \in \Theta\delta_{AB} \wedge P1' \wedge (\mathit{up}(A.b) \Rightarrow \neg A.b.\mathit{ctr}(f))\\
& \quad \wedge (\mathit{up}(A.p) \Rightarrow \neg A.p.\mathit{ctr}(f)) \wedge Pr\}\\
& \text{therefore, according to the definitions of } CTR(f) \text{ and } A.\mathit{ctr}(f),\\
& \{CTR(f) \in \Theta\delta_{AB} \wedge A \notin CTR(f) \wedge P1' \wedge Pr\}
\end{array}$$

What we really want to conclude in *post*(X1a), however, is $CTR(f) = \emptyset$— not just $A \notin CTR(f)$. This is easily done by strengthening the above proof outline with the following:

POnly(A): For all controllers C: $C \neq A$: $C \notin CTR(f)$

POnly(A) is initially *true* because $A \in CTR(f) \wedge P1'$ holds. It is not invalidated by any assignment, because the only variables assigned to are those of $A.p$ and $A.b$. So, *POnly*(A) remains *true* throughout the execution of X1.

The derivation of a protocol for X2 is basically the same, except that A is replaced by B and *false* is interchanged by *true*. Doing so results in the proof outline shown in Figure 1.

4.1 Implementing *P3*

Property *P3* of Section 2 is satisfied by the protocol in Figure 1 as long as there are exactly two controllers. When there are more than two controllers,

$$\{CTR(f) \in \Theta\delta_{AB} \wedge P1' \wedge Pr \wedge POnly(A) \wedge A \in CTR(f)\}$$

X1b : $\langle A.b.ctr(f) := \mathit{false} \rangle$ **at** $A.b$

$$\{CTR(f) \in \Theta\delta_{AB} \wedge P1' \wedge POnly(A) \wedge A.p.ctr(f) \wedge B \notin CTR(f) \wedge (up(A.b) \Rightarrow \neg A.b.ctr(f))\}$$

X1a : $\langle A.p.ctr(f) := \mathit{false} \rangle$ **at** $A.p$

$$\{CTR(f) \in \Theta\delta_{AB} \wedge P1' \wedge Pr \wedge POnly(A) \wedge B \notin CTR(f) \wedge (up(A.b) \Rightarrow \neg A.b.ctr(f)) \wedge (up(A.p) \Rightarrow \neg A.p.ctr(f))\}$$

therefore, according to the definitions of $CTR(f)$, $A.ctr(f)$, and $POnly(B)$

$$\{CTR(f) \in \Theta\delta_{AB} \wedge P1' \wedge Pr \wedge POnly(B) \wedge CTR(f) = \emptyset\}$$

X2b : $\langle B.b.ctr(f) := \mathit{true} \rangle$ **at** $B.b$

$$\{CTR(f) \in \Theta\delta_{AB} \wedge P1' \wedge POnly(B) \wedge \neg B.p.ctr(f) \wedge (up(B.b) \Rightarrow B.b.ctr(f))\}$$

X2a : $\langle B.p.ctr(f) := \mathit{true} \rangle$ **at** $B.p$

$$\{CTR(f) \in \Theta\delta_{AB} \wedge P1' \wedge Pr \wedge POnly(B) \wedge (up(B.b) \Rightarrow B.b.ctr(f)) \wedge (up(B.p) \Rightarrow B.p.ctr(f))$$

therefore, according to the definitions of $CTR(f)$ and $B.ctr(f)$,

$$\{B \in \Theta\delta_{AB} \wedge P1' \wedge Pr \wedge POnly(B) \wedge B \in CTR(f)\}$$

Figure 1: Hand-off Protocol for A and B

a controller must query other controllers in order to determine which owns a flight. Doing so is inefficient, so we instead consider having each controller C maintain a variable $C.ctrID(f)$ that names the owner of flight f. As with $C.ctr(f)$, we represent the value of $C.ctrID(f)$ in a manner that tolerates a single site failure:

$$C.ctrID(f)\colon \begin{cases} C.p.ctrID(f) & \text{if } up(C.p) \\ C.b.ctrID(f) & \text{if } \neg up(C.p) \end{cases} \tag{6}$$

This variable can be used to implement the Boolean $C.ctr(f)$ by defining:

$$C.ctr(f)\colon (C.ctrID(f) = C)$$

Thus, the assignment "$C.ctr(f) := \mathit{true}$" would be replaced by "$C.ctrID(f) := C$", and "$C.ctr(f) := \mathit{false}$" would be replaced by "$C.ctrID(f) := X$" for any value $X \neq C$.

We can rewrite *P3* as the following:

$$\begin{aligned} P3'\colon\ & (\exists C\colon (C.ctrID(f) = C)) \Rightarrow \\ & \qquad (\exists C\colon (C.ctrID(f) = C) \wedge (\forall C'\colon C'.ctrID(f) = C)). \end{aligned}$$

For the protocol of Figure 1, *P3'* holds when *at*(X2b) is *true* because the the antecedent is *false*. Furthermore, if we explicitly assign $A.ctrID(f) := B$ as the assignments X1b and X1a, then *P3'* holds throughout the execution, provided C' ranges over the set $\{A, B\}$. For the other controllers, additional statements are needed, shown in Figure 2.

Since Z(A, B) in Figure 2 changes the values of $C.ctrID(f)$, it should be executed when $CTR(f) = \emptyset$ holds, because otherwise its execution may violate *P3'*. Thus, Z(A, B) would have to be started no earlier than *after*(X1a) and terminate by *at*(X2a). Unfortunately, Z(A, B) may take a significant amount of time—even though its component statements can be executed in parallel, the time to execute Z(A, B) will include some communication and synchronization overhead. This extra time could make satisfying *P2* hard or impossible.

Property *P3'* is perhaps a bit too strong. In fact, all that is really required is that a controller be able to communicate with the process that owns a flight. For example, $C.ctrID(f)$ could be the start of a path of controllers, terminating with the current owner. The scheme where $C.ctrID(f)$ indicates the current owner is equivalent to requiring that this path have a length of 1. But, longer paths are also acceptable.

Let $C \leadsto C'$ denote that $C.ctrID(f) = C'$, and let $C \leadsto^* C'$ denote the transitive closure of $\leadsto$. Using this notation, *P3'* can be expressed as:

$$
\begin{array}{lll}
 & \{true\} & \\
\mathrm{Z}(A, B): & \textbf{cobegin} & \\
 & \|_{C: C \notin \{A,B\}}: \{true\} & \\
\mathrm{Zp}: & & \langle C.p.ctrlD(f) := B \rangle \textbf{ at } C.p \\
 & & \{up(C.p) \Rightarrow (C.p.ctrlD(f) = B)\} \\
 & \|_{C: C \notin \{A,B\}}: \{true\} & \\
\mathrm{Zb}: & & \langle C.b.ctrlD(f) := B \rangle \textbf{ at } C.b \\
 & & \{up(C.b) \Rightarrow (C.b.ctrlD(f) = B)\} \\
 & \textbf{coend} & \\
 & \{up(C.p) \Rightarrow (C.p.ctrlD(f) = B) & \\
 & \quad \wedge\ up(C.b) \Rightarrow (C.b.ctrlD(f) = B)\} & \\
 & \text{therefore, according to the definitions of } C.ctrlD(f) & \\
 & \{(\forall C \notin \{A, B\} : C \leadsto B)\} &
\end{array}
$$

Figure 2: Hand-off Protocol for Controllers other than A and B

P3': $(\exists C : C \leadsto C) \Rightarrow (\exists C : C \leadsto C \wedge (\forall C' : C' \leadsto C))$.

We weaken *P3'* as follows:

P3'': $(\exists C : C \leadsto C) \Rightarrow (\exists C : C \leadsto C \wedge (\forall C' : C' \leadsto^* C))$.

P3'' is left invariant by the protocol in Figure 1. *P3''* is also an invariant of the protocol of Figure 2 provided $B \leadsto B \vee B \leadsto A$ initially holds. From *post*(Z(A, B)) and *post*(X2a), we conclude that as long as the execution of Z(A, B) completes before another hand-off starts, *P3'* will hold once Z(A, B) and the protocol in Figure 1 have both terminated. Since *P3'* implies $B \leadsto B \vee B \leadsto A$, the system is once again in a state from which a hand-off can be performed. Hence, Z(A, B) can begin executing at any point during the hand-off from A to B—because its precondition, $B \leadsto B \vee B \leadsto A$, holds throughout the protocol of Figure 1. And, Z(A, B) must complete before a subsequent hand-off has started.

4.2 Implementation using Messages

So far, the protocol we have derived consists of assignment statements to various variables that reside on separate processes. The protocol consists of the three processes, as follows:

cobegin
X1b : $\langle A.b.ctr(f){:=}\ \mathit{false}\rangle$ **at** $A.b$
X1a : $\langle A.p.ctr(f){:=}\ \mathit{false}\rangle$ **at** $A.p$
X2b : $\langle B.b.ctr(f){:=}\ \mathit{true}\rangle$ **at** $B.b$
X2a : $\langle B.p.ctr(f){:=}\ \mathit{true}\rangle$ **at** $B.p$

Zp : $\|_{C:\notin\{A,B\}}: \langle C.p.ctrID(f){:=}\ B\rangle$ **at** $C.p$

Zb : $\|_{C:\notin\{A,B\}}: \langle C.b.ctrID(f){:=}\ B\rangle$ **at** $C.b$

coend

An actual implementation would require that each assignment statement be executed by the processor whose variable is being set. Furthermore, the assignment statements of the first process must be sequenced. This sequencing will be accomplished in our implementation by processor $B.p$, since this processor starts the protocol. If $B.p$ crashes, then $B.b$ will take over the sequencing. Because all assignments are constants to variables, when taking over, $B.b$ can simply start at the beginning of the sequence—it not need to know how far $B.p$ got before failing.

$B.b$ does need to know when $B.p$ has finished executing the hand-off protocol. Otherwise, a crash of $B.p$ might cause $B.b$ to re-execute the hand-off from A to B after f has been later handed off to another controller, in which case $B.b$ would undo that later hand-off. Hence, $B.b$ must be notified of the completion of the hand-off before any subsequent hand-offs are started. We represent the fact that a hand-off from A to B is in progress with a variable $B.b.xfr$, whose value is initially $\perp$.

In order to continue the implementation using messages, some further details of the AAS system services must be given.

- Communication between resilient processes uses **send** and **receive**. If some process sends a message m to a resilient process C, then m is enqueued at $C.p$ if $C.p$ has not crashed and enqueued at $C.b$ if $C.p$ has crashed. Furthermore, **send** does not return control until the message has been enqueued at the remote process. The remote process may crash after enqueueing m but before delivering m, in which case m is lost.

- The primary of a resilient process communicates with its backup using **log**. Like **send**, **log** does not return control until the message is enqueued by the remote process. A **log** that is executed when there is no backup (for example, when the backup has crashed or when **log** is executed by the backup itself) does nothing and immediately returns control.

- Until the primary of a resilient process crashes, the backup delivers only messages sent by **log**.
- When primary $C.p$ crashes, $C.b$ takes over by first processing any enqueued messages sent by $C.p$ using **log**. It then executes the user-defined recovery protocol. And, finally, it receives messages sent to C.

We also use a variable in each process to represent the value of variables $C.p.ctrID(f)$ and $C.b.ctrID(f)$. A simple approach would be to introduce $C.p.owner(f)$ and $C.b.owner(f)$, such that:

$C.p.ctrID(f)$: $C.p.owner(f)$
$C.b.ctrID(f)$: $C.b.owner(f)$.

Doing so, however, is inefficient (as well as difficult given the AAS communication primitives). Consider X1b in the hand-off protocol. To implement X1b, $B.p$ would send a message to $A.b$ instructing it to execute $A.b.owner(f) := B$. Since X1b must complete before X1a starts, $B.p$ cannot start X1a before $A.b$ completes its assignment. The result is two end-to-end message delays.

A more efficient hand-off protocol can be implemented using the following definitions of $C.p.ctrID(f)$ and $C.b.ctrID(f)$. Let the predicate $E_C(f, X)$ mean that $C.b$ has enqueued but not yet processed a log from $C.p$ that requests the execution of $C.b.owner(f) := X$, and let $V_C(f)$ be the value of X in the most recent such **log** message. Then, we define:

$C.p.ctrID(f)$: $C.p.owner(f)$

$$C.b.ctrID(f)\colon \begin{cases} C.b.owner(f) & \text{if } (\forall X: \neg E_C(f, X)) \\ V_C(f) & \text{if } (\exists X: E_C(f, X)) \end{cases}$$

$B.p$ can cause the execution of X1b followed by X1a simply by sending a single message to A requesting execution of $owner(f) := B$. Upon delivery of this message, $A.p$ first executes a **log** so $A.b$ learns of the message. Since **log** does not return until $E_A(f, B)$ holds, $post$(X1b) holds when **log** returns. $A.p$ can then establish $post$(X1a) by executing $C.p.owner(f) := B$.

cobegin

$\{pre(\text{X1b})\}$
$\langle$**log** $''xfr{:=}\ (f, A)''\rangle$ **at** $B.p$
$\langle$**send** $''owner(f){:=}\ B''$ **to** $A\rangle$ **at** $B.p$
$\langle$**wait for** $''ack''$ **from** $A\rangle$ **at** $B.p$
$\{post(\text{X1a}) \wedge pre(\text{X2b})\}$
$\langle$**log** $''owner(f){:=}\ B''\rangle$ **at** $B.p$
$\{post(\text{X2b}) \wedge pre(\text{X2a})\}$
$\langle B.p.owner(f){:=}\ B\rangle$ **at** $B.p$
$\{post(\text{X2a})\}$
$\langle \forall C : C \notin \{A, B\}$: **send** $''owner(f){:=}\ B''$ **to** $C\rangle$ **at** $B.p$
$\langle$**log** $''xfr{:=}\ \perp''\rangle$ **at** $B.p$

$\|_{C:C\neq B}$: $\langle$**when deliver** $''owner(f){:=}\ X''\rangle$ **at** $C.p$
$\langle$**log** $''owner(f){:=}\ X''\rangle$ **at** $C.p$
$\{(C = A) \Rightarrow post(\text{X1b}) \wedge (C \neq A) \Rightarrow post(\text{Zb})\}$
$\langle C.p.owner(f){:=}\ X\rangle$ **at** $C.p$
$\{(C = A) \Rightarrow post(\text{X1a})$
$\wedge\ (C \neq A) \Rightarrow (post(\text{Zp}) \wedge post(\text{Zb}))\}$
$\langle$**send** "ack" **to** $X\rangle$ **at** $C.p$

$\|_C$: $\langle$**when deliver** $''x{:=}\ v''$ **from** $C.p$ **do** $C.b.x{:=}\ v\rangle$ **at** $C.b$

$\|_C$: $\langle$**when** $C.p$ **fails do**
if $xfr = (f, X)$
then start hand-off of f **from** X **to** $C\rangle$ **at** $C.b$

coend

Figure 3: Complete Hand-off Protocol

The complete hand-off protocol is shown in Figure 3. The assertions in the code refer to Figures 1 and 2.

Acknowledgements Scott Stoller provided helpful comments on a draft of this paper. We also would like to thank Mary Jodrie and Alan Moshel for bringing the problem to our attention and for helping us develop the requirements.

References

[1] F. Cristian, B. Dancey and J. Dehn. Fault-tolerance in the Advanced Automation System. In *Proceedings of 20th International Symposium on Fault-Tolerant Computing*, Newcastle Upon Tyne, UK, 26-28 June 1990), pp. 6–17.

[2] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM* 12(10):576–580 (October 1969).

[3] Richard D. Schlichting and Fred B. Schneider. Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems. *ACM Transactions on Computer Systems* 3(1):222–238 (August 1983).

[4] Fred B. Schneider. *On Concurrent Programming*. To appear.