A METHOD AND TOOL FOR ANALYZING FAULT-TOLERANCE IN SYSTEMS

A Dissertation
Presented to the Faculty of the Graduate School
of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by
Scott David Stoller
May 1997

A METHOD AND TOOL FOR ANALYZING FAULT-TOLERANCE IN SYSTEMS

Scott David Stoller, Ph.D.

Cornell University 1997

As computers are integrated into systems that have stringent fault-tolerance requirements, there is a growing need for techniques to establish that these systems actually satisfy those requirements. Informal arguments do not supply the desired level of assurance for critical systems. This dissertation presents a rigorous, automated approach to analyzing distributed systems, with a focus on checking fault-tolerance requirements, and describes a prototype implementation of the analysis. The analysis is a novel hybrid of ideas from stream-processing semantics of networks of processes, abstract interpretation of programs, and symbolic computation. The underlying principles of the analysis method are general, but specialized techniques—such as the use of perturbations to represent changes to normal behavior caused by failures—are developed to deal efficiently with the types of systems and requirements that arise in establishing fault-tolerance. The method is illustrated with three examples: the Oral Messages algorithm for Byzantine Agreement, due to Lamport, Shostak and Pease, a standard protocol for FIFO reliable broadcast, and a (subtly) flawed protocol for fault-tolerant moving agents.

Biographical Sketch

Scott David Stoller was born on October 31, 1968, in New Jersey, U.S.A. His childhood was, in retrospect, uneventful. He was graduated *summa cum laude* from Princeton University with a Bachelor's degree in Physics in 1990. After spending one year as a graduate student in the Physics Department at Cornell, he transferred to the Computer Science Department. In the spring of 1992, he took Anil Nerode's logic course. There he met Yanhong Annie Liu, who has made his life richer in countless ways. In May 1996, they were married. In August 1996, they joined the faculty of Indiana University in Bloomington.

To all my teachers,

especially my parents

Acknowledgements

I would like to thank my thesis advisor, Fred Schneider, for his guidance, support, knowledge, and insights, which have been invaluable to the research in this thesis and beyond. The opportunity to work with him has been a pleasure and a privilege.

I would also like to thank Bob Constable for sharing his enthusiasm for and insights into logic, programming languages, my research (everywhere it turned), and many other topics.

I thank Annie Liu for her love, encouragement, criticisms, and suggestions, which have contributed greatly to this work.

I would also like to thank Rich Zippel, Robbert van Renesse, Doug Howe, David Gries, Tom Henzinger, Stuart Allen, Wilfred Chen, Lorenzo Alvisi, Thomas Yan, Pei-Hsin Ho, Yaron Minsky, and Mark Hayden for many enjoyable and thought-provoking discussions.

This material is based on work supported in part by the NASA/ARPA grant NAG-2-893, AFOSR grant F49620-97-1-0013, and ARPA/RADC grant F30602-96-1-0317. 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.

Finally, I thank my parents, for their love and for preparing me for life.

Table of Contents

List of Figures

Chapter 1

Introduction

As computers are integrated into systems that have stringent fault-tolerance requirements, there is a growing need for techniques to establish that these systems actually satisfy those requirements. These systems include safety-critical systems, from digital flight control systems to factory automation systems, and business-critical systems, from traditional applications, like distributed databases, to nascent ones, like systems for electronic commerce. Moreover, as networks of workstations (NoWs) become an increasingly popular platform for large-scale computations of all kinds, fault-tolerance is becoming an issue for general-purpose computing.

Establishing fault-tolerance of a system involves three major steps: (1) identifying possible failures of each component, (2) determining requirements on system behavior for each combination of component failures, and (3) checking whether system behavior actually satisfies these requirements. Step 3 is sometimes tackled with informal arguments. However, informal arguments do not supply the desired level of assurance for critical systems. This thesis presents a rigorous, automated approach to analyzing distributed systems, with a focus on checking fault-tolerance requirements. The underlying principles of the analysis method are general, but specialized techniques are developed to deal efficiently with the types of systems and requirements that arise in establishing fault-tolerance.

1.1 Establishing Fault-Tolerance

The first step in establishing fault-tolerance of a system is to identify possible failures of each component. Each failure corresponds to one way in which a component's actual behavior might diverge from its normal (specified) behavior. For example, one commonly-considered failure of processors is the *crash* failure, which causes the processor to halt. A more severe failure is the *Byzantine* failure, which causes the processor to execute an arbitrary sequence of instructions, unrelated to the program it would have executed in the absence of the failure. Combinations of failures that might occur in a system are called *failure scenarios*; thus, a failure scenario for a system is simply an assignment of a failure to each component of that system. Since components sometimes (hopefully most of the time!) do not fail, we introduce a special failure called OK, corresponding to absence of failure; in other words, OK indicates that the divergence from normal behavior is nothing (or “zero”).

The second step is to determine requirements on system behavior in the failure scenarios. Fault-tolerance requirements can be expressed as a function b such that, for each failure scenario fs of the system, $b(fs)$ is a condition that the system's behavior should satisfy in that failure scenario. For example, an aircraft control system might be required to provide normal service despite the Byzantine failure of any one component. More precisely, for every failure scenario in which at most one component is faulty, the signals sent by the control system to the actuators should be the same as if no failures had occurred.

The third step is to check whether a system will satisfy its fault-tolerance requirements. Experience has shown that informal arguments about fault-tolerance are error-prone and do not supply the desired level of assurance for critical systems [ORSvH95]. For example, one might guess that replication and voting is an easy (albeit expensive) way to achieve fault-tolerance. However, the extensive literature on Byzantine agreement, and errors like the one described in [LR93], show that efficiently coordinating non-faulty replicas in the presence of arbitrary behavior by faulty replicas is a difficult problem.

The difficulty of analyzing fault-tolerance by informal methods has inspired the development and application of rigorous methods. One approach is to apply general-purpose proof-based verification techniques. Work on SIFT [W^+78], an aircraft control computer, is a classic example; indeed, this is one of the earliest applications of any rigorous approach to fault-tolerance. Mechanized support, in the form of a theorem-proving system, typically helps to manage the large and complex proofs. The use of these general-purpose verification tools offers an attractive conceptual economy. However, most people who design and validate fault-tolerant systems are not experts in mathematical logic or formal verification, so methods that require them to construct large proofs (even with support from a theorem-proving system) are problematic. Proof techniques designed specifically for verification of fault-tolerance have been proposed [CdR93,Web93,PJ94,Sch94]. These techniques do facilitate proofs of fault-tolerance, but still require considerable logical expertise of the user.

Automated verification techniques have received increasing attention in recent years, largely as a result of advances in temporal-logic model-checking [CGL94] and automata- and process-based verification techniques [Hol91,Kur94,CS96]. The techniques are largely based on exhaustive exploration of finite state spaces. They are particularly well-suited to hardware verification and have been applied predominantly thereto. Relatively little work has been done on automated analysis of fault-tolerant systems, partly because the protocols of interest are more typical of software than hardware, and exhaustive search of the state space of interesting software systems is often infeasible.

1.2 Overview of this Work

This thesis explores a specialized approach to analysis of distributed systems, focusing on fault-tolerance properties. Our approach is not based on exhaustive state-space exploration. Instead, it is a novel hybrid of ideas from stream-processing (or data-flow) semantics of networks of processes [Kah74,Bro87,Bro90], abstract interpretation of programs [AH87], and symbolic computation. An important feature of our approach is that flexible and powerful abstraction mechanisms are incorporated directly into the framework.[1] Having these mechanisms plays a crucial role in making fault-tolerance analysis tractable.

Our use of abstraction aims to exploit separation of concerns: analysis of failures is separated as much as possible from other aspects of system analysis. To facilitate this separation of concerns, the analysis is parameterized by possible occurrences of failures, and system behavior is analyzed separately for different failure scenarios. A different and more common (e.g., [LJ92,CdR93,Web93,PJ94,LM94]) approach is to model failures as events that occur non-deterministically during a computation; however, this makes it difficult to separate the effects of failures from other aspects of system behavior and hence to model the former more finely than the latter.

Our analysis methods rest on a separation of concerns in specifications: fault-tolerance requirements are separated as much as possible from other correctness requirements. This separation allows the analysis to ignore aspects of the system that do not directly impact its fault-tolerance. For example, in a system with replicated processors, detailed analysis of how the results from different replicas are combined (e.g., voted) may be needed, but other aspects of the processing (e.g., the particular state machine implemented by each replica)

[1] Following literature on abstract interpretation (e.g., [AH87]) and program refinement (e.g., [KMP94]), we use "abstraction" in the sense of "approximation". This has little to do with the meaning of "abstraction" in the theory of functional programming languages (e.g., [Rea89]) or abstract data types.

are treated in our approach by coarse approximations. Such abstraction is crucial for making the analysis tractable.

Our analysis uses a fixed-point calculation to determine three kinds of information that, together, characterize system behavior:

Values: The data sent in messages.

Multiplicities: The number of times each value is sent.

Orderings: The order in which values are sent.

Values and their multiplicities are approximated using *abstract values*, each representing a set of possible values, as in abstract interpretation [AH87]. We also use *symbolic values*, which are expressions composed of constants and variables, to capture additional relationships between values. Orderings are approximated by allowing partial orders, rather than just total orders. This support for approximation of all three kinds of information allows irrelevant aspects of a system to be suppressed and allows compact representation of the highly non-deterministic behavior that failures can cause.

For example, suppose one process of a system sends to another process a message containing a number, then possibly sends a second message containing the same number. The data in the first message could be represented using abstract value $\mathbf{N}$, representing the natural numbers, and symbolic value X, where X is a variable that denotes the actual value sent. The multiplicity of the first message is 1, which is (overloaded as) an abstract value that represents only the number one; since this abstract multiplicity determines the multiplicity uniquely, it doesn't matter what symbolic multiplicity is used. The second message message would be represented using the same abstract value $\mathbf{N}$ and, to show that the values in the two messages are equal, the same symbolic value X. The multiplicity of the second message is ?, which is an abstract value that we define to represent zero or one; the symbolic multiplicity might be a different variable.

We require that all approximations used in modeling a component be *conservative*, i.e., they must over-estimate, rather than under-estimate, the component's possible behaviors. The use of conservative approximations ensures that the information (values, multiplicities, and orderings) determined by the analysis includes *all* possible behaviors of the system being analyzed. But because approximations are being used, the values, multiplicities, and orderings may represent other behaviors as well. Thus, analyzing the approximation—rather than the system it approximates—never gives false positives but may give false negatives. In other words, the analysis may fail to show that all of a system's possible behaviors satisfy a fault-tolerance requirement, even if they do. The possibility of false negatives is an inevitable consequence of the approximations that enable efficient and automated analysis of systems with intractably large state spaces.

To support efficient and convenient fault-tolerance analysis, we extend the analysis framework sketched above—hereafter called the *non-perturbational framework*—to obtain the *perturbational framework*, in which perturbations (changes) due to failures are made explicit. Here, the effects of a failure are represented as changes to the original outputs of the faulty component. Changes to the outputs of a component change the inputs to components that use those outputs, and changes to a component's inputs generally cause changes to its outputs. Each component is characterized, in part, by how it propagates changes. For example, one might normally describe the behavior of a majority-voter as follows: if a majority of its inputs are equal, then its output is the majority value among its inputs. Intuitively, the justification for using a majority-voter to mask the effects of failures corresponds more directly to the fact that a majority-voter propagates changes to its inputs consistently with the rule: if the inputs are originally equal and at most a minority of them change, then the output is unchanged.

For example, consider the graph in Figure 1.1, which we use to represent the behavior of a two-stage replicated pipeline having one faulty component F_1. The meaning of such graphs is defined formally in Chapter 3; here we give only an informal explanation. The nodes of the graph correspond to components. Edge $\langle x, y\rangle$ is labeled with a representation of messages sent from x to y. The graph represents both the failure-free behavior of the system and the behavior in a specified failure scenario—here, the failure scenario in which F_1 suffers a Byzantine failure, and the other components do not fail. In figures, dots on the circumference of a node indicate a failure other than OK for that component; the specific failure is identified in the text (as in the previous sentence).

Edges are labeled with *ms-atoms*, which represent sets of messages. There are two kinds of ms-atoms: *perturbed* ms-atoms, which contain square brackets, and *new* ms-atoms, which do not. Each perturbed ms-atom contains two parts: an original part (preceding the square brackets) and a perturbation (enclosed in square brackets). In this figure, the edge from F_1 to S has a new ms-atom; the other edges all have perturbed ms-atoms. The failure-free behavior of the system is represented by the original parts of the perturbed ms-atoms. Thus, we see that the source S sends a natural number, represented by the symbolic value X, to processors F_1–F_3. Each of these processors applies a function, represented by the symbol

F, to its input and sends the result to the corresponding processor in the next stage of the pipeline. Processors G_1–G_3 apply a function represented by G to their input and send the result, namely $G(F(X))$, to a 3-input voter V, which selects the majority of its inputs and sends the result to actuator A.

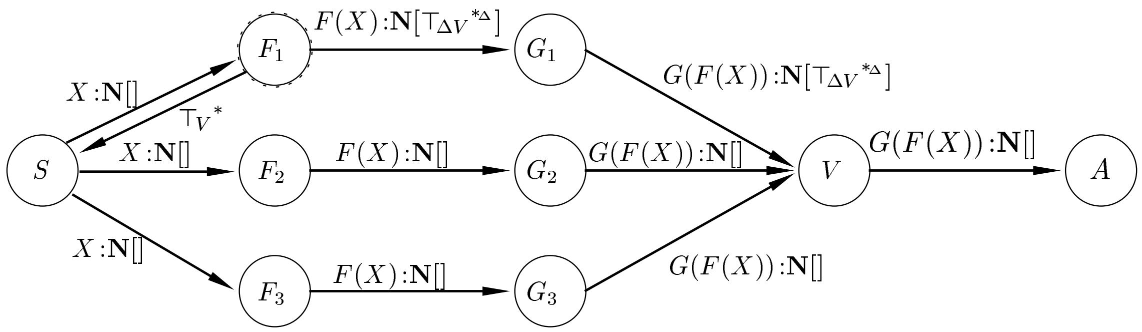

Figure 1.1: Example of graphical representation of system behavior.

The changes to this behavior caused by failure of F_1 are represented by the perturbations in the perturbed ms-atoms as well as by the new ms-atoms, which represent messages having no analogue in the failure-free behavior of the system. Thus, the graph represents the case where F_1 fails by sending perturbed messages to G_1 and new messages to S. The symbol $\top_{\Delta V}$ in a ms-atom denotes an arbitrary change to the data sent in the message, and the superscript $*_\Delta$ denotes an arbitrary change to the multiplicity. Thus, the graph of Figure 1.1 reflects that when F_1 fails, it might send an arbitrary number of arbitrary messages to G_1. The abstract value $\top_V$ represents all concrete values; thus, when F_1 fails, it might also send an arbitrary number of arbitrary messages to S. Since G_1's input is perturbed arbitrarily, so its output is function G applied to an arbitrary value. Without specific information about function G, the result of the application is itself just an arbitrary value, so the output of G is also perturbed arbitrarily. As before, symbol $\top_{\Delta V}$ represents an arbitrary perturbation. The output of the voter is not perturbed—empty square brackets in a perturbed ms-atom are used to denote "no change". So, if the fault-tolerance requirement for this system is that the input to the actuator is unchanged in this failure scenario, then this graph states that the system satisfies this requirement.

Including perturbations in ms-atoms allows the sensitivity of a component to perturbations of its inputs to be expressed directly. Without this notation, this information sometimes would have to be encoded awkwardly in the values and multiplicities, and sometimes would not be expressible at all. For example, consider again the system shown in Figure 1.1. In a framework without explicit perturbations, there is no way to express whether or not the new messages sent from F_1 to S confuse the source and cause changes to the data that S sends to F_2 and F_3. The problem is that even if the data sent by S changes, it is still represented by the symbolic value X. In the graph in Figure 1.1, the square brackets on the source's outputs are empty; this states that outputs from S are indeed unchanged by the additional messages sent from F_1 to S.

The use of explicit perturbations also allows fault-tolerance requirements to be conveniently expressed as constraints on the acceptable perturbations to system behavior. Recall that in the non-perturbational framework, the condition associated with a failure scenario is simply a predicate that the system behavior in that failure scenario must satisfy. In the perturbational framework, the condition associated with a failure scenario is allowed to be a relation that must hold between the system's failure-free behavior and its behavior in that failure scenario. For example, a typical fault-tolerance requirement is that inputs to certain components are unchanged in certain failure scenarios; a weaker requirement is that inputs to those components be either unchanged or absent.

Explicit perturbations may be unfamiliar in fault-tolerance analysis, but they are analogous to familiar techniques for analysis of numerical error [Sca62]. Error analysis focuses on how numerical errors introduced in one part of the computation are propagated by subsequent computation. Analogously, fault-tolerance analysis with explicit perturbations focuses on how perturbations introduced by failures are propagated during subsequent execution of the system. Note that the separation of error analysis from other aspects of correctness of a numerical computation is analogous to separation of fault-tolerance analysis from other correctness concerns.

Feasibility of the Approach. The computational complexity of our analysis method depends on the number of failure scenarios for which the analysis must be performed and on the cost of the analysis for each failure scenario. The number of failure scenarios sometimes can be reduced by use of symmetry arguments (if one failure-scenario is a symmetric variant of another) and by abstracting from the timing of failures (making each failure scenario specify less precisely when the failure occurs). The cost of the analysis for one failure scenario depends largely on the complexity of the fault-tolerance mechanisms—since their behavior must be analyzed—and on the extent to which separation of concerns can be achieved, i.e., the extent to which other mechanisms in the system can be ignored in the analysis. Complex fault-tolerance mechanisms—for example, protocols involving many rounds of communication—will take longer to analyze than simpler ones, but this seems inevitable. Separation of concerns is a design principle underlying many fault-tolerant systems, so achieving a similar separation of concerns in the analysis is a natural goal. The key is to find approximations that are coarse enough for tractability and precise enough to validate interesting systems (i.e., precise enough not to yield false negatives).

The practical utility of our (or any) approach to fault-tolerance analysis can be determined only by trying it on a wide range of fault-tolerant systems. Therefore, not only does this thesis introduce an approach, but it examines the applicability of that approach to some important classes of fault-tolerant systems. Classic algorithms for Byzantine agreement and reliable broadcast are analyzed. New protocols for fault-tolerant computation with moving agents are also analyzed. These analyses were performed using a prototype tool, described in Appendix B, that implements the analysis and provides a graphical interface to it.

1.3 Outline of the Dissertation

Chapter 2 presents a framework that incorporates abstraction mechanisms but ignores failures. Chapter 3 extends that framework for fault-tolerance analysis. First, a non-perturbational framework is described, in which each component is parameterized by its possible failures, allowing the component's behavior to be described separately for each possible failure. Limitations of this approach are discussed, motivating the introduction of explicit perturbations, which enable expression of additional correlations between the failure-free and faulty behaviors of a component.

Chapter 4 analyzes algorithms for two classic problems in fault-tolerance: Byzantine agreement [LSP82] and reliable broadcast [HT94]. These examples illustrate analysis of systems subject to Byzantine failures and crash failures, respectively. Chapter 5 applies our framework to fault-tolerant moving agents, a more recent concern. This illustrates how our analysis methods can be applied to protocols that employ cryptographic techniques.

Chapter 6 discusses related work and presents several ideas for future work. Appendix A contains an index of symbols. Appendix B describes our prototype implementation of and graphical interface for the analysis.

Chapter 2

Analyzing Systems that Never Fail

Our system model is based on Gilles Kahn's stream-processing model of parallel and distributed systems [Kah74]. For concreteness, we, like Kahn, consider systems of components that communicate through asynchronous FIFO channels. However, this assumption is not essential to our approach.

The system model presented in this chapter differs from other descendants of Kahn's model mainly by having mechanisms that allow approximation of system behavior. The traditional purpose of stream-processing models is to provide compositional semantics for networks of communicating processes, thereby serving as the foundation of compositional frameworks for (manually) proving properties. In contrast, our goal is to develop a foundation for automated analysis. Approximations are necessary to make this analysis tractable. The approximations we use abstract from aspects of a system that do not directly impact fault-tolerance.

Section 2.1 presents a model of concrete processes. Section 2.2 describes abstraction mechanisms used to approximate a system's behavior. Section 2.3 discusses the abstract representation of processes and defines the fixed-point analysis of system behavior. Section 2.4 relates the two models. It shows soundness of the fixed-point analysis: if an appropriate fixed-point exists at the abstract level, then that fixed-point represents all possible behaviors of the system of processes. Finally, Section 2.5 discusses conditions under which a fixed-point is guaranteed to exist and under which iterative calculation of the fixed-point is guaranteed to terminate.

2.1 Concrete Model

Kahn's original model [Kah74] handles only *determinate* systems, systems in which each component is:

Deterministic: For each possible sequence of inputs, the component has exactly one possible sequence of outputs.

Strict: The input ("receive") primitives are restricted so that at any time, a component can be waiting to receive a message from at most one component. This ensures that components are insensitive to non-determinism in message reception order caused by an asynchronous FIFO network.

After summarizing Kahn's model, we present an extension based on [Bro87,Bro90] that eliminates these restrictions.

2.1.1 Kahn's Model of Determinate Systems

A system is a collection of communicating components. Each component is represented by a function describing its input-output behavior. An *input-output function* takes as argument the sequences of messages received by a component (during some computation) and returns the sequences of messages sent by that component as a result of receiving those messages. An input-output function is sometimes called a stream-processing function [BD92], stream transformer [DS89], or history function [BA81], because it maps a stream of input messages (the input history) to a stream of output messages (the output history).

If we depict the behavior of a system by a graph in which each node corresponds to a component and each edge is labeled with the sequence of messages sent from the source to the target, then an input-output function for a component takes as argument the sequences of messages labeling the inedges of the node corresponding to that component and returns the sequences of messages labeling the outedges of that node. Figure 2.1 shows such a graph. Note that double angle brackets ($\langle\!\langle \cdot \rangle\!\rangle$) construct sequences. Component C receives inputs from components A and B. Applying the input-output function associated with C to those input sequences yields the sequences shown on the edges from C to D and E. In this case, the input-output function associated with C happens to forward messages from A to D and from B to E.

These ideas are formalized as follows. A system comprises a set of named components. The names serve as addresses for specifying the recipient of a message. Let *Name* denote

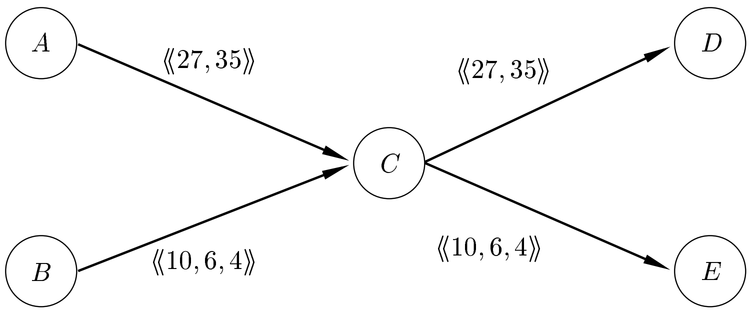

Figure 2.1: Example of graphical representation of system behavior.

the names of the system components. For example, for the system pictured in Figure 2.1, $Name = \{A, B, C, D, E\}$. Let $CVal$ (mnemonic for "Concrete Values") be the set of values that can be transmitted in messages. A (concrete) history (of the messages sent along a channel) has signature

$$CHist \triangleq Name \to Seq(CVal), \tag{2.1}$$

where $\triangleq$ means "equal by definition", and for any set S, $Seq(S)$ is the set of finite and infinite sequences of elements of S (with zero-based indexing). Also, for any sets S and T, a function of signature $S \to T$ is a total function from S to T. When a history ch is regarded as the input to a component x, $ch(y)$ is the sequence of messages sent by y to x; when ch is regarded as the output of a component x, $ch(y)$ is the sequence of messages sent by x to y. For example, in Figure 2.1, the input history of component C is

$$\begin{aligned} ch_0 = (\lambda x\!:\!Name.\ &\mathbf{if}\ x = A\ \mathbf{then}\ \langle\!\langle 27, 35 \rangle\!\rangle \\ &\mathbf{else\ if}\ x = B\ \mathbf{then}\ \langle\!\langle 10, 6, 4 \rangle\!\rangle \\ &\mathbf{else}\ \varepsilon), \end{aligned} \tag{2.2}$$

where ε is the empty sequence, and the output history of component C is

$$\begin{aligned} (\lambda x\!:\!Name.\ &\mathbf{if}\ x = D\ \mathbf{then}\ \langle\!\langle 27, 35 \rangle\!\rangle \\ &\mathbf{else\ if}\ x = E\ \mathbf{then}\ \langle\!\langle 10, 6, 4 \rangle\!\rangle \\ &\mathbf{else}\ \varepsilon). \end{aligned}$$

Partial ordering $\leq_{CHist}$ on $CHist$ is the pointwise extension of the prefix ordering $\leq_{Seq}$ on sequences. More explicitly,

$$ch_1 \leq_{CHist} ch_2 \triangleq (\forall x \in Name : ch_1(x) \leq_{Seq} ch_2(x)). \tag{2.3}$$

Each component of a system—whether a piece of hardware, a process (in the usual sense), a moving agent, etc.—is represented as a process. A *determinate process* is a function with signature

$$\mathit{DProcess} \triangleq \mathit{CHist} \twoheadrightarrow \mathit{CHist}, \tag{2.4}$$

where the two-headed arrow indicates a restriction to monotonic and continuous functions. The argument of the function contains the input sequences, and the result contains the corresponding output sequences. Informally, the restriction to monotonic functions ensures that providing additional inputs to a component can't cause the component to produce fewer outputs (i.e., the component can't "take back" outputs it already emitted). Continuity ensures that a component can't produce outputs only after receiving an infinite sequence of inputs.

A system is represented by a function $np \in \mathit{Name} \rightarrow \mathit{DProcess}$ (np is mnemonic for "name → process"), which describes how each component of the system behaves. And, a run of a system is represented by a function with signature

$$\mathit{CRun} \triangleq \mathit{Name} \rightarrow \mathit{CHist}. \tag{2.5}$$

For $cr \in \mathit{CRun}$, we adopt the convention that $cr(x)$ is the input history of component x in the run, i.e., $cr(x)(y)$ is the sequence of messages sent to x by y. For example, the graph in Figure 2.1 corresponds to the run

$$\begin{aligned}(\lambda x\!:\!\mathit{Name}.\ &\mathbf{if}\ x = C\ \mathbf{then}\ ch_0\\ &\quad\mathbf{else\ if}\ x = D\ \mathbf{then}\ (\lambda y\!:\!\mathit{Name}.\ \mathbf{if}\ y = C\ \mathbf{then}\ \langle\!\langle 27, 35\rangle\!\rangle\ \mathbf{else}\ \varepsilon)\\ &\quad\mathbf{else\ if}\ x = E\ \mathbf{then}\ (\lambda y\!:\!\mathit{Name}.\ \mathbf{if}\ y = C\ \mathbf{then}\ \langle\!\langle 10, 6, 4\rangle\!\rangle\ \mathbf{else}\ \varepsilon)\\ &\quad\mathbf{else}\ (\lambda y\!:\!\mathit{Name}.\ \varepsilon)),\end{aligned}$$

where ch_0 is defined by (2.2).

The run representing the behavior of a system np of determinate processes is defined as follows. For a run $cr \in \mathit{CRun}$, the input history of component x is $cr(x)$. The output history of x on those inputs is $np(x)(cr(x))$. Taken together, these new output histories for each component define new input histories for each component; specifically, these new input histories form the run

$$\mathit{step}(np)(cr) \triangleq (\lambda y\!:\!\mathit{Name}.\ (\lambda x\!:\!\mathit{Name}.\ np(x)(cr(x))(y))). \tag{2.6}$$

For a run cr that represents a complete execution of the system, this new run must equal the run cr that we started with, since the output histories defined by cr must already reflect processing of all messages in the input histories defined by cr. Thus, the behavior of the system is represented by a run cr satisfying

$$(\forall x \in \mathit{Name} : (\forall y \in \mathit{Name} : cr(y)(x) = np(x)(cr(x))(y))); \tag{2.7}$$

more precisely, it is represented by the least such run, where the ordering $\leq_{\mathit{CRun}}$ on CRun is the pointwise extension of the ordering $\leq_{\mathit{CHist}}$ on concrete histories [Kah74]. Equivalently, this run can be characterized as the least fixed-point of $\mathit{step}(np) \in \mathit{CRun} \to \mathit{CRun}$.[1] Thus, the run representing the behavior of system np is

$$\mathit{crun}(np) \triangleq \mathrm{lfp}(\mathit{step}(np)). \tag{2.8}$$

This model has two main attractions for us, besides its elegance. First, input-output functions provide abstraction. Just as an abstract data type hides internal details of objects, an input-output function hides the local state and internal computations used to implement the component, describing only the externally visible behavior of the component. In our framework, this abstraction provides a convenient separation of local and global analyses. Local analysis is done on each component to determine an input-output function that represents its behavior. The global analysis, embodied as a fixed-point calculation, determines the entire system's behavior—the possible histories of messages.

Second, defining a system's behavior as a fixed-point facilitates verification and simulation. It facilitates verification by allowing powerful induction rules in proving properties of the fixed-point. It facilitates simulation by providing a simple and effective procedure for computing (finite prefixes of) the system's behavior.

The computation of least fixed-points is based on the following definitions and classic theorem.

Basic Fixed-Point Theory. The *upper-closure* of an element x of a partial order $\langle S, \leq_S \rangle$ is $\{y \in S \mid x \leq_S y\}$. Note that single angle brackets ($\langle \cdot \rangle$) construct tuples. We often omit the ordering $\leq_S$ when it is obvious from context. A *chain* of a partial order $\langle S, \leq_S \rangle$ is an increasing sequence of elements of S; thus, the set of chains of a partial order is given by

$$\mathit{Chain}(\langle S, \leq_S \rangle) = \{\sigma \in \mathit{Seq}(S) \mid (\forall i \in (\mathit{dom}(\sigma) \setminus \{0\}) : \sigma[i-1] \leq_S \sigma[i])\}. \tag{2.9}$$

where for a sequence σ, $|\sigma|$ is the length of σ and $\mathit{dom}(\sigma)$ is the index set of σ, i.e., $\mathit{dom}(\sigma) = \{i \in \mathsf{N} \mid i < |\sigma|\}$. An ω-*chain* is a chain of length ω. Let $\langle\!\langle g(i) \rangle\!\rangle_{i \in \mathsf{N}}$ denote the sequence

$$\langle\!\langle g(0), g(1), g(2), \ldots \rangle\!\rangle,$$

[1] As discussed below, existence of this fixed-point follows from Theorem 2.1.

where N denotes the natural numbers. An *ω-complete partial order* (abbreviated ω-cpo) is a partial order $\langle S, \leq_S \rangle$ such that every ω-chain σ of $\langle S, \leq_S \rangle$ has a least upper bound, denoted $\text{lub}(\sigma)$. A function $f \in S \to S$ is *continuous* if it is (1) monotonic and (2) preserves least upper bounds of ω-chains, i.e., for all ω-chains σ, $f(\text{lub}(\sigma)) = \text{lub}(\langle\!\langle f(\sigma[i]) \rangle\!\rangle_{i \in \mathsf{N}})$. Among the most celebrated and useful properties of ω-cpos is [Gun92, chapter 4]:

Theorem 2.1. Let $\langle S, \leq_S \rangle$ be an ω-cpo. Let f be a continuous function in $S \to S$, and let $x \in S$ be such that $x \leq_S f(x)$. Then f has a fixed-point in the upper-closure of x in S. Furthermore, the least such fixed-point is $\text{lub}(\langle\!\langle f^i(x) \rangle\!\rangle_{i \in \mathsf{N}})$.

Note that $f^i(x)$ denotes f applied i times to x.

Computing Runs. It is easy to check that *CRun* is an ω-cpo with least element

$$\perp_{CRun} \triangleq (\lambda x : Name.\ \perp_{CHist}), \tag{2.10}$$

where the least element $\perp_{CHist}$ of *CHist* is

$$\perp_{CHist} \triangleq (\lambda x : Name.\ \varepsilon). \tag{2.11}$$

Recall that determinate processes are, by definition (2.4), monotonic and continuous. It is easy to check that this restriction on determinate processes implies that for all $np \in Name \to DProcess$, $step(np)$ is monotonic and continuous. It follows from Theorem 2.1 that for all $np \in Name \to DProcess$, $step(np)$ has a least fixed-point in *CHist*, given by

$$crun(np) = \text{lub}(\langle\!\langle step(np)^i(\perp_{CRun}) \rangle\!\rangle_{i \in \mathsf{N}}).$$

Thus, prefixes of the system's behavior can be calculated by starting with $\perp_{CRun}$ and repeatedly applying $step(np)$.

2.1.2 Concrete Model of Non-Determinate Systems

Recall that Kahn's model deals only with determinate systems, i.e., systems whose components are deterministic and strict. Restriction to determinate systems is a severe limitation, and approaches to eliminating this restriction have been suggested [BA81,BM82,SN85,Bro87, Jon89,Bro90,Rus90]. Here, we will adopt a slight variant of an approach due to Broy [Bro87, Bro90]. Two ideas are involved: one to handle non-determinism, and a second to handle non-strictness. The idea for handling non-determinism is to represent a non-deterministic process as a set of determinate processes: each determinate process in the set corresponds to one possible behavior of the non-deterministic process. To see why this first idea alone is insufficient to model non-strictness, consider a non-strict process *first* that waits for an input from process x_1 or x_2 then echoes that input on its output. If we try to represent this process as a set of determinate processes, the obvious candidate is $\{dp_1, dp_2\}$, where dp_i waits for an input from process x_i. However, this is not equivalent to *first*, because if the only input comes from (say) x_1, then dp_2 blocks forever, producing no output, so $\{dp_1, dp_2\}$ might produce no output on this input; in contrast, *first* definitely produces an output. One solution is to indicate that dp_2 represents the behavior of *first* only when the input contains a message from x_2. More generally, we associate with each determinate process dp in the set an *input-restriction*, which is the set of inputs for which dp represents a possible behavior of the component [Bro87,Bro90]. Thus, (non-determinate) processes are in

$$Process \triangleq Set(IRProcess), \tag{2.12}$$

where input-restricted processes are in

$$IRProcess \triangleq DProcess \times Set(CHist). \tag{2.13}$$

To summarize, for an input-restricted process p, for each $\langle dp, ir\rangle \in p$, p can behave like the determinate process dp, but only on inputs in ir. Thus, the set of possible runs of a system $np \in Name \to Process$ of non-determinate processes is

$$\begin{aligned}
cruns(np) \triangleq \{cr \in CRun \mid\ & \\
& (\exists h \in Name \to IRProcess : \\
& \quad \wedge (\forall x \in Name : h(x) \in np(x)) \\
& \quad \wedge (\forall x \in Name : enabled(h(x), \langle\!\langle step(\pi_1 \circ h)^i(\bot_{CRun})(x)\rangle\!\rangle_{i \in \mathsf{N}})) \\
& \quad \wedge cr = \mathrm{lfp}(step(\pi_1 \circ h)))\},
\end{aligned} \tag{2.14}$$

where π_i projects the ith component of a tuple, $\circ$ denotes function composition, and an input-restricted process $p \in IRProcess$ is enabled for a chain $\sigma \in Chain(CHist)$ of inputs iff

$$enabled(p, \sigma) \triangleq (\sigma \cup \{\mathrm{lub}(\sigma)\}) \subseteq \pi_2(p). \tag{2.15}$$

For notational convenience, we sometimes (as here) regard a sequence as the set of its elements.

The conjunction in (2.14) is written using Lamport's bullet-style notation [Lam93]. In this notation, the conjunction $b_1 \wedge b_2 \wedge \cdots \wedge b_n$ is written

$$\begin{aligned} &\wedge\, b_1 \\ &\wedge\, b_2 \\ &\vdots \\ &\wedge\, b_n \end{aligned}$$

Bullet-style notation can be used for disjunction ($\vee$) as well.

2.1.3 A Running Example

To illustrate this model, we return to the two-stage replicated pipeline of Figure 1.1. Here, we consider a failure-free version of that system; failures will be considered in Chapter 3. The two stages compute the functions $F \in \mathsf{N} \to \mathsf{N}$ and $G \in \mathsf{N} \to \mathsf{N}$, respectively.[2] The system contains a source S, which sends a number i to three processors F_1, F_2, and F_3, which each compute $F(i)$ then send the result to the next stage in the pipeline. Processors G_1, G_2, and G_3 in the next stage compute G of their input then send the results to a 3-input voter V. For convenience, we assume that processors F_1–F_3 and G_1–G_3 are well-behaved on unexpected inputs—specifically, that each input not in N is treated as if it were the input "0". Voter V waits for an input from each G_i, computes the majority of those inputs, and then sends the result to an actuator A. More precisely, the voter computes a 3-way majority function maj, which is any function in $\mathsf{N}^3 \to \mathsf{N}$ such that if any two of its three arguments have the same value i, then the value of the majority function on those arguments is i.

Source S may contain physical sensors (e.g., a keyboard or an air-speed indicator). Therefore, we model the source as a non-deterministic process $CSrc(\{F_1, F_2, F_3\})$. For $dests \in Set(Name)$, $CSrc(dests)$ is a process that non-deterministically selects a natural number and then sends it to each component named in $dests$. Non-deterministic process $CSrc(dests)$ is defined in terms of a family of determinate processes: $CSrc_i(dests)$ is a determinate process that sends the number i to the components named in $dests$. Note that $CSrc(dests)$ is trivially strict, since it ignores input messages; thus, the input-restriction associated with each $CSrc_i(dests)$ is $CHist$, which is no restriction at all.

$$CSrc(dests) \;=\; \bigcup_{i \in \mathsf{N}} \{\langle CSrc_i(dests), CHist\rangle\} \tag{2.16}$$

$$CSrc_i(dests) \;=\; (\lambda h\!:\!CHist.\ (\lambda x\!:\!Name.\ \textbf{if}\ x \in dests\ \textbf{then}\ \langle\!\langle i \rangle\!\rangle\ \textbf{else}\ \varepsilon)) \tag{2.17}$$

[2] F is overloaded: it is both a symbol in our language of values (defined in Section 2.2.2) and the name of the mathematical function represented by that symbol. Similarly for G and for maj below.

Processes for F_1–F_3 and G_1–$G-3$ are defined as instances of $CComp(src, dest, \phi)$, which is a process that receives values from src, computes a function ϕ on those values, and sends the results to $dest$:

$$\begin{aligned} CComp(src, dest, \phi) &= \{\langle CComp'(src, dest, \phi), CHist\rangle\} && (2.18)\\ CComp'(src, dest, \phi) &= (\lambda h\colon CHist.\ (\lambda x\colon Name. && (2.19)\\ &\qquad\qquad \textbf{if } x = dest \textbf{ then } \overline{\phi}(h(src)) \textbf{ else } \varepsilon)), \end{aligned}$$

where $\overline{\phi}$ is a pointwise extension of ϕ to sequences of concrete values, with values not in N treated as zero. For example, the behavior of F_1 is described by the process $CComp(S, G_1, F)$.

The voter V in this system is a process $CVoter(G_1, G_2, G_3, A)$, where for any $s_1, s_2, s_3, dest \in Name$, $CVoter(s_1, s_2, s_3, dest)$ waits (i.e., produces no output) until it has received an input from each of s_1, s_2, and s_3, computes a majority using the first input from each of s_1, s_2, and s_3, and then sends the result to $dest$.[3]

$$\begin{aligned} CVoter(s_1, s_2, s_3, dest) &= \{\langle CVoter'(srcs, dest), CHist\rangle\} && (2.20)\\ CVoter'(s_1, s_2, s_3, dest) &= (\lambda h\colon CHist.\ (\lambda x\colon Name. && (2.21)\\ &\qquad \textbf{if } x = dest \textbf{ then}\\ &\qquad\quad \textbf{if } h(s_1) = \varepsilon \vee h(s_1) = \varepsilon \vee h(s_1) = \varepsilon \textbf{ then } \varepsilon\\ &\qquad\quad \textbf{else } \langle\!\langle maj(h(s_1)[0], h(s_2)[0], h(s_3)[0])\rangle\!\rangle\\ &\qquad \textbf{else } \varepsilon)) \end{aligned}$$

It is easy to generalize this definition to voters that have an arbitrary number of inputs.

The actuator is the process $CAct$, which is just a message sink:

$$\begin{aligned} CAct &= \{\langle CAct', CHist\rangle\} && (2.22)\\ CAct' &= (\lambda h\colon CHist.\ (\lambda x\colon Name.\ \varepsilon)) && (2.23) \end{aligned}$$

For this system, $Name = \{S, F_1, F_2, F_3, G_1, G_2, G_3, V, A\}$. Let np^{re} (re is mnemonic for "running example") be the obvious mapping from names to processes: $np^{re}(S) = CSrc(\{F_1, F_2, F_3\})$, *etc.* It is easy to check that

$$cruns(np^{re}) = \bigcup_{i \in \mathsf{N}} \{cr^{re}(i)\} \qquad (2.24)$$

[3]This defines a strict voter. Alternatively, we could have defined a non-strict voter, which produces an output immediately after receiving two equal inputs.

$$
\begin{aligned}
cr^{re}(i) \quad = \quad & (\lambda y\!:\!Name.\,(\lambda x\!:\!Name. \\
& \quad \textbf{if } x = S \wedge y \in \{F_1, F_2, F_3\} \textbf{ then } \langle\!\langle i \rangle\!\rangle \\
& \quad \textbf{else if } \langle x, y\rangle \in \{\langle F_1, G_1\rangle, \langle F_2, G_2\rangle, \langle F_3, G_3\rangle\} \textbf{ then } \langle\!\langle F(i) \rangle\!\rangle \\
& \quad \textbf{else if } x \in \{G_1, G_2, G_3\} \wedge y = V \textbf{ then } \langle\!\langle G(F(i)) \rangle\!\rangle \\
& \quad \textbf{else if } x = V \wedge y = A \textbf{ then } \langle\!\langle G(F(i)) \rangle\!\rangle \\
& \quad \textbf{else } \varepsilon)).
\end{aligned}
\tag{2.25}
$$

Each run $cr^{re}(i)$ can be represented by a graph like the one in Figure 2.1 (page 11). To obtain a finite graphical representation of the infinite family of runs given by (2.25), abstractions (such as those described in the next section) are employed.

2.2 Representation of Runs

Computing the exact set of runs for a system is, in general, infeasible. Thus, the model described in Section 2.1—hereafter called the *concrete model*—is inadequate for automated analysis of systems. This inadequacy motivates the development of a system model that incorporates approximations. To distinguish our new model from the concrete one we already discussed, we sometimes refer to the new model as the *abstract model.*

In the abstract model, a system's behavior is approximated by:

Values: The data sent in messages.

Multiplicities: The number of times each value is sent.

Orderings: The order in which values are sent.

This section, then, describes how each of these three kinds of information is represented in our framework.

Recall that in the concrete model, runs are characterized by the sequences of messages sent between each pair of components. Thus, our abstract model is based on a language for approximating sequences of messages. Specifically, we approximate sequences of messages by partially-ordered sets of ms-atoms: each ms-atom approximates a set of messages, and the partial order approximates the order in which those messages appear in the sequences. Note that this order corresponds to the order in which those messages are sent and, since channels are assumed to be FIFO, also the order in which they are received. Let L denote the set of ms-atoms. A particular definition of L appears below; for now, it suffices to say that each element of L approximates a set of messages. A strict[4] partial order on a set S is an element of

$$Order(S) \triangleq \{\prec \in S \times S \mid \prec \text{ is acyclic and transitive}\}. \tag{2.26}$$

Note that $\in$ has lower precedence than all set constructors, such as $\times$. The strictly-partially-ordered sets (*posets* for short) over a set T is just a subset of T together with a partial order on that subset:

$$POSet(T) \triangleq \{\langle S, \prec\rangle \in Set(T) \times Set(T \times T) \mid \ \prec \in Order(S)\}. \tag{2.27}$$

Informally, a poset $\langle S, \prec\rangle \in POSet(L)$ approximates a sequence σ of messages if there exists a correspondence between elements of S and elements of σ such that: (1) each ms-atom in S approximates the set of corresponding messages, and (2) if $\ell_1 \prec \ell_2$, then all messages corresponding to ℓ_1 precede all messages corresponding to ℓ_2 (this definition is formalized in Section 2.4.1).

At the concrete level, sequences of messages are aggregated into concrete histories *CHist*, defined in (2.1), to represent all the inputs or outputs of a component. Analogously, at the abstract level, we aggregate posets of ms-atoms into histories:

$$Hist \triangleq Name \to POSet(L), \tag{2.28}$$

Histories are interpreted using the same conventions as concrete histories: when a history $h \in Hist$ is regarded as the input to a component x, $h(y)$ is the sequence of messages sent by y to x; when h is regarded as the output of a component x, $h(y)$ is the sequence of messages sent by x to y.

Just as concrete histories are aggregated into concrete runs to represent the overall behavior of a system, in the abstract model, we aggregate histories into runs:

$$Run \triangleq Name \to Hist. \tag{2.29}$$

As in the concrete model, we adopt the convention that, for a run $r \in Run$ and a component x, history $r(x)$ represents the inputs to x. Informally, the meanings of histories and runs are pointwise extensions of the meaning of posets of ms-atoms. Note that a run $r \in Run$ can be interpreted as a labeled directed graph with set of nodes *Name* and with edge $\langle x, y\rangle$ labeled with $r(y)(x)$. Since an (abstract) run can approximate many concrete runs, it suffices to use a single abstract run to approximate the behavior of a system. Thus, the result of our analysis is a single run that approximates—or *represents*—all concrete runs of a system.

[4]For partial orders, "strict" means "not reflexive". This is unrelated to strictness of processes.

The remainder of this section defines the set L of ms-atoms, illustrates the definitions with an example, and discusses our approximation of message orderings.

2.2.1 ms-atoms

A ms-atom has signature

$$L \triangleq \mathit{Mul} \times \mathit{Val} \times \mathit{Tag}. \tag{2.30}$$

The multiplicity *Mul* indicates how many messages are represented by the ms-atom; the value *Val* describes the data in those messages. For example, by analogy with regular expressions, we use a multiplicity of ? to denote zero or one messages, and a multiplicity of $*$ to denote an arbitrary number of messages. For values, we use, for example, a value of $\mathbf{N}$ to indicate that the data is a natural number. These are all examples of abstract values: each specifies a set of possible concrete value. Symbolic values—expressions that represent a single but sometimes undetermined concrete value—are also used in *Val* and *Mul*. The sets *Val* and *Mul* and our treatment of message orderings are discussed in the following subsections. The tag is a technicality. It allows multiple ms-atoms with the same value and multiplicity to appear on an edge; an alternative is to use bags (i.e., multisets) of ms-atoms with signature $\mathit{Mul} \times \mathit{Val}$.

2.2.2 Values

Values are approximated by a set of possibilities. Each of these possibilities is determined by a symbolic value and an abstract value, rather than (say) a single concrete value. Let *AVal* and *SVal* denote the sets of abstract and symbolic values, respectively. Values have signature

$$\mathit{Val} \triangleq \mathcal{P}_{\mathit{fin}}(\mathit{SVal} \times \mathit{AVal}) \setminus \{\emptyset\}, \tag{2.31}$$

where $\mathcal{P}_{\mathit{fin}}(S)$ is the set of finite subsets of S, and $\setminus$ denotes set difference.

Abstract Values

An abstract value characterizes the concrete values that might be transmitted in particular messages. This idea is familiar from abstract interpretation [AH87]: each element of an abstract interpretation represents a set of concrete values. For example, one way to abstract the real numbers is to keep track only of the sign. The corresponding abstract interpretation is $\{0, \mathbf{R}_-, \mathbf{R}_+, \mathbf{R}\}$, where 0 represents only itself, $\mathbf{R}_-$ represents the non-positive real numbers, $\mathbf{R}_+$ represents the non-negative real numbers, and $\mathbf{R}$ represents all real numbers. The abstract value **R** is included, even though it does not determine the sign, in case the sign of a value is not known.

For generality, we do not fix particular abstract values. The abstract framework is parameterized by the set $AVal$ of abstract values and their meaning, given by $[\![]\!]_{AVal} \in Interp_{Set}(AVal)$, where for a set S,

$$Interp_{Set}(S) \triangleq S \rightarrow Set(CVal). \tag{2.32}$$

Symbolic Values

Symbolic values track relationships between values. To motivate the need for this, consider the two-stage replicated pipeline described in Section 2.1.3 and depicted in Figure 1.1. The outputs of voter V depend on equality relationships among its inputs. Abstract values can provide some information about these relationships. However, if two inputs both have abstract value **N**, there is no way to tell (from this fact alone) whether they are equal.

Additional relationships can be determined using symbolic values, which are expressions composed of constants Con and variables Var . We assume the sets Con and Var are disjoint and do not contain the underscore symbol. Informally, a constant represents the same value in every run of the system, while a variable represents values that may be different in different concrete runs of the system.[5]. Symbolic values are obtained using a form of symbolic computation, which we will integrate into the input-output functions and hence into the fixed-point analysis. To continue the pipeline example, note that if multiple inputs to the voter are represented by the same symbolic value, then they are equal.

Variables. Each variable is associated with a single component; we say the variable is *local to* that component. Informally, the value of that variable is determined by the behavior of that component. Making each variable local to a *single* component avoids name clashes that would otherwise cause trouble when components are assembled to form a system. For example, suppose one component nondeterministically selects and outputs a real number, representing it by the value $\{\langle X, \mathbf{R}\rangle\}$. Note that this value contains a single possibility $\langle X, \mathbf{R}\rangle$, where X is a variable and **R** is an abstract value. In isolation, this representation is fine. But suppose another component in the system also nondeterministically selects and outputs a real number, representing it by the same value $\{\langle X, \mathbf{R}\rangle\}$. Since the two components may choose different values, there might not be a single interpretation of X that "matches"

[5]This is made precise by the semantics given in Section 2.4.1

the overall behavior of the system (i.e., that matches the output of both components). This problem is avoided if each component uses a local variable to represent its output.

Let $Var(x)$ denote the set of variables local to component x. We require that $Var(x)$ and $Var(y)$ be disjoint for $x \neq y$. Var is

$$Var = \bigcup_{x \in Name} Var(x) \tag{2.33}$$

For example, in Figure 1.1, variable X is local to the source S.

Constants. The value of a constant is fixed not by the behavior of one component but by an *interpretation* $\rho_c \in Interp(Con)$, where for any set S,

$$Interp(S) \triangleq S \rightarrow CVal \tag{2.34}$$

For example, a constant min might be interpreted as the function that returns the minimum of a set of numbers; a constant *encrypt* might be interpreted as DES encryption. In Figure 1.1, constants F and G represent the functions computed by the first and second stage, respectively, of the pipeline.

Syntax of Symbolic Values. Symbolic values are expressions built from constants and variables:

$$SVal_0 \triangleq Sym \cup \{s(s_1, \ldots, s_n) \mid s \in Sym \wedge s_1 \in SVal_0 \wedge \cdots \wedge s_n \in SVal_0\}, \tag{2.35}$$

where the set Sym of symbols is

$$Sym \triangleq Con \cup Var. \tag{2.36}$$

For example, if the constant min is interpreted as above, then the symbolic value $\mathrm{min}(X, Y, Z)$ represents the minimum of the values represented by X, Y, and Z.

The set of symbolic values is obtained from $SVal_0$ by adding the underscore symbol, which is used as a *wildcard*:[6]

$$SVal \triangleq SVal_0 \cup \{\_\} \tag{2.37}$$

As in pattern-matching in the programming language ML [MTH90], the wildcard can represent any value. This is especially significant when the wildcard appears in ms-atoms

[6]We could allow the wildcard to appear within larger symbolic values, but this slightly complicates the semantics, and it's not clear whether it is useful.

that may represent multiple messages. Without the wildcard, a ms-atom containing a value expressed using n elements of $SVal \times AVal$ could represent only sets of messages containing at most n distinct concrete values; if the value contains a wildcard, the ms-atom could represent a set of messages containing arbitrarily many distinct concrete values.

Notational Conventions. Since abstract values are analogous to types, we sometimes write $s : a$ to denote $\langle s, a \rangle \in SVal \times AVal$. We often elide the braces around singleton sets; for example, we might write $X : \mathbf{R}_+$ to denote $\{\langle X, \mathbf{R}_+ \rangle\} \in Val$. We sometimes elide the wildcard; for example, we might write $\mathbf{R}_+$ to denote $\_ : \mathbf{R}_+$. These conventions are used in Figure 1.1; for example, the value in the new ms-atom on edge $\langle F_1, S \rangle$ is actually $\{\langle \_, \top_V \rangle\}$.

Running Example Revisited. The concrete runs (2.25) of the system introduced in Section 2.1.3 are represented by Figure 2.2. In this example, $X \in Var(S)$, $\{F, G\} \subseteq Con$,

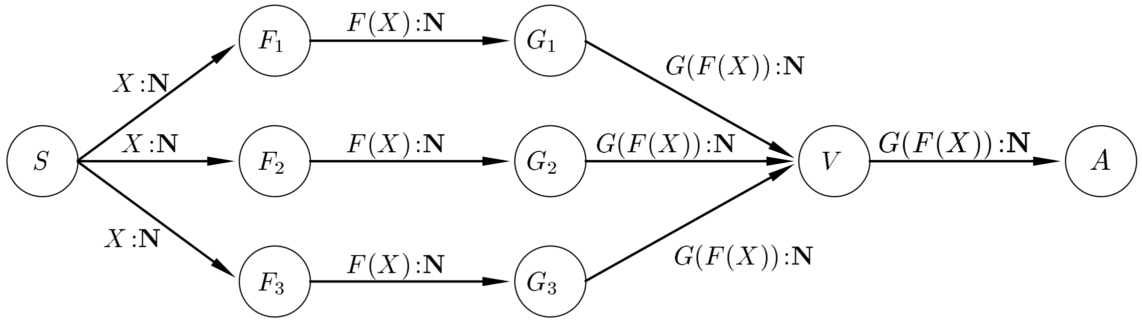

Figure 2.2: Run for running example.

and $[\![\mathbf{N}]\!]_{AVal} = \mathsf{N}$. When a run is represented using a graph, edges labeled with $\langle \emptyset, \emptyset \rangle$ are elided. Also, if the poset labeling an edge is a singleton—that is, it contains only a single ms-atom ℓ—then we display the poset simply as the ms-atom ℓ. For example, the poset on edge $\langle S, F_1 \rangle$ in Figure 2.2 is, when written more explicitly, $\langle \{\langle 1, X : \mathbf{N}, 0 \rangle\}, \emptyset \rangle$. This is a singleton poset—i.e., it is a pair whose first component is a singleton set, and whose second component is the empty partial ordering—so it can be displayed as the ms-atom $\langle 1, X : \mathbf{N}, 0 \rangle$. Since the multiplicity in this ms-atom is 1, and since the tag is 0, according to the notational conventions described in Section 2.2.3, this ms-atom can be written $X : \mathbf{N}$, as it appears in the figure.

Note that the voter's inputs in Figure 2.2 are represented by the same symbolic value. Thus, the voter's inputs are equal, and the voter's output is also represented by that symbolic value. Section 2.3 describes how this run is obtained. Section 2.4.1 defines the notion of a run representing a concrete run; informally, it means that for each edge, the messages in the concrete run can be obtained from the poset of ms-atoms by duplicating each ms-atom some number of times consistent with its multiplicity, linearizing the copies consistently with the partial order, then substituting for each variable a concrete value that is consistent with the abstract values.

2.2.3 Multiplicity

Uncertainty in the number of messages sent during a computation stems from various sources, including:

Non-determinism of components. Faulty components are often non-deterministic.

Non-determinism of message arrival-order. This may cause uncertainty in the number of outputs of a component.

Approximation of values. Approximating a component's inputs may cause uncertainty in the number of its outputs.

Approximation of "loops" of communication. When a set of components send messages back and forth in "loops" of communication, determining whether the computation terminates is, in general, impossible. Thus, determining the number of messages is also, in general, impossible.

Uncertainty in the number of messages is handled in the abstract framework by using *multiplicities*, which are approximations of numbers of messages. For example, a component subject to *send-omission failures* [HT94, Section 2.3], which cause a component to possibly omit the sending of each message normally produced, might emit each output with a multiplicity of either zero or one.

Since multiplicities approximate numbers (of messages), we can represent them with elements of

$$Mul \triangleq \mathcal{P}_{\mathit{fin}}(SVal \times AMul) \setminus \{\emptyset\}, \tag{2.38}$$

where the set of *abstract multiplicities* is

$$AMul = \{a \in AVal \mid [\![a]\!]_{AVal} \in (Set(\mathsf{N}) \setminus \{\emptyset, \{0\}\})\} \tag{2.39}$$

We exclude $\{0\}$ from the possible meanings of abstract multiplicities, because ms-atoms are used to represent messages, and a ms-atom with abstract multiplicity denoting $\{0\}$ would represent no messages.

Abstract Multiplicities. Abstract multiplicities are analogous to the superscripts in regular expressions. Recall, for example, that the regular expression a^* represents sequences of a's of arbitrary length, and the regular expression $a^?$ represents sequences of a's of length 0 or 1. To promote the resemblance between ms-atoms and regular expressions, we assume in examples that $AVal$ contains the following elements with the following meanings:

$$[\![1]\!]_{AVal} = \{1\} \tag{2.40}$$

$$[\![?]\!]_{AVal} = \{0, 1\} \tag{2.41}$$

$$[\![*]\!]_{AVal} = \mathsf{N} \tag{2.42}$$

$$[\![+]\!]_{AVal} = \mathsf{N} \setminus \{0\} \tag{2.43}$$

For example, the outputs of a component subject to send-omission failures might have abstract multiplicity "?".

Symbolic Multiplicities. Symbolic multiplicities track relationships between multiplicities of different messages. They play an important role in the analysis of systems whose fault-tolerance involves atomicity. Atomicity properties are typically of the form: "All non-faulty components do *action*, or none of them do." Such properties correlate multiplicities of actions (e.g., message receptions) at different sites.

For example, the atomicity requirement in reliable broadcast is: for each broadcast message, either every non-faulty process delivers the message, or none do. Since a process may crash before or after sending a particular message, the transmission and hence delivery of a message is not guaranteed. This would be reflected by the abstract multiplicity being ? instead of 1. If two processes each receive a message with abstract multiplicity "?", however, we could not determine whether the atomicity requirement is being satisfied. However, if two processes each receive a message with multiplicity M :?, where M is a variable, then we know that either both processes received the message (i.e., M is interpreted as zero), or neither did (i.e., M is interpreted as one). Reliable broadcast is analyzed in detail in Section 4.1.

Notational Conventions. The notational conventions for values apply to multiplicities as well. To foster the resemblance between ms-atoms and regular expressions, we sometimes write multiplicities as superscripts. This notation is used only if the tag is zero; in other words, we sometimes write the ms-atom $\langle mul, val, 0\rangle$ as val^{mul}. When writing a ms-atom as val^{mul}, we usually elide the multiplicity if it is $\{\langle \_, 1\rangle\} \in Mul$. Thus, the ms-atom $\langle\{\langle \_, 1\rangle\}, val, 0\rangle$ may be written simply as val.

These conventions are used in Figure 2.3, which represents the behavior of the two-stage replicated pipeline when F_1 suffers a Byzantine failure.[7] For example, the ms-atom on edge $\langle F_1, S\rangle$ is, when written more explicitly, $\langle *, \top_V, 0\rangle$; since the tag is zero, this ms-atom is displayed as ${\top_V}^*$. Note that the multiplicity in this ms-atom is, when written more explicitly, $\{\langle \_, *\rangle\}$, representing an arbitrary number of messages. The multiplicity in the ms-atom on edge $\langle S, F_1\rangle$ is $\{\langle \_, 1\rangle\}$, so it is elided. As another example, in Figure 2.2, the multiplicities are all $\{\langle \_, 1\rangle\}$, and the tags are all zero, so the multiplicities and tags are all elided.

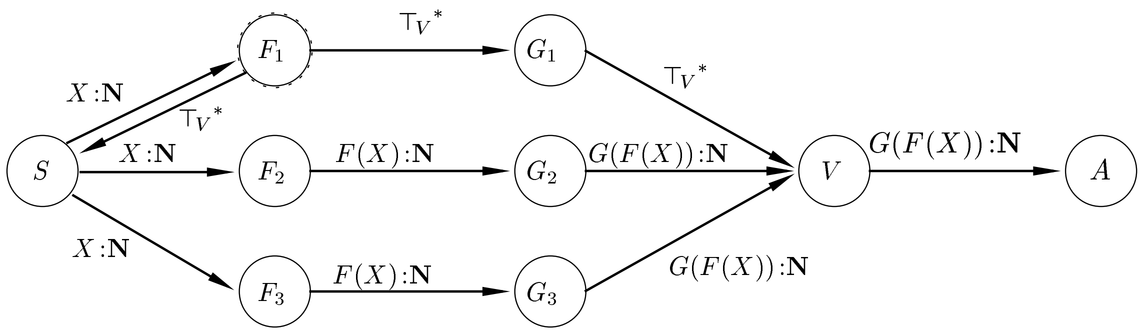

Figure 2.3: Run for two-stage replicated pipeline when F_1 suffers a Byzantine failure.

2.2.4 Tags

Recall that tags are introduced to allow multiple ms-atoms with the same value and multiplicity to appear on an edge. One might ask: "If the values are the same, why not just combine those ms-atoms into a single ms-atom with a 'larger' multiplicity? This would eliminate the need for tags." That approach is indeed possible but sometimes undesirable, since ordering information may be lost when the ms-atoms are merged. For example, consider the

[7]Figure 1.1 represents the same system in the same failure scenario. However, Figure 1.1 is based on the perturbational framework of Chapter 3, while Figure 2.3 uses the non-perturbational framework presented in this chapter.

poset containing the three ms-atoms

$$\begin{aligned} \ell_X^0 &= \langle 1, X\!:\!\mathbf{N}, 0\rangle \\ \ell_Y &= \langle 1, Y\!:\!\mathbf{N}, 0\rangle \\ \ell_X^1 &= \langle 1, X\!:\!\mathbf{N}, 1\rangle, \end{aligned}$$

with the ordering $\ell_X^0 \prec \ell_Y \prec \ell_X^1$. Note that ℓ_X^0 and ℓ_X^1 differ only in their tag. This poset represents sequences of three messages containing natural numbers and such that the first and last messages in the sequence contain the same number. If we merge the two ms-atoms containing X into a single ms-atom, such as $\langle 2, X\!:\!\mathbf{N}, 0\rangle$, we will not be able to express that exactly one occurrence of the number represented by X appears before the occurrence of the number represented by Y.

2.2.5 Message Ordering

The partial ordering in a history (2.28) approximates the orderings between the messages represented by the ms-atoms in that history. Since a history represents the messages transmitted along a single channel (i.e., between a single pair of components), only orderings between messages sent on the same channel are reflected in our representation (2.29) of runs. As in Kahn's model, orderings between messages on different channels are ignored. This simplifies the semantics considerably. The disadvantage is that the behavior of non-strict components sensitive to inter-channel orderings cannot be specified exactly; the output ms-atoms of such a component must represent that component's outputs for each possible interleaving of the inputs from different sources. Some ideas on extending the framework to include inter-channel orderings are mentioned in Chapter 6.

2.3 Representation of Components

By analogy with definition (2.4) of determinate processes, input-output functions have signature[8]

$$IOF \triangleq \{f \in \mathit{Hist} \rightarrow \mathit{Hist} \mid \mathit{tagUniform}(f)\}, \tag{2.44}$$

where $\mathit{tagUniform}(f)$ is a sanity condition requiring that renaming of tags in the argument of f causes no change in the output of f except possibly renaming of tags. This requirement is sensible because tags are artifacts of the formalism; they don't appear in actual messages. To

[8] The analogy is not perfect, since (2.44) contains $\rightarrow$, whereas (2.4) contains $\twoheadrightarrow$. The reasons for this are discussed in Section 2.4.4.

formalize this requirement, we introduce the equality $=_{Hist}$ on *Hist*. Informally, $h_1 =_{Hist} h_2$ means that h_1 and h_2 are the same up to renaming of tags—in other words, that h_1 can be obtained from h_2 by renaming the tags (or *vice versa*). A "renaming of tags" is a function in $Tag \stackrel{\text{inj}}{\to} Tag$, where for any sets S and T, $S \stackrel{\text{inj}}{\to} T$ is the set of injections from S to T. We extend such a function $h \in Tag \stackrel{\text{inj}}{\to} Tag$ to a function $\overline{h} \in L \to L$ by applying h to the tag and leaving the value and multiplicity unchanged:

$$\overline{h}(\langle mul, val, tag\rangle) = \langle mul, val, h(tag)\rangle.$$

Now, equality on $POSet(L)$ up to renaming of tags is:

$$\begin{aligned}\langle S_1, \prec_1\rangle =_{POSet(L)} \langle S_2, \prec_2\rangle \;\triangleq\; & (\exists h \in Tag \stackrel{\text{inj}}{\to} Tag : \\ & \quad \wedge\, S_1 = (\bigcup_{x \in S_2} \{\overline{h}(x)\}) \\ & \quad \wedge \prec_1 = (\bigcup_{\langle x,y\rangle \in \prec_2} \langle \overline{h}(x), \overline{h}(y)\rangle)).\end{aligned}$$

Equality $=_{Hist}$ is the pointwise extension of $=_{POSet(L)}$. With the definition of $=_{Hist}$ in hand, the definition of *tagUniform* is easy:

$$tagUniform(f) \triangleq (\forall in_1, in_2 \in Hist : in_1 =_{Hist} in_2 \;\Rightarrow\; f(in_1) =_{Hist} f(in_2)). \tag{2.45}$$

System Behavior as a Fixed-Point. As in the concrete case, a system's behavior is characterized by a fixed-point. For $nf \in Name \to IOF$, the system's behavior is represented by $\mathrm{lfp}(step(nf))$, if this fixed-point exists. We can always look for a fixed-point by repeated application of $step(nf)$ starting from $\perp_{Run}$, where

$$\perp_{Run} \triangleq (\lambda x\!:\!Name.\ \perp_{Hist}) \tag{2.46}$$

$$\perp_{Hist} \triangleq (\lambda x\!:\!Name.\ \langle\emptyset, \emptyset\rangle). \tag{2.47}$$

A fixed-point r has been found if $step(nf)(r) =_{Run} r$, where the equality $=_{Run}$ on *Run* is the pointwise extension of the equality $=_{Hist}$ on *Hist*.

In contrast to the concrete case, this fixed-point might fail to exist. Section 2.5 discusses the reasons for this and gives additional requirements that ensure existence of a fixed-point.

2.3.1 Notation for Functions

We freely use standard mathematical constructs, such as logical formulas and lambda expressions, in definitions of functions. We also use the following constructs from the functional programming language CAML Light [Ler97], a dialect of Standard ML [MTH90].

Conditionals. The conditional expression **if** b **then** e_1 **else** e_2 has the obvious meaning.

Comments. Comments begin with `(*` and end with `*)`.

Binding. The binding construct is

$$\begin{array}{l}\textbf{let } \mathit{var} = \mathit{expr}_1 \\ \textbf{in } \mathit{expr}_2\end{array} \tag{2.48}$$

where *var* is a variable and expr_1 and expr_2 are expressions. The result of evaluating (2.48) is the result of evaluating expr_2 in a context in which *var* is bound to the result of evaluating expr_1. For example, the Fibonacci function can be written

$$\begin{array}{l} f = (\lambda i{:}\mathbf{N}.\ \textbf{if } i = 0 \vee i = 1 \textbf{ then } 1 \\ \qquad\qquad \textbf{else let } v_1 = f(i-1) \\ \qquad\qquad\qquad \textbf{in let } v_2 = f(i-2) \\ \qquad\qquad\qquad \textbf{in } v_1 + v_2) \end{array}$$

To save horizontal space, we sometimes (as above) do not fully indent sequences of **let** expressions.

Pattern matching. The pattern-matching construct is

$$\begin{array}{l}\textbf{match } \mathit{expr}_0 \textbf{ with} \\ \mid \mathit{patt}_1 \to \mathit{expr}_1 \\ \mid \mathit{patt}_2 \to \mathit{expr}_2 \\ \vdots \\ \mid \mathit{patt}_n \to \mathit{expr}_n\end{array} \tag{2.49}$$

where each expr_i is an expression and each patt_i is a pattern. A pattern is composed of data constructors (e.g., tuple or sequence constructors, or any constant) and variables. This construct evaluates expr_0 and attempts to match the resulting value v against the patterns, in order of appearance. A match occurs if v can be obtained from the pattern by some instantiation of the variables in the pattern; as a special case, the wildcard pattern $\_\_$ matches anything.[9] If pattern patt_i yields the first match, then the variables in patt_i are bound to the values that cause patt_i to equal v, and the result of evaluating (2.49) is the result of evaluating expr_i in a context augmented with these bindings. For example, the following

[9] Note that the wildcard symbolic value is $\_$, while the wildcard pattern is $\_\_$.

function accepts a number or pair of numbers, and returns the given number or the sum of the given pair of numbers, respectively.

$$f = (\lambda x\!:\!(\mathbf{N} \cup (\mathbf{N}\times\mathbf{N})).\ \textbf{match}\ x\ \textbf{with}\\ \mid \langle x_1, x_2\rangle \to x_1 + x_2\\ \mid \_ \to x)$$

2.3.2 Running Example

To illustrate the abstract model, we give input-output functions that represent the processes given in Section 2.1.3 for the replicated pipeline example. An input-output function f represents a process p if, whenever an input history h represents a concrete input history ch, then the output history $f(h)$ represents each possible concrete output history of p on input ch. (The notion of an input-output function representing a process is formalized in Section 2.4.1.)

Definition of ***Src*****.** Source S is represented by input-output function $\mathit{Src}(\{F_1, F_2, F_3\})$, where

$$\mathit{Src}(\mathit{dests}) = (\lambda h\!:\!\mathit{Hist}.\ (\lambda x\!:\!\mathit{Name}.\ \textbf{if}\ x \in \mathit{dests}\ \textbf{then}\ \langle\{\langle 1, X\!:\!\mathbf{N}, 0\rangle\}, \emptyset\rangle\ \textbf{else}\ \langle\emptyset,\emptyset\rangle)), \quad (2.50)$$

with $X \in \mathit{Var}(S)$. We have used some of the notational conventions described in Section 2.2; for example, the multiplicity 1 abbreviates $\{\langle \_, 1\rangle\}$, and the value $X : \mathbf{N}$ abbreviates $\{\langle X, \mathbf{N}\rangle\}$.

Definition of ***Comp*****.** Processors F_i and G_i are represented by appropriate instances of

$$\mathit{Comp}(\mathit{src}, \mathit{dest}, \mathit{op}) = (\lambda h\!:\!\mathit{Hist}.\ (\lambda x\!:\!\mathit{Name}.\ \textbf{if}\ x = \mathit{dest}\ \textbf{then}\ \overline{\mathit{apOp}(\mathit{op}, \mathbf{N})}(h(\mathit{src}))\\ \textbf{else}\ \langle\emptyset,\emptyset\rangle)), \quad (2.51)$$

where for $\mathit{op} \in \mathit{Sym}$, $\mathit{aval} \in \mathit{AVal}$, and $\mathit{val} \in \mathit{Val}$, the value $\mathit{apOp}(\mathit{op}, \mathit{aval})(\mathit{val}) \in \mathit{Val}$ is defined as follows:

- If the abstract value associated with s is aval, then $\mathit{apOp}(\mathit{op}, \mathit{aval})(\mathit{val})$ is value obtained by applying the operator op to each symbolic value s in val (think of aval as the domain and range of op).

- Otherwise, we take the result of applying of op to val to be arbitrary; specifically, $\mathit{apOp}(\mathit{op}, \mathit{aval})(\mathit{val})$ is $\top_V$ (or, written out in full, $\{\langle \_, \top_V\rangle\}$), which represents all concrete values:

$$[\![\top_V]\!]_{\mathit{AVal}} = \mathit{CVal}. \quad (2.52)$$

Thus,

$$apOp(op, aval)(val) = \bigcup_{\langle s,a\rangle \in val} \{\textbf{if } a = aval \textbf{ then } \langle apply(op, \langle\!\langle s \rangle\!\rangle), aval\rangle \\ \textbf{else } \langle \_, \top_V\rangle\}. \tag{2.53}$$

For a symbol $op \in Sym$ and a sequence $\langle\!\langle s_1, \ldots, s_n\rangle\!\rangle \in SVal$, $apply(op, \langle\!\langle s_1, \ldots, s_n\rangle\!\rangle)$ returns the symbolic value $op(s_1, \ldots, s_n)$ if all the s_i are in $SVal_0$; otherwise (i.e., if any s_i is a wildcard), it returns a wildcard.

$\overline{apOp(op, aval)}$ is an extension of $apOp(op, aval)$ from values to posets of ms-atoms; it operates on the values, leaves all multiplicities unchanged, and changes tags if necessary in order to avoid "collisions". For example, if the poset S contains $\langle 1, X : \top_V, 0\rangle$ and $\langle 1, Y : \top_V, 0\rangle$, then a retagging is needed to avoid a "collision", and we have (e.g.)

$$\overline{apOp(F, \mathbf{N})}(\langle S, \emptyset\rangle) = \{\langle 1, \_ : \top_V, 0\rangle, \langle 1, \_ : \top_V, 1\rangle\}.$$

Definition of *Voter***.** The voter is defined in terms of two auxiliary functions: *ballot*, which extracts a vote from a poset of ms-atoms, and *tally*, which uses *ballot* to extract a set of votes from a poset of ms-atoms, then tallies those votes to determine the majority (if any). Extracted votes and outputs of *tally* are both elements of $Mul \times (SVal \times AVal)$, indicating the multiplicity of the vote and the value voted for. The definitions of *ballot* and *tally* are given below. The function representing the voter tests whether it has received some input from each source. If not, it just "waits", i.e., produces no output (represented by the empty poset $\langle \emptyset, \emptyset\rangle$); this corresponds to the **then** branch of the conditional. If it has received some input from each source, it calls *tally*; this corresponds to the **else** branch of the conditional.

$$\begin{aligned} Voter(srcs, dest, aval) = (\lambda h\!:\!Hist.\ (\lambda x\!:\!Name.\ & \\ & \textbf{if } x = dest \textbf{ then} \\ & \quad \textbf{if } (\exists y \in srcs : \pi_1(h(y)) = \emptyset) \textbf{ then } \langle \emptyset, \emptyset\rangle \\ & \quad \textbf{else let } \langle mul, val\rangle = tally(ballot, srcs, aval, h) \\ & \qquad \textbf{in } \langle \{\langle mul, \{val\}, 0\rangle\}, \emptyset\rangle \\ & \textbf{else } \langle \emptyset, \emptyset\rangle)). \end{aligned} \tag{2.54}$$

Definition of *ballot***.** Ballot condenses the input $S \in POSet(L)$ from a component into a single multiplicity and a single element of $SVal \times AVal$. Roughly, if S contains only a single element of $SVal \times AVal$ (i.e., a single value with cardinality one), then *ballot* returns that value and the associated multiplicity. Otherwise, *ballot* uses a coarse approximation: it returns a "top" (i.e., an arbitrary multiplicity and value). It would be easy to make *ballot* more precise, by having it extract all the elements of $SVal \times AVal$ in S (with the associated multiplicities) and return them as a set. This additional precision is not needed to analyze the running example, because non-faulty components do send exactly one value, and it doesn't matter how coarsely inputs from faulty processes are approximated, since those inputs are arbitrary anyway. This approximation in *ballot* reflects our general style in writing input-output functions for examples: we write out the "base cases" (which typically correspond to singleton sets) exactly but don't bother to accumulate sets of possibilities, unless that is necessary to make the analysis sufficiently precise. Of course, whether an input-output function is "sufficiently precise" depends on the histories on which that function will be evaluated and hence on the rest of the system being analyzed.

The definition of *ballot* is, for $S \in POSet(L)$,

$$\begin{aligned} ballot(S) = \mathbf{match}\ &\pi_1(S)\ \mathbf{with} \\ &\mid \{\langle mul, val, tag\rangle\} \to \mathbf{match}\ val\ \mathbf{with} \\ &\qquad\qquad\qquad\qquad \mid \{s : a\} \to \langle mul, s : a\rangle \\ &\qquad\qquad\qquad\qquad \mid \_ \to \text{(* approximate *)} \\ &\qquad\qquad\qquad\qquad\qquad \langle mul, \_ : \top_V\rangle \\ &\mid \_ \to \text{(* approximate *)} \\ &\qquad \langle\{\_ : *\}, \_ : \top_V\rangle \end{aligned} \tag{2.55}$$

Definition of *tally*. The function $tally(ballot, srcs, aval, h)$ extracts and counts ballots cast in history $h \in Hist$ by components named in $srcs \in Seq(Name)$, using the parameter $ballot \in POSet(L) \to (Mul \times (SVal \times AVal))$ to extract the ballots. The remaining parameter $aval \in AVal$ is the "type" of values expected by the voter: if all the ballots have abstract value *aval*, or if there is a majority of equal ballots with abstract value *aval*, then the voter's output has abstract value *aval*; otherwise, the voter's output value may be arbitrary. When the abstract value of the voter's output is *aval*, the symbolic part is determined as follows. Generally, the output is just a (symbolic) application of the operator *maj* to the symbolic values in the ballots. However, if a majority of those symbolic values are the same, then that application can be simplified (this is a form of symbolic computation), yielding just that majority symbolic value. Finally, since we don't allow wildcards inside applications, if the application can't be simplified and one of the ballots contains a wildcard, *tally* just returns a wildcard. The definition of *tally* is

$$
\begin{array}{l}
\mathit{tally}(\mathit{ballot}, \mathit{srcs}, \mathit{aval}, h) = \\
\quad \textbf{let } \mathit{bllts} = (\overline{\mathit{ballot} \circ h})(\mathit{srcs}) \\
\quad \textbf{in let } \mathit{mul} = \textbf{if } (\forall i \in \mathit{dom}(\mathit{bllts}) : \mathit{definite}(\pi_1(\mathit{bllts}[i]))) \textbf{ then } \{\langle \_, 1\rangle\} \textbf{ else } \{\langle \_, ?\rangle\} \\
\quad \textbf{in let } \mathit{val} = \textbf{if } \text{at least } \lceil (|\mathit{srcs}|+1)/2 \rceil \text{ ballots are for some } \langle s, a\rangle \in \mathit{SVal} \times \mathit{AVal} \textbf{ then} \\
\qquad\qquad\qquad \textbf{if } (s = \_) \vee (a \neq \mathit{aval}) \textbf{ then } \langle \_, \top_V\rangle \textbf{ else } \langle s, a\rangle \\
\qquad\qquad \textbf{else if } \overline{\pi_1 \circ \pi_2}(\mathit{bllts}) \in \mathit{Seq}(\mathit{SVal}_0) \wedge \overline{\pi_2 \circ \pi_2}(\mathit{bllts}) \in \mathit{Seq}(\{\mathit{aval}\}) \textbf{ then} \\
\qquad\qquad\qquad\quad \langle \mathit{apply}(\mathit{maj})(\overline{\pi_1 \circ \pi_2}(\mathit{bllts})), \mathit{aval}\rangle \\
\qquad\qquad\qquad \textbf{else } \langle \_, \top_V\rangle \\
\quad \textbf{in } \langle \mathit{mul}, \mathit{val}\rangle
\end{array}
\tag{2.56}
$$

where the overline denotes pointwise extension (to sequences), a multiplicity is *definite* iff it can't denote zero, i.e.,

$$\mathit{definite}(\mathit{mul}) \triangleq (\forall x \in \mathit{mul} : 0 \notin [\![\pi_2(x)]\!]_{\mathit{AVal}}), \tag{2.57}$$

and for a set S, $|S|$ is the size of S. The constant maj has interpretation $\rho_c(\mathit{maj}) = \mathit{maj}$.[10] This definition of *tally* incorporates some easily removable approximations; for example, it ignores symbolic multiplicities.

Definition of *Act*. The actuator is represented by

$$\mathit{Act} = (\lambda h : \mathit{Hist}.\ (\lambda x : \mathit{Name}.\ \langle \emptyset, \emptyset\rangle)). \tag{2.58}$$

2.4 Semantics and Soundness

This section relates the system model described in sections 2.2 and 2.3 to the concrete model presented in Section 2.1. Informally, soundness asserts that a run obtained from the fixed-point analysis represents all possible concrete runs of the system of interest. Soundness allows conditions on a concrete system to be re-cast as conditions on that (abstract) run: if that run satisfies a certain condition, then all concrete runs represented by that run satisfy a related condition, so all concrete runs of the system satisfy that related condition.

For example, suppose we want to check for the replicated pipeline described in Section 2.1.3 that all of the inputs to the voter V are equal. According to the semantics below, all concrete runs represented by a run r satisfy that condition if all input ms-atoms of the voter

[10]As mentioned in Footnote 2, *maj* is overloaded; that's why it appears on both sides of this equation

V in run r together contain only a single symbolic value and that symbolic value is not the wildcard. Thus, the system has this property provided the run r in Figure 2.2 (page 23) satisfies the predicate

$$b(r) = (\lambda r\!:\!Run.\ \mathbf{let}\ S = \cup_{x \in Name} \overline{\pi_1 \circ \pi_2}(\pi_1(r(V)(x))) \\ \mathbf{in}\ |S| = 1 \wedge \_ \notin S),$$

which it does.

2.4.1 Semantics

It is convenient to allow partial interpretations in the semantics. For $S \subseteq Sym$, the set of partial interpretations of S is

$$interp(S) \triangleq S \rightharpoonup CVal, \tag{2.59}$$

where $S \rightharpoonup T$ is the set of partial functions from S to T. For a partial (or total) function f, $dom(f)$ is the domain of f. The ordering on partial interpretations is

$$\rho_1 \leq_{interp} \rho_2 \triangleq dom(\rho_1) \subseteq dom(\rho_2) \wedge (\forall s \in dom(\rho_1) : \rho_1(s) = \rho_2(s)). \tag{2.60}$$

Semantics of Posets of ms-atoms. The semantics of posets of ms-atoms was described informally just below (2.27):

> Informally, a poset $\langle S, \prec \rangle \in POSet(L)$ approximates a sequence σ of messages if there exists a correspondence between elements of S and elements of σ such that: (1) each ms-atom in S approximates the set of corresponding messages, and (2) if $\ell_1 \prec \ell_2$, then all messages corresponding to ℓ_1 precede all messages corresponding to ℓ_2.

The correspondence between the elements of $\langle S, \prec \rangle$ and the elements of σ is embodied in a function $g \in dom(\sigma) \xrightarrow{\text{onto}} S$, where $\xrightarrow{\text{onto}}$ indicates a restriction to surjective (onto) functions.[11] Informally, $g(i)$ is the ms-atom representing the i'th element of σ. We use a predicate $compat_{POSet(L)}$ to check that the correspondence g

(1) respects the values and multiplicities of the ms-atoms; more explicitly: (1a) the concrete value $\sigma[i]$ is represented by the value in $g(i)$, and (1b) the number of elements of σ associated with each ms-atom ℓ is represented by the multiplicity in ℓ;

[11] Thus, $S \xrightarrow{\text{onto}} T$ contains the functions f in $S \to T$ that satisfy $(\forall y \in T : \exists x \in S f(x) = y)$.

(2) respects the ordering on the poset; more explicitly, if $\ell_1 \prec \ell_2$ and $g(i_1) = \ell_1$ and $g(i_2) = \ell_2$, then $i_1 < i_2$.

Note that these two conditions correspond to the two conditions in the informal description.

Condition (1) requires formalizing the notion of a concrete value cv being represented by a value $v \in \mathit{Val}$. This involves two conditions: one based on the symbolic part of v, and one based on the abstract part of v. The condition based on the symbolic part is expressed by extending a given partial interpretation $\rho \in \mathit{interp}(\mathit{Sym})$ of symbols to work on all non-wildcard symbolic values. The extension is done by a recursive definition of the structure of the symbolic value, which corresponds (in a sense) to evaluation of the symbolic value. If, at some point in this "evaluation", an "error" occurs (e.g., the operator in an expression does not denote a function), the evaluation aborts and returns a dummy value $\perp$ (which is required not to be in CVal). For $\rho \in \mathit{interp}(\mathit{Sym})$, the extension $\overline{\rho} \in \mathit{SVal}_0 \rightarrow (\mathit{CVal} \cup \{\perp\})$ is given by

$$\overline{\rho}(s) \;=\; \begin{cases} \rho(s) & \text{if } s \in \mathit{dom}(\rho) \\ \perp & \text{otherwise} \end{cases}$$

$$\overline{\rho}(s(s_1, \ldots, s_n)) \;=\; \begin{cases} \rho(s)(\langle \overline{\rho}(s_1), \ldots, \overline{\rho}(s_n) \rangle) & \text{if } \wedge\, s \in \mathit{dom}(\rho) \\ & \quad \wedge\, \langle \overline{\rho}(s_1), \ldots, \overline{\rho}(s_n) \rangle \in \mathit{dom}_0(\rho(s)) \\ \perp & \text{otherwise} \end{cases}$$

where $\mathit{dom}_0(f)$ is the domain of f, if f is a function, and $\emptyset$ otherwise. We use this extension to define the predicate $\mathit{compat}_{\mathit{Val}}$, which checks whether a concrete value cv is represented by a value $v \in \mathit{Val}$:

$$\mathit{compat}^{\rho}_{\mathit{Val}}(\mathit{val}, cv) \triangleq (\exists \langle s, a \rangle \in \mathit{val} : \;\wedge\, cv \in [\![a]\!]_{\mathit{AVal}} \qquad \wedge\, s = \_ \vee cv = \overline{\rho}(s)). \tag{2.61}$$

The definition of $\mathit{compat}_{\mathit{POSet}(L)}$ has 3 conjuncts, corresponding to conditions (1a), (1b), and (2) above. Given a partial interpretation $\rho \in \mathit{interp}(\mathit{Sym})$ of symbols, poset $\langle S, \prec \rangle \in \mathit{POSet}(L)$ represents sequence $\sigma \in \mathit{Seq}(\mathit{CVal})$ under correspondence $g \in \mathit{dom}(\sigma) \xrightarrow{\text{onto}} S$ iff $\mathit{compat}^{\rho}_{\mathit{POSet}(L)}(S, \prec, \sigma, g)$ holds, where

$$\begin{aligned} \mathit{compat}^{\rho}_{\mathit{POSet}(L)}(S, \prec, \sigma, g) \triangleq \;& \wedge\, (\forall \ell \in S : (\forall i \in g^{\mathit{inv}}(\ell) : \mathit{compat}^{\rho}_{\mathit{Val}}(\pi_2(\ell), \sigma[i]))) \\ & \wedge\, (\forall \ell \in S : \mathit{compat}^{\rho}_{\mathit{Val}}(\pi_1(\ell), |g^{\mathit{inv}}(\ell)|)) \\ & \wedge\, (\forall \langle \ell_1, \ell_2 \rangle \in \prec : g^{\mathit{inv}}(\ell_1) \prec_{\mathit{Set}(\mathsf{N})} g^{\mathit{inv}}(\ell_2)) \end{aligned} \tag{2.62}$$

where $g^{\mathit{inv}}(y)$ is the pre-image of y under g, i.e.,

$$g^{\mathit{inv}}(y) \triangleq \{x \in \mathit{dom}(g) \mid g(x) = y\} \tag{2.63}$$

and $\prec_{Set(\mathsf{N})}$ is a strict partial order on sets of natural numbers:

$$S_1 \prec_{Set(\mathsf{N})} S_2 \triangleq (\forall i_1 \in S_1 : (\forall i_2 \in S_2 : i_1 < i_2)). \tag{2.64}$$

The meaning of a poset $\langle S, \prec\rangle \in POSet(L)$ is the set of sequences of concrete values that it represents:

$$[\![\langle S, \prec\rangle]\!]^{\rho}_{POSet(L)} \triangleq \{\sigma \in Seq(CVal) \mid (\exists g \in dom(\sigma) \xrightarrow{\text{onto}} S : compat^{\rho}_{POSet(L)}(S, \prec, \sigma, g))\}, \tag{2.65}$$

Note that if $\overline{\rho}(s) = \bot$, then the condition $cv = \overline{\rho}(s)$ in $compat^{\rho}_{Val}$ cannot hold, so pairs containing s in a value are effectively ignored. Thus, increasing a partial interpretation ρ (with respect to $\leq_{interp}$) can only increase $[\![\langle S, \prec\rangle]\!]^{\rho}_{POSet(L)}$ (with respect to $\subseteq$); in other words, $[\![]\!]^{\rho}_{POSet(L)}$ is monotonic in ρ.

Recall that tags in ms-atoms are just a technical device to allow multiple ms-atoms with the same value and multiplicity to appear in a set. Thus, renaming tags should have no effect on the meaning of a poset of ms-atoms. It is easy to check that $[\![]\!]_{POSet(L)}$ is independent of tags, i.e.,

$$\begin{gathered}(\forall \rho \in interp(Sym) : (\forall S_1 \in POSet(L) : (\forall S_2 \in POSet(L) : \\ S_1 =_{POSet(L)} S_2 \Rightarrow [\![S_1]\!]^{\rho}_{POSet(L)} = [\![S_2]\!]^{\rho}_{POSet(L)}))).\end{gathered}$$

Semantics of Histories. The meaning of histories is a straightforward extension of $[\![]\!]_{POSet(L)}$. For $\rho \in interp(Sym)$,

$$[\![h]\!]^{\rho}_{Hist} \triangleq \{ch \in CHist \mid (\forall x \in Name : ch(x) \in [\![h(x)]\!]^{\rho}_{POSet(L)})\}. \tag{2.66}$$

Note that $[\![]\!]^{\rho}_{Hist}$ is monotonic in ρ, i.e.,

$$\begin{gathered}(\forall \rho_1, \rho_2 \in interp(Sym) : (\forall h \in Hist : \\ \rho_1 \leq_{interp} \rho_2 \Rightarrow [\![h]\!]^{\rho_1}_{Hist} \subseteq [\![h]\!]^{\rho_2}_{Hist})).\end{gathered} \tag{2.67}$$

Semantics of Input-Output Functions. Informally, an input-output function f represents a process p if, whenever an input history $in \in Hist$ represents the concrete input history of p, then $f(in)$ represents the concrete output history of p. The values of the local variables $lvar$ of the component can be chosen freely to "match" the concrete values in a sequence, while the values of other variables may already be constrained. The values chosen for the local variables in $f(in)$ can depend on the inputs to the process. However, to ensure soundness, this dependence must be monotonic, i.e., additional inputs can't change the values chosen for those local variables. In other words, as more inputs are received, the values of more local variables are determined, and those values are unchanged when yet more inputs are received. Formally, the dependence of the values of local variables on the concrete inputs is captured in a monotonic and continuous function $g \in \mathit{CHist} \twoheadrightarrow \mathit{interp}(\mathit{lvar})$. For $p \in \mathit{Process}$, $\rho_c \in \mathit{Interp}(\mathit{Con})$, $\mathit{lvar} \subseteq \mathit{Var}$, and $f \in \mathit{IOF}$,

$$\begin{aligned} p \sqsubset_{\mathit{IOF}}^{\rho_c,\mathit{lvar}} f \triangleq {} & (\forall \langle dp, ir\rangle \in p : (\exists g \in \mathit{CHist} \twoheadrightarrow \mathit{interp}(\mathit{lvar}) : \\ & (\forall \rho_e \in \mathit{Interp}(\mathit{Var} \setminus \mathit{lvar}) : (\forall \mathit{in} \in \mathit{Hist} : (\forall \mathit{ch} \in \mathit{ir} : \\ & \mathit{ch} \in [\![\mathit{in}]\!]_{\mathit{Hist}}^{\rho_c \cup \rho_e \cup g(\mathit{ch})} \Rightarrow dp(\mathit{ch}) \in [\![f(\mathit{in})]\!]_{\mathit{Hist}}^{\rho_c \cup \rho_e \cup g(\mathit{ch})}))))). \end{aligned} \tag{2.68}$$

Note that we use union to combine functions with disjoint domains (if functions are regarded as sets of pairs, this does not even require overloading $\cup$). If $p \sqsubset_{\mathit{IOF}}^{\rho_c,\mathit{lvar}} f$, we say f *represents* p.

Semantics of Systems. An abstract system comprises a mapping $\mathit{nf} \in \mathit{Name} \to \mathit{IOF}$ and a partial interpretation $\rho_a \in \mathit{interp}(\mathit{Con})$ of constants. The partial interpretation allows the user to specify, for example, that *maj* represents a majority function, or that the constant *encrypt* represents DES encryption. The meaning of systems is essentially a pointwise extension of the meaning of input-output functions: for $\mathit{np} \in \mathit{Name} \to \mathit{Process}$, $\mathit{nf} \in \mathit{Name} \to \mathit{IOF}$, and $\rho_a \in \mathit{interp}(\mathit{Con})$,

$$\mathit{np} \sqsubset_{\mathit{Sys}} \langle \mathit{nf}, \rho_a\rangle \triangleq (\exists \rho_c \in \mathit{Interp}(\mathit{Con}, \rho_a) : (\forall x \in \mathit{Name} : \mathit{np}(x) \sqsubset_{\mathit{IOF}}^{\rho_c, \mathit{Var}(x)} \mathit{nf}(x))), \tag{2.69}$$

where the extensions of a partial interpretation $\rho \in \mathit{interp}(S)$ are

$$\mathit{Interp}(S, \rho) \triangleq \{\rho_1 \in \mathit{Interp}(S) \mid \rho \leq_{\mathit{interp}} \rho_1\}. \tag{2.70}$$

If $\mathit{np} \sqsubset_{\mathit{Sys}} \langle \mathit{nf}, \rho_a\rangle$, we say the abstract system $\langle \mathit{nf}, \rho_a\rangle$ *represents* the concrete system *np*. To reduce clutter, we have left *Var* implicit in the notation for abstract systems.

Semantics of Runs. For $r \in \mathit{Run}$ and $\rho_a \in \mathit{interp}(\mathit{Con})$,

$$\begin{aligned} [\![r]\!]_{\mathit{Run}}^{\rho_a} \triangleq {} & \{\mathit{cr} \in \mathit{CRun} \mid (\exists \rho_c \in \mathit{Interp}(\mathit{Con}, \rho_a) : (\exists \rho_v \in \mathit{Interp}(\mathit{Var}) : \\ & (\forall x \in \mathit{Name} : \mathit{cr}(x) \in [\![r(x)]\!]_{\mathit{Hist}}^{\rho_c \cup \rho_v})))\}. \end{aligned} \tag{2.71}$$

The existential quantification over ρ_v reflects the intuition that the interpretations of the variables in a run can be chosen freely to "match" the concrete values.

2.4.2 Soundness

Although fixed-points are not, in general, guaranteed to exist, if iteration of $step(nf)$ does lead to a fixed-point r, soundness means that r represents all possible finite behaviors of the concrete system, i.e., that $cruns^{fin}(np) \subseteq [\![r]\!]^{\rho_a}_{Run}$, where

$$cruns^{fin}(np) = \{cr \in cruns(np) \mid (\forall x \in Name : (\forall y \in Name : |cr(y)(x)| < \omega))\}, \quad (2.72)$$

The restriction to finite behaviors is only for convenience; the framework and semantics can be extended to deal with infinite behaviors as well.

Soundness is expressed by the following theorem:

Theorem 2.2. For all $np \in Name \to Process$, all $nf \in Name \to IOF$, all $\rho_a \in interp(Con)$, and all $i_{fp} \in \mathsf{N}$, if $np \sqsubset_{Sys} \langle nf, \rho_a \rangle$, and if $r = step(nf)^{i_{fp}}(\bot_{Run})$ is a fixed-point of $step(nf)$, then all finite runs of the system are represented by r, i.e., $cruns^{fin}(np) \subseteq [\![r]\!]^{\rho_a}_{Run}$.

Proof: Let $\rho_c \in Interp(Con, \rho_a)$ witness the existential quantification in $np \sqsubset_{Sys} \langle nf, \rho_a \rangle$. Consider any $cr_0 \in cruns^{fin}(np)$. By definition (2.14) of $cruns$, there exists $h \in Name \to IRProcess$ such that

$$\begin{aligned} &\wedge \ (\forall x \in Name : h(x) \in np(x)) \\ &\wedge \ (\forall x \in Name : enabled(h(x), \langle\!\langle step(\pi_1 \circ h)^i(\bot_{CRun})(x) \rangle\!\rangle_{i \in \mathsf{N}})) \\ &\wedge \ cr_0 = \mathrm{lfp}(step(\pi_1 \circ h)). \end{aligned}$$

Let $cr[i] = step(\pi_1 \circ h)^i(\bot_{CRun})$ and $r[i] = step(nf)^i(\bot_{Run})$. We show by induction that

$$(\forall i \in \mathsf{N} : (\forall x \in Name : cr[i](x) \in [\![r[i](x)]\!]^{\rho_c \cup \rho_v[i]}_{Hist})), \quad (2.73)$$

where $\rho_v[i] = \cup_{x \in Name} g(x)(cr[i](x))$, where for all x, $g(x) \in CHist \to interp(Var(x))$ is a witness for the existential quantification in $np(x) \sqsubset^{\rho_c, Var(x)}_{IOF} nf(x)$ when the outermost universal quantification there is instantiated with $\langle dp, ir \rangle = h(x)$.

Base Case. For $i = 0$, the claim is that $(\forall x \in Name : \bot_{CRun}(x) \in [\![\bot_{Run}(x)]\!]^{\rho_c \cup \rho_v[0]}_{Hist})$, which follows easily from the definitions.

Step Case. Using the induction hypothesis as the antecedent of the implication in definition (2.68) of $\sqsubset^{\rho_c, Var(x)}_{IOF}$, we get

$$(\forall x \in Name : \pi_1(h(x))(cr[i](x)) \in [\![nf(x)(r[i](x))]\!]^{\rho_c \cup \rho_v[i]}_{Hist}). \quad (2.74)$$

Monotonicity of all the $\pi_1(h(x))$ implies $cr[i] \leq_{CRun} cr[i+1]$. Monotonicity of all the $g(x)$ then implies $\rho_v[i] \leq_{interp} \rho_v[i+1]$. So, by monotonicity of $[\![]\!]^{\rho}_{Hist}$ in ρ (from (2.67)), (2.74) still holds if $\rho_v[i]$ is replaced with $\rho_v[i+1]$. From the resulting equation and definition (2.6) of *step*, we get $(\forall x \in \mathit{Name} : cr[i+1](x) \in [\![r[i+1](x)]\!]^{\rho_c \cup \rho_v[i+1]}_{Hist})$. This completes the proof of (2.73).

Finally, we show that (2.73) implies $cr_0 \in [\![r]\!]^{\rho_c}_{Run}$. Since cr_0 is finite, there exists $i_0 \in \mathsf{N}$ such that $(\forall i \geq i_0 : cr_0 = cr[i])$. The desired result is obtained by instantiating the universal quantification in (2.73) with $i = \max(i_{fp}, i_0)$. ∎

2.4.3 Invariants

In writing and verifying input-output functions, it is sometimes convenient to introduce assumptions about the values of variables. These assumptions are embodied in an invariant that has signature

$$\mathit{Invar} \triangleq \{I \in \mathit{Name} \to \mathit{Set}(\mathit{interp}(\mathit{Var})) \mid (\forall x \in \mathit{Name} : I(x) \subseteq \mathit{interp}(\mathit{Var}(x)))\}. \quad (2.75)$$

For $I \in \mathit{Invar}$ and $x \in \mathit{Name}$, $I(x)$ contains the partial interpretations of $\mathit{Var}(x)$ that satisfy the invariant. Note that this definition of *Invar* excludes correlations between values of variables local to different components; otherwise, $\sqsubset_{IOF}$ could not be verified independently for different components.

To accommodate invariants in the semantics, we just restrict quantifications over interpretations of variables to being with respect to the invariant. The changes are as follows. For $I_l \subseteq \mathit{interp}(\mathit{lvar})$ and $I_e \subseteq \mathit{Interp}(\mathit{Var} \setminus \mathit{lvar})$, define $p \sqsubset^{\rho_c, \mathit{lvar}, I_l, I_e}_{IOF} f$ as in (2.68), but with $\mathit{interp}(\mathit{lvar})$ replaced with I_l, and $\mathit{Interp}(\mathit{Var} \setminus \mathit{lvar})$ replaced with I_e. Extend an abstract system to include an invariant as the third component of the tuple. For $I \in \mathit{Invar}$, define $np \sqsubset_{Sys} \langle nf, \rho_a, I\rangle$ as in (2.69), but with $\sqsubset^{\rho_c, \mathit{Var}(x)}_{IOF}$ replaced with $\sqsubset^{\rho_c, \mathit{Var}(x), I(x), I \downarrow (\mathit{Name} \setminus \{x\})}_{IOF}$, where for $S \subseteq \mathit{Name}$, the restriction of I to S is

$$I \downarrow S = \{\rho \in \mathit{Interp}(\cup_{x \in S} \mathit{Var}(x)) \mid (\forall y \in S : (\lambda v \colon \mathit{Var}(y).\ \rho(v)) \in I(y))\}. \quad (2.76)$$

For $I \in \mathit{Invar}$, define $[\![r]\!]^{\rho_a, I}_{Run}$ as in (2.71), but with $\mathit{Interp}(\mathit{Var})$ replaced with $I \downarrow \mathit{Name}$. Modifying the proof of soundness to accommodate invariants is straightforward.

2.4.4 Sanity Conditions for Input-Output Functions

As mentioned in Section 2.1, monotonicity and continuity are typically required of input-output functions in stream-processing models, in order to eliminate from consideration input-output functions that don't represent any process. We discuss these two conditions in turn.

Monotonicity. Monotonicity of input-output functions is defined below (see (2.82)). Informally, a history in *Hist* represents a set of concrete histories. Moving up in the ordering on *Hist* corresponds to either extending some of those concrete histories (i.e., replacing them with concrete histories that are larger with respect to $\leq_{CHist}$) or adding "new" concrete histories (i.e., concrete histories that aren't extensions of old ones). The former aspect of $\leq_{Hist}$ corresponds directly to the prefix orderings typical of stream-processing models. The latter aspect is analogous to orderings used in abstract interpretation. Typically, it corresponds to an increase in the set of possible behaviors at some point in the program (e.g., to an increase in the set of possible values that may be sent by some **send** statement). It is interesting to note that this also corresponds (albeit indirectly) to further execution of the system. For example, consider analysis of a pair of processes that repeatedly reply to each other. As the system continues to execute, the processes send additional messages to each other, so the set of possible values in the messages they have exchanged (taken collectively) increases.

Monotonicity of input-output functions can be interpreted as the following two sanity conditions, corresponding to the two aspects of the ordering:

1. providing additional inputs to a component can't cause the component to produce fewer outputs;

2. enlarging the set of possible inputs of a component can't cause the set of possible outputs of the component to shrink.

There is no technical difficulty in augmenting the definition of *IOF* with monotonicity requirement (2.82), though this does require parameterizing the definition of *IOF* by *lvar* and ρ_a. We omit this requirement for two reasons. First, since we assume that the input-output functions used in an analysis have been shown to represent the appropriate processes, (other) sanity conditions are otiose. Second, this requirement can complicate input-output functions by forcing "uniform" use of approximations. An input-output function might fail to be monotonic simply because finer approximations are used on larger inputs. Assuming the finer approximation is needed on large inputs, a monotonicity requirement would force finer approximations to be used on smaller inputs as well. However, assuming the input-output function represents the process of interest, there is no compelling reason to require this.

Continuity. Continuity is defined only for functions between ω-cpos. As discussed in Section 2.5.4, *Hist* and *Run* are not ω-cpos, unless additional conditions are imposed, so in general, it is not sensible to require continuity of input-output functions. The conditions in Section 2.5 that ensure existence of fixed-points also trivially ensure continuity of input-output functions.

2.5 Termination of Fixed-Point Calculations

Since our goal is automated analysis, of interest are conditions under which the fixed-point can be computed in a finite number of steps. Of course, when analyzing any particular system, one can seek a fixed-point without knowing whether one exists, by iterating $step(nf)$ until either a fixed-point is found (i.e., applying $step(nf)$ again has no effect, except possibly renaming of tags) or computational resources are exhausted.

It is more satisfying to know before starting the computation whether this iteration will terminate with a fixed-point. To help state the relevant conditions, we define an *ascending* chain of a partial order $\langle S, \leq_S\rangle$ to be a chain $\sigma \in Chain(\langle S, \leq_S\rangle)$ in which no two consecutive elements are equal, i.e., $(\forall i \in (dom(\sigma) \setminus \{0\}) : \sigma[i-1] \neq \sigma[i])$. It follows from antisymmetry of the partial order that all elements of an ascending chain are distinct. The basic observation is that in a partial order with no infinite ascending chains, fixed-points can be computed in a finite number of steps. To see this, let S and f be as in Theorem 2.1. If S has no infinite ascending chains, then the chain $\langle\!\langle f(i)\rangle\!\rangle_{i\in\mathsf{N}}$ converges to the least fixed point in a finite number of steps.

If we assume that S has no infinite ascending chains, then some of the other hypotheses of Theorem 2.1 can be weakened. In a partial order with no infinite ascending chains, ω-chains trivially have least upper bounds (to see this, note that an ω-chain can contain only a finite number of distinct elements, so the largest of these is the least upper bound of the chain). Similarly, monotonic functions are trivially continuous (to see this, note that, as in the previous remark, $\mathrm{lub}(\sigma) = x_{max}$, where x_{max} is the largest element in σ, so $f(\mathrm{lub}(\sigma)) = f(x_{max})$; monotonicity of f implies that $f(x_{max})$ is the largest element in the ω-chain $f(\sigma)$, so $\mathrm{lub}(f(\sigma)) = f(x_{max}) = f(\mathrm{lub}(\sigma))$, as desired). Thus, we have the following corollary of Theorem 2.1.

Corollary 2.3. Let $\langle S, \leq_S\rangle$ be a partial order with no infinite ascending chains. Let f be a monotonic function in $S \to S$, and let $x \in S$ be such that $x \leq_S f(x)$. Then f has a fixed-point in the upper-closure of x in S. Furthermore, the least such fixed-point is $f^i(x)$, where i is any natural number satisfying $f^i(x) = f^{i+1}(x)$.

In order to apply Corollary 2.3 to ensure termination of the fixed-point calculation for $\mathit{step}(\mathit{nf})$, we define a partial order $\leq_{\mathit{Run}}$ on Run and find conditions under which:

1. $\mathit{step}(\mathit{nf})$ is monotonic;

2. $\bot_{\mathit{Run}} \leq_{\mathit{Run}} \mathit{step}(\mathit{nf})(\bot_{\mathit{Run}})$;

3. $\langle \mathit{Run}, \leq_{\mathit{Run}} \rangle$ has no infinite ascending chains.

Since *step* is defined in terms of input-output functions, we discharge the first obligation by defining orderings on *Hist* and showing that monotonicity of input-output functions with respect to these orderings implies monotonicity of *step* with respect to $\leq_{\mathit{Run}}$. To discharge the second obligation, we show that $\bot_{\mathit{Run}} \leq_{\mathit{Run}} \mathit{step}(\mathit{nf})(\mathit{Run})$ holds given an additional technical assumption about the input-output functions. To discharge the third obligation, we give conditions under which *Run* has no infinite ascending chains. Finally, we discuss why *Run* is not in general ω-complete.

2.5.1 Monotonicity of *step*

The partial ordering on *Run* is defined in terms of partial orderings on *Hist*, which are, in turn, defined in terms of a partial ordering on $\mathit{POSet}(L)$.

Ordering on Posets of ms-atoms. Roughly, the partial ordering on $\mathit{POSet}(L)$ is: $\langle S_1, \prec_1 \rangle$ is less than $\langle S_2, \prec_2 \rangle$ if each sequence in $[\![\langle S_1, \prec_1 \rangle]\!]_{\mathit{POSet}(L)}$ is a prefix of a sequence in $[\![\langle S_2, \prec_2 \rangle]\!]_{\mathit{POSet}(L)}$. To make this precise, we need to quantify appropriately over interpretations of the variables. When interpreting the output ms-atoms of a component x, the values of its local variables $\mathit{Var}(x)$ can be chosen freely, while the values of other variables may already be constrained. So, given an interpretation $\rho_c \in \mathit{Interp}(\mathit{Con})$ of constants, and a set *lvar* of local variables, we define a pre-order

$$S_1 \preceq^{\rho_c, \mathit{lvar}}_{\mathit{POSet}(L)} S_2 \triangleq (\forall \rho_v \in \mathit{Interp}(\mathit{Var}) : (\exists \rho_l \in \mathit{Interp}(\mathit{lvar}) : [\![S_1]\!]^{\rho_c \cup \rho_v}_{\mathit{POSet}(L)} \preceq_{\mathit{Set}(\mathit{Seq})} [\![S_2]\!]^{\rho_c \cup (\rho_v \oplus \rho_l)}_{\mathit{POSet}(L)})), \tag{2.77}$$

where the pre-order $\preceq_{\mathit{Set}(\mathit{Seq})}$ on sets of sequences is

$$S_1 \preceq_{\mathit{Set}(\mathit{Seq})} S_2 \triangleq (\forall \sigma_1 \in S_1 : (\exists \sigma_2 \in S_2 : \sigma_1 \leq_{\mathit{Seq}} \sigma_2)), \tag{2.78}$$

and where for functions f and g, $(f \oplus g)$ is f updated with g, i.e., $(f \oplus g)(x)$ is $g(x)$ for $x \in dom(g)$, and is $f(x)$ otherwise.

It is easy to check that $\preceq^{\rho_c, lvar}_{POSet(L)}$ is, in fact, a pre-order, i.e., that it is reflexive and transitive. It is not a partial order because it lacks antisymmetry. This lack of antisymmetry has two causes: first, the pre-order $\prec_{Set(Seq)}$ is not antisymmetric; second, *Run* contains distinct runs with the same meaning, as discussed in Section 2.5.4. We construct a partial ordering by:

$$\begin{aligned} S_1 \leq^{\rho_c, lvar}_{POSet(L)} S_2 \triangleq\ & \vee (S_1 \preceq^{\rho_c, lvar}_{POSet(L)} S_2) \wedge (S_2 \not\preceq^{\rho_c, lvar}_{POSet(L)} S_1) \\ & \vee S_1 =_{POSet(L)} S_2. \end{aligned} \tag{2.79}$$

The first disjunct is a strict partial ordering; the second disjunct makes the ordering reflexive. It is easy to check that $\leq^{\rho_c, lvar}_{POSet(L)}$, like $[\![]\!]^{\rho}_{POSet(L)}$, is independent of tags. It is easy to check that $\leq_{POSet(L)}$ is a partial order on $POSet(L)$ quotiented by $=_{POSet(L)}$, i.e., on $POSet(L)$ with elements related by $=_{POSet(L)}$ considered equivalent. Informally, the construction of $\leq_{POSet(L)}$ ensures antisymmetry by removing orderings in $\preceq_{POSet(L)}$ between semantically equivalent posets and between posets whose meanings are related in both directions by $\leq_{Set(Seq)}$ (e.g., if one is the prefix-closure of the other).

Orderings on Histories. The local variables in a set of ms-atoms are always the variables associated with the sender, so the ordering on histories depends on whether the histories are regarded as input histories or output histories. For histories regarded as inputs,

$$h_1 \leq^{\rho_c}_{InHist} h_2 \triangleq (\forall x \in Name : h_1(x) \leq^{\rho_c, Var(x)}_{POSet(L)} h_2(x)). \tag{2.80}$$

For histories regarded as output of a component with local variables *lvar*,

$$h_1 \leq^{\rho_c, lvar}_{OutHist} h_2 \triangleq (\forall x \in Name : h_1(x) \leq^{\rho_c, lvar}_{POSet(L)} h_2(x)). \tag{2.81}$$

Thus, an input-output function f is defined to be monotonic with respect to $lvar \subseteq Var$ (intuitively, these are the local variables of the component f represents) and $\rho_a \in interp(Con)$ iff

$$\begin{aligned} & (\forall \rho_c \in Interp(Con, \rho_a) : (\forall h_1 \in Hist : (\forall h_2 \in Hist : \\ & \quad h_1 \leq^{\rho_c}_{InHist} h_2 \Rightarrow f(h_1) \leq^{\rho_c, lvar}_{OutHist} f(h_2)))). \end{aligned} \tag{2.82}$$

Ordering on Runs. The ordering on runs is just a pointwise extension of the ordering on histories regarded as inputs:

$$r_1 \leq^{\rho_a}_{Run} r_2 \triangleq (\forall \rho_c \in Interp(Con, \rho_a) : (\forall y \in Name : r_1(y) \leq^{\rho_c}_{InHist} r_2(y))). \tag{2.83}$$

Monotonicity of *step*.

Theorem 2.4. For all $nf \in Name \to IOF$ and all $\rho_a \in interp(Con)$, if for all $x \in Name$, $nf(x)$ is monotonic with respect to $Var(x)$ and ρ_a, then $step(nf)$ is monotonic with respect to $\leq_{Run}^{\rho_a}$.

Proof: Assume $r_1 \leq_{Run}^{\rho_a} r_2$. Using monotonicity of $nf(x)$, then expanding the definition of $\leq_{OutHist}^{\rho_c, Var(x)}$ and recognizing that $step(nf)(r)(y)(x) = nf(x)(r(x))(y)$, we have

$$\begin{aligned}&(\forall \rho_c \in Interp(\rho_a) : (\forall x \in Name : (\forall y \in Name :\\ &\qquad step(nf)(r_1)(y)(x) \leq_{POSet(L)}^{\rho_c, Var(x)} step(nf)(r_2)(y)(x)))),\end{aligned}$$

which is equivalent to $step(nf)(r_1) \leq_{Run}^{\rho_a} step(nf)(r_2)$. ∎

Semantic Ordering vs. Syntactic Ordering. To obtain the most general orderings on histories, we have defined the orderings directly in terms of the semantics. A more "syntactic" ordering might be more convenient for verifying monotonicity of particular functions. However, such characterizations are more restrictive, hence not as widely applicable. For example, it seems difficult to formulate a general "syntactic" ordering with respect to which the input-output functions used in analysis of reliable broadcast in Chapter 4 are monotonic, because they wouldn't be monotonic if max were replaced with min in their definitions.

2.5.2 The First Step

To show that the first application of step yields a larger run, i.e., that $\bot_{Run} \leq_{Run}^{\rho_a} step(nf)(\bot_{Run})$, we need an additional assumption about input-output functions. This is necessary because $\bot_{Run}$ is not a least element in $\langle Run, \leq_{Run}\rangle$. To see this, note that $[\![\bot_{Run}]\!]_{Run}^{\rho_a} = \bot_{CRun}$, so any "meaningless" run (i.e., any run r such that $[\![r]\!]_{Run}^{\rho_a} = \emptyset$) is less than $\bot_{Run}$.

As in the definition of the strict part of $\leq_{POSet(L)}$, it is necessary here to ensure that semantically equivalent posets of ms-atoms are not substituted for each other. When we start the fixed-point calculation with $\bot_{CRun}$ represented by $\bot_{Run}$, the empty sequence of messages on each edge is represented by the empty poset. If, after one step, some edges still have the empty sequence of messages on them, then we require that those empty sequences still be represented by the empty poset, rather than some semantically equivalent poset. More precisely, if a process p has no initial outputs to a component y, then we require that $f(\bot_{Hist})(y) = \langle \emptyset, \emptyset \rangle$. Formally, we augment the definition (2.68) of $p \sqsubset_{IOF}^{\rho_c} f$ with the conjunct

$$(\forall y \in Name : noInitialOut(p, y) \Rightarrow f(\bot_{Hist})(y) = \langle \emptyset, \emptyset \rangle), \qquad (2.84)$$

where

$$noInitialOut(p, y) \triangleq (\forall \langle dp, ir \rangle \in p : \bot_{CHist} \in ir \Rightarrow dp(\bot_{CHist})(y) = \varepsilon). \tag{2.85}$$

This suffices to establish the following theorem.

Theorem 2.5. For all $np \in Name \to Process$, all $nf \in Name \to IOF$, and all $\rho_a \in interp(Con)$, if $np \sqsubset_{Sys} \langle nf, \rho_a \rangle$, then $\bot_{Run} \leq_{Run}^{\rho_a} step(nf)(\bot_{Run})$.

Proof: Expanding the definitions of $step(nf)$ and $\leq_{Run}$, we need to show that for all $\rho_c \in Interp(Con, \rho_a)$,

$$\begin{aligned}(\forall x \in Name : (\forall y \in Name : \;\; \vee \;\; & nf(x)(\bot_{Hist})(y) =_{Hist} \langle \emptyset, \emptyset \rangle \\ \vee \;\; & \wedge \;\; (\forall \rho_v \in Interp(Var) : (\exists \rho_l \in Interp(Var(x)) : \\ & \qquad \{\varepsilon\} \preceq_{Set(Seq)} [\![nf(x)(\bot_{Hist})(y)]\!]_{POSet(L)}^{\rho_c \cup (\rho_v \oplus \rho_l)})) \\ & \wedge \;\; (\exists \rho_v \in Interp(Var) : \\ & \qquad [\![nf(x)(\bot_{Hist})(y)]\!]_{POSet(L)}^{\rho_c \cup \rho_v} \not\preceq_{Set(Seq)} \{\varepsilon\})\end{aligned} \tag{2.86}$$

Suppose p might initially send a message to y, i.e., there exists $\langle dp, ir \rangle \in p$ such that $\bot_{Hist} \in ir \wedge dp(\bot_{CHist})(y) \neq \varepsilon$. Then by definition of $\sqsubset_{IOF}^{\rho_c, Var(x)}$,

$$\begin{aligned}(\exists \rho_l \in Interp(Var(x)) : (\forall \rho_e \in Interp(Var \setminus Var(x)) : \\ dp(\bot_{CHist})(y) \in [\![nf(x)(\bot_{Hist})(y)]\!]_{POSet(L)}^{\rho_c \cup \rho_e \cup \rho_l})).\end{aligned}$$

It is easy to check that this implies the second disjunct in (2.86). Otherwise—that is, if p does not initially send a message to y—the antecedent in (2.84) holds, so the first disjunct in (2.86) holds. ∎

2.5.3 Finite Ascending Chains

We give below a simple set of conditions that ensures *Run* has no infinite ascending chains. We also give corresponding conditions on input-output functions that ensure the necessary closure property, namely, that application of $step(nf)$ to a run satisfying those conditions yields a run that also satisfies those conditions. If the input-output functions for a system nf satisfy these corresponding conditions, then the fixed-point iteration for that system is guaranteed to terminate.

The conditions on *Run*, and hence the corresponding conditions on the input-output functions, are stronger than necessary. These conditions do hold for the input-output functions used in the running example. For classes of systems for which these conditions are too restrictive, more flexible (but probably more complicated) conditions could be used to establish termination of the fixed-point iteration (provided it does terminate).

Restrictions on Runs. All ascending chains in *Run* are finite iff all ascending chains in *POSet*(L) are finite, so we look at conditions for ensuring the latter. One set of sufficient conditions is as follows. For each $n > 0$, let FAC_n be the subset of *POSet*(L) that contains an element $\langle S, \prec_S \rangle$ iff

1. the size of S is at most n;

2. the size of each element of *Mul* and *Val* occurring in (a ms-atom in) S is at most n. Note that the size of (say) a multiplicity is its size (cardinality) as a set.

Furthermore, we require that *AVal* have size at most n. These conditions together ensure FAC_n has only finite ascending chains. Let $Run_n = Name \to (Name \to FAC_n)$.

One might think that the following weaker condition on *AVal* is sufficient: require that *AVal* have no infinite ascending chains with respect to the ordering $a_1 \leq_{AVal} a_2 \triangleq [\![a_1]\!]_{AVal} \subseteq [\![a_2]\!]_{AVal}$. However, this condition is too weak; roughly, it deals only with *singleton* sets of abstract values, not with larger sets of abstract values. For example, suppose $AVal = \cup_{i \in \mathsf{N}} \{x_i^0, x_i^1\}$, with the meanings defined by recursion on i: for $\alpha \in \{0, 1\}$, $[\![x_0^\alpha]\!]_{AVal} = \{\alpha\}$ and $\left[\!\left[x_{i+1}^\alpha\right]\!\right]_{AVal} = x_i^\alpha \cup \{\max(x_i^\alpha) + 2, \max(x_i^{\alpha+1 \bmod 2}\} \setminus \{\max(x_i^\alpha)\}$. It is easy to show that *AVal* has no infinite ascending chains but *POSet*(L) does.

Restrictions on Input-Output Functions. To ensure that

$$r \in Run_n \Rightarrow step(nf)(r) \in Run_n,$$

it suffices to require that each input-output function satisfy:

1. if the size of each input poset is at most n, then the size of each output poset is at most n;

2. if the size of each *Mul* and *Val* in the input is at most n, then the size of each *Mul* and *Val* in the output is at most n.

2.5.4 *Run* is not an ω-cpo

Not all ω-chains in *Run* have a least upper bound. To see this, consider first the ω-chain $\langle\!\langle \langle \cup_{j \leq i} \{\langle 1, val, j \rangle\}, \emptyset \rangle \rangle\!\rangle_{i \in \mathsf{N}}$ of *POSet*(L), where *val* is any value and we have taken *Tag* to be N. This ω-chain has no least upper bound, because $\langle \{\langle *, val, tag \rangle\}, \emptyset \rangle$ and $\langle \{\langle ?, val, 0 \rangle, \langle *, val, 0 \rangle\}, \emptyset \rangle$ are incomparable upper bounds of it. They are incomparable (with respect to $\leq^{\rho_c}_{POSet(L)}$) because they have the same meaning but are not "syntactically" equal (i.e., are not related by $=_{POSet(L)}$). By analogous reasoning, the ω-chain

$$\langle\!\langle \langle (\lambda e\!:\!Name \times Name.\ \cup_{j \leq i} \{\langle 1, val, j\rangle\}), \emptyset \rangle \rangle\!\rangle_{i \in \mathsf{N}}$$

of *Run* has no least upper bound.

Intuitively, $POSet(L)$ and *Run* are not ω-cpo's because posets of ms-atoms are not *canonical* representations of sets of sequences of concrete values; in particular, there are different posets with the same meaning. Although simple examples like the one above are easily prohibited, a general solution is complicated by abstract values whose meanings overlap and by algebraic identities among constants.

2.6 Sanity Conditions for ms-atoms

The definitions in Section 2.2 could be augmented with numerous sanity conditions. For example, we could associate an arity with each symbol, and require that all interpretations assign to each symbol a function of appropriate arity. We could also introduce a type system to ensure statically that functions are applied only to values in their domain. Although it is easy to formulate conditions of this nature that ensure that each ms-atom in isolation is meaningful (i.e., represents some set of messages), it is difficult to find equally powerful sanity conditions on *sets* of ms-atoms, because ensuring that a set of ms-atoms represents some set of messages requires checking satisfiability of the constraints implied by the symbolic values. Such checks would be feasible only if the framework were specialized to specific abstract values and to constants with specific interpretations. Since these sanity conditions are not needed to ensure soundness of our analysis, we omit them. Constructing specialized versions of the framework for specific application areas may be worthwhile, since it would facilitate use of the framework for synthesis of fault-tolerant systems, by helping ensure that refinement does not lead to an unimplementable design.

To illustrate the difficulty of ensuring that a set of ms-atoms is meaningful, we consider the set of ms-atoms $\{X : \mathbf{R}_+, X+1 : \mathbf{R}_-\}$, where the constants + and 1 have their usual interpretation. Note that the interpretation $\rho_c(+)$ of + as a constant symbol is distinct from its meaning $[\![+]\!]_{AVal}$ as an abstract value. It should be clear that no set of messages is represented by this set of ms-atoms, because there is no value for X that satisfies all the constraints. Using the semantics in Section 2.4.1 and the interpretation of $\mathbf{R}_-$ and $\mathbf{R}_+$ in Section 2.2.2, this is equivalent to the claim that there is no real number x such that $x \geq 0$

and $x + 1 \leq 0$. It follows, again according to the semantics in Section 2.4.1, that an input-output function that returns such ms-atoms in its output does not represent any process. Since soundness of the analysis is a meaningful issue only for input-output functions that do represent processes, omitting these sanity conditions does not cause unsoundness.

Chapter 3

Analyzing Systems that Fail

This chapter describes two methods for doing fault-tolerance analysis. The first, presented in Section 3.1, extends the framework introduced in Chapter 2 to specify how a component behaves when it fails. Some of the limitations of this approach are discussed in Section 3.2.

The remainder of the chapter then describes an extension to this framework that overcomes these limitations. We increase the expressiveness of specifications in order to capture non-trivial relationships between values in the failure-free and faulty executions. The effects of failures are represented explicitly as *changes* (or *perturbations*) to failure-free behavior. Section 3.3 extends the concrete model to capture correlations between a component's failure-free and faulty behaviors. Sections 3.4 and 3.5 then extend the representations of runs and components, respectively, in the abstract system model.

3.1 Fault-Tolerance Analysis Without Perturbations

Any form of fault-tolerance analysis will require descriptions of possible component failures and fault-tolerance requirements. For our method, these are dealt with in Sections 3.1.1 and 3.1.2, respectively. Section 3.1.3 illustrates these definitions using the running example introduced in Chapter 2.

3.1.1 Behavior of Failure-Prone Systems

A component's behavior depends on what failures it can suffer. This dependency can be reflected by parameterizing each component by its possible failures. Let *Fail* be the set of all failures of interest, with distinguished element $OK \in \mathit{Fail}$ representing absence of failure. A process is represented by a function p whose domain is the set of possible failures of this process and such that, for each $fail \in dom(p)$, $p(fail)$ describes the component's behavior when that failure is present. This parameterization is used at the concrete and abstract levels; thus, we have

$$Process_F \triangleq \{p \in Fail \rightharpoonup Process \mid OK \in dom(p)\} \tag{3.1}$$

$$IOF_F \triangleq \{f \in Fail \rightharpoonup IOF \mid OK \in dom(f)\} \tag{3.2}$$

For example, we use $crash \in Fail$ to indicate that a component crashes at some unspecified time during a computation. Consider a process $p \in Process_F$ subject only to crash failures, i.e., $dom(p) = \{OK, crash\}$. Suppose the output from p to q on a certain input history h_{in} is $F(X)^1$, i.e., $p(OK)(h_{in})(q)$ is the singleton poset containing the ms-atom $F(X)^1$. Since p might crash before or after sending this message, $p(crash)(h_{in})(q)$ must represent both of these possiblities; for example, $p(crash)(h_{in})(q)$ might be the singleton poset containing the ms-atom $F(X)^?$. Note that in our terminology, a failure—that is, an element of *Fail*, such as *crash*—is not itself an event that occurs during a computation; rather, a failure is simply a token indicating what (if any) erroneous behavior occurs during a computation.

Recall that a failure scenario associates a failure with each component of a system. This is true at the concrete and abstract levels. At both levels, a system is represented by a function with signature $Name \to (Fail \to S)$ for some S (S is either *Process* or *IOF*). For any function f with such a signature, the set of failure scenarios for f is

$$FS(f) \triangleq \{fs \in Name \to Fail \mid (\forall x \in Name : fs(x) \in dom(f(x)))\}. \tag{3.3}$$

It is convenient to define

$$fs_{OK} \triangleq (\lambda x : Name.\ OK). \tag{3.4}$$

We say that a component x is *non-faulty* in failure scenario fs iff $fs(x) = OK$.

The concrete runs of a concrete system $np \in Name \to Process_F$ for a given failure scenario $fs \in FS(np)$ are given by

$$cruns_F(np)(fs) \triangleq cruns(\lambda x : Name.\ np(x)(fs(x))). \tag{3.5}$$

The behavior of an abstract system $nf \in Name \to IOF_F$ for failure scenario $fs \in FS(nf)$ is represented by $run_F(nf)(fs) = \mathrm{lfp}(step_F(nf, fs))$, if it exists, where

$$step_F(nf, fs) = step(\lambda x : Name.\ nf(x)(fs(x))). \tag{3.6}$$

The meaning of IOF_F is given by a slight extension of the definition (2.68) of $\sqsubset_{IOF}$. For $p \in Process_F$ and $f \in IOF_F$,

$$p \sqsubset^{\rho_c, lvar}_{IOF_F} f \triangleq dom(p) = dom(f) \wedge (\forall fail \in dom(p) : p(fail) \sqsubset^{\rho_c, lvar}_{IOF} f(fail)). \tag{3.7}$$

3.1.2 Fault-Tolerance Requirements

A fault-tolerance requirement imposes conditions on the system's possible behaviors in certain failure scenarios. Since a system's possible behaviors are approximated as a run, a fault-tolerance requirement is expressed as a mapping from failure scenarios to predicates on runs. We use a mapping, rather than just a single predicate on runs, so that requirements involving graceful degradation, in which stronger requirements are associated with failure scenarios involving fewer or less catastrophic failures, can be expressed. A predicate on runs is a function with signature $Run \to \mathbf{B}$, where the booleans are

$$\mathbf{B} = \{\text{true}, \text{false}\}. \tag{3.8}$$

A system $nf \in Name \to IOF_F$ satisfies a fault-tolerance requirement $b \in FS(nf) \to (Run \to \mathbf{B})$ iff for each $fs \in FS(nf)$, $run_F(nf)(fs)$ exists and satisfies $b(fs)$. The only sanity requirement on b is that it be independent of tags, i.e.,

$$(\forall fs \in dom(b) : (\forall r_1, r_2 \in Run : r_1 =_{Run} r_2 \Rightarrow b(fs)(r_1) = b(fs)(r_2))). \tag{3.9}$$

Verifying that a concrete system satisfies its fault-tolerance requirement requires checking that the processes are represented by the input-output functions of an abstract system nf, and for each failure scenario $fs \in FS(nf)$, checking that $run_F(nf)(fs)$ exists and satisfies $b(fs)$.

3.1.3 Running Example

We develop concrete and abstract models of a two-stage replicated pipeline in which the processing components F_i and G_i produce arbitrary values when faulty. These models build on the definitions for the failure-free two-stage replicated pipeline in Sections 2.1.3 and 2.3.2.

Concrete System. Process $CComp_F(src, dest, \phi)$ describes a processing component that normally behaves the same as $CComp(src, dest, \phi)$, defined in (2.18), and that produces arbitrary values when faulty.

$$\begin{aligned} CComp_F(src, dest, \phi) = (\lambda fail : \{OK, valFail\}.\ & \\ & \textbf{if } fail = OK \textbf{ then } CComp(src, dest, \phi) \\ & \textbf{else } CValFail(src, dest, CVal)), \end{aligned} \tag{3.10}$$

where the process $CValFail(src, dest, S)$ sends an arbitrary value in $S \subseteq CVal$ to $dest \in Name$ whenever it receives a value from $src \in Name$:

$$CValFail(src, dest, S) = \bigcup_{g \in dests \to Seq(S)} \{\langle CValFail_g(src, dest), CHist\rangle\} \tag{3.11}$$

$$CValFail_g(src, dest) = (\lambda h\colon CHist.\ (\lambda x\colon Name.\ \textbf{if}\ x = dest\ \textbf{then}\ g(x)[0..(|h(src)|-1)]\ \textbf{else}\ \varepsilon)), \tag{3.12}$$

where for any sequence σ and any natural numbers i and j, $\sigma[i..j]$ is the contiguous subsequence of σ from position i to position j (inclusive); as a special case, we define $\sigma[0..(-1)] = \varepsilon$.

At this point, we won't consider failures of the source, voter, or actuator, so these components correspond to the same processes as before, but with a trivial lambda abstraction wrapped around; e.g., for the source,

$$CSrc_F(dests) = (\lambda fail\colon \{OK\}.\ CSrc(dests)). \tag{3.13}$$

Concrete Runs. The concrete runs of this system can be computed using definition (3.5) of $cruns_F$. Let np_F^{re} be the obvious mapping from $Name$ to $Process_F$: $np_F^{re}(S) = CSrc_F(\{F_1, F_2, F_3\})$, *etc.* As an example, consider the failure scenario fs_1 in which only F_1 fails. It is easy to check that

$$cruns_F(np_F^{re})(fs_1) = \bigcup_{i \in \mathsf{N}, cv \in CVal} cr_F^{re}(i, cv),$$

where $cr_F^{re}(i, cv) \in CRun$ is the same as $cr^{re}(i)$ except that the sequence of concrete values from F_1 to G_1 is replaced with $\langle\!\langle cv \rangle\!\rangle$, and the sequence from G_1 to V is replaced with $\langle\!\langle \phi_2(cv) \rangle\!\rangle$.

Abstract System. The input-output functions representing the processors F_i and G_i are appropriate instances of

$$Comp_F(src, dest, op) = (\lambda fail\colon \{OK, valFail\}.\ \textbf{if}\ fail = OK\ \textbf{then}\ Comp(src, dest, op)\ \textbf{else}\ ValFail(src, dest, \top_V)), \tag{3.14}$$

where $\top_V$ is defined by (2.52), and $ValFail(src, dest, aval)$ sends an arbitrary value represented by $aval \in AVal$ to $dest \in Name$ when it receives an input from $src \in Name$. The definition of *ValFail* is similar in structure to definition (2.51) of *Comp*:

$$ValFail(src, dest, aval) = (\lambda h\!:\!Hist.\ (\lambda x\!:\!Name.\ \textbf{if}\ x = dest\ \textbf{then}\ \overline{arbval(aval)}(h(src))\ \textbf{else}\ \langle\emptyset,\emptyset\rangle)), \tag{3.15}$$

where $arbval(aval)(val) = \{\langle\_, aval\rangle\}$, and $\overline{arbval(aval)}$ is the pointwise extension of $arbval(aval)$ from values to posets of ms-atoms; as in the extension of *apOp* discussed below (2.53), re-tagging may be needed to avoid collisions.

Again, since we won't consider for now failure of the source, voter, or actuator, the input-output functions for those components are the same as in Section 2.3.2, but with a trivial lambda abstraction wrapped around; e.g., for the source,

$$Src_F(dests) = (\lambda fail\!:\!\{OK\}.\ Src(dests)). \tag{3.16}$$

Fault-Tolerance Requirement. Suppose the fault-tolerance requirement for this system is: if at most one component F_i or G_i fails, then the input to the actuator remains the same as in the failure-free case. Thus, the fault-tolerance requirement is

$$b^{re} = (\lambda fs\!:\!FS(nf_F^{re}).\ (\lambda r\!:\!Run.\ \ |\{x \in Name \mid fs(x) \neq OK\}| \leq 1 \Rightarrow b_0(fs, r))) \tag{3.17}$$

where nf_F^{re} is the obvious mapping in $Name \to IOF_F$ for this example, and b_0 is a predicate that expresses that the inputs to the actuator are unchanged. Finding a suitable predicate b_0 is slightly tricky. A natural attempt is: $b_0(fs, r)$ holds iff the abstract inputs to the actuator are the same as in the failure-free run, i.e., iff $r(A) =_{Run} run_F(nf)(fs_{OK})(A)$. Note that $run_F(nf)(fs_{OK})(A)$ is the run in Figure 2.2. Thus, more specifically, $b_0(fs, r)$ says that the sole input ms-atom to the actuator is from V and has multiplicity of one and value $G(F(X))\!:\!\mathbf{N}$.

This specification is both unnecessarily restrictive and too weak. These problems both result from the tacit assumption that the variable X represents the output of the source in faulty runs. It is unnecessarily restrictive because in general, the input-output function for the source could, as a result of inputs received from a faulty component, use a different variable to represent the source's output, even if the output itself is not really affected (at the concrete level). For example, in Figure 1.1, faulty component F_1 sends new inputs to source S; those new inputs could cause the input-output function for S to use in its output a different variable than it would have used otherwise. The specification is too weak because, if the source does receive inputs from a faulty component, those inputs may cause the value represented by X to change, in which case this specification does *not* ensure that the input value of the actuator is unchanged.

To remedy the weakness of this specification, we take $b_0(fs, r)$ to be the conjunction of two conditions:

1. The poset $run_F(nf)(fs_{OK})(A)$ uniquely determines the inputs to the actuator as a function of the interpretation of the variables that appear in it. More precisely, we require that history $run_F(nf)(fs_{OK})(A)$ is unambiguous. A history is *unambiguous* if each poset of ms-atoms in it is unambiguous. A poset of ms-atoms is *unambiguous* if the ordering is total and each value and multiplicity in each ms-atom is unambiguous. An element of *Val* (including elements of *Mul*) is *unambiguous* if it is a singleton set $\{\langle s, a\rangle\}$ and either $s \in SVal_0$ or $|[\![a]\!]_{AVal}| = 1$.

2. The concrete values represented by variables that appear in $run_F(nf)(fs_{OK})(A)$ are unaffected by failures. More precisely, we require that in each failure scenario fs of interest, the input history of the actuator A is structurally-unaffected. Structurally-unaffected is the least predicate satisfying the following recursive definition. The input history of component x of system nf in failure scenario fs is *structurally-unaffected* if $run_F(nf)(fs)(x) =_{Run} run_F(nf)(fs_{OK})(x)$ and, for each variable Y that appears in $run_F(nf)(fs)(x)$, Y is local to a non-faulty component whose inputs are unambiguous and structurally-unaffected.

We illustrate these two concepts with some examples. First, we illustrate "unambiguous". Let ρ be an interpretation of variables, i.e., $\rho \in Interp(Var)$. The value $\{X : \mathbf{N}, Y : \mathbf{N}\}$ is ambiguous because it can represent either $\rho(X)$ or $\rho(Y)$. On the other hand, the value $X : \mathbf{N}$ (note that we have elided the curly braces around a singleton value, as per the notational conventions on page 23) is unambiguous, because for a given interpretation ρ, it can represent only one fixed concrete value, namely, $\rho(X)$. The poset $\langle\{\ell_0, \ell_1\}, \emptyset\rangle$, where $\ell_0 = X{:}\mathbf{N}$ and $\ell_1 = X{:}\mathbf{N}$ (note that we have elided the multiplicity (of one) and the tag (of zero), as per the notational conventions on page 26), is ambiguous, because it can represent either $\langle\!\langle \rho(X), \rho(Y)\rangle\!\rangle$ or $\langle\!\langle \rho(Y), \rho(X)\rangle\!\rangle$. On the other hand, the poset $\langle\{\ell_0, \ell_1\}, \{\langle \ell_0, \ell_1\rangle\}\rangle$ is unambiguous, because it can represent only $\langle\!\langle \rho(X), \rho(Y)\rangle\!\rangle$.

We illustrate "structurally-unaffected" using Figure 2.3 (page 26), which shows the behavior of a two-stage replicated pipeline when component F_1 suffers a Byzantine failure. The input history of source S is not structurally-unaffected, because it is not the same as the input history of the source in failure scenario fs_{OK} (shown in Figure 2.2). Informally, this reflects the fact that the inputs of S are potentially affected by the failure of F_1. Since variable X is local to S, the valued represented by X may depend on the inputs of S. Thus, the value of X is potentially affected by the failure of F_1. Since X appears in the input history of component G_2, the input to G_2 is also potentially affected by this failure. This explains informally why the input history of G_2 in Figure 2.3 is not structurally-unaffected.

The introduction of explicit perturbations in Section 3.4 is motived in part by the awkwardness of this specification and the concomitant difficulty of being sure that it has the intended meaning. In the perturbational framework, the requirement that inputs to the actuator are unchanged can be expressed concisely as the property that the perturbations associated with those inputs equal the "identity" perturbation, which denotes unchangedness (details are in Section 3.5.1).

The reader might wonder whether a history containing unambigous posets should be considered unambiguous, since in a sense, such a history does not uniquely determine the inputs to a component as a function of the interpretation of the variables in the history—specifically, the history says nothing about the ordering between inputs from different senders. This is true, but it is not cause for concern, because in our abstract system models, orderings between inputs from different senders are always ignored. For example, suppose the fault-tolerance requirement for some system is that the inputs to a certain component are unaffected by failures. Consider the input histories of that component in the runs computed for failure scenario fs_{OK} and for some other failure scenario. Suppose these input histories are equal, unambiguous, and structurally-unaffected. These input histories may still represent multiple interleavings of concrete inputs from different senders. Since this "ambiguity" exists independently of whether component failures are being considered, it has no impact on fault-tolerance analysis and therefore can be ignored in our definition of "unambiguous". In terms of our example, since each interleaving of concrete inputs represented by the input histories is a possible input to the component in the absence of failures, each of these interleavings must also be an acceptable input to the component in the presence of failures.

With the input-output functions and fault-tolerance requirement just given, we can check whether the system satisfies its fault-tolerance requirement by computing, for each failure scenario $fs \in FS(nf_F^{re})$, the fixed-point $r = \mathrm{lfp}(step_F(nf_F^{re})(fs))$ and checking whether $b^{re}(fs)(r)$ is true.[1] The outcome in this case is affirmative. Of course, if it is not already established that the input-output functions represent the appropriate processes, this must be checked

[1] An obvious optimization, based on definition (3.17) of b^{re}, is to compute the fixed-point only for failure scenarios in which at most one component is faulty.

as well. Neither of these two obligations involves verifying properties of the potentially complicated functions computed by the two stages of the pipeline; for example, verifying that $CComp_F$ is represented by $Comp_F$ requires little more than checking that $CComp_F$ is deterministic and stateless. Thus, the abstractions introduced so far do allow separation of concerns in this example.

3.2 Motivation for Changes

This section discusses some limitations of the analysis method just described.

3.2.1 Expressiveness

In systems where the appropriate input histories are unambiguous and structurally-unaffected, a predicate in the style of (3.17) seems to capture (albeit awkwardly) the intended meaning, i.e., that the inputs to the actuator are unaffected by failures. However, even when the system is fault-tolerant, using input-output functions that make the appropriate input histories unambiguous might be awkward, and—worse—the appropriate input histories might not be structurally-unaffected. We discuss these two issues in turn.

Unambiguous

For concrete systems comprising only determinate components, it is always possible to construct an abstract model of the system such that the input-output functions produce only unambiguous ms-atoms. Roughly, this can be done by introducing constant symbols for *all* of the functions computed by the components, including boolean functions used in the guards of conditionals and loops, as well as functions used to compute output values. For example, consider a process that sends $\phi(x)$ to A_1 if the input value x satisfies some condition, and sends $\phi(x)$ to A_2 otherwise. An input-output function representing this process might produce on input X an output history in which the messages sent to A_1 are represented by $F(X)^{M_1(X):?}$ and the messages sent to A_2 are represented by $F(X)^{M_2(X):?}$. This representation is unambiguous. Note that M_1 and M_2 represent the predicate in the conditional, and F represents ϕ. If we "simplified" the input-output function by omitting (say) M_1, then the output to A_1 would be simply $F(X)^?$, which is ambiguous.

Abstract systems with unambiguous posets can be constructed even if the system contains non-determinate components. This requires introducing local variables to represent explicitly all choices made by the components. For example, consider a process *first* that forwards its first input to A. Suppose *first* receives X_1 from S_1 and X_2 from S_2. A natural but ambiguous representation of its output to A is the poset $\langle\{\langle 1, \{F(X_1), F(X_2)\}, 0\rangle\}, \emptyset\rangle$. One unambiguous representation is the poset $\langle\{\ell_1, \ell_2\}, \{\langle \ell_1, \ell_2\rangle\}\rangle$, where $\ell_i = F(X_1)^{Y_i:?}$. Note that Y_1 and Y_2 represent the outcome of the non-determinate choice, and $\prec = \{Y_1, Y_2\}$. The same technique of introducing local variables applies to internal non-determinism as well as non-strictness.

In summary, constructing input-output functions that produce unambiguous outputs sometimes requires introducing additional symbols in their outputs. Furthermore, the outputs of an input-output function often contain the symbols that appear in its inputs, so when computing the run for a system, the symbols introduced in the outputs of one input-output function are often propagated by other input-output functions. Thus, a symbol introduced to make an output unambiguous may eventually appear in many ms-atoms in a run. Introducing and propagating these symbols makes computation of runs more expensive and clutters the runs, making them harder to read (reading runs is sometimes useful, e.g., to see why a system does not satisfy its fault-tolerance requirement). Since often we care only about the sensitivity of values to changes caused by failures, a more concise representation can be achieved by specifying this information directly, rather than encoding it in additional symbols.

Structurally-Unaffected

We give two examples of concrete systems for which it is impossible to find input-output functions such that the input histories of the acutuators are structurally-unaffected. Roughly speaking, such input-output functions do not exist if non-determinate components see any effects of failures. Both examples can be analyzed using the perturbational framework, which is described in Sections 3.4 and 3.5.

Impossibility Example 1. Consider a system comprising a source S, processing components F_1–F_3 that perform some triplicated computation, a voter V, an unreplicated processing component G, which is assumed not to fail, and an actuator A. The output of the voter is sent to both A and G, and the output of G is sent to A. The failure-free behavior of the system is represented by the run in Figure 3.1. Variable Y is local to G, allowing for some non-determinacy or non-strictness of the component. Of course, the physical component being modeled by G may really be determinate but with behavior that depends on aspects of the system that have been abstracted from, such as timing conditions, load, or

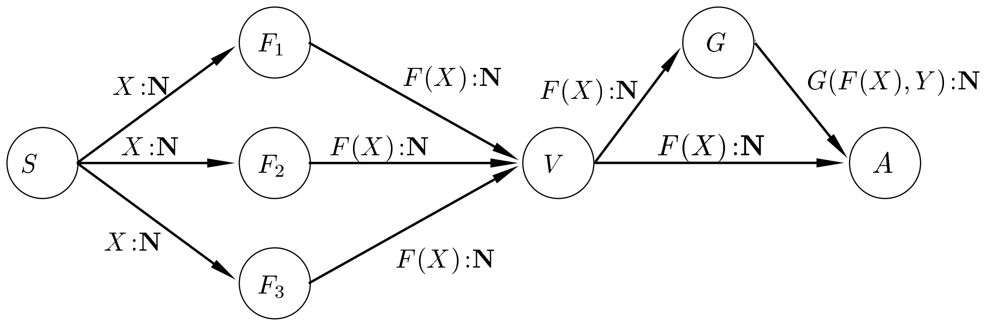

Figure 3.1: Impossibility example 1: failure-free behavior.

physical environment. We model such components as being non-determinate, and we use local variables to represent the values they produce.

Consider a failure scenario in which F_1 suffers a failure that causes it to send an arbitrary value to the voter and to G. The behavior of the system in that case is represented by the run in Figure 3.2. As in Figure 1.1, faulty components are distinguished in figures by dots on their circumference. Note that the input history of A is not structurally-unaffected, because

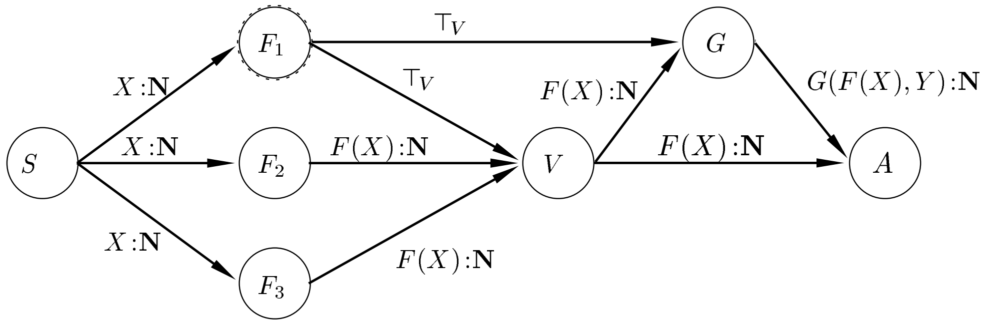

Figure 3.2: Impossibility example 1: faulty run.

it contains variable Y, which is local to G, and the inputs to G have changed. In particular, if the value represented by Y can be affected by the arbitrary input to G from F_1, then the system is not fault-tolerant; if it cannot be affected (e.g., because G ignores inputs from all processes except V), then the system is fault-tolerant. There is no way to express in this framework whether or not the value of Y is sensitive to the new input, because there is no way to express correlations between a component's inputs and non-deterministic choices made by that component.

Impossibility Example 2. The above example involves a failure that causes a message to be sent between components that don't communicate in failure-free executions. This next example does not involve such communication. Consider a system with a source S that sends values X_1 and X_2 along separate channels C_1 and C_2, respectively, to a component G, which, for each i, processes values received from C_i and sends the results to actuator A_i. Note that component G happens to be non-deterministic, as in the previous example; variables Y_1 and Y_2 are local to G. The failure-free behavior of the system is represented by the run in Figure 3.3.

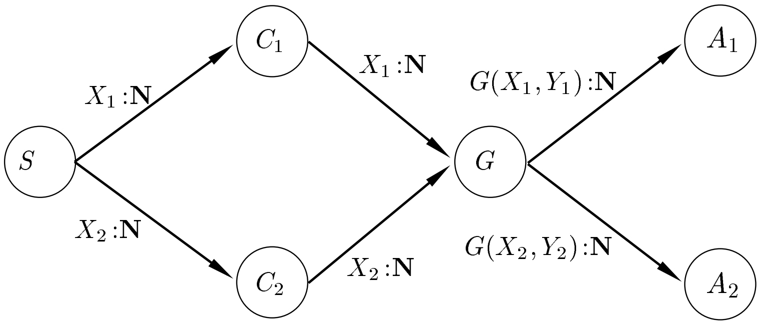

Figure 3.3: Impossibility example 2: failure-free behavior.

Now, suppose each channel might fail, causing the channel to corrupt transmitted values. The fault-tolerance requirement is that if C_1 fails, then the input of A_2 is unaffected, and likewise for C_2 and A_1. The run shown in Figure 3.4 represents the behavior of the system when C_1 fails. Note that the input history of A_2 is not structurally-unaffected, because it

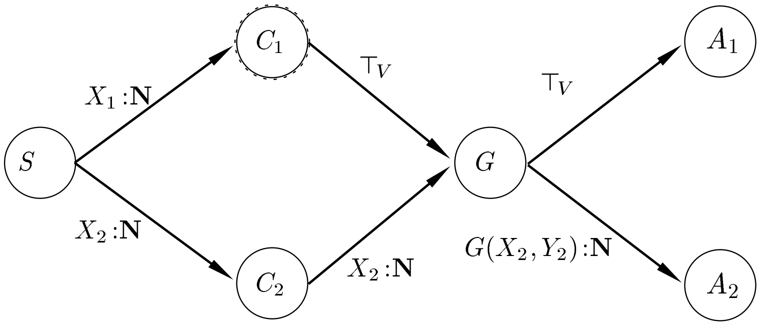

Figure 3.4: Impossibility example 2: faulty run.

contains variable Y_2, which is local to G, and the inputs to G have changed. If G processes inputs from C_1 and C_2 independently, then the value of Y_2 is unaffected by the faulty input from C_1 (and likewise for Y_1 and C_2), and the system is fault-tolerant; otherwise, it is not. However, in the framework of Section 3.1, there is no way to indicate whether or not the value of Y_2 is affected by the faulty input.

Incidentally, this example can also be used to illustrate that the condition

$$run_F(nf)(fs)(A_2) = run(nf)(fs_{OK})(A_2)$$

is unnecessarily restrictive: when the input-output function for G sees the faulty value from C_1, it could use Y_1 instead of Y_2 in its output to A_2, even if the represented value is unchanged. In that case, $run_F(nf)(fs)(A_2) \neq run(nf)(fs_{OK})(A_2)$, so the abstract input to A_2 is not structurally-unaffected.

3.2.2 Non-Trivial Relationships between Original and Perturbed Values

We now consider a different axis along which the method described in section 3.1 has too limited expressiveness. If only failures that corrupt values arbitrarily are considered, as in all of the above examples, then the corresponding perturbations are either "unchanged" or "arbitrarily changed". This single bit of information can be encoded in symbolic values, by using the same or different variable names, respectively, to represent those values. The method described in Section 3.1 uses such an encoding. However, non-trivial relationships between values in the failure-free and faulty computations cannot be expressed with that method. We give two examples that involve such relationships.

Example 1: ECC. Error-correcting codes (ECCs) are widely used when transmitting data over unreliable channels or storing data on unreliable media. Our goal in this example is to characterize abstractly the fault-tolerance provided by an ECC, so that we can analyze larger systems that use ECCs together with other fault-tolerance mechanisms. To illustrate the importance of explicit perturbations for this purpose, we consider a simple system comprising only a source S, an encoder E, an unreliable channel C, a decoder D, and a receiver R. Constant function F represents the ECC function. The failure-free behavior of the system is represented by the run in Figure 3.5. The abstract value W is a set of bit-vectors ("words").

Suppose the channel may fail by corrupting at most k bits of the transmitted value, and that a k-bit ECC is used. The fault-tolerance property of a k-bit ECC is: if the input value

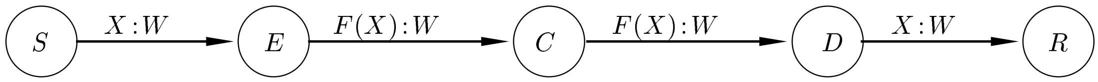

Figure 3.5: Failure-free behavior of system with ECC.

to the decoder differs in at most k bits from the input value to the encoder, then the decoder outputs the value given to the encoder. The fault-tolerance requirement for this system is that the input to receiver R is unaffected by failure of channel C.

To analyze this system, we need to find an input-output function for channel C that expresses that at most k bits are corrupted in each output. For concreteness, consider what the output of that input-output function should be on an input Y. The most accurate symbolic value is something like $corrupt(Y, Z)$, where Z is a local variable of the channel (note that the faulty channel is non-deterministic). We would like the abstract value in the output of C to denote the set of words that differ from Y in at most k bits; if this set cannot be expressed, then we will not be able to conclude that the decoder outputs the original value Y.

Indeed, this set cannot be expressed as an abstract value in the framework of Section 3.1, since it would require an abstract value whose meaning depends on the interpretation of a variable (namely, Y). However, this set can be expressed directly in the perturbational framework that will be introduced in Section 3.4.

The perturbational framework represents one approach to dealing with this and similar systems. An alternative approach is to extend the definition of abstract values to allow abstract values whose meanings depend on the interpretation of symbolic values. Such abstract values are considered in Chapter 5.

Example 2: Median. Non-trivial relationships between original and perturbed values also arise in replicated systems in which different replicas produce approximately equal (but not necessarily identical) values. For example, consider a system with replicated sensors that, when non-faulty, produce values that are within ε of some physical quantity. Suppose sensors fail by producing arbitrary values. We model this system using a component E, representing the environment, that sends the actual value X of the measured physical quantity to components S_i, representing the sensors. We take the failure-free behavior of the system to correspond to the ideal case in which each sensor outputs (actual value) X to component M. Component M sends the median of its inputs to component F, which computes some control function and sends the result to an actuator A. The behavior of this system is represented by the run in Figure 3.6.

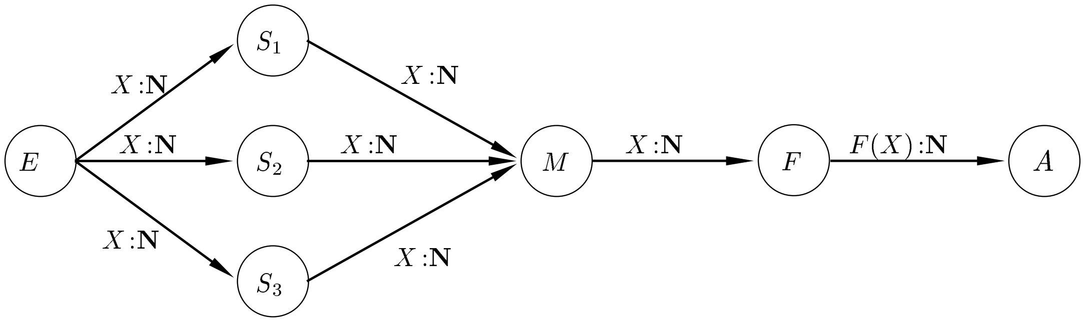

Figure 3.6: Idealized behavior of system with median.

Consider failure scenarios in which a majority of the sensors are non-faulty and produce values within ε of their input value X, but a minority fail by producing arbitrary values. It is natural to model both of these deviations from the ideal behavior as perturbations. The analysis would then be based on how the median propagates perturbations: if all the inputs are originally all equal (to X), and a majority of the inputs change by at most ε, then the output changes by at most ε. However, this analysis requires the ability to express that the perturbed inputs to M are within ε of (the concrete value represented by the variable) X. As in the previous example, there is no way to express this set in the framework of Section 3.1.

3.3 Concrete Model with Failures and Correlations

This section describes the concrete model that is later used in Section 3.5.2 to give a semantics for our perturbational framework. To see why representation (3.1) of processes is inadequate for this purpose, consider a component that non-deterministically selects and outputs a single number. Suppose this component can fail only by crashing. Thus, at worst a failure causes the component to output nothing; a failure cannot cause the component to output a different value than in the failure-free computation. Defining a process $p \in \mathit{Process}_F$ that describes this component and reflects this fact is problematic. The root of the problem is that the definition of $\mathit{Process}_F$ forces one to describe completely *separately* the possible behaviors in the cases $\mathit{fail} = \mathit{OK}$ and $\mathit{fail} = \mathit{crash}$. Recall that the interpretation of $\mathit{fail} = \mathit{crash}$ is that a crash occurs at some unspecified time during a computation; thus, the possible behaviors of $p(crash)$ are to: (1) non-deterministically select and output any number and then crash, or (2) crash (before doing anything else) and output nothing.

If one tries to determine solely from information in p how the component's behavior might change as a consequence of a crash failure, in the absence of information about correlations between the behaviors in $p(OK)$ and $p(crash)$, the only safe (conservative) assumption is that each possible behavior of $p(OK)$ may be changed by a crash into any possible behavior of $p(crash)$. With this conservative approximation, it would appear that a crash could change the value output by p, while in fact, it cannot.

The root of the innacuracy is that elements of $Process_F$ do not describe correlations between the possible behaviors of a component in the presence and absence of failures. Continuing the above example, we would like to express that a behavior in which the process outputs a number n is changed by occurrence of a crash only into: (1) a behavior in which the process outputs n and then crashes, or (2) a behavior in which the process crashes (before doing anything else) and outputs nothing. So, in this example, the behavior in which the non-faulty process outputs n is related (by a crash) to the behaviors in which the faulty process outputs n or outputs nothing but is not related to the behavior in which the faulty process outputs n', where $n' \neq n$.

To reflect such correlations, we now model a process as a function of type $Fail \rightharpoonup Set(IRProcess \times IRProcess)$. The interpretation of $\langle irp, irp' \rangle \in p(fail)$ is that the process can behave like irp in the absence of failures, and that failure $fail$ can cause p's behavior to change from that of irp to that of irp'. We impose two sanity conditions. The first condition, $origIndep_C$, requires that the set of failure-free behaviors of the process be independent of the failure:

$$origIndep_C(p) \triangleq (\forall fail_1, fail_2 \in dom(p) : \overline{\pi_1}(p(fail_1)) = \overline{\pi_1}(p(fail_2))), \tag{3.18}$$

where $\overline{\pi_1}$ is the pointwise extension of π_1 from tuples to sets of tuples.

An input to a pair $\langle irp, irp' \rangle \in p(fail)$ is a pair $\langle \sigma, \sigma' \rangle$ of concrete histories, where σ is the input to irp in a failure-free execution of the system, and σ' is the input to irp' in a faulty execution. A pair $\langle irp, irp' \rangle \in p(fail)$ of input-restricted processes is enabled only if both processes are enabled on their respective inputs; this convention is reflected in the definition of $cruns_{FC}$ below. Thus, to ensure that the set of failure-free behaviors is independent of the failure-mode, we must require that enabledness of each input-restricted process in $\overline{\pi_1}(p(fail)))$ is independent of the component's inputs in the faulty computation. This is ensured by the second sanity condition, *consistent*, defined by

$$\begin{aligned} consistent(p) \triangleq (\forall fail \in dom(p) : (\forall \langle \sigma, \sigma' \rangle \in Chain(CHist) \times Chain(CHist) : \\ (\forall irp \in \overline{\pi_1}(p(fail)) : enabled(irp, \sigma) \Rightarrow (\exists irp' \in IRProcess : \\ \langle irp, irp' \rangle \in p(fail) \wedge enabled(irp', \sigma'))))). \end{aligned} \tag{3.19}$$

Thus, processes are elements of

$$\begin{aligned} Process_{FC} \triangleq \{p \in Fail \rightharpoonup Set(IRProcess \times IRProcess) \mid \; & \wedge origIndep_C(p) \\ & \wedge consistent(p)\}. \end{aligned} \tag{3.20}$$

The possible behaviors of a concrete system $np \in Name \to Process_{FC}$ in failure scenario $fs \in FS(np)$ are represented by a set $cruns_{FC}(np)(fs) \subseteq CRun \times CRun$. The interpretation of $\langle cr, cr' \rangle \in cruns_{FC}(np)(fs)$ is that cr is a possible failure-free run of the system and that the failures in fs can cause the system's behavior to change from cr to cr'. Formally,

$$\begin{aligned} cruns_{FC}(np)(fs) \triangleq \{\langle cr_1, cr_2 \rangle \in CRun \times CRun \mid \\ (\exists h \in Name \to IRProcess \times IRProcess : (\forall x \in Name : \\ \wedge\, h(x) \in np(x)(fs) \\ \wedge\, (\forall \alpha \in \{1,2\} : cr_\alpha \in cruns(\lambda x \!:\! Name.\, \{\pi_\alpha(h(x))\})))\}. \end{aligned} \tag{3.21}$$

The set of failure-free runs of the system is given by $\overline{\pi_1}(cruns_{FC}(np)(fs))$. Conditions $origIndep_C$ and *consistent* together ensure that the set of failure-free runs is independent of the failure scenario:

$$\begin{aligned} (\forall np \in Name \to Process_{FC} : (\forall fs_1, fs_2 \in FS(np) : \\ \overline{\pi_1}(cruns_{FC}(np)(fs_1)) = \overline{\pi_1}(cruns_{FC}(np)(fs_2)))). \end{aligned} \tag{3.22}$$

3.3.1 Running Example

Definitions of $CSrc_{FC}$, $CVoter_{FC}$, and $CAct_{FC}$. We do not consider failures of the source S, voter V, or actuator A, so the elements of $Process_{FC}$ that describe those components are closely related to the processes $CSrc$, $CVoter$, $CAct \in Process$ defined in Section 2.1.3. The function $nonfaulty_{FC} \in Process \to Process_{FC}$ converts an element of *Process* into a failure-free element of $Process_{FC}$:

$$nonfaulty_{FC}(p) \triangleq (\lambda fail \!:\! \{OK\}.\; \bigcup_{irp \in p} \{\langle irp, irp \rangle\}). \tag{3.23}$$

So, for $dests, srcs \in Set(Name)$ and $dest \in Name$,

$$
\begin{aligned}
CSrc_{FC}(dests) &= nonfaulty_{FC}(CSrc(dests)) && (3.24)\\
CVoter_{FC}(srcs, dest) &= nonfaulty_{FC}(CVoter(srcs, dest)) && (3.25)\\
CAct_{FC} &= nonfaulty_{FC}(CAct), && (3.26)\\
& && (3.27)
\end{aligned}
$$

Definition of $CComp_{FC}$. For processors F_1–F_3 and G_1–G_3, we assume the same failure modes as in Section 3.1.3. These components are represented by appropriate instances of $CComp_{FC}$, where for $src, dest \in Name$ and $\phi \in \mathsf{N} \to \mathsf{N}$,

$$
\begin{aligned}
CComp_{FC}(src, dest, \phi) = (\lambda fail\colon \{OK, valFail\}.& && (3.28)\\
&\textbf{if } fail = OK \textbf{ then}\\
&\quad CComp(src, dest, \phi) \times CComp(src, dest, \phi)\\
&\textbf{else } CComp(src, dest, \phi) \times CValFail(CVal, \{dest\})),
\end{aligned}
$$

where $CComp$ and $CValFail$ are defined by (2.18) and (3.11), respectively. The clause for $fail = OK$ has been simplified using the fact that $CComp(src, dest, \phi)$ is a singleton set, so there is no need to explicitly restrict to pairs $\langle cr, cr' \rangle$ such that $cr = cr'$.

The concrete runs of this system can be computed using $cruns_{FC}$. Let np^{re}_{FC} be the obvious mapping from $Name$ to $Process_{FC}$: $np^{re}_{FC}(S) = CSrc_{FC}(\{F_1, F_2, F_3\})$, *etc.* As before, let fs_1 be the failure scenario in which only F_1 fails. It is easy to check that

$$
\begin{aligned}
cruns_{FC}(np^{re}_{FC})(fs_{OK}) &= \bigcup_{i \in \mathsf{N}} \langle cr^{re}(i), cr^{re}(i) \rangle\\
cruns_{FC}(np^{re}_{FC})(fs_1) &= \bigcup_{i \in \mathsf{N}, cv \in CVal} \langle cr^{re}(i), cr^{re}_{F}(i, cv) \rangle.
\end{aligned}
$$

3.4 Perturbational Framework: Representation of Runs

The set of possible behaviors of a system in a particular failure-scenario is represented by a single run. That run represents the system's failure-free behaviors as well as its possible behaviors in the given failure scenario. For example, in Figure 1.1, the failure-free behavior is described by the original parts of the perturbed ms-atoms, while the new ms-atoms and the perturbations in the perturbed ms-atoms together indicate how the failure-free behavior is changed by failures.

More generally, we introduce in this section a new set Run_{FC} of runs extended with perturbations and new ms-atoms. The meaning of an element of Run_{FC} is a set of *pairs* of concrete runs. For a run r computed for a system in failure scenario fs, the interpretation of a pair $\langle cr, cr' \rangle$ in the meaning of r is that the system's behavior may change from cr to cr' as a consequence of the failures in fs.

The definition of Run_{FC} is analogous to definition (2.29) of Run, except that the underlying set of ms-atoms is extended with perturbations and new ms-atoms. The extended set of ms-atoms is

$$L_{FC} \triangleq L_{per} \cup L_{new}, \tag{3.29}$$

where the sets L_{per} of perturbed ms-atoms and L_{new} of new ms-atoms are defined by

$$L_{per} \triangleq Mul \times Val \times \Delta Mul \times \Delta Val \times Tag \tag{3.30}$$

$$L_{new} \triangleq Mul \times Val \times Tag, \tag{3.31}$$

where ΔMul and ΔVal are perturbations to the multiplicity and value, respectively. Perturbations are represented similarly to values: an abstract part describing the possible changes in value, and a symbolic part representing the perturbed value itself, i.e., the concrete value in the faulty execution. Possible changes are described by binary relations over $CVal$; if the relation relates cv to cv', then the concrete value can change from cv to cv'. Thus, possible changes in value are represented by elements of a new set $\Delta AVal$, and each element of $\Delta AVal$ is interpreted as a binary relation over $CVal$. Formally, the interpretation of $\Delta AVal$ is given by a function $[\![]\!]_{\Delta AVal} \in \Delta AVal \to Set(CVal \times CVal)$.

For convenience, we assume that $id \in \Delta AVal$ denotes the identity relation:

$$[\![id]\!]_{\Delta AVal} = \{\langle x, x' \rangle \in CVal \times CVal \mid x = x'\} \tag{3.32}$$

and that the "top" element $\top_{\Delta V} \in \Delta AVal$ denotes an arbitrary change:

$$[\![\top_{\Delta V}]\!]_{\Delta AVal} = CVal \times CVal. \tag{3.33}$$

For any $a \in AVal$, we define $a_{\Delta} \in \Delta AVal$ by

$$[\![a_{\Delta}]\!]_{\Delta AVal} = [\![a]\!]_{AVal} \times [\![a]\!]_{AVal}. \tag{3.34}$$

The notational conventions introduced for Val are also used for ΔVal; for example, we might write $\top_{\Delta V}$ to denote $\{\langle \_, \top_{\Delta V} \rangle\} \in \Delta Val$. In the ms-atom on edge $\langle F_1, G_1 \rangle$ in Figure 1.1, the perturbation $\top_{\Delta V}$ indicates that the data sent from F_1 to G_1 may change arbitrarily when F_1 fails.

By analogy with definition (2.39) of $AMul$, we take $\Delta AMul$ to be an appropriate subset of $\Delta AVal$, namely, elements that relate natural numbers only to natural numbers:

$$\Delta AMul = \{\delta a \in \Delta AVal \mid (\forall \langle x, y\rangle \in [\![\delta a]\!]_{\Delta AVal} : x \in \mathsf{N} \Rightarrow y \in \mathsf{N})\}. \tag{3.35}$$

Here, δa is just a bound variable, like a in (2.39); we include δ in the name only as a reminder that this variable ranges over some set of perturbations. Note that for $a \in AMul$, a_Δ (defined by (3.34)) is in $\Delta AMul$. For example, in the ms-atom on edge $\langle F_1, G_1\rangle$ in Figure 1.1, the superscript $*_\Delta \in \Delta AMul$ indicates that the number of messages sent from F_1 to G_1 may change arbitrarily when F_1 fails.

The symbolic and abstract parts of a perturbation are aggregated in the same way as the symbolic and abstract parts of a value: by analogy with definitions (2.31) and (2.38) of Val and Mul, respectively, we define

$$\Delta Val \triangleq \mathcal{P}_{fin}(SVal \times \Delta AVal) \setminus \{\emptyset\} \tag{3.36}$$

$$\Delta Mul \triangleq \mathcal{P}_{fin}(SVal \times \Delta AMul) \setminus \{\emptyset\}. \tag{3.37}$$

Histories and runs are defined as before, except over L_{FC} instead of L:

$$Hist_{FC} \triangleq Name \to POSet(L_{FC}) \tag{3.38}$$

$$Run_{FC} \triangleq Name \to Hist_{FC}. \tag{3.39}$$

Notational Conventions. We sometimes write a ms-atom $\langle mul, val, 0\rangle \in L_{new}$ as val^{mul}. Similarly, we sometimes write a ms-atom $\langle val, mul, \delta mul, \delta val, 0\rangle \in L_{per}$ as $val^{mul}[\delta val^{\delta mul}]$. We sometimes elide a change of $\langle \_, id\rangle$; for example, the ms-atom $\langle val, mul, id, id, 0\rangle \in L_{per}$ may be written $val^{mul}[]$. The empty brackets are retained to distinguish this from the shorthand for ms-atoms in L_{new}.

To illustrate these definitions, consider the perturbed ms-atom $(X : \mathbf{N})^*[(X : id)^{id}]$. The original part of this ms-atom represents an arbitrary sequence of messages all containing the same number, represented by X. In the perturbed behavior, the same number of messages are sent—because the change in multiplicity is the identity relation id—and the messages all contain the original value—because the perturbation contains the same variable X, and because the change in value is the identity relation id. In short, the messages represented by this ms-atom are unchanged. Note that the ms-atom $(X : \mathbf{N})^*[id^{id}]$ has the same meaning.

As another example, consider the ms-atom $X : \mathbf{N}[Y : \top_{\Delta V}]$. The original part of this ms-atom represents a single message containing a number, represented by X. In the perturbed behavior, the multiplicity is unchanged—because the change in multiplicity is the identity relation id, which is elided—but the value changes arbitrarily and is now represented by Y. As a variation on this, consider the ms-atom $X : \mathbf{N}[X : \top_{\Delta V}]$. Although the abstract part of the perturbation to the value allows an arbitrary change, the symbolic part shows that the value is still represented by X, so this ms-atom has the same meaning as $X : \mathbf{N}[X : id]$.

As a final example, consider the ms-atom $\mathbf{N}[id^{?_\Delta}]$. The original part of this ms-atom represents a single message containing a number. When failures occur, the data in the message is unchanged—because the change in value is the identity relation id—but the number of messages might change to zero or remain at one, since $[\![?_\Delta]\!]_{\Delta AVal}$ contains the pairs $\langle 1, 0\rangle$ and $\langle 1, 1\rangle$. In short, there is a possibility that the original behavior is unchanged, but there is also a possibility that no message is sent.

3.4.1 Running Example

The replicated pipeline's behavior in failure scenario fs_{OK} (recall that fs_{OK} is defined on page 50) is represented by almost the same run as in Figure 2.2; the only difference is that each ms-atom val^{mul} is replaced with $val^{mul}[]$. In other words, all of the perturbations are the identity perturbation.

The system's behavior in failure scenario fs_1 is represented by the run in Figure 3.7.

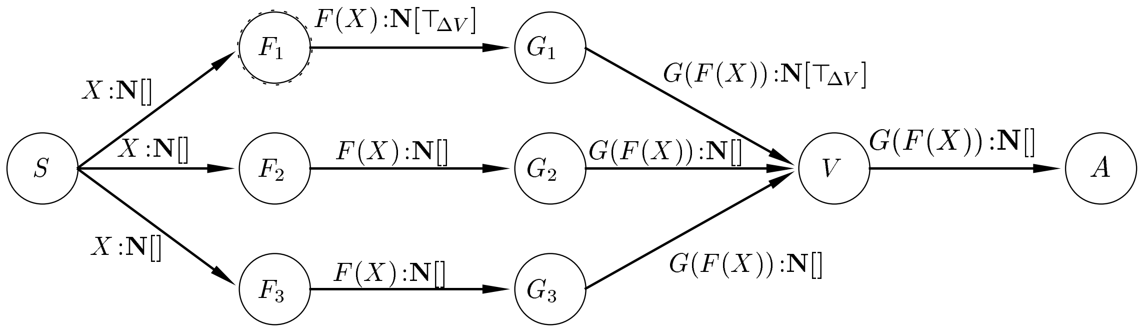

Figure 3.7: Run for running example when component F_1 fails.

3.5 Perturbational Framework: Representation of Components

Components are represented by input-output functions over the extended type L_{FC} of ms-atoms. In addition to the familiar sanity requirement of tag-uniformity, we impose a sanity requirement analogous to $origIndep_C$ in the definition (3.20) of processes. Thus,

$$IOF_{FC} \triangleq \{f \in Fail \rightharpoonup (Hist_{FC} \to Hist_{FC}) \mid \begin{array}{l} \wedge\ tagUniform_{FC}(f) \\ \wedge\ origIndep(f)\}, \end{array} \tag{3.40}$$

where $tagUniform_{FC}(f)$ ensures that renaming of tags in the argument of f causes no change in the output of f except possibly renaming of tags, and *origIndep* ensures that for all $fail \in dom(f)$, the original part of $f(fail)$'s output (i.e., the part ignoring changes and new outputs) depends only on the original part of $f(fail)$'s input. Note that no analogue of *consistent* is needed, because input-output functions are not input-restricted: they are total functions. Formally, the definition of $tagUniform_{FC}$ is definition (2.45) of *tagUniform* with *Hist* replaced with $Hist_{FC}$:

$$tagUniform_{FC}(f) \triangleq (\forall in_1, in_2 \in Hist_{FC} : in_1 =_{Hist_{FC}} in_2 \Rightarrow f(in_1) =_{Hist_{FC}} f(in_2)), \tag{3.41}$$

where $=_{Hist_{FC}}$ denotes equality of histories up to renaming of tags; the definition of $=_{Hist_{FC}}$ is similar to the definition of $=_{Hist}$.

The definition of *origIndep* uses a function $orig \in Set(L_{FC}) \to Set(L)$ that projects the original part of a set of ms-atoms. Roughly, $orig(S)$ is obtained from S by eliminating the perturbations from the perturbed ms-atoms and dropping the new ms-atoms entirely. The only technicality is that retagging may be necessary to avoid "collisions". This is similar to the retagging needed in the definition of $\overline{apOp}$ in Section 2.3.2. Here, a collision occurs if the set S of ms-atoms contains two elements that differ only in their perturbations; eliminating the perturbations would cause those two elements to appear identical. To avoid this collision of identities, we would change the tag in one of those ms-atoms. Using *orig*, the definition of *origIndep* is straightforward:

$$\begin{array}{l} origIndep(f) \triangleq (\forall fail \in dom(f) : (\forall in_1, in_2 \in Hist_{FC} : \\ \qquad\qquad \overline{orig}(in_1) =_{Hist} \overline{orig}(in_2) \\ \qquad\qquad \Rightarrow \overline{orig}(f(fail)(in_1)) =_{Hist} \overline{orig}(f(fail)(in_2)))), \end{array} \tag{3.42}$$

where $\overline{orig} \in Hist_{FC} \to Hist$ is the extension of *orig* from $Set(L_{FC})$ to $Hist_{FC}$.

For future reference, we remark that there is a family of bijections naturally associated with *orig*. In particular, for $S \in Set(L_{FC})$, $b_{orig}(S)$ is a bijection from $S \cap L_{per}$ to $orig(S)$ that preserves values and multiplicities; in other words, $b_{orig}(S)$ satisfies

$$\begin{array}{l} \wedge\ b_{orig}(S) \in (S \cap L_{per}) \xrightarrow{\text{biject}} orig(S) \\ \wedge\ (\forall \ell \in S \cap L_{per} : \pi_1(b_{orig}(S)(\ell)) = \pi_1(\ell) \wedge \pi_2(b_{orig}(S)(\ell)) = \pi_2(\ell)), \end{array} \tag{3.43}$$

where for sets S and T, $S \xrightarrow{\text{biject}} T$ is the set of bijections from S to T. This family of bijections will be convenient for formulating the semantics in Section 3.5.2.

The behavior of a system $nf \in \mathit{Name} \to \mathit{Process}_{FC}$ is represented by the run $\mathit{run}_{FC}(nf)(fs) = \mathrm{lfp}(\mathit{step}_F(nf, fs))$, if it exists. The equality $=_{\mathit{Run}_{FC}}$ on Run_{FC} is the pointwise extension of the equality $=_{\mathit{Hist}_{FC}}$ on Hist_{FC}. Thus, a fixed-point of $\mathit{step}_F(nf, fs)$ is an element r of Run_{FC} satisfying $\mathit{step}_F(nf, fs)(r) =_{\mathit{Run}_{FC}} r$.

3.5.1 Running Example.

The fault-tolerance requirement for the replicated pipeline is the same as in Section 3.1.3. In the perturbational model, the fault-tolerance requirement still has the form (3.17), but b_0 is defined in terms of perturbations. In particular, $b_0(fs, r) = \overline{\mathit{unchanged}}(r(A))$, where the predicate $\overline{\mathit{unchanged}}$ on histories is the pointwise extension of the predicate *unchanged* on posets of ms-atoms. Roughly, the predicate *unchanged* on posets of ms-atoms asserts that the poset is totally-ordered and contains no new ms-atoms, and that the perturbations to the value and multiplicity in each perturbed ms-atom in the history are unchanged. A perturbation is unchanged if the abstract part of the perturbation is *id* and the symbolic part of the perturbation does not "force" the value to change, i.e., the symbolic value either contains a wildcard or contains all of the symbolic values in the original value. The condition that the poset be totally ordered assures that messages occur in the same order in the original and perturbed computations. Formally,

$$\begin{aligned} \mathit{unchanged}(\langle S, \prec\rangle) \triangleq\ & \wedge\ \mathit{totalOrd}(\langle S, \prec\rangle) \\ & \wedge\ (S \cap L_{new}) = \emptyset \\ & \wedge\ (\forall \langle \mathit{mul}, \mathit{val}, \delta \mathit{mul}, \delta \mathit{val}, \mathit{tag}\rangle \in (S \cap L_{per}) : \\ & \qquad \mathit{unchanged}_{\mathit{Val}}(\mathit{val}, \delta \mathit{val}) \wedge \mathit{unchanged}_{\mathit{Val}}(\mathit{mul}, \delta \mathit{mul})) \end{aligned} \tag{3.44}$$

where

$$\mathit{totalOrd}(\langle S, \prec\rangle) \triangleq (\forall x, y \in S : x = y \vee x \prec y \vee y \prec x) \tag{3.45}$$

and for $\mathit{val} \in \mathit{Val}$ and $\delta \mathit{val} \in \Delta \mathit{Val}$,

$$\begin{aligned} \mathit{unchanged}_{\mathit{Val}}(\mathit{val}, \delta \mathit{val}) \triangleq\ & \wedge\ \overline{\pi_2}(\delta \mathit{val}) = \{\mathit{id}\} \\ & \wedge\ \ \vee\ \_ \in \overline{\pi_1}(\delta \mathit{val}) \\ & \qquad \vee\ \overline{\pi_1}(\mathit{val}) \subseteq \overline{\pi_1}(\delta \mathit{val}). \end{aligned} \tag{3.46}$$

The meaning of *unchanged* is discussed further in Section 3.5.2.

Definitions of Src_{FC} and Act_{FC}. Source S ignores its inputs and doesn't fail, so the input-output function for it is nearly the same as (2.50); the only modification to the source's outputs is that they now contain id, indicating that they are unchanged:

$$\begin{aligned} Src_{FC}(dests) = (\lambda fail\!:\!\{OK\}.\ (\lambda h\!:\!Hist_{FC}.\ (\lambda x\!:\!Name.\ & \\ & \mathbf{if}\ x \in dests\ \mathbf{then}\ \langle\{\langle 1, X\!:\!\mathbf{N}, \_\!:id, \_\!:id, 0\rangle\}, \emptyset\rangle \\ & \mathbf{else}\ \langle\emptyset,\emptyset\rangle))). \end{aligned} \tag{3.47}$$

Actuator A produces no outputs, so the input-output function for it is simply

$$Act_{FC} = (\lambda fail\!:\!\{OK\}.\ (\lambda h\!:\!Hist_{FC}.\ (\lambda x\!:\!Name.\ \langle\emptyset,\emptyset\rangle))). \tag{3.48}$$

Definition of $Comp_{FC}$. Processors F_1–F_3 and G_1–G_3 are represented by appropriate instances of $Comp_{FC}$. Informally, for $src, dest \in Name$ and $op \in Sym$, $Comp_{FC}(src, dest, op)$ normally applies operator op to each input from src and sends the results to $dest$. If a "value failure" occurs (i.e., $fail = valFail$), then the perturbations to those outputs are $\top_{\Delta V}$.

The application of an operator to a perturbed or new value is handled by the function $apOp_F$. A "perturbed or new value" is represented by an element of $(Val \times \Delta Val) \cup Val$; elements of $Val \times \Delta Val$ correspond to values in perturbed ms-atoms, and elements of Val to values in new ms-atoms. For $op \in Sym$, $aval \in AVal$, and $x \in Val \cup Val \times \Delta Val$, $apOp_F(OK)(op, aval)(x)$ is the result of applying the operator op to each symbolic value in x, provided the associated abstract value is $aval$; as in definition (2.53) of $apOp$, $aval$ can be thought of as representing the domain and range of op. Formally, $apOp_F(OK)$ is defined by

$$\begin{aligned} & apOp_F(OK)(op, aval)(x) \triangleq \\ & \quad \mathbf{match}\ x\ \mathbf{with} \\ & \quad |\ \langle val, \delta val\rangle \to \langle apOp(op, aval)(val), apOp_\Delta(op, aval)(val, \delta val)\rangle \\ & \quad |\ val \to apOp(op, aval)(val) \end{aligned} \tag{3.49}$$

where $apOp(op, aval)$ is defined in (2.53), and $apOp_\Delta$ is defined similarly by

$$\begin{aligned} apOp_\Delta(op, aval)(val, \delta val) = \bigcup_{\langle s, \delta a\rangle \in \delta val} \{ & \mathbf{if}\ \pi_2(val) = \{aval\} \wedge \delta a = id\ \mathbf{then} \\ & \quad \mathbf{if}\ s = \_\ \mathbf{then}\ \langle \_, id\rangle\ \mathbf{else}\ \langle op(s), id\rangle \\ & \mathbf{else}\ \langle \_, \top_{\Delta V}\rangle\}. \end{aligned} \tag{3.50}$$

Behavior of processors that suffer value failures is described using $apOp_F(valFail)$ rather than $apOp_F(OK)$. For $op \in Sym$, $aval \in AVal$, and $x \in Val \cup (Val \times \Delta Val)$,

$$apOp_F(valFail)(op, aval)(x)$$

is the result of applying the operator op to each symbolic value in the "original part" of x, (i.e., the first component of a perturbed value, and no part of a new value) to obtain the original part of the output. The perturbation or new value in the output is simply "top", representing arbitrary values. Formally,

$$\begin{aligned} apOp_F(valFail)(op, aval)(x) \triangleq\ & \textbf{match } x \textbf{ with} \\ & \mid \langle val, \delta val\rangle \to \langle apOp(op, aval)(val), \{\langle \_, \top_{\Delta V}\rangle\}\rangle \\ & \mid val \to \{\langle \_, \top_V\rangle\}. \end{aligned} \tag{3.51}$$

Given the definition of $apOp_F$, the definition of $Comp_{FC}$ is simple:

$$\begin{aligned} Comp_{FC}(src, dest, op) = (\lambda fail\!:\!\{OK, valFail\}.\ (\lambda h\!:\!Hist_{FC}.\ (\lambda x\!:\!Name.\ & \\ \textbf{if } x = dest \textbf{ then } \overline{apOp_F(fail)(op, \mathbf{N})}(h(src)) & \\ \textbf{else } \langle\emptyset, \emptyset\rangle))), & \end{aligned} \tag{3.52}$$

where $\overline{apOp_F(fail)(op, aval)}$ is the extension of $apOp_F(fail)(op, aval)$ to posets of ms-atoms: the extension is done by letting $apOp_F$ operate on the value and perturbation in each perturbed ms-atom and on the value in each new ms-atom. Retagging may be necessary in the extension to avoid "collisions".

Definition of $Voter_{FC}$. The voter is represented by an input-output function that tallies the original ballots (i.e., ignoring changes and new ms-atoms) to obtain an original result t_o, then tallies the changed and new ballots to obtain a perturbed result t_p, then compares these two results to determine the perturbation in its output. The function *tally* defined in (2.56) is used in both cases to tally the ballots. Recall that the first argument of *tally* is a function used to extract ballots from a poset of ms-atoms. Ballots based on the original parts of the input ms-atoms are extracted with the function $ballot_o = ballot \circ orig$, where *ballot* is defined in (2.55) and *orig* is defined following (3.42). Ballots reflecting perturbations and new ms-atoms in the input are extracted with the function $ballot_p$ defined in Figure 3.8. As in definition (2.55) of *ballot*, we approximate rather than accumulate sets of possibilities in the definition of $ballot_p$.

The definition of $Voter_{FC}$ appears in Figure 3.9. If the input history contains no perturbed ms-atom from some component in *src*, then there is no point in tallying the original ballots, so we simply let t_o equal $\perp$. Similarly, if the input history contains no perturbed or new ms-atom from some component in *src*, then there is no point in tallying the perturbed/new ballots, so we simply let t_p equal $\perp$. It follows from these comments (and the fact that *tally* never returns $\perp$) that if $t_p = \perp$, then $t_o = \perp$; this justifies the comment "this can't happen" in the definition of $Voter_{FC}$.

$$
\begin{array}{l}
ballot_p(\langle S, \prec_S \rangle) = \textbf{match } S \textbf{ with} \\
\quad \mid \{\langle mul, val, \delta mul, \delta val, tag \rangle\} \rightarrow \\
\qquad \textbf{let } mul' = \textbf{if } \pi_2(\delta mul) = \{id\} \textbf{ then } mul \textbf{ else } \{\langle \_, * \rangle\} \\
\qquad \textbf{in match } \delta val \textbf{ with} \\
\qquad\quad \mid \{s : \delta a\} \rightarrow \textbf{if } \pi_2(val) = \{aval\} \wedge \delta a = id \textbf{ then} \langle mul', s : aval \rangle \\
\qquad\qquad\qquad \textbf{else } \langle mul', s : \top_V \rangle \\
\qquad\quad \mid \_ \rightarrow \text{(* approximate *)} \\
\qquad\qquad \langle mul', \_ : \top_V \rangle \\
\quad \mid \{\langle mul, val, tag \rangle\} \rightarrow \textbf{match } val \textbf{ with} \\
\qquad\qquad\qquad \mid \{s : a\} \rightarrow \langle mul, s : a \rangle \\
\qquad\qquad\qquad \mid \_ \rightarrow \text{(* approximate *)} \\
\qquad\qquad\qquad\quad \langle mul, \_ : \top_V \rangle \\
\quad \mid \_ \rightarrow \text{(* approximate *)} \\
\qquad \langle \{\_ : *\}, \_ : \top_V \rangle
\end{array}
$$

Figure 3.8: Definition of $ballot_p$.

$$
\begin{aligned}
&\mathit{Voter}_{FC}(\mathit{srcs}, \mathit{dest}, \mathit{aval}) = \\
&\quad (\lambda \mathit{fail}\!:\!\{\mathit{OK}\}.\ (\lambda h\!:\!\mathit{Hist}_{FC}.\ (\lambda x\!:\!\mathit{Name}. \\
&\qquad \mathbf{if}\ x \neq \mathit{dest}\ \mathbf{then}\ \langle \emptyset, \emptyset \rangle \\
&\qquad \mathbf{else\ let}\ t_o = \mathbf{if}\ (\exists \mathit{src} \in \mathit{srcs} : (\pi_1(h(\mathit{src})) \cap L_{per}) = \emptyset)\ \mathbf{then}\ \bot \\
&\qquad\qquad\qquad \mathbf{else}\ \mathit{tally}(\mathit{ballot}_o, \mathit{srcs}, \mathit{aval}, h) \\
&\qquad\quad \mathbf{in\ let}\ t_p = \mathbf{if}\ (\exists \mathit{src} \in \mathit{srcs} : \pi_1(h(\mathit{src})) = \emptyset)\ \mathbf{then}\ \bot \\
&\qquad\qquad\qquad\quad \mathbf{else}\ \mathit{tally}(\mathit{ballot}_p, \mathit{srcs}, \mathit{aval}, h) \\
&\qquad\quad \mathbf{in\ match}\langle t_o, t_p \rangle\ \mathbf{with} \\
&\qquad\qquad \mid \langle \bot, \bot \rangle \to \langle \emptyset, \emptyset \rangle \\
&\qquad\qquad \mid \langle \langle \mathit{mul}, \mathit{val} \rangle, \bot \rangle \to (\text{* this can't happen *}) \\
&\qquad\qquad\qquad\qquad\qquad\quad \langle \emptyset, \emptyset \rangle \\
&\qquad\qquad \mid \langle \bot, \langle \mathit{mul}, \mathit{val} \rangle \rangle \to \langle \{ \langle \mathit{mul}, \{\mathit{val}\}, 0 \rangle \}, \emptyset \rangle \\
&\qquad\qquad \mid \langle \langle \mathit{mul}, \mathit{val} \rangle, \langle \mathit{mul}', \mathit{val}' \rangle \rangle \to \\
&\qquad\qquad\quad \mathbf{let}\ \delta \mathit{val} = \mathbf{if}\ \pi_1(\mathit{val}) = \pi_1(\mathit{val}') \land \pi_1(\mathit{val}) \neq \_\ \mathbf{then}\ \langle \pi_1(\mathit{val}), \mathit{id} \rangle \\
&\qquad\qquad\qquad\qquad\quad \mathbf{else}\ \langle \_, \top_{\Delta V} \rangle \\
&\qquad\qquad\quad \mathbf{in\ let}\ \delta \mathit{mul} = \mathbf{if}\ \mathit{definite}(\mathit{mul}) \land \mathit{definite}(\mathit{mul}')\ \mathbf{then}\ \langle \_, \mathit{id} \rangle\ \mathbf{else}\ \langle \_, ?_{\Delta} \rangle \\
&\qquad\qquad\quad \mathbf{in}\ \langle \{ \langle \mathit{mul}, \{\mathit{val}\}, \{\delta \mathit{mul}\}, \{\delta \mathit{val}\}, 0 \rangle \}, \emptyset \rangle))),
\end{aligned}
\tag{3.53}
$$

Figure 3.9: Definition of Voter_{FC}.

3.5.2 Semantics

Semantics of Posets of ms-atoms. The definition of $[\![]\!]_{POSet(L_{FC})}$ is a straightforward extension of definition (2.65) of $[\![]\!]_{POSet(L)}$. For $\rho \in \mathit{interp}(\mathit{Sym})$,

$$\begin{aligned}[\![\langle S, \prec\rangle]\!]^{\rho}_{POSet(L_{FC})} \triangleq \{&\sigma \in \mathit{Seq}(\mathit{CVal}) \times \mathit{Seq}(\mathit{CVal}) \mid \\ &(\exists g \in \mathit{dom}(\sigma) \xrightarrow{\text{onto}} S \cap L_{per} : (\exists g' \in \mathit{dom}(\sigma') \xrightarrow{\text{onto}} S : \\ &\mathit{compat}^{\rho}_{POSet(L_{FC})}(S, \prec, \sigma, \sigma', g, g')\},\end{aligned} \tag{3.54}$$

where the correspondences g and g' must satisfy conditions related to the original part of S, the new part of S, the perturbations in S, and the ordering $\prec$; these four conditions are formalized as the four conjuncts, respectively, in

$$\begin{aligned}&\mathit{compat}^{\rho}_{POSet(L_{FC})}(S, \prec, \sigma, \sigma', g, g') \triangleq \\ &\quad \wedge\ \mathit{compat}^{\rho}_{POSet(L)}(\mathit{orig}(S), \overline{b_{orig}(S)}(\prec \cap (L_{per} \times L_{per})), \sigma, b_{orig}(S) \circ g) \\ &\quad \wedge\ \mathit{compat}^{\rho}_{POSet(L)}(S \cap L_{new}, \prec \cap (L_{new} \times L_{new}), \sigma', g') \\ &\quad \wedge\ (\forall \ell \in S \cap L_{per} :\ \wedge\ (\forall i \in g^{inv}(\ell) : (\forall i' \in g'^{\,inv}(\ell) : \mathit{compat}^{\rho}_{\Delta Val}(\pi_4(\ell), \sigma[i], \sigma'[i']))) \\ &\qquad\qquad\qquad\qquad \wedge\ \mathit{compat}^{\rho}_{\Delta Val}(\pi_3(\ell), |g^{inv}(\ell)|, |g'^{inv}(\ell)|)) \\ &\quad \wedge\ (\forall \langle \ell_1, \ell_2\rangle \in \prec : g'^{inv}(\ell_1) \prec_{Set(\mathsf{N})} g'^{inv}(\ell_2)),\end{aligned} \tag{3.55}$$

where $b_{orig}(S)$ is defined by (3.43) and $\overline{b_{orig}(S)}$ is the pointwise extension of $b_{orig}(S)$ from $S \cap L_{per}$ to $\mathit{Order}(S \cap L_{per})$, and the predicate used in the third conjunct to check that the perturbations in each ms-atom relate the original values in σ to the perturbed values in σ' is

$$\begin{aligned}\mathit{compat}^{\rho}_{\Delta Val}(\delta val, cv, cv') \triangleq (\exists \langle s, \delta a\rangle \in \delta val :\ &\wedge\ \langle cv, cv'\rangle \in [\![\delta a]\!]_{\Delta AVal} \\ &\wedge\ s = \_ \vee cv' = \overline{\rho}(s)),\end{aligned} \tag{3.56}$$

where $\overline{\rho}$ is defined by (2.61). It is easy to check that $[\![]\!]_{POSet(L_{FC})}$ is independent of tags.

Semantics of Histories. The meaning of histories is a straightforward extension of the meaning of posets of ms-atoms. For $\rho \in \mathit{interp}(\mathit{Sym})$,

$$\begin{aligned}[\![h]\!]^{\rho}_{Hist_{FC}} \triangleq \{&\langle ch, ch'\rangle \in \mathit{CHist} \times \mathit{CHist} \mid \\ &(\forall x \in \mathit{Name} : \langle ch(x), ch'(x)\rangle \in [\![h(x)]\!]^{\rho}_{POSet(L_{FC})})\}.\end{aligned} \tag{3.57}$$

Note that $[\![]\!]^{\rho}_{Hist_{FC}}$ is monotonic in ρ, i.e.,

$$\begin{aligned}&(\forall \rho_1, \rho_2 \in \mathit{interp}(\mathit{Sym}) : (\forall h \in \mathit{Hist}_{FC} : \\ &\quad \rho_1 \leq_{interp} \rho_2 \Rightarrow [\![h]\!]^{\rho_1}_{Hist_{FC}} \subseteq [\![h]\!]^{\rho_2}_{Hist_{FC}}))).\end{aligned} \tag{3.58}$$

Semantics of Input-Output Functions. By analogy with definition (2.68) of $\sqsubset_{IOF}$, we define for $p \in Process_{FC}$, $f \in IOF_{FC}$, $\rho_c \in Interp(Con)$, and $lvar \subseteq Var$,

$$
\begin{aligned}
& p \sqsubset^{\rho_c, lvar}_{IOF_{FC}} f \triangleq \\
& \quad \wedge\ dom(p) = dom(f) \\
& \quad \wedge\ (\forall fail \in dom(p) : (\forall \langle \langle dp, ir \rangle, \langle dp', ir' \rangle \rangle \in p(fail) : \\
& \qquad (\exists g \in (CHist \times CHist) \twoheadrightarrow interp(lvar) : (\forall \rho_e \in Interp(Var \setminus lvar) : \\
& \qquad\quad (\forall in \in Hist_{FC} : (\forall pch \in ir \times ir' : pch \in [\![in]\!]^{\rho_c \cup \rho_e \cup g(pch)}_{Hist_{FC}} \\
& \qquad\qquad \Rightarrow \langle dp(\pi_1(pch)), dp'(\pi_2(pch)) \rangle \in [\![f(fail)(in)]\!]^{\rho_c \cup \rho_e \cup g(pch)}_{Hist_{FC}})))))).
\end{aligned}
\tag{3.59}
$$

Semantics of Systems. For $nf \in Name \rightarrow Process_{FC}$ and $\rho_a \in interp(Con)$, the semantics of the abstract system $\langle nf, \rho_a \rangle$ is given by $\sqsubset_{Sys_{FC}}$, whose definition is the same as definition (2.69) of $\sqsubset_{Sys}$, except with $\sqsubset_{IOF}$ replaced with $\sqsubset_{IOF_{FC}}$.

Semantics of Runs. By analogy with definition (2.71) of $[\![]\!]_{Run}$, we define for $\rho_a \in interp(Con)$,

$$
\begin{aligned}
[\![r]\!]^{\rho_a}_{Run_{FC}} \triangleq \{ & \langle cr_1, cr_2 \rangle \in CRun \times CRun \mid \\
& (\exists \rho_c \in Interp(Con, \rho_a) : (\exists \rho_v \in Interp(Var) : \\
& \quad (\forall x \in Name : \langle cr_1(x), cr_2(x) \rangle \in [\![r(x)]\!]^{\rho_c \cup \rho_v}_{Hist_{FC}})))\}.
\end{aligned}
\tag{3.60}
$$

Semantics of *unchanged*. Recall that for $h \in Hist_{FC}$, $\overline{unchanged}(h)$ means that the concrete behavior represented by h does not change, i.e., is the same in the failure-free and faulty executions. This idea is formalized by the theorem

$$
\begin{aligned}
& (\forall h \in Hist_{FC} : (\forall \rho \in interp(Sym) : \\
& unchanged(h) \Rightarrow [\![h]\!]^{\rho}_{Hist_{FC}} = \cup_{\sigma \in [\![\overline{orig}(S)]\!]^{\rho}_{Hist}} \{\langle \sigma, \sigma \rangle\}).
\end{aligned}
\tag{3.61}
$$

As mentioned in Section 3.1.3, $\overline{unchanged}$ says nothing about the relative order of messages on different channels; indeed, it couldn't possibly say anything about inter-channel orderings, since $Hist_{FC}$ and $CHist$ do not.

3.5.3 Soundness

Soundness plays the same role for the perturbational framework as for the framework of Chapter 2 (see the comments in the beginning of Section 2.4). The development here is closely analogous to the development in Section 2.4.2.

For convenience, we again consider only finite runs. Let $cruns^{fin}_{FC}(np)(fs)$ contain the pairs of finite runs in $cruns_{FC}(np)(fs)$. Soundness is established by the following theorem, whose proof is almost identical to the proof of Theorem 2.2.

Theorem 3.1. For all $np \in Name \to Process_{FC}$, all $nf \in Name \to IOF_{FC}$, all $\rho_a \in interp(Con)$, all $fs \in FS(np)$, and all $i_{fp} \in \mathsf{N}$, if $np \sqsubset_{Sys_{FC}} \langle nf, \rho_a\rangle$, and if $r = step_F(nf, fs)^{i_{fp}}(\bot_{Run})$ is a fixed-point of $step_F(nf, fs)$, then $cruns^{fin}_{FC}(np) \subseteq [\![r]\!]^{\rho_a}_{Run_{FC}}$.

Proof: Let $\rho_c \in Interp(Con, \rho_a)$ witness the existential quantification in $np(x) \sqsubset_{Sys_{FC}} \langle nf, \rho_a\rangle$. Consider any $fs \in FS(np)$. The definition of $\sqsubset_{IOF_{FC}}$ ensures $fs \in FS(nf)$. Consider any $pcr_0 \in cruns^{fin}_{FC}(np)$. By definition (3.21) of $cruns_{FC}$, there exists $h \in Name \to IRProcess$ such that

$$\begin{aligned}
&\wedge \quad (\forall x \in Name : h(x) \in np(x)(fs(x)))\\
&\wedge \quad (\forall \alpha \in \{1,2\} : \pi_\alpha(pcr_0) \in cruns(\lambda x{:}Name.\ \{\pi_\alpha(h(x))\})).
\end{aligned}$$

Let

$$pcr[i] = \langle step(\pi_1 \circ \pi_1 \circ h)^i(\bot_{CRun}), step(\pi_1 \circ \pi_2 \circ h)^i(\bot_{CRun})\rangle$$

and $r[i] = step_F(nf, fs)^i(\bot_{Run})$. We show by induction that

$$(\forall i \in \mathsf{N} : (\forall x \in Name : pcr[i](x) \in [\![r[i](x)]\!]^{\rho_c \cup \rho_v[i]}_{Hist_{FC}})), \tag{3.62}$$

where $pcr[i](x)$ denotes the pointwise application of $pcr[i]$ to x, and

$$\rho_v[i] = \cup_{x \in Name}\, g(x)(pcr[i](x)),$$

where for all x, $g(x) \in CHist \times CHist \to interp(Var(x))$ is a witness for the existential quantification in $np(x) \sqsubset^{\rho_c, Var(x)}_{IOF} nf(x)$ when the universal quantification over $dom(p)$ is instantiated with $fs(x)$ and the universal quantification over $p(fail)$ is instantiated with $h(x)$.

Base Case. For $i = 0$, the claim is that $(\forall x \in Name : \bot_{CRun}(x) \in [\![\bot_{Run}(x)]\!]^{\rho_c \cup \rho_v[0]}_{Hist})$, which follows easily from the definitions.

Step Case. Using the induction hypothesis and the definition (3.59) of $\sqsubset^{\rho_c, Var(x)}_{IOF_{FC}}$, then simplifying using the definition (2.6) of $step$, we get

$$(\forall x \in Name : pcr[i+1](x) \in [\![nf(x)(fs(x))(r[i](x))]\!]^{\rho_c \cup \rho_v[i]}_{Hist_{FC}}). \tag{3.63}$$

Monotonicity of all the $\pi_1(\pi_1(h(x)))$ and all the $\pi_1(\pi_2(h(x)))$ imply

$$\pi_1(pcr[i]) \leq_{CRun} \pi_1(pcr[i+1]) \ \wedge\ \pi_2(pcr[i]) \leq_{CRun} \pi_2(pcr[i+1]).$$

Monotonicity of all the $g(x)$ then implies $\rho_v[i] \leq_{interp} \rho_v[i+1]$. So, by monotonicity of $[\![]\!]^{\rho}_{Hist_{FC}}$ in ρ (from (3.58)), (3.63) still holds if $\rho_v[i]$ is replaced with $\rho_v[i+1]$. From the resulting equation and definition (3.6) of $step_F$, we get $(\forall x \in Name : pcr[i+1](x) \in [\![r[i+1](x)]\!]^{\rho_c \cup \rho_v[i+1]}_{Hist_{FC}})$. This completes the proof of (3.62).

Finally, we show that (3.62) implies $pcr_0 \in [\![r]\!]^{\rho_c}_{Run}$. Since both runs in pcr_0 are finite, there exists $i_0 \in \mathsf{N}$ such that $(\forall i \geq i_0 : pcr_0 = pcr[i])$. The desired result is obtained by instantiating the universal quantification in (3.62) with $i = \max(i_{fp}, i_0)$. ∎

3.5.4 Termination of Fixed-Point Calculations

The development here is closely analogous to the development in Section 2.4.2.

Orderings. The orderings are defined in exactly the same way as in Chapter 2, except that they are built on an ordering $\preceq_{Set(Seq^2)}$ on sets of pairs of sequences, instead of an ordering $\preceq_{Set(Seq)}$ on sets of sequences. Thus, the pre-order on posets of ms-atoms is

$$\begin{aligned} S_1 \preceq^{\rho_c, lvar}_{POSet(L_{FC})} S_2 \triangleq (\forall \rho_v \in Interp(Var) : (\exists \rho_l \in Interp(lvar) : \\ [\![S_1]\!]^{\rho_c \cup \rho_v}_{POSet(L_{FC})} \preceq_{Set(Seq^2)} [\![S_2]\!]^{\rho_c \cup (\rho_v \oplus \rho_l)}_{POSet(L_{FC})})). \end{aligned} \tag{3.64}$$

where $\preceq_{Set(Seq^2)}$ is defined by

$$S \preceq_{Set(Seq^2)} S' \triangleq (\forall x \in S_1 : (\exists x' \in S_2 : \pi_1(x) \leq_{Seq} \pi_1(x') \wedge \pi_2(x) \leq_{Seq} \pi_2(x'))). \tag{3.65}$$

The definitions of $\leq^{\rho_c, lvar}_{POSet(L_{FC})}$, $\leq^{\rho_c}_{InHist_{FC}}$, $\leq^{\rho_c}_{OutHist_{FC}}$, and $\leq^{\rho_a}_{Run_{FC}}$ are exactly analogous to (2.79), (2.80), (2.81), and (2.83), respectively, with the obvious substitutions: replace L with L_{FC}, *Hist* with $Hist_{FC}$, *etc.* The definition of monotonicity for elements of IOF_{FC} is obtained from the definition (2.82) of monotonicity for *IOF* by quantifying over failures and applying the above substitutions; thus $f \in IOF_{FC}$ is monotonic with respect to $lvar \subseteq Var$ and $\rho_a \in interp(Con)$ iff

$$\begin{aligned} (\forall fail \in dom(f) : (\forall \rho_c \in Interp(Con, \rho_a) : (\forall h_1 \in Hist_{FC} : (\forall h_2 \in Hist_{FC} : \\ h_1 \leq^{\rho_c}_{InHist_{FC}} h_2 \Rightarrow f(fail)(h_1) \leq^{\rho_c, lvar}_{OutHist_{FC}} f(fail)(h_2))))). \end{aligned} \tag{3.66}$$

Monotonicity of $step_F$. As in Section 2.5.1, monotonicity of the "step" function follows from monotonicity of the input-output functions. This is expressed by the following theorem.

Theorem 3.2. For all $nf \in Name \to IOF_{FC}$ and all $\rho_a \in interp(Con)$, if for all $x \in Name$, $nf(x)$ is monotonic with respect to $Var(x)$ and ρ_a, then for all $fs \in FS(nf)$, $step_F(nf, fs)$ is monotonic with respect to $\leq^{\rho_a}_{Run_{FC}}$.

The proof is almost identical to the proof of Theorem 2.4.

The First Step. As in Section 2.5.2, a conjunct must be added to the definition of input-output functions to ensure that $\perp_{Run} \leq^{\rho_a}_{Run_{FC}} step_F(nf, fs)(\perp_{Run})$. By analogy with (2.84), the conjunct that must be added to definition (3.59) of $p \sqsubset^{\rho_c}_{IOF_{FC}} f$ is

$$\begin{aligned}&(\forall fail \in dom(p) : (\forall y \in Name :\\ &\quad noInitialOut_{FC}(p(fail), y) \Rightarrow f(fail)(\perp_{Hist})(y) = \langle\emptyset, \emptyset\rangle)),\end{aligned} \tag{3.67}$$

where

$$\begin{aligned}noInitialOut_{FC}(p, y) \triangleq\ &(\forall \langle\langle dp, ir\rangle, \langle dp', ir'\rangle\rangle \in p : \perp_{CHist} \in ir \wedge \perp_{CHist} \in ir'\\ &\Rightarrow dp(\perp_{CHist})(y) = \varepsilon \wedge dp'(\perp_{CHist})(y) = \varepsilon).\end{aligned} \tag{3.68}$$

This suffices to establish the following theorem.

Theorem 3.3. For all $np \in Name \to Process_{FC}$, all $nf \in Name \to IOF_{FC}$, and all $\rho_a \in interp(Con)$, if $np \sqsubset_{Sys_{FC}} \langle nf, \rho_a\rangle$, then for all $fs \in FS(nf)$, $\perp_{Run} \leq^{\rho_a}_{Run} step_F(nf, fs)(\perp_{Run})$.

The proof is similar to the proof of Theorem 2.5.

Finite Ascending Chains. The comments in Section 2.5.3 apply here as well. An analogue of FAC_n for the perturbational framework is obtained by adding the requirement that $\Delta AVal$ have size at most n.

Chapter 4

Two Classic Problems in Fault-Tolerance

We have presented two analysis frameworks: the non-perturbational framework in Section 3.1, and the perturbational framework in Sections 3.4 and 3.5. To illustrate the use of these two frameworks, we apply each to one classic problem in fault-tolerance.

The non-perturbational framework is applied to a protocol for reliable broadcast that tolerates patterns of crash failures that don't partition the network [HT94, section 6]. This example demonstrates the power of symbolic multiplicities for efficient analysis of systems subject to crash failures. Showing that the protocol satisfies the basic requirements of validity, agreement and integrity [HT94, section 3] is straightforward, because these properties depend mainly on equalities between multiplicities. Showing that the protocol also provides FIFO message delivery requires analyzing inequalities between multiplicities. Invariants are useful for this.

Next, the perturbational framework is applied to a protocol for Byzantine Agreement. A seminal paper by Lamport, Shostak, and Pease defines the problem of Byzantine Agreement and presents two solutions [LSP82]. We analyze the first of those, namely, the Oral Messages algorithm.[1] We use the perturbational framework for this problem, because the correctness requirements are easily expressed in terms of acceptable changes.

One motivation for analyzing a Byzantine Agreement algorithm that has already been proved correct [LM94] is to allow that algorithm to be used as a benchmark for comparison of different verification methods. Also, analysis of a Byzantine Agreement algorithm could

[1]Their second solution, the Signed Messages algorithm, requires digital signatures and can only be analyzed using the techniques presented in Chapter 5.

provide a starting point for analysis of more complicated systems, such as digital flight control systems [DBC91], in which a Byzantine Agreement algorithm is only one of the fault-tolerance mechanisms.

4.1 Reliable Broadcast

Section 4.1.1 introduces a reliable broadcast protocol and its specification; both are adopted from [HT94]. Section 4.1.2 discusses modeling of crash failures for this protocol and gives an analysis of relationships between multiplicities. This discussion motivates the fault-tolerance requirement in Section 4.1.3 and the definitions of the input-output functions in Section 4.1.4.

4.1.1 Reliable Broadcast Protocol

Consider a system with clients $C_1, \ldots, C_n$ and corresponding servers $S_1, \ldots, S_n$. A function $\mathit{nbrs} \in \mathit{Name} \rightarrow \mathit{Set}(\mathit{Name})$ describes the connectivity of the network. We assume each client can communicate directly only with the corresponding server, so $\mathit{nbrs}(C_i) = \{S_i\}$. A server can communicate directly with its client and other of the servers, so $\mathit{nbrs}(S_i)$ satisfies $\{C_i\} \subseteq \mathit{nbrs}(S_i) \subseteq \{C_i\} \cup \{S_1, \ldots, S_n\} \setminus \{S_i\}$.

Informally, the reliable broadcast protocol in [HT94, section 6.3] is as follows. A client C_i initiates a broadcast of a message by sending the message to its server S_i. When a server receives a message, it checks whether it has received that message before. If so, it ignores the message; if not, it sends the message to all of its neighbors. When a client receives a message from its server, we say it *delivers* that message.

Following [HT94, section 3.1], we assume:

Known-Sender: Each message contains the name of its sender.

Uniqueness: Clients send each message at most once. This is easily implemented by including a unique identifier, such as a sequence number or timestamp, in each message.

The known-sender assumption is captured by having client x send messages with abstract value $\mathit{MF}(x)$ (mnemonic for "<u>m</u>essage <u>f</u>rom x"). The meaning of $\mathit{MF}(x)$ depends on the message format. For example, if messages are tuples containing the sender's name, a sequence number that is an element of N, and data of type D, then

$$[\![\mathit{MF}(x)]\!]_{\mathit{AVal}} = \bigcup_{i \in \mathsf{N}, d \in D} \langle x, i, d \rangle. \tag{4.1}$$

There are two approaches to modeling the second assumption: the unique identifier can be modeled explicitly or implicitly. As an example of explicit modeling, we could make (say) sequence numbers explicit by using abstract values of the form $MF'(x,i)$, where

$$[\![MF'(x,i)]\!]_{AVal} = \bigcup_{d \in D} \langle x, i, d \rangle.$$

Modeling the unique identifier implicitly is preferable because it yields a more general model: it abstracts from particular schemes for generating unique identifiers. The uniqueness assumption is expressed directly by using a different variable to represent each message and asserting, as an invariant, that the values of these variables are distinct. For $x \in \mathit{Name}$, let $\mathit{Var}_M(x) \subseteq \mathit{Var}(x)$ be the variables used to represent messages broadcast by x. The invariant is

$$\begin{aligned} I_{RB}(x) = \{\rho \in \mathit{interp}(\mathit{Var}(x)) \mid & \ \mathbf{let}\ S = \mathit{dom}(\rho) \cap \mathit{Var}_M(x) \\ & \ \mathbf{in}\ (\forall v_1 \in S : (\forall v_2 \in S \setminus \{v_1\} : \rho(v_1) \neq \rho(v_2))))\}. \end{aligned} \tag{4.2}$$

For illustration, consider the system with $n = 3$ and with each server having the other two servers as neighbors. Suppose client C_1 broadcasts a single message $X : MF(C_1)$, and the other clients broadcast no messages. Assuming the clients and servers run the protocol sketched above, the run in Figure 4.1 represents the failure-free behavior of this system. Since multiplicities play a central role in analysis of reliable broadcast, we do not elide any multiplicities in figures in Section 4.1.

Specification. The defining properties of *reliable broadcast* are [HT94, section 3]:

Validity: If a client C_i broadcasts a message m and corresponding server S_i is non-faulty, then C_i eventually delivers m.

Integrity: For each message $m \in [\![MF(x)]\!]_{AVal}$, each client having a non-faulty server delivers m at most once and only if m was previously broadcast by x.

Agreement: If a client having a non-faulty server delivers a message m, then all clients of non-faulty servers eventually deliver m.

FIFO reliable broadcast must also satisfy

FIFO Order: If a client broadcasts a message m before it broadcasts a message m', then no client of a non-faulty server delivers m' unless it has previously delivered m.

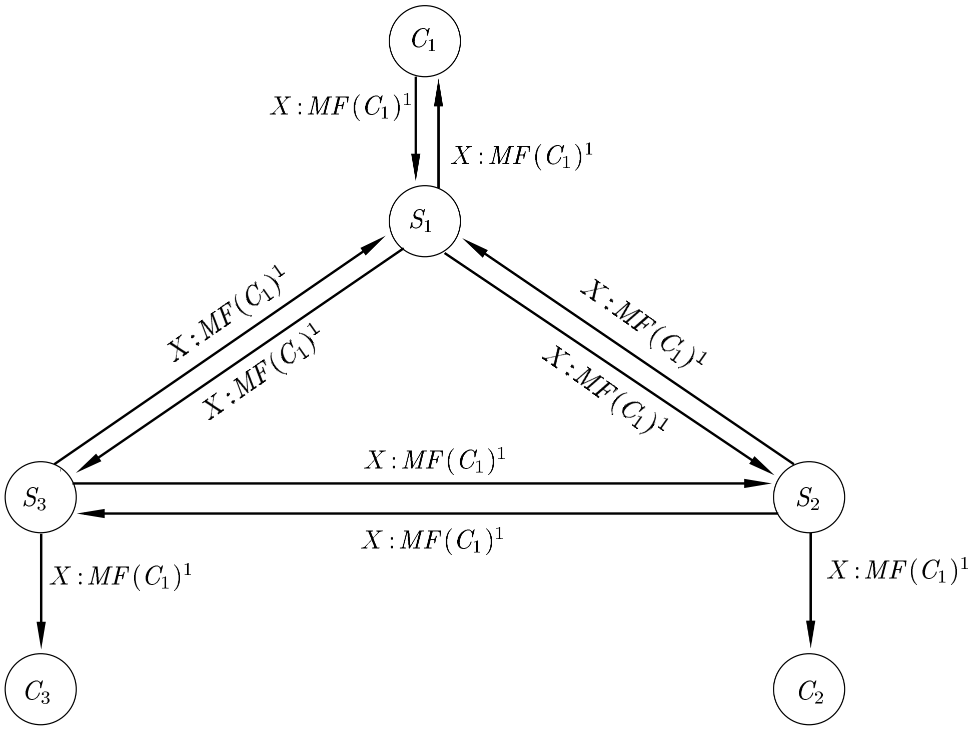

Figure 4.1: Failure-free behavior of the reliable broadcast protocol.

These properties are required to hold in failure scenarios in which servers crash, provided the network remains connected. The network is *connected* in failure scenario *fs* if for each pair $\langle x, y\rangle$ of servers that are non-faulty in *fs*, there is a sequence $\sigma \in \mathit{Seq}(\mathit{Name})$ that starts with x, ends with y, and satisfies

$$(\forall i \in \mathit{dom}(\sigma) \setminus \{0\} : \mathit{fs}(\sigma[i]) = \mathit{OK} \wedge \sigma[i] \in \mathit{nbrs}(\sigma[i-1])). \tag{4.3}$$

Section 4.1.3 formalizes this specification in our framework.

4.1.2 Relationships Between Multiplicities

When analyzing systems that experience crash failures, there is a spectrum of alternatives for the set *Fail*, corresponding to the inclusion of different amounts of timing information. At one end of the spectrum, the timing of the crash can be abstracted completely: $\mathit{crash} \in \mathit{Fail}$ indicates that the component crashes at an unspecified time during execution. To include timing information, a family $\cup_i\{\mathit{crash}_i\}$ of crash failures may be used, where crash_i denotes a crash that occurs at (logical) time i. For example, one might take crash_i to denote a crash that occurs after the component sends i messages. Or, if a synchronous system is being modeled using a distinguished value— called "tick"—to model the passage of time [Bro90, BD92], one might take $crash_i$ to denote a crash that occurs after i ticks.

Abstracting from timing of failures reduces the number of failure scenarios that need to be analyzed and thereby makes the analysis more efficient. Thus, if the resulting approximations are not too coarse (and if the system really is fault-tolerant), then the analysis based on $Fail = \{OK, crash\}$ more efficiently establishes that the system satisfies its fault-tolerance requirement. The analysis of reliable broadcast in this section illustrates the importance of tracking relationships between multiplicities in order to avoid false negatives. We start by sketching an analysis that tracks equalities between multiplicities. That analysis suffices to show that the protocol provides reliable broadcast but does not show that it provides FIFO reliable broadcast. To show that FIFO delivery is provided, the analysis must also track inequalities between multiplicities. We describe two ways of modifying the analysis to do so.

Tracking Equalities Between Multiplicities

Suppose the input-output function representing a server propagates only abstract multiplicities, using the wildcard for all symbolic multiplicities. The effect of a crash is expressed using an abstract multiplicity of ? instead of 1 in the server's outputs, to reflect the possibility of the server crashing before sending the message. More concretely, consider the system with $n = 3$ described above. Consider the failure scenario fs_1 in which only S_1 is faulty (i.e., $fs(S_1) = crash$). Since S_1 might crash at any time, all messages it sends have abstract multiplicity ? instead of 1. The inputs to the non-faulty servers have indefinite multiplicities (i.e., multiplicities not satisfying *definite*, defined by (2.57)), so the outputs of those servers also have indefinite multiplicities. Thus, the result of the analysis is a run just like the one in Figure 4.1, except that on every edge except $\langle C_1, S_1\rangle$, the multiplicity in the ms-atom is ? rather than 1. This is too coarse an approximation of the system's behavior, because this run has interpretations in which C_2 delivers X and C_3 does not (and *vice versa*), and such concrete runs do not satisfy Agreement.

One solution is for the input-output function representing a server to introduce and propagate symbolic multiplicities: if C_2 and C_3 receive X with the same symbolic multiplicity, then Agreement is ensured. An input-output function *server* that does this works roughly as follows (see Section 4.1.4 for details). Let $s_{rcv} \in SVal$ be the "symbolic maximum" of the multiplicities with which the server received a message m; for example, if a message is received with multiplicity X :? from one source and with multiplicity Y :? from another, then s_{rcv} is the symbolic value $\max(X, Y)$. If $fail = OK$, *server* outputs message m with symbolic multiplicity s_{rcv}. If $fail = crash$, *server* introduces for message m a variable v whose value indicates whether that server crashed before outputting that message; m is output with symbolic multiplicity $\min(v, s_{rcv})$, since a server outputs a message if it receives the message and does not crash too soon. Input-output function *server* uses the following naming scheme for the variables denoted above by v: the value (zero or one) of $c.i.x.y \in Var(x)$ indicates whether server x crashes before it can relay to component y the i'th message broadcast by client c. We use 0-based indexing, so the first message broadcast by a client corresponds to $i = 0$.

We illustrate the analysis for this method of modeling the system by using the same system and same failure scenario as above. Let nf_{RB} be the mapping from *Name* to $Process_F$ for this system, using the input-output functions in Section 4.1.4. Figure 4.2 shows the run $step_F(nf_{RB}, fs_1)^3(\bot_{Run})$, corresponding to a partial execution of the protocol; in other words, the fixed-point has not yet been reached. We see in this figure that S_1 has received message X with multiplicity 1 and sent X to each of its neighbors y with symbolic multiplicity $\min(C_1.0.S_1.y, \max(1))$. Since $C_1.0.S_1.y$ is zero or one, this symbolic multiplicity simplifies to $C_1.0.S_1.y$, as shown in the figure. Servers S_2 and S_3 have received messages from S_1 and forwarded them to their neighbors. In the next step, S_2 and S_3 would each output X with the maximum of the two symbolic multiplicities with which they received it.[2] That step yields the fixed-point, which is shown in Figure 4.3. Clients C_2 and C_3 of the non-faulty servers deliver X with the same symbolic multiplicity, so Agreement is satisfied.

Tracking Inequalities Between Multiplicities

The analysis just described suffices to show that the protocol sketched in Section 4.1.1 provides reliable broadcast.[3] However, the analysis is too weak to show that the protocol provides FIFO delivery, even though it does. For example, suppose C_1 sends two messages X_0 and X_1, in that order. The input-output function described above for servers handles each broadcast message independently, so the result of the analysis is a run similar to the one in Figure 4.3, except that on each edge, the singleton poset $\langle \ell, \emptyset \rangle$ is replaced with the poset $\langle \{\ell_0, \ell_1\}, \emptyset \rangle$, where ℓ_i is ℓ with X replaced with X_i and with $C_1.0.S_1.y$ replaced with

[2]Note that the output of S_1 does not change as a result of receiving the messages from S_2 and S_3, because $\min(C_1.0.S_1.y, \max(1, C_1.0.S_1.S_2, C_1.0.S_1.S_3))$ simplifies to $C_1.0.S_1.y$. This simplification relies on the fact that all of the variables in the former expression represent values in $\{0, 1\}$.

[3]Although only the Agreement requirement was discussed, it is easy to see that Validity and Integrity also follow from the results of the analysis.

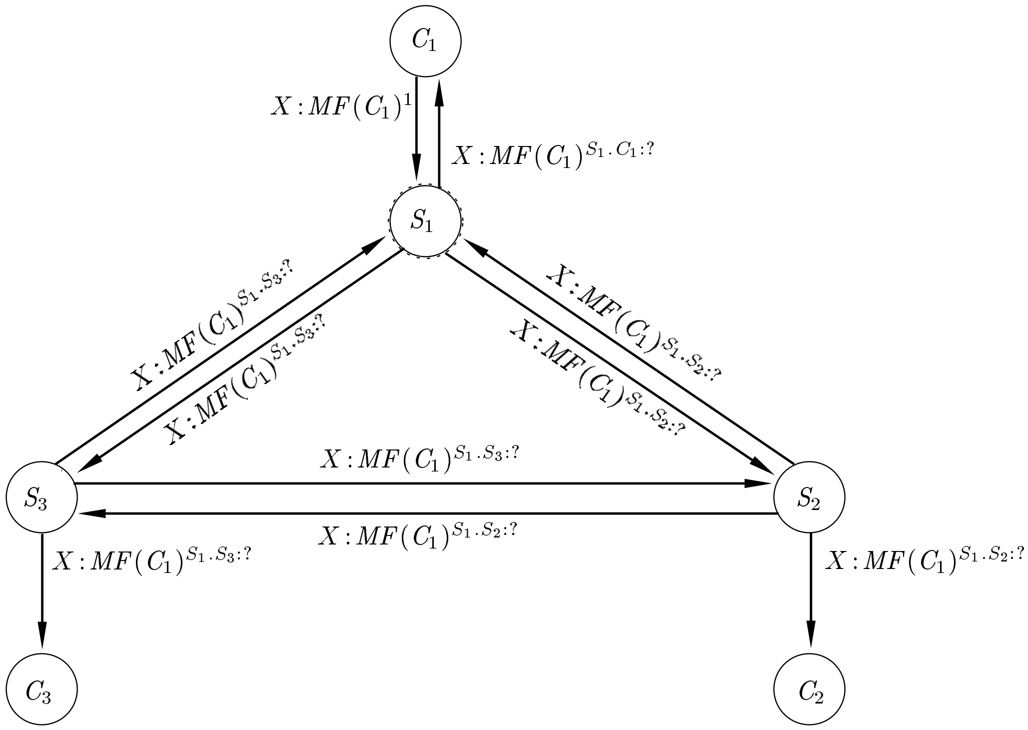

Figure 4.2: Initial behavior of the reliable broadcast protocol when S_1 crashes; more precisely, the run $step_F(nf_{RB}, fs_1)^3(\perp_{Run})$. In the figure, $x.y$ abbreviates $C_1.0.x.y$.

$C_1.i.S_1.y$. For example, the input to C_2 and C_3 (from S_2 and S_3, respectively) is

$$\langle\{X_0 : MF(C_1)^{\max(C_1.0.S_1.S_2, C_1.0.S_1.S_3):?},\ X_1 : MF(C_1)^{\max(C_1.1.S_1.S_2, C_1.1.S_1.S_3):?}\}, \emptyset\rangle. \tag{4.4}$$

This run satisfies validity, integrity, and agreement, but it represents concrete runs that do not satisfy FIFO Order. For example, for an interpretation ρ_{so} such that

$$\begin{aligned} \rho_{so}(C_1.0.S_1.S_2) &= 0 \\ \rho_{so}(C_1.0.S_1.S_3) &= 0 \\ \rho_{so}(C_1.1.S_1.S_2) &= 1, \end{aligned}$$

clients C_2 and C_3 both appear to deliver X_1 but not X_0.

The imprecision in this particular analysis stems from an imprecision in modeling crash failures. No constraints are given between the values of variables used in symbolic multiplicities of different messages, so the output ms-atoms of a faulty server represent executions in which that server fails to send an arbitrary *subset* of its original outputs. In other words, the input-output function sketched above actually represents servers subject to send-omission

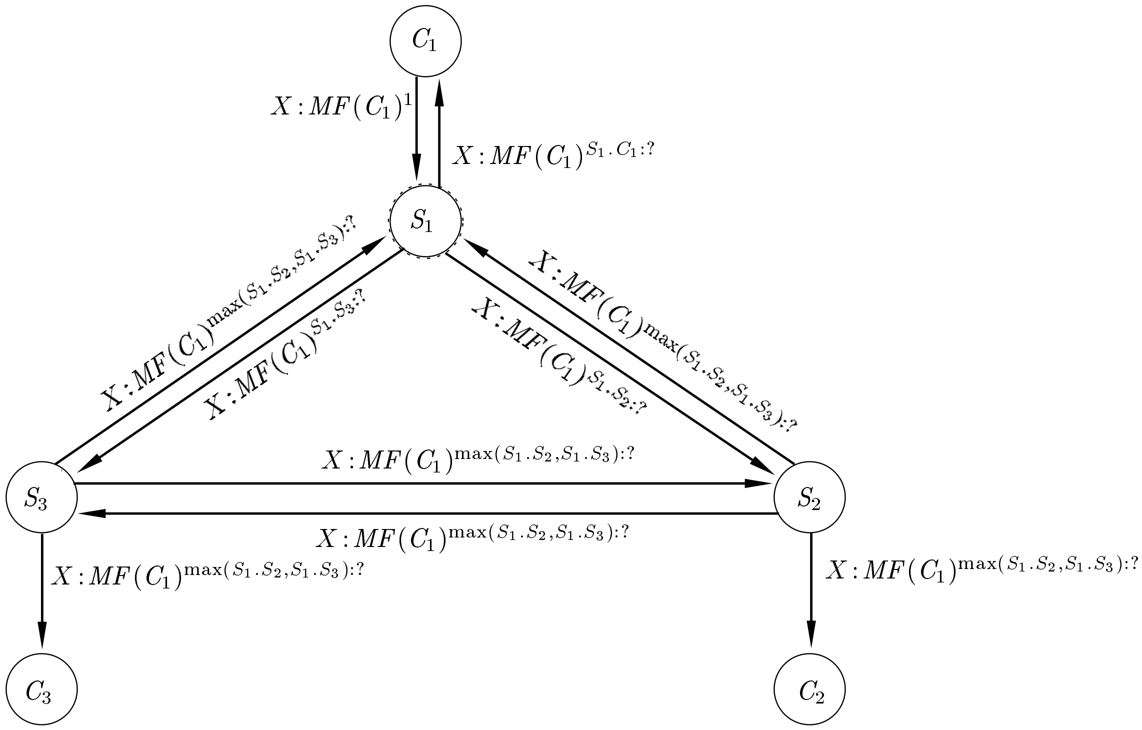

Figure 4.3: Behavior of the reliable broadcast protocol when S_1 crashes, i.e., the run $run_F(nf_{RB})(fs_1)$. In the figure, $x.y$ abbreviates $C_1.0.x.y$.

failures; recall from Section 2.2.3 that send-omission failures cause a component to possibly omit the sending of each message normally produced [HT94, Section 2.3]. Thus, the analysis sketched above shows that the protocol provides reliable broadcast despite send-omission failures. (Crash failures can be regarded as a special case of send-omission failures.)

To establish that the protocol provides FIFO delivery even in the event of crash failures, the analysis must reflect the prefix property of crashes: a component that crashes sends only a prefix of its original outputs (in other words, fails to send an arbitrary *suffix* of its original outputs). This implies that later messages are sent with multiplicities less than or equal to the multiplicities of earlier messages. There are two ways of expressing these inequalities: encode them using combinations of max and min, or express them in an invariant. We discuss each of these two approaches in turn.

Tracking Inequalities using Max and Min. This approach requires changing the meaning of $c.i.x.y$ slightly, so that the value of $\min(\cup_{j \leq i}\{c.j.x.y\})$ indicates whether server x crashes before it can relay to component y the i'th message broadcast by client c. To continue the example in which client C_1 broadcasts X_0 and X_1, the analysis based on this approach yields as the input to C_2 and C_3 the totally-ordered poset (written as a sequence)

$$\begin{aligned}\langle\!\langle &X_0 : MF(C_1)^{\max(C_1.0.S_1.S_2,C_1.0.S_1.S_3):?},\\ &X_1 : MF(C_1)^{\max(\min(C_1.0.S_1.S_2,C_1.1.S_1.S_2)\,\min(C_1.0.S_1.S_3,C_1.1.S_1.S_3)):?}\rangle\!\rangle\end{aligned} \tag{4.5}$$

instead of yielding (4.4). The more precise symbolic multiplicities in the input of S_2 and S_3 have allowed more precise symbolic multiplicities in their outputs and a more precise ordering on the poset. Roughly, these strengthenings are justified by the fact that, for all interpretations of the variables, the multiplicity of X_0 is less than or equal to the multiplicity of X_1; this fact follows easily from monotonicity of max and from the arithmetic fact $\min(i_0, i_1) \leq i_0$. We omit details of this approach, since the approach based on invariants is more elegant and efficient.

Tracking Inequalities using Invariants. This approach retains the original meaning of $c.i.x.y$ and simply asserts that for $i \leq j$, $\rho(c.j.x.y) \leq \rho(c.i.x.y)$. This prohibits interpretations like ρ_{so}. Thus, we strengthen invariant $I_{RB}(x)$ from (4.2) with the conjunct

$$\begin{aligned}(\forall c \in \mathit{Client} : (\forall i, j \in \mathsf{N} : (\forall y \in \mathit{nbrs}(x) :&\\ \{c.i.x.y, c.j.x.y\} \subseteq \mathit{dom}(\rho) \wedge i \leq j \Rightarrow\ &\wedge\ \rho(c.i.x.y) \in \{0,1\}\\ &\wedge\ \rho(c.j.x.y) \leq \rho(c.i.x.y)))),\end{aligned} \tag{4.6}$$

where the set of clients is $\mathit{Client} = \{C_1, \ldots, C_n\}$. To continue again the example in which client C_1 broadcasts X_0 and X_1, this analysis yields the run in Figure 4.4. The inputs to C_2 and C_3 are the same as in (4.4), except that the poset is totally-ordered and the interpretations of the variables are restricted by the invariant (4.6). Details of this approach appear in Section 4.1.4.

4.1.3 Fault-Tolerance Requirement

The fault-tolerance requirement for FIFO reliable broadcast is

$$b(\mathit{fs})(r) = (\text{the network is connected in } \mathit{fs}) \Rightarrow b_0(\mathit{fs}, r),$$

where $b_0(\mathit{fs}, r)$ is the conjunction of the following five predicates, which together express the assumptions (namely, Known-Sender and Uniqueness) and requirements described in Section 4.1.1. Predicates formalizing the assumptions are included here because the predicates formalizing the requirements depend on the assumptions; specifically, if the assumptions did

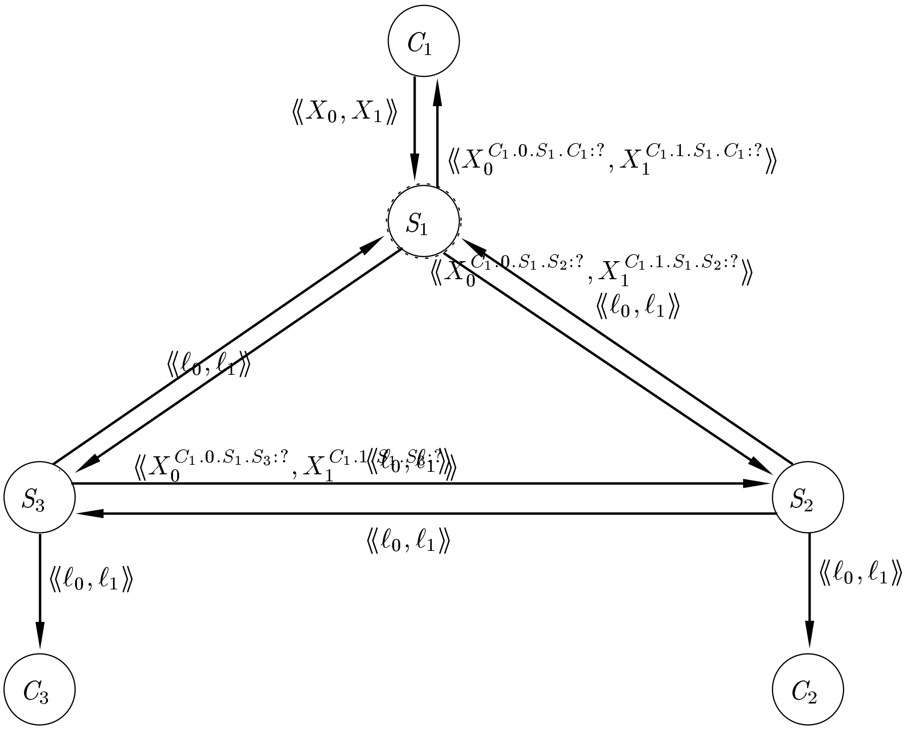

Figure 4.4: Behavior of the reliable broadcast protocol when S_1 crashes. In the figure, totally-ordered posets are written as sequences, the abstract value $MF(C_1)$ is elided, and $\ell_i = X_i\!:\!MF(C_1)^{\max(C_1.i.S_1.S_2, C_1.i.S_1.S_3):?}$.

not hold, the other predicates would not have the intended meaning. For each predicate, we indicate in parentheses the conditions from Section 4.1.1 to which that predicate roughly corresponds.

1. (*Known-Sender*) For each client C_i, every output ms-atom of C_i contains a value of the form $\{v\!:\!MF(C_i)\}$ for some $v \in Var_M(C_i)$.

2. (*Uniqueness*) For each client C_i and each $v \in Var_M(C_i)$, v appears in at most one output ms-atom of C_i, and that ms-atom has multiplicity $\{s\!:\!1\}$ for some $s \in SVal$.

3. (*Integrity*) Every input ms-atom of every client contains a value that occurs in some client's output.

4. (*Agreement*, *Validity*) For each client C_i and each $v \in Var_M(C_i)$ that appears in C_i's output:

(a) If $fs(S_i) = OK$, then for each client C_j such that $fs(S_j) = OK$, C_j delivers v exactly once, i.e., C_j's input contains exactly one ms-atom containing v, and that ms-atom is of the form $v\!:\!MF(C_i)^{s:1}$ for some $s \in SVal$.

(b) If $fs(S_i) = crash$, then there exists $s \in SVal_0$ such that for each client C_j such that $fs(S_j) = OK$, C_j's input contains exactly one ms-atom containing v, and that ms-atom is of the form $v\!:\!MF(C_i)^{s:?}$.

5. (*FIFO Order*) For each client C_i, each $\langle \ell_0, \ell_1 \rangle \in \pi_2(r(S_i)(C_i))$, and each client C_j such that $fs(S_j) = OK$, C_j does not deliver the message represented by ℓ_1 unless it has previously delivered the message represented by ℓ_0. More precisely, let ℓ_0' be the unique ms-atom in $r(C_j)(S_j)$ containing the same variable as ℓ_0, and similarly for ℓ_1'.[4] We require that $\langle \ell_0', \ell_1' \rangle \in \pi_2(r(C_j)(S_j))$ and that the symbolic multiplicities s_0 and s_1 in ℓ_0' and ℓ_1', respectively, satisfy $s_1 \leq_{SV} s_0$, where the relation $\leq_{SV}$ on symbolic values captures the inequalities implied by the invariant (4.6) and by the meaning of min and max:

$$
\begin{array}{rlcl}
s \leq_{SV} s' = & \textbf{match } \langle s, s' \rangle \textbf{ with} & & \\
 & \mid \langle c.i.x.y, c'.i'.x'.y' \rangle & \rightarrow & c' = c \wedge i' \leq i \wedge x' = x \wedge y' = y \\
 & \mid \langle \max(S), \max(S') \rangle & \rightarrow & (\forall s \in S : (\exists s' \in S' : s \leq_{SV} s')) \\
 & \mid \langle \min(S), \min(S') \rangle & \rightarrow & (\forall s' \in S' : (\exists s \in S : s \leq_{SV} s')) \\
 & \mid \_ & \rightarrow & \text{false}
\end{array}
\tag{4.7}
$$

4.1.4 Input-Output Functions

With the invariant defined by (4.2) and (4.6), server S_i is represented by the input-output function $server(S_i, nbrs(S_i))$, where for $me \in Name$ and $nbrs \subseteq Name$, $server(me, nbrs)$, defined in Figure 4.5, works as follows. If the destination x is not a neighbor, then the server sends no messages to x. Predicate *noRepeats* (defined below) checks, roughly, whether the input satisfies the known-sender and uniqueness assumptions. If not, *server* just "gives up" and returns a poset representing arbitrary outputs. Otherwise, the server's outputs are computed as follows. For $n \in nbrs$, $mf(n)$ is just the set of "messages" (more precisely, values representing messages) received from n. Set *msgs* is the set of all messages received by this server. Since the server relays every message, *msgs* is also the set of messages that will appear in the server's outputs. Thus, the set O of output ms-atoms is given by a union

[4] Existence of ℓ_0' and ℓ_1' is guaranteed by the previous requirement.

over $msgs$. In this union, function mul_{RB} (defined below) is used to compute the multiplicity associated with each output message; this is done in the manner sketched in Section 4.1.2. Next, the order $\prec$ in which the server outputs the messages is computed using the following rule: if every neighbor that sent v_2 sent v_1 first (in other words, if v_1 precedes v_2 in the poset representing the input from that neighbor), then this server definitely receives v_1 before v_2, so $v_1 \prec v_2$. Predicate $precede(S, v_1, v_2)$ (defined below) checks whether value v_1 precedes value v_2 in poset S. Finally, the output poset is computed from O and $\prec$. The following paragraphs give details of the auxiliary functions used in the definition of $server$.

$$
\begin{array}{l}
server(me, nbrs) = \\
\quad (\lambda fail\!:\!\{OK, crash\}.\ (\lambda h\!:\!Hist_{FC}.\ (\lambda x\!:\!Name. \\
\quad\quad \textbf{if } x \notin nbrs \textbf{ then } \langle \emptyset, \emptyset \rangle \\
\quad\quad \textbf{else if } \neg(\forall n \in nbrs : noRepeats(\pi_1(h(n)))) \textbf{ then } \langle \{\langle \_\!:\!*, \_\!:\!\top_V, 0\rangle\}, \emptyset\rangle \\
\quad\quad \textbf{else let } mf = (\lambda n\!:\!nbrs.\ \overline{\pi_2}(\pi_1(h(n)))) \\
\quad\quad\quad \textbf{in let } msgs = \cup_{n \in nbrs} mf(n) \\
\quad\quad\quad \textbf{in let } \prec = \{\langle v_1, v_2\rangle \in msgs \times msgs \mid \\
\quad\quad\quad\quad\quad\quad (\forall n \in nbrs : v_2 \in mf(n) \Rightarrow \wedge\, v_1 \in mf(n) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \wedge\, precede(h(n), v_1, v_2))\} \\
\quad\quad\quad \textbf{in let } O = \cup_{v \in msgs}\{\langle mul_{RB}(nbrs, h, fail, v, me, x, msgs, \prec), v, 0\rangle\} \\
\quad\quad\quad \textbf{in } \langle O, \{\ell_1, \ell_2 \in O \times O \mid \pi_2(\ell_1) \prec \pi_2(\ell_2)\}\rangle)))
\end{array}
$$

Figure 4.5: Definition of $server$.

Predicate $noRepeats(S)$ checks that each message in $S \in POSet(L)$ has the required format and is sent at most once:

$$
\begin{aligned}
noRepeats(S) = (\forall \ell \in S : (\exists c \in Name : (\exists v \in Var_M(c) : \\
&\wedge\, \pi_2(\ell) = \{v\!:\!MF(c)\} \wedge \overline{\pi_2}(\pi_1(\ell)) \subseteq \{1, ?\} \\
&\wedge\, (\forall \ell' \in S \setminus \{\ell\} : v \notin \overline{\pi_1}(\pi_2(\ell')))))).
\end{aligned} \tag{4.8}
$$

Predicate $precede(S, v_1, v_2)$ checks whether $v_1 \in Val$ is definitely sent (and hence received) before $v_2 \in Val$ in the sequences of messages represented by $S \in POSet(L)$:

$$
\begin{aligned}
&precede(S, v_1, v_2) = \\
&\quad (\forall \ell_1, \ell_2 \in \pi_1(S) : (\pi_2(\ell_1) = v_1 \wedge \pi_2(\ell_2) = v_2) \Rightarrow \wedge\, \langle \ell_1, \ell_2\rangle \in \pi_2(S) \\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \wedge\, \overline{\pi_1}(\pi_1(\ell_2)) \leq_{Set(SV)} \overline{\pi_1}(\pi_1(\ell_1))),
\end{aligned} \tag{4.9}
$$

where the extension of $\leq_{SV}$ to sets of symbolic values is

$$S \leq_{Set(SV)} S' = (\forall s \in S : (\forall s' \in S' : s \leq_{SV} s')). \tag{4.10}$$

Multiplicity $mul_{RB}(nbrs, h, fail, v, me, x, msgs, \prec)$, defined in Figure 4.6, is the multiplicity with which server $me \in Name$ with neighbors $nbrs \subseteq Name$ sends message $v \in Val$ to neighbor $x \in Name$, given inputs $h \in Hist$ and failure $fail \in \{OK, crash\}$, and with $msgs$ and $\prec$ as in the definition of *server*. It is computed as follows. First, the multiplicity with which the server received value v is computed; the abstract and symbolic parts of this multiplicity are a_{rcv} and s_{rcv}, respectively. If $fail = OK$, then the multiplicity with which v is received is also the multiplicity with which v is relayed by the server. Otherwise (i.e., if the server crashes), the symbolic multiplicity with which v is relayed is the minimum of s_{rcv} and a variable that indicates whether the server crashed before sending v. If the inputs from c are totally ordered, then a variable of the form $c.i.me.x$ is used, as described in Section 4.1.2; otherwise, there is no easy way to associate an index i with the message,[5] a "fresh" variable—in particular, a variable not of that form, whose value is therefore not constrained by (4.6)—is used instead. Function $freshvar(c, v, me, x) \in Var(me)$ is assumed to return such a variable.

Other functions used in the definition of mul_{RB} are as follows. Recall that *apply* was defined following (2.53). For a poset p, $linearize(p)$ returns a sequence that is some linearization of p; in the definition of mul_{RB}, it doesn't matter which linearization is returned, because max is commutative. For a value v, $getClnt(v)$ is c if $\overline{\pi_2}(v) = \{MF(c)\}$ and is undefined otherwise. Note that the check of *noRepeats* in *server* ensures that the application of *getClnt* in mul_{RB} will be defined. Similarly, $getSym(v)$ is s if $\overline{\pi_1}(v) = \{s\}$ and is undefined otherwise. For a totally-ordered poset p, $getIndex(x, p)$ returns the least $i \in \mathsf{N}$ such that $p[i] = x$ (where p is regarded as a sequence), if it exists, and is undefined otherwise.

A simplification routine $simplify \in SVal \rightarrow SVal$ is used to simplify expressions involving max and min. The invariant (4.6) justifies assuming during the simplification that each variable of the form $c.i.x.y$ represents a value in $\{0, 1\}$. Thus, powerful Boolean simplification procedures can be used. In particular, putting the expressions in some canonical form (e.g., disjunctive normal form, or ordered binary decision diagrams) helps ensure termination of the fixed-point iteration. For examples with single failures, the analysis terminates even if

[5]The index set could be generalized to be some partial order $\langle S, \prec_S \rangle$, instead of the natural numbers. The invariant could be extended to require that variables of the form $c.s.x.y$, where $s \in S$, satisfy inequalities corresponding to the partial ordering $\prec_S$. This would give a more precise analysis in cases where the output posets of clients are not totally-ordered.

$$
\begin{array}{l}
\mathit{mul}_{RB}(\mathit{nbrs}, h, \mathit{fail}, v, me, x, \mathit{msgs}, \prec) = \\
\quad \textbf{let } a_{rcv} = \textbf{if } (\exists n \in \mathit{nbrs} : (\exists \ell \in \pi_1(h(n)) : \pi_2(\ell) = v \wedge 1 \in \pi_2(\pi_1(\ell)))) \textbf{ then } 1 \\
\qquad\qquad \textbf{else } ? \\
\quad \textbf{in let } s_{rcv} = \textbf{let } S = \cup_{n \in \mathit{nbrs}} \textbf{ let } S_1 = \{\ell \in \pi_1(h(n)) \mid \pi_2(\ell) = v\} \\
\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in } \cup_{\ell \in S_1} \overline{\pi_1}(\pi_1(\ell)) \\
\qquad\qquad \textbf{in } \mathit{apply}(\max, \mathit{linearize}(\langle S, \emptyset \rangle)) \\
\quad \textbf{in if } \mathit{fail} = \mathit{OK} \textbf{ then } \{\langle \mathit{simplify}(s_{rcv}), a_{rcv} \rangle\} \\
\qquad \textbf{else let } c = \mathit{getClnt}(v) \\
\qquad\quad \textbf{in let } p = \langle \{v' \in \mathit{msgs} \mid \pi_2(v') = \{\mathit{MF}(c)\}\}, \prec \rangle \\
\qquad\quad \textbf{in let } s_F = \textbf{if } \mathit{totalOrd}(p) \textbf{ then} \\
\qquad\qquad\qquad\qquad \textbf{let } i = \mathit{getIndex}(v, p) \\
\qquad\qquad\qquad\qquad \textbf{in } c.i.me.x \\
\qquad\qquad\qquad \textbf{else } \mathit{freshvar}(c, v, me, x) \\
\qquad\quad \textbf{in } \{\langle \mathit{simplify}(\mathit{apply}(\min, \langle\!\langle s_F, s_{rcv} \rangle\!\rangle)), ? \rangle\}
\end{array}
$$

Figure 4.6: Definition of mul_{RB}.

simplify does only trivial simplifications (e.g., $\mathit{simplify}(\max(\max(m, n), m)) = \max(m, n)$). Additional simplifications are needed to analyze failure scenarios involving multiple crashes, because the symbolic multiplicities have a more complicated structure. In particular, for single crashes, they have the form $\max(\cdots)$, as in Figure 4.3; for double crashes, they have the form $\min(\cdots, \max(\cdots), \cdots)$, as in Figure 4.7; for triple crashes, they have the form $\min(\cdots, \max(\cdots, \min(\cdots), \cdots), \cdots)$; and so on. Thus, simplifications involving combinations of min and max are needed.

In principle, this input-output function *server* representing a server can be used with any input-output function representing a client, though for the analysis to be useful, the latter should produce outputs satisfying the Known-Sender and Uniqueness conditions.

4.1.5 Examples

An example of the analysis for a failure scenario involving a single crash appears in Figure 4.4.

For an example involving two crashes, consider a system with $n = 4$, and consider the failure scenario in which S_1 and S_2 crash. The result of the analysis is shown in Figure 4.7.

Note that this run satisfies the fault-tolerance requirement.

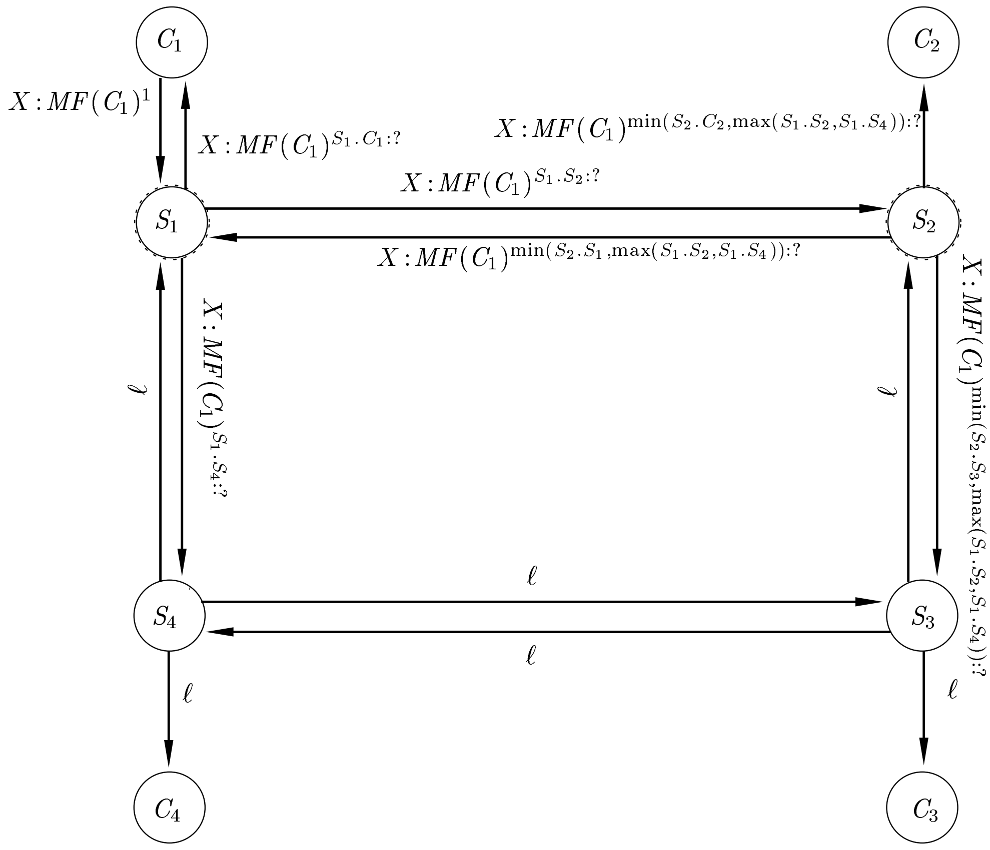

Figure 4.7: Behavior of the reliable broadcast protocol when S_1 and S_2 crash. In the figure, $\ell = X:MF(C_1)^{\max(S_1.S_4,\min(S_2.S_3,S_1.S_2)):?}$.

4.2 Byzantine Agreement

Now consider a system comprising a commander C, lieutenants $L_1, \ldots, L_n$, and armies $A_1, \ldots, A_n$. The goal of a Byzantine Agreement protocol is for the commander to disseminate a command to the lieutenants, so they can then act on this command. A lieutenant L_i acts on (or "decides on", in the terminology of [LSP82]) a command by sending a message containing that command to its army A_i. Thus, the set of components in the system is

$$\mathit{Name} = \{C\} \cup \mathit{Ltnts} \cup (\bigcup_{i \in [1..n]} \{A_i\}) \tag{4.11}$$

where

$$
\begin{aligned}
Ltnts &= \bigcup_{i \in [1..n]} \{L_i\} \\
[i..j] &\triangleq \{k \in \mathsf{N} \mid i \leq k \leq j\}.
\end{aligned}
$$

The Oral Messages algorithm of [LSP82] to solve this problem works under the following assumptions about the underlying communication mechanism:

Integrity: Every message that is sent is delivered correctly.

Known-Sender: The receiver of a message knows who sent it.

Missing-Message-Detection: The absence of a message can be detected.

The first two assumptions are built into our framework: Integrity, because our definition of *step* never removes messages from a run; Known-Sender, because histories classify messages by sender, and input-output functions take histories as arguments. Missing-Message-Detection is not satisfied by our framework but can be encoded using standard techniques [Bro90,BD92]. In particular, we adopt the following convention: when a component in reality omits to send a message, this omission is modeled by sending a distinguished value *tmout* ("timeout"). By this convention, receiving *tmout* corresponds to detecting absence of a message. This convention is used at both the concrete and abstract levels, so we treat timeout as both a concrete value and an abstract value, with $[\![tmout]\!]_{AVal} = \{tmout\}$.

The basic Oral Messages algorithm of [LSP82] further assumes that the commander and lieutenants can communicate with each other directly. We associate with each component $x \in Name$ the set $nbrs(x) \subseteq Name$ of its neighbors, i.e., the set of components with which it can communicate directly. We take:

$$
\begin{aligned}
nbrs(C) &= Ltnts \\
nbrs(L_i) &= \{C, A_i\} \cup Ltnts \setminus \{L_i\} \\
nbrs(A_i) &= \{L_i\}.
\end{aligned}
$$

Fault-Tolerance Requirement. We consider Byzantine failures of the commander and lieutenants. A Byzantine-faulty component may send an arbitrary number of arbitrary values to each of its neighbors. A Byzantine failure is represented by failure $ByzFail \in Fail$. The fault-tolerance requirement is

$$
b(fs)(r) = |\{x \in Name \mid fs(x) \neq OK\}| \leq \lfloor n/3 \rfloor \Rightarrow b_0(fs, r),
$$

where $b_0(fs, r)$ is the conjunction of the following conditions:

1. If the commander is non-faulty (i.e., $fs(C) = OK$), then the inputs of the armies associated with non-faulty lieutenants are unchanged compared to the failure-free behavior, i.e., for each such army A, $unchanged(r(A))$.

2. If the commander is faulty (i.e., $fs(C) = ByzFail$), then all armies associated with non-faulty lieutenants receive the same value, i.e., for some $s \in SVal_0$, for each such army A, $r(A)$ is a singleton poset $\{\ell\}$, and ℓ has original multiplicity "one", unperturbed multiplicity, and perturbed value s:

$$\wedge\, \overline{\pi_2}(\pi_1(\ell)) = \{1\} \wedge unchanged_{Val}(\pi_1(\ell), \pi_3(\ell)) \wedge \overline{\pi_1}(\pi_4(\ell)) = \{s\}. \tag{4.12}$$

This specification has a pleasant locality property: it refers only to the inputs of the armies. In a framework without explicit perturbations, the specification would have to involve the commander's inputs and outputs as well.

The Oral Messages algorithm of [LSP82] is essentially a recursive application of majority voting. The interesting aspects of its behavior can already be seen in the case $n = 3$, which exhibits a single level of recursion plus the base case. So, we take $n = 3$ in the detailed part of the exposition and then sketch the extension to arbitrary n. The algorithm is described in Section 4.2.1. Section 4.2.2 shows the results of the analysis.

4.2.1 Oral Messages Algorithm

We describe the algorithm informally before formalizing it as input-output functions. The commander sends a command, represented by variable $X \in Var(C)$, to each lieutenant. Each lieutenant forwards the value it receives from the commander to the other lieutenants. When a lieutenant has received values from the commander and all of the other lieutenants, it takes the majority of those values using a majority function ϕ_{maj} (*cf.* Section 2.1.3) and sends the result to its army. More precisely, L_i computes $\phi_{maj}(cv_1, cv_2, cv_3)$, where cv_i is the value that L_i received from the commander and for $j \neq i$, cv_j is the value received from L_j. If any of the received values is *tmout*, then some default value is used in its place. The run in Figure 4.8 represents the failure-free behavior of this algorithm.

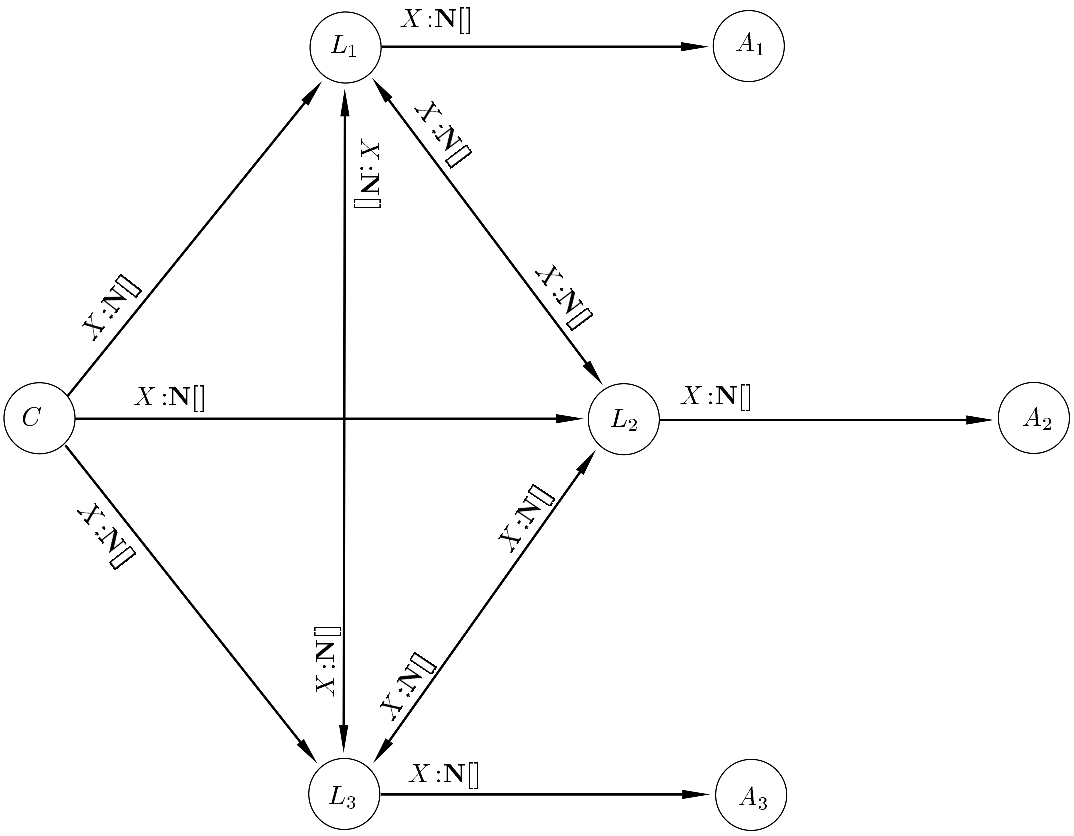

Figure 4.8: Failure-free behavior of the Oral Messages algorithm.

Definition of ***Cmdr*****.** The input-output function representing the commander is $\mathit{Cmdr}(\mathit{Ltnts})$, where for $\mathit{dests} \in \mathit{Set}(\mathit{Name})$,

$$\begin{aligned} \mathit{Cmdr}(\mathit{dests}) = (\lambda \mathit{fail}\!:\!\{\mathit{OK}, \mathit{ByzFail}\}.\ (\lambda h\!:\!\mathit{Hist}.\ (\lambda x\!:\!\mathit{Name}.\ & \\ & \mathbf{if}\ x \notin \mathit{dests}\ \mathbf{then}\ \langle \emptyset, \emptyset \rangle \\ & \mathbf{else\ if}\ \mathit{fail} = \mathit{OK}\ \mathbf{then}\ \langle 1, X\!:\!\mathbf{N}, \mathit{id}, \mathit{id}, 0 \rangle \\ & \quad \mathbf{else}\ \langle 1, X\!:\!\mathbf{N}, \_\!:\top_{\Delta V}, \_\!:+_\Delta, 0 \rangle))), \end{aligned} \tag{4.13}$$

The use of $+_\Delta$ rather than $*_\Delta$ is justified by the convention for modeling Missing-Message-Detection: even a faulty process must send at least a timeout.

Definition of ***Ltnt*****.** The input-output function representing a lieutenant is composed of two main pieces: one that relays the value received from the commander, and one that handles voting.

Relaying is captured by the function *relay* defined in Figure 4.9, where

$$\mathit{arbnew} = \langle \{\top_V{}^*\}, \emptyset \rangle \tag{4.14}$$

For *fail* $\in \{OK, ByzFail\}$, $S \in POSet(L_{FC})$, $var \in Var$, and $aval \in AVal$, the original part of the output of $relay(fail, S, var, aval)$ is determined as follows. If there are no inputs, then there are no outputs. If S is a singleton containing a value of the form $\{s\!:\!aval\}$, then that value is relayed (i.e., included in the output). If neither of those cases applies, then the output is $\{var\!:\!aval\}$. At the concrete level, this third case may correspond to receiving a timeout (or other value not in *aval*) and relaying some default value, or to receiving multiple values in *aval* from the (faulty) commander and relaying the first of them. We don't need to distinguish these two possibilities at the abstract level, because they both arise only if the commander is faulty, in which case it doesn't matter what value is relayed, provided the same value is relayed to all of the other lieutenants. The equality of the values relayed to different lieutenants is reflected by using a local variable *var* (instead of just a wildcard) to represent the relayed value.

Perturbations to the relayed output are determined by the parameter *fail* and by perturbations and new ms-atoms in the input S. For $fail = OK$, then the perturbation to the output is determined roughly as follows: if the inputs are unchanged (i.e., the perturbation is *id*), then so are the outputs; if the inputs do change, then we conservatively assume the relayed value may change to any other value in *aval*, so we take the perturbation to the output to be $aval_{\Delta}$, whose meaning is given by (3.34).

For $fail = ByzFail$, perturbations to the output and new outputs are determined as follows. Arbitrary new outputs may be sent before the original output; these are represented by *arbnew*. The original output value may change arbitrarily, so we take the perturbation to be $\top_{\Delta V}$. Perturbations to the multiplicities are determined by $rly\Delta Mul(fail, S)$, whose definition reflects the convention for modeling Missing-Message-Detection: if the lieutenant definitely receives a message from the commander, then it definitely relays some value (possibly just *tmout*). This is a special case of Broy and Dendorfer's time progress property [BD92].

For $cmdr \in Name$, $ltnts \in Seq(Name)$, $me \in Name$, $var \in Var$, and $army \in Name$, input-output function $Ltnt(cmdr, ltnts, me, var, army)$ represents a lieutenant named *me* with army A in a system in which the commander is *cmdr* and the sequence of lieutenants is *ltnts*; also, when this lieutenant's input from the commander contains multiple symbolic values, this lieutenant uses *var* to represent the value received from the commander and relayed to the other lieutenants. This input-output function works as follows. Function *relay* is used to determine the value received from the commander and relayed to the other lieutenants; this value is contained in the ms-atom in the singleton poset returned by *relay*.

$$
\begin{array}{l}
\mathit{relay}(\mathit{fail}, S, \mathit{var}, \mathit{aval}) = \\
\quad \textbf{match } \pi_1(S) \cap L_{per} \textbf{ with} \\
\quad \mid \emptyset \rightarrow \textbf{if } \mathit{fail} = \mathit{OK} \wedge (\pi_1(S) \cap L_{new}) = \emptyset \textbf{ then } \langle \emptyset, \emptyset \rangle \\
\qquad\quad \textbf{else } \mathit{arbnew} \\
\quad \mid \{\langle \mathit{mul}, \mathit{val}, \delta \mathit{mul}, \delta \mathit{val}, \mathit{tag} \rangle\} \rightarrow \\
\qquad \textbf{let } \mathit{val}_1 = \textbf{match } \mathit{val} \textbf{ with} \\
\qquad\qquad\qquad \mid \{s\!:\!a\} \rightarrow \textbf{if } a = \mathit{aval} \wedge s \neq \_ \textbf{ then } s\!:\!a \textbf{ else } \mathit{var}\!:\!\mathit{aval} \\
\qquad\qquad\qquad \mid \_ \rightarrow \mathit{var}\!:\!\mathit{aval} \\
\qquad \textbf{in let } \delta \mathit{val}_1 = \textbf{if } \mathit{fail} = \mathit{OK} \textbf{ then} \\
\qquad\qquad\qquad\qquad \textbf{if } (\pi_1(S) \cap L_{new}) = \emptyset \wedge \pi_2(\delta \mathit{val}) = \{\mathit{id}\} \textbf{ then} \\
\qquad\qquad\qquad\qquad\quad \{\pi_1(\mathit{val}_1)\!:\!\mathit{id}\} \\
\qquad\qquad\qquad\qquad \textbf{else } \{\_\!:\mathit{aval}_\Delta\} \\
\qquad\qquad\qquad \textbf{else } \{\_\!:\top_{\Delta V}\} \\
\qquad \textbf{in } \langle \{\langle \mathit{rlyMul}(S), \{\mathit{val}_1\}, \mathit{rly\Delta Mul}(\mathit{fail}, S), \delta \mathit{val}_1, \mathit{tag} \rangle\}, \emptyset \rangle \\
\quad \mid \_ \rightarrow \textbf{let } \delta \mathit{val} = \textbf{if } \mathit{fail} = \mathit{OK} \textbf{ then } \mathit{aval}_\Delta \textbf{ else } \top_{\Delta V} \\
\qquad\quad \textbf{in } \langle \{\langle \mathit{rlyMul}(S), \mathit{var}\!:\!\mathit{aval}, \mathit{rly\Delta Mul}(\mathit{fail}, S), \delta \mathit{val}, 0 \rangle\}, \emptyset \rangle
\end{array}
$$

$$
\mathit{rlyMul}(S) = \textbf{if } (\exists \ell \in \pi_1(S) : \mathit{definite}(\pi_1(\ell))) \textbf{ then } \{\_\!:1\} \textbf{ else } \{\_\!:?\}
$$

$$
\begin{array}{l}
\mathit{rly\Delta Mul}(\mathit{fail}, S) = \textbf{if } (\exists \ell \in \pi_1(S) : \mathit{definite}(\pi_1(\ell)) \wedge \pi_2(\pi_3(\ell)) \subseteq \{\mathit{id}, +_\Delta\}) \textbf{ then} \\
\qquad\qquad\qquad\qquad \textbf{if } \mathit{fail} = \mathit{OK} \textbf{ then } \{\_\!:\mathit{id}\} \textbf{ else } \{\_\!:+_\Delta\} \\
\qquad\qquad\qquad \textbf{else if } \mathit{fail} = \mathit{OK} \textbf{ then } \{\_\!:?_\Delta\} \textbf{ else } \{\_\!:*_\Delta\}
\end{array}
$$

Figure 4.9: Definition of *relay*, with two auxiliary functions.

Input-output function $Voter_{FC}$, defined by (3.53), is used to handle voting, with the result of *relay* (representing the value received from the commander) being used as the vote of this lieutenant. The output is then determined by considering the destination. If the lieutenant is non-faulty, it sends its decision—that is, the outcome of the vote—to its army. To other lieutenants, the result of *relay* is always sent. A non-faulty lieutenant sends nothing to the commander, but a faulty lieutenant sends arbitrary messages to the commander.

The input-output function for lieutenant L_i is $Ltnt(C, \langle\!\langle L_i \rangle\!\rangle_{i\in[1..n]}, L_i, X_i, A_i)$, where $X_i \in Var(L_i)$ and

$$
\begin{array}{l}
Ltnt(cmdr, ltnts, me, var, army) = \\
\quad (\lambda fail\!:\!\{OK, ByzFail\}.\ (\lambda h\!:\!Hist.\ (\lambda x\!:\!Name. \\
\qquad \textbf{let } relay_0 = relay(fail, h(cmdr), var, \mathbf{N}) \\
\qquad \textbf{in let } decis = Voter_{FC}(ltnts, army, \mathbf{N})(OK)(h \oplus (\lambda x\!:\!\{me\}.\ relay_0))(army) \\
\qquad \textbf{in if } x = army \textbf{ then} \\
\qquad\qquad\quad \textbf{if } fail = OK \textbf{ then } decis \\
\qquad\qquad\quad \textbf{else if } \pi_1(decis) = \emptyset \textbf{ then } arbnew \\
\qquad\qquad\qquad \textbf{else } \overline{arbchng}(decis) \\
\qquad\quad \textbf{else if } x \in ltnts \setminus \{me\} \textbf{ then } relay_0 \\
\qquad\quad \textbf{else if } x = cmdr \wedge fail \neq OK \textbf{ then } arbnew \\
\qquad\quad \textbf{else } \langle \emptyset, \emptyset \rangle))),
\end{array}
\tag{4.15}
$$

where for $\ell \in L_{FC}$,

$$
\begin{array}{rl}
arbchng(\ell) = & \textbf{match } \ell \textbf{ with} \\
& \mid \langle mul, val, tag \rangle \to \langle *, \top_V, tag \rangle \\
& \mid \langle mul, val, \delta mul, \delta val, tag \rangle \to \langle mul, val, \top_{\Delta V}, +_\Delta, tag \rangle
\end{array}
\tag{4.16}
$$

and $\overline{arbchng}$ is the pointwise extension of *arbchng* from ms-atoms to $POSet(L_{FC})$ (as usual, retagging may be needed in the extension). Recall that $\oplus$ is defined following (2.78).

Definition of ***Army*****.** The input-output function *Army* for an army is equal to Act_{FC}, defined in (3.48).

Extending the Definitions to Arbitrary n**.** For $n > 3$, the Oral Messages algorithm proceeds by recursion. Byzantine Agreements are performed among smaller and smaller sets of lieutenants, until a base case (a singleton set) is reached. The results of the recursive invocations of Byzantine Agreement are repeatedly combined using majority functions (of appropriate arity) to determine the result of the top-level Byzantine Agreement. To prevent confusion between messages generated by different recursive calls to the Byzantine Agreement protocol, each message is tagged with an identifier indicating which invocation of the protocol it belongs to. A simple scheme for choosing these identifiers is described in [LSP82].

Extending the definitions in this section to arbitrary n involves little more than the additional bookkeeping needed to keep track of these identifiers. There is one *caveat*: if a message with uncertain identifier is received from a lieutenant (e.g., if the value is $\top_V$), then as a conservative approximation, we assume this might confuse determination of the values received from that lieutenant in all invocations of the protocol, so we use $\top_V$ for all those values.

4.2.2 Analysis of Perturbed Behavior

For $n = 3$, the algorithm is required to tolerate a Byzantine failure of any one component. We consider first the effects of a lieutenant failure and then turn to the effects of a commander failure.

Lieutenant Failure. Let nf_{BA} be the obvious mapping from $Name$ to $Process_{FC}$ for this example. Let fs_L be the failure scenario in which only L_2 is faulty. Figure 4.10 shows the run $run_{FC}(nf_{BA})(fs_L)$. To reduce clutter, where the posets on the edges in both directions between a pair of components are non-empty (e.g., for components L_1 and L_2), those two edges together are drawn as a single two-headed arrow, and each poset is positioned closer to its source (e.g., $X : \mathbf{N}[]$ is on edge $\langle L_1, L_2 \rangle$). The non-faulty lieutenants receive two unchanged values and one changed value. Voting masks the changed input, leaving the non-faulty lieutenants' outputs unchanged. It is easy to check that this run satisfies the fault-tolerance requirement. The analysis of failure of L_1 or L_3 is a symmetric variant of this analysis.

Comments on the Definition of *Ltnt*. A faulty lieutenant may send arbitrary values at any time, even before it receives or sends messages in the non-faulty execution. In the run $step_F(nf_{BA}, fs_L)(\bot_{Run})$ shown in Figure 4.11, the ms-atoms on the outgoing edges of L_2 represent those outputs; note that these ms-atoms are equal to *arbnew*, defined by (4.14). In the fixed-point shown in Figure 4.10, some of these ms-atoms have been "replaced" by ms-atoms in L_{per} containing arbitrary changes; in other words, messages once represented by the former are later represented by the latter. The omission of those occurrences of *arbnew*

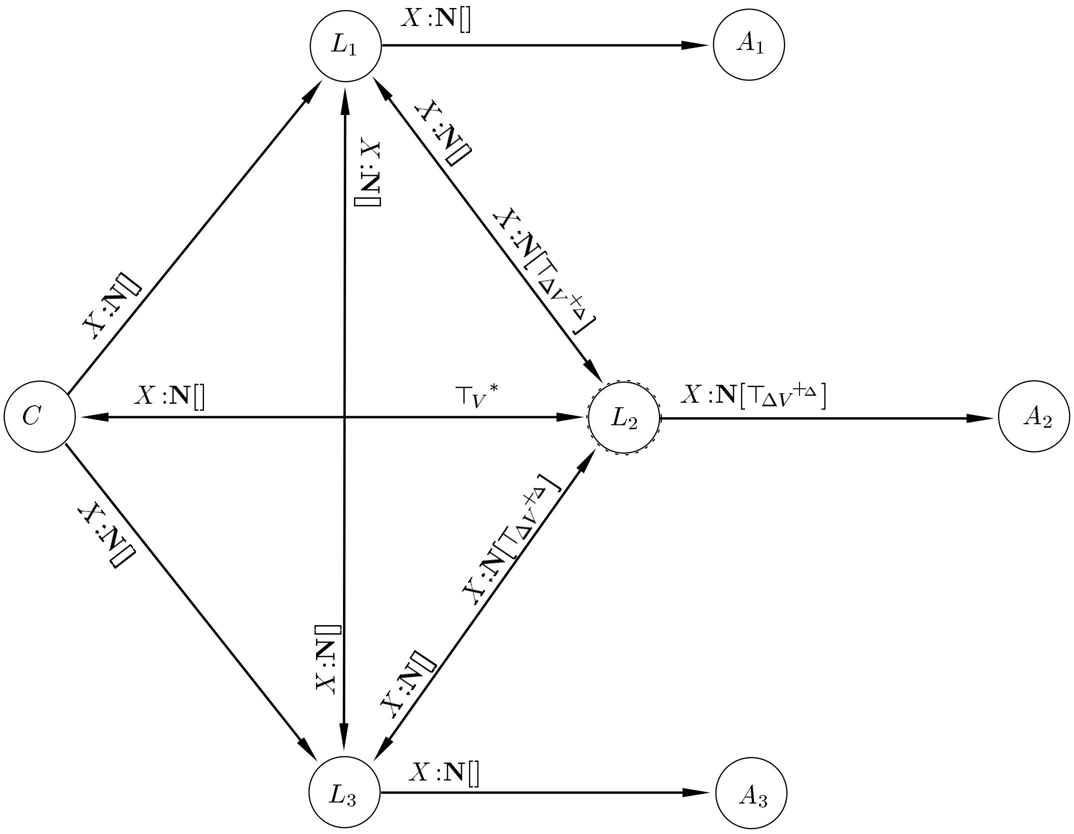

Figure 4.10: Behavior of the Oral Messages Algorithm when L_2 is faulty.

when the lieutenant receives inputs is a design decision; it would also have been correct to define *relay* so it retains those ms-atoms. The freedom to do either (and still have a monotonic input-output function) reflects the flexibility of our definition of $\leq_{Hist_{FC}}$: it does not introduce an artificial separation between perturbed behavior and new behavior. It is easy to check that the run in Figure 4.10 satisfies the fault-tolerance requirement.

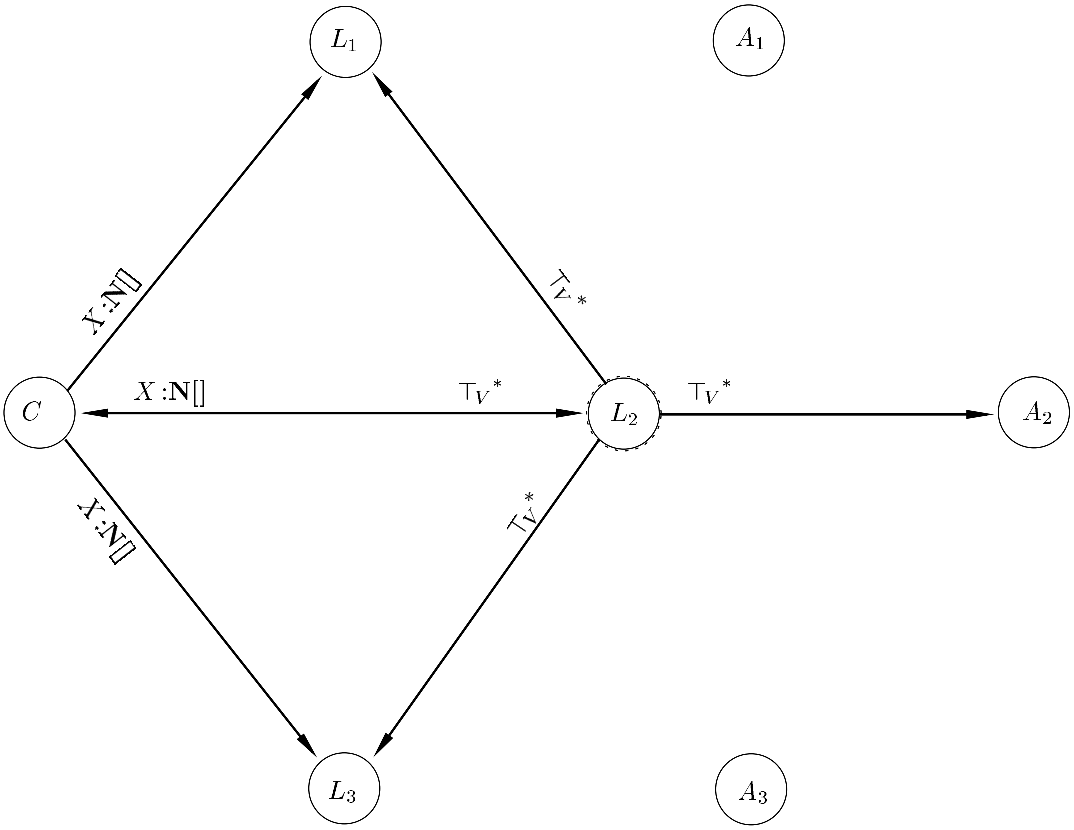

Figure 4.11: The run $step_F(nf_{BA}, fs_L)(\perp_{Run})$.

Commander Failure. Let fs_C be the failure scenario in which only the commander is faulty. Figure 4.12 shows the run $run_{FC}(nf_{BA})(fs_C)$. Since each lieutenant L_i relays the same value X_i to the other two lieutenants, all three lieutenants apply the majority function to the same sequence of values when they compute their output. It is easy to check that the run in Figure 4.12 satisfies the fault-tolerance requirement.

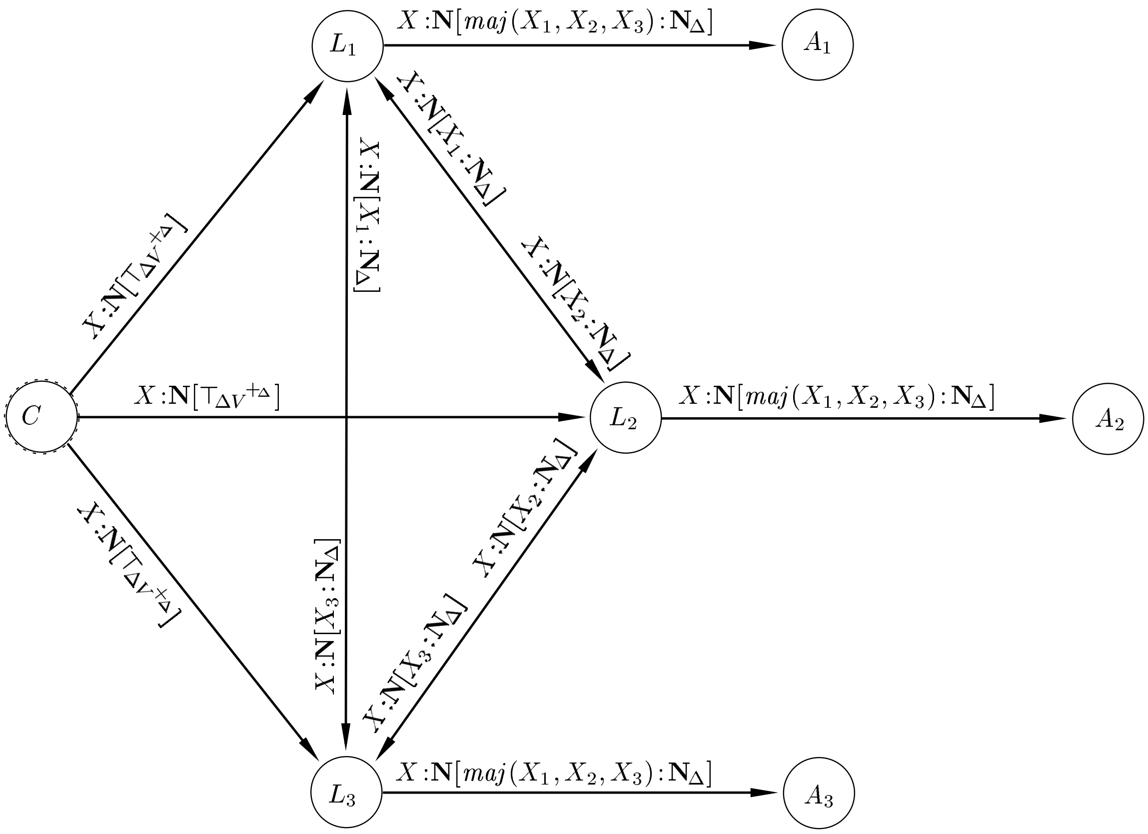

Figure 4.12: Behavior of the Oral Messages Algorithm when C is faulty.

Chapter 5

Fault-Tolerance for Moving Agents

An interesting paradigm for programming distributed systems is *moving agents*. In this paradigm, an agent is not tied to a particular site but rather moves from site to site in a network. For example, an agent that starts at site S might move to site S_1 in order to access some service (e.g., a database) available there. The agent might then determine that it needs to access a service located at site S_2 and move there. If the agent has gathered all of the information it needs, it might finish by moving to a final site A to deliver the result of the computation (A may be the same as S).

For our current purposes, it does not matter whether code is shipped from site to site; the essential points are that the thread of control moves from site to site, and that the sequence of services used by a moving agent is generally not known when the computation starts, since it may depend on information obtained as the computation proceeds.

There are two fault-tolerance issues: protecting moving agents from faulty servers and protecting servers from faulty moving agents. This chapter examines protocols for protecting moving agents from servers that may suffer Byzantine failures.

5.1 Fault-tolerance for Moving Agents

To illustrate new problems that arise with moving agents, we consider a two-stage moving agent, analogous to the two-stage pipeline discussed in Chapters 2 and 3. By analogy with the replicated pipeline, one might try to make a two-stage moving agent fault-tolerant by having it access multiple replicas of each service, with a majority vote on the results after the final stage. This coresponds to the moving agent shown in Figure 5.1: it starts at a source S, accesses service F, which is replicated at sites F_1, F_2, F_3, then accesses service

G, which is replicated at sites G_1, G_2, G_3.[1] Since G is the last service it needs, the agent moves to a consolidator B, which is responsible for delivering the result of the computation to the "actuator" A. Like a voter, the consolidator computes the majority of the values it receives and sends the result to the actuator; in addition, as discussed in detail below, the consolidator uses an authentication mechanism to determine which values are invalid and should be excluded from the vote.

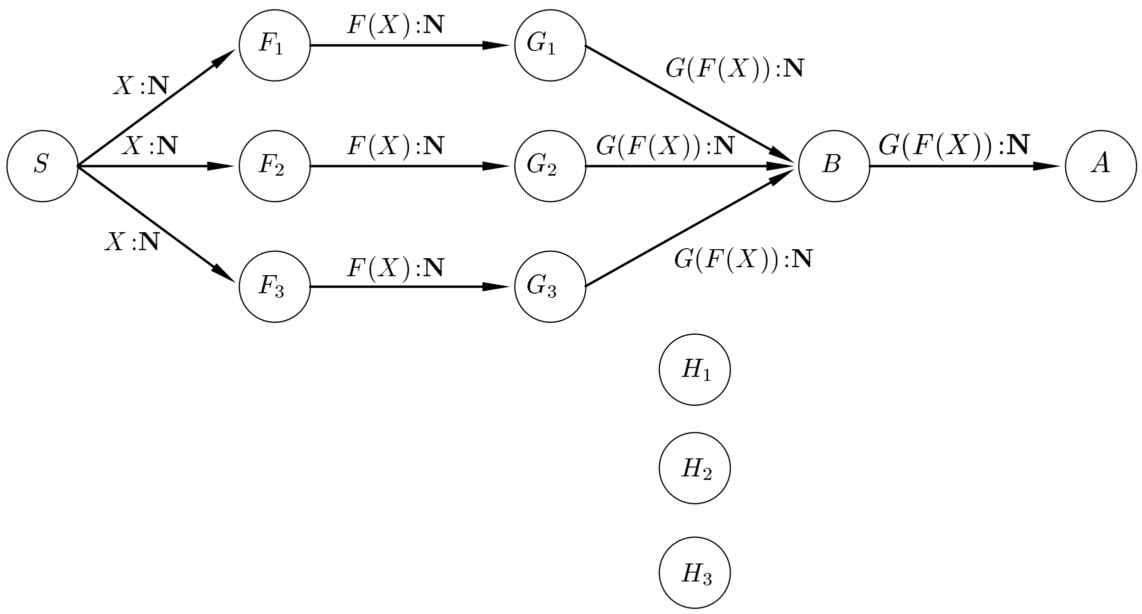

Figure 5.1: Run of replicated two-stage moving agent.

A typical moving agent accesses only some of the available services. To reflect this, the system shown in Figure 5.1 includes a service H comprising replicas H_1–H_3 not used by this particular agent. The fault-tolerance requirement for this system is that inputs to the actuator should be unaffected by Byzantine failure of a minority of the replicas of each service used by the moving agent and by Byzantine failure of all replicas of each service not used by the moving agent.

Faulty components can spoof (i.e., send messages that appear to be from other components) and eavesdrop (i.e., obtain copies of messages sent to other components). From the perspective of the recipient of a message, the possibility of spoofing causes uncertainty about the identity of the sender of the message, since the message might not be from the purported

[1]Of course, S, F_1, *etc*, here name denote different components than in previous examples: the mapping from names to input-output functions is different.

sender. This uncertainty is modeled in our framework by using input-output functions that are *independent* of the purported names of the senders in the input history. A simple way to ensure this is to use only input-output functions of the form

$$(\lambda \mathit{fail}\!:\! S.\ (\lambda h\!:\! \mathit{Hist}.\ f(\mathit{fail}, \cup_{x \in \mathit{Name}} \pi_1(h(x))))),$$

where $S \subseteq \mathit{Fail}$ and $f \in (S \times \mathit{Set}(L)) \to \mathit{Hist}$. Since we hide the information that identifies the sender of each message, the accuracy of this information is irrelevant.

Eavesdropping is modeled in a similar manner to Missing-Message-Detection in Section 4.2. A faulty component (the "eavesdropper") can send a special value *evsdrp*. The output of a component that receives this value must contain a possibility of sending copies of all subsequent outputs to the easvesdropper.[2] In examples, we assume a server is able to eavesdrop on all components except actuators; actuators communicate only with consolidators. This convention is used at both the concrete and abstract levels, so we treat *evsdrp* as both a concrete value and an abstract value, with $[\![\mathit{evsdrp}]\!]_{\mathit{AVal}} = \{\mathit{evsdrp}\}$.

Consider the consolidator B in Figure 5.1. How does it decide which inputs are valid (i.e., should be included in majority votes)? One might be tempted to say that the consolidator should treat messages from G_1, G_2, and G_3 as valid and messages from other components as invalid. This proposal is inappropriate for moving agents, because it assumes the consolidator knows in advance that the last service visited by the moving agent will be service G—the sequence of services visited by a moving agent is generally not known in advance.

At the other extreme, suppose the consolidator considers all inputs valid. Whenever the consolidator receives the same value from a majority of the replicas of some service, it sends that value to the actuator. Due to the possibility of spoofing, checking that a value was received from different replicas of the same service requires cryptographic techniques, as discussed below. This scheme tolerates some failure scenarios but not those involving the failure of services not used by the moving agent. For example, if H_1 alone fails, then with this scheme, the consolidator would ignore any incorrect values that H_1 sends, since none of the other replicas of service H would send incorrect values. If H_1, H_2, and H_3 all fail and send the same incorrect value to the consolidator, then with this scheme, the consolidator would send that incorrect value to the actuator.

There are various ways to fix this problem. Informally, in a computation with failures, a message is considered valid if it has visited the same sequence of services as a message

[2]For this purpose, we allow an exception to the rule in the previous paragraph; since *evsdrp* is not really sent in messages, spoofing is irrelevant.

in the failure-free computation. We consider a protocol in which digital signatures [RSA78] are used by the consolidator to reliably determine this.[3] We assume digital signatures are implemented using public-key cryptography and that each component knows its own private key and the public key of every other component. We also assume each component knows which service is provided by each component.

Each message is signed—to foil spoofing by faulty components—and augmented with information about the sequence of services that should have been visited and about the sequence of services actually visited. For the former, each source and server include in each outgoing message the next service to be visited by that moving agent. More specifically, a source includes in outgoing messages the first service (or consolidator, if no services are being used) to be visited by the moving agent, and a server includes in each outgoing message the service (or consolidator) to be visited next by the moving agent embodied in that message. The consolidator will need to verify the entire "history" of the moving agent (i.e., the entire sequence of visited services), so a server x also includes in the outgoing message one of the incoming messages that embodied the arrival of that moving agent at x; by induction, that incoming message contains the history of the moving agent up to the arrival of the moving agent at x.

Recall that sources and servers sign every outgoing message. The signatures both prevent lying about the sequence of services that should have been visited (e.g., prevent tampering with the input message included in the output message) and document the sequence of services actually visited. The consolidator tests validity of a message by checking that it was originated by a source, that the consolidator itself is the declared destination of the message, and that the sequence of declared destinations in the message is consistent with the signatures. Of course, the consolidator also checks each of the signatures and considers the message invalid if any of those checks fail. We say a set of messages is valid if each message is valid, all the messages visited the same sequence of services and contain the same data value, and the messages collectively are last signed by a majority of the replicas of some service. When the consolidator receives a valid set of messages, it forwards the common data value to the actuator.

We now describe this protocol in more detail. A message sent by a source with private key k has the form

$$sign(\langle data, dest \rangle, k) \tag{5.1}$$

[3] A protocol based on shared secrets is described in [MvRSS96].

where *data* is the data carried by the message, *dest* is the first service or consolidator to be visited, and $sign(x, k)$ represents x digitally signed with key k. For convenience, we assume each source initiates at most one moving agent; this restriction is easily removed by including (say) a sequence number in *data* of (5.1). A message received, processed, and forwarded by a server x with private key k has the form

$$sign(\langle data, dest, m\rangle, k) \tag{5.2}$$

where *data* and *dest* are as before, and m is the original message received by x that caused it to send this message.

To conveniently describe protocols that send messages of these forms in our framework, we introduce some notation. Let $Data \in AVal$ represent the data values carried by moving agents, and let $Key \subseteq CVal$ represent the cryptographic keys used for digital signatures. For $cv \in CVal$ and $k \in Key$, the concrete value $sign(cv, k)$ is cv signed with key k.

Let $Src \subseteq Name$ be the set of sources, i.e., the unreplicated components trusted to initiate moving agents. Let $Svc \subseteq Con$ be the set of (names of) services. We require that different elements of Svc denote different services at the concrete level; this is similar to the Uniqueness requirement for messages in the reliable broadcast example in Section 4.1.1. Formally, this requirement means that we consider only partial interpretation of constants satisfying

$$(\forall svc, svc' \in Svc : svc \neq svc' \Rightarrow \rho_a(svc) \neq \rho_a(svc')). \tag{5.3}$$

For convenience, we assume each server offers a single service. Thus, there is a function $provides \in Name \rightarrow (Svc \cup \{\bot\})$ that returns the service provided by a component; it returns $\bot$ for components (such as sources and actuators) that don't provide a service. Each consolidator offers its own unique consolidating service; that is, associated with each consolidator x is some service provided only by x.

Messages of the forms (5.1) and (5.2) are represented by an abstract value $Msg \in AVal$, whose meaning is the smallest set satisfying

$$\begin{aligned} [\![Msg]\!]_{AVal} = {} & \textbf{let } S_0 = [\![Data]\!]_{AVal} \times Svc \\ & \textbf{in let } S = [\![Data]\!]_{AVal} \times Svc \times [\![Msg]\!]_{AVal} \\ & \textbf{in } \textstyle\bigcup_{k \in Key} \overline{sign}(S_0 \cup S, k), \end{aligned} \tag{5.4}$$

where $\overline{sign}$ is the pointwise extension of $sign$ with respect to its first argument. To construct messages of the forms (5.1) and (5.2), we introduce constants $msg_0 \in Con$ and $msg \in Con$, respectively, with interpretations

$$\begin{aligned} \rho_a(msg_0) &= (\lambda\langle k, data, dest\rangle : Key \times [\![Data]\!]_{AVal} \times Svc. \\ &\qquad sign(\langle data, dest\rangle, k)) \\ \rho_a(msg) &= (\lambda\langle k, data, dest, msg\rangle : Key \times [\![Data]\!]_{AVal} \times Svc \times [\![Msg]\!]_{AVal}. \\ &\qquad sign(\langle data, dest, msg\rangle, k)) \end{aligned}$$

A set $KC \subseteq Con$ of key constants is used to represent the keys in Key. We adopt the following convention: for $x \in Name$, $K_x \in KC$ represents x's private key. Since these are the only keys we are interested in, we take $KC = \bigcup_{x \in Name} K_x$. We require that private keys be distinct. Thus, we assume the partial interpretation ρ_a of constants satisfies

$$\begin{aligned} &\wedge(\forall kc \in KC : \rho_a(kc) \in Key) \\ &\wedge(\forall kc_1 \in KC : (\forall kc_2 \in KC : kc_1 \neq kc_2 \Rightarrow \rho_a(kc_1) \neq \rho_a(kc_2))) \end{aligned} \tag{5.5}$$

Define $prin \in KC \rightarrow Name$ (short for "principal") by $prin(K_x) = x$.

The processing done by a service $svc \in Svc$ is represented by an operator $\widetilde{svc} \in Con$. For example, if a moving agent carrying symbolic data value v visits service F, the moving agent will leave carrying symbolic data value $\widetilde{F}(v)$.

The behavior of this protocol for the two-stage moving agent considered above is shown in Figure 5.2, where

$$m^0 = msg_0(K_S, X, F) \tag{5.6}$$

$$m_i^1 = msg(K_{F_i}, \widetilde{F}(X), G, m^0) \tag{5.7}$$

$$m_i^2 = msg(K_{G_i}, \widetilde{G}(\widetilde{F}(X)), B, m_i^1) \tag{5.8}$$

where $X \in Var(S)$ represents the data sent by the source S. To see that this protocol prevents spoofing by faulty replicas of service H, consider, for example, the case where those faulty components obtain a copy of message m^0 by eavesdropping, and then send the consolidator messages containing m^0. The consolidator will find these messages invalid, because they are not signed by providers of the declared destination of m^0 (namely, service F). Of course, attempts by the faulty replicas of service H to change the declared destination of m^0 will cause the consolidator's check of the source's signature to fail.

5.1.1 Voting After Each Stage

This protocol provides some fault-tolerance but does not satisfy the fault-tolerance requirement on page 106, namely, that a moving agent tolerate simultaneous Byzantine failure of a

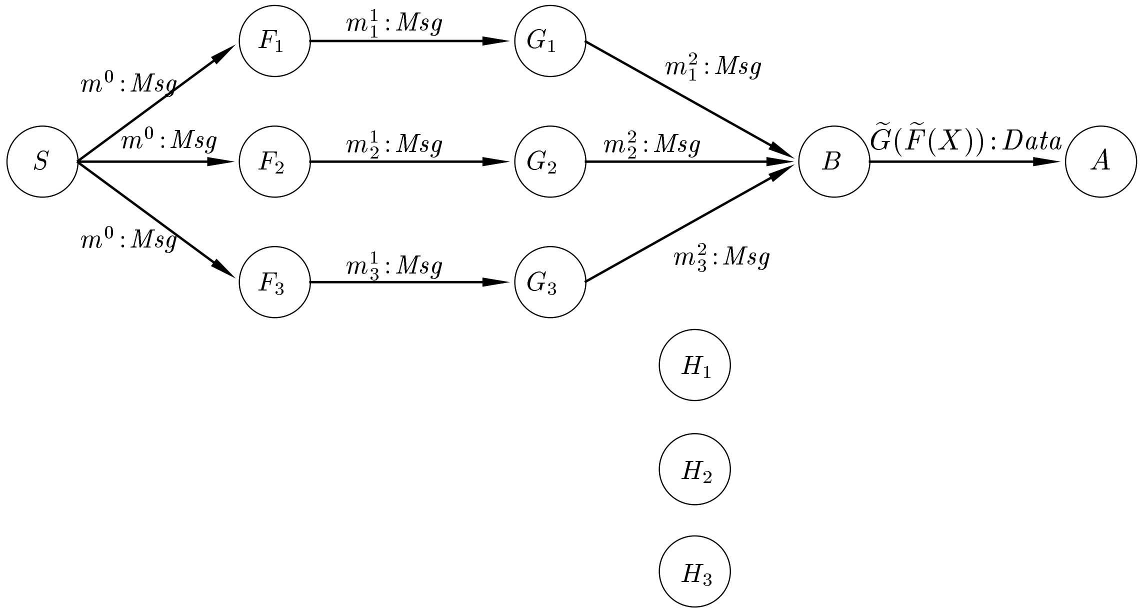

Figure 5.2: Run of replicated two-stage moving agent, with authentication.

minority of replicas of *each* service it uses. A protocol that uses only the sparse pattern of communication shown in Figure 5.1 or Figure 5.2 cannot satisfy this requirement, because the effects of failures in different stages are cumulative. For example, the above protocol does not tolerate simultaneous failure of F_1 and G_2, because then two of the consolidator's three inputs might be corrupted by the failures. Incidentally, the same argument applies to the two-stage pipeline of chapter 3.

To make computation more robust, each server can send its outgoing messages to *all* replicas of the next service, instead of just one, and validity tests and voting are incorporated into each stage of the computation after the first.[4] The validity test and voting are just as described for consolidators.

This change to the protocol requires one clarification. Recall that a server included in each outgoing message the unique incoming message that caused it to send that message. Now, a server sends messages only after receiving valid messages with the same value from a majority of the replicas of some service; so, we add that the server may include in the outgoing messages any one of these incoming messages.[5]

[4]Intermediate levels of fault-tolerance can be achieved by voting after every few stages, rather than after every stage.

[5]The reader who wonders whether more than one incoming message should be included in the outgoing message is referred to the comments below.

Note that the only remaining difference between a server and a consolidator is that a consolidator does not perform application-specific computation (i.e., does not apply an operator) and does not include authentication information in its outputs (it sends unadorned data values to the actuator).

The resulting pattern of communication is shown in Figure 5.3. Each G_j might include any one of its three input messages in its output, so the value it outputs is $ms_j \times \{Msg\}$, where

$$ms_j = \{m^2_{1,j}, m^2_{2,j}, m^2_{3,j}\}, \tag{5.9}$$

where $m^2_{i,j}$ is a message signed by F_i then G_j:

$$m^2_{i,j} = msg(K_{G_j}, \widetilde{G}(\widetilde{F}(X)), B, m^1_i). \tag{5.10}$$

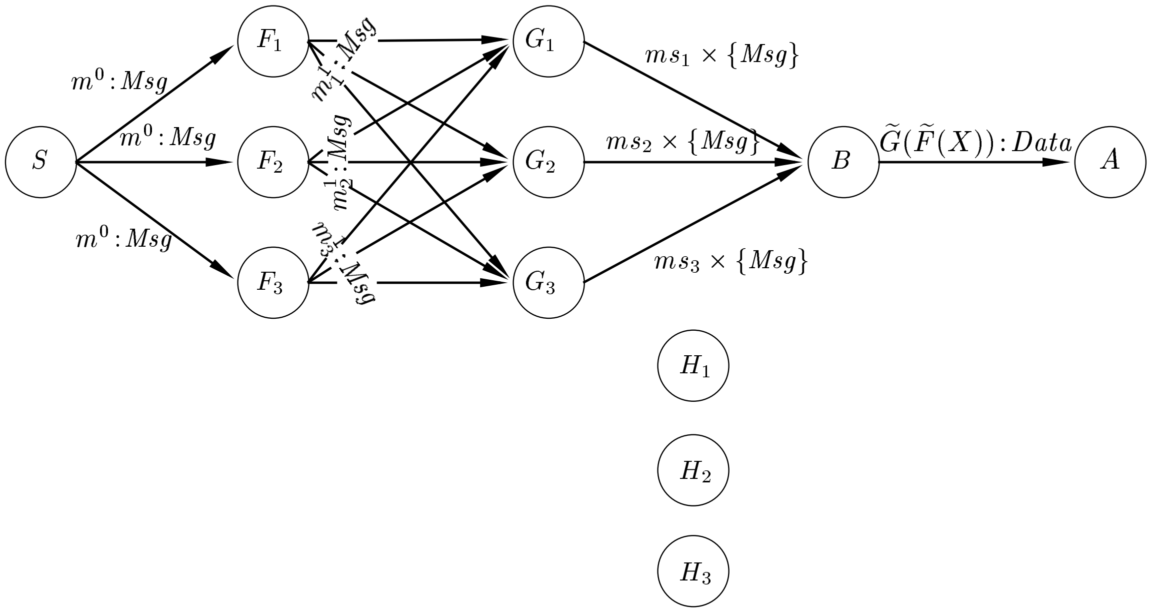

Figure 5.3: Run of replicated two-stage moving agent, with authentication and with voting after each stage. Each skewed ms-atom labels each of the three edges it crosses.

Comments on the protocol. Before proceeding with the modeling and analysis of the protocol sketched above, we observe that the protocol does not satisfy the fault-tolerance requirement on 106. Modifying the protocol to satisfy this fault-tolerance requirement is straightforward (see Section 5.3.3). We choose to analyze this protocol—rather than a more robust one—for two reasons. First, it seems appropriate to "test" the analysis on an incorrect protocol. The analysis of this protocol then demonstrates both positive and negative results; specifically, the analysis shows that the protocol enables a moving agent to tolerate Byzantine failure of a minority of the replicas of each service used by the moving agent or Byzantine failure of all replicas of each service not used by the moving agent, but not both simultaneously. Second, this development reflects our original expectations: the protocol was developed and analyzed with the idea that it would satisfy the original fault-tolerance requirement, but the analysis proved our expectations incorrect.

5.1.2 The Effects of Byzantine Failures

In the analysis of Byzantine agreement in Section 4.2, we assumed a Byzantine-faulty component could output any value. This behavior was modeled using the abstract value $\top_V$. This model of Byzantine-faulty components is too coarse when cryptography-based protocols are being analyzed. A fundamental assumption of cryptography is the infeasibility of malicious entities to randomly guess certain kinds of information, such as cryptographic keys or signed messages. Of course, $\top_V$ represents all concrete values, including keys and signed messages. To reflect the infeasibility of malicious entities to randomly guess certain values, we need to use in the outputs of Byzantine-faulty components abstract values that represent all concrete values that can be generated from specified sets of cryptographic information. The cryptographic information known by a faulty component is specified by a set of keys, represented by key constants, and a set of signed messages, represented by elements of *SMsg*, which is the smallest set satisfying

$$SMsg \triangleq SMsg_0 \cup \bigcup_{kc \in KC, data \in SVal_0, dest \in Svc, m \in SMsg} \{msg(kc, data, dest, m)\}, \tag{5.11}$$

where

$$SMsg_0 \triangleq \bigcup_{kc \in KC, data \in SVal_0, dest \in Svc} \{msg_0(kc, data, dest)\}. \tag{5.12}$$

For $kcs \subseteq KC$ and $ms \subseteq SMsg$, the abstract value $Arb(kcs, ms)$ represents all concrete values that can be generated from the specified cryptographic information. To formalize this, we first consider the meanings of elements of KC and *SMsg*.

For elements of KC, we assume the meanings are given by a partial interpretation of constants satisfying (5.5). When giving the meaning of $Arb(kcs, ms)$, we must consider not only keys (i.e., elements of *Key*) that are represented by key constants but also keys that are not represented by any key constant. To show soundness of a protocol, it suffices to assume that Byzantine-faulty components cannot randomly guess cryptographic information generated from keys being used by non-faulty components. Of course, those keys are all represented by key constants. Thus, it does not matter for our purposes whether keys not represented by key constants can be guessed by faulty components. For generality, we assume they can be. Accordingly, we take $Arb(kcs, ms)$ to represent concrete values that can be generated using keys explicitly represented in kcs and keys not represented by any key constant.

For elements of $SMsg$, the only subtlety involves the symbolic value $data$, which may contain variables. In Chapter 3, the meaning of an abstract value is independent of the interpretation of variables. This restriction forces an overly coarse approximation here, since it implies (for example) that $Arb(kcs, \{msg_0(S, X, F)\})$ must represent messages from source S to service F containing an arbitrary data value. Thus, in this approximation, if Arb is used to represent the outputs of faulty components, those components would appear to be able to change the data in a message without invalidating the signature. To avoid this problem, we allow the meaning of abstract values to depend on the interpretation of variables.

This requires only minor changes to the framework described in Section 3.1. We parameterize $[\![]\!]_{AVal}$ by a partial interpretation of symbols; thus, instead of $[\![]\!]_{AVal} \in Interp_{Set}(AVal)$, we have $[\![]\!]_{AVal} \in interp(Sym) \twoheadrightarrow Interp_{Set}(AVal)$, where the ordering on $Interp_{Set}(AVal)$ is

$$\rho_1 \leq_{Interp_{Set}(S)} \rho_2 \triangleq (\forall s \in S : \rho_1(s) \neq \emptyset \Rightarrow \rho_1(s) = \rho_2(s)). \tag{5.13}$$

Since ρ is a partial interpretation, it might not give values for all of the symbols on which the meaning of an abstract value a depends. For technical convenience, instead of using a partial function, we encode this undefinedness by taking $[\![a]\!]^{\rho}_{AVal} = \emptyset$ in those cases. Note that with this correspondence in mind (namely, $\rho_1(s) = \emptyset$ corresponds to $s \notin dom(\rho)$, (5.13) is essentially the same as (2.60). The monotonicity and continuity requirements for $[\![]\!]_{AVal}$ ensure that the meaning of an abstract value a, once "defined" (i.e., non-empty), is not changed by extending ρ.

To check that these changes are reasonable, we also give the revised definition of the meanings of posets of ms-atoms, *etc.* The definitions in Section 2.4.1 are mostly unchanged, the sole exception being the definition of $compat_{Val}$ in (2.61), to which we add a ρ:

$$\begin{aligned} compat^{\rho}_{Val}(val, cv) \triangleq (\exists \langle s, a\rangle \in val :\ & \wedge\ cv \in [\![a]\!]^{\rho}_{AVal} \\ & \wedge\ s = \_ \vee cv = \overline{\rho}(s)). \end{aligned} \tag{5.14}$$

Now, note that for an abstract value a with $[\![a]\!]^{\rho}_{AVal} = \emptyset$, the condition $cv \in [\![a]\!]^{\rho}_{AVal}$ does not hold, so such abstract values are effectively ignored when determining the meaning of a poset of ms-atoms, just as uninterpreted symbols are effectively ignored. Monotonicity of $[\![]\!]^{\rho}_{AVal}$ with respect to ρ ensures that $[\![]\!]^{\rho}_{POSet(L)}$ and $[\![]\!]^{\rho}_{Hist}$ are still monotonic with respect to ρ. With this observation, the proof of soundness goes through just as before.

Returning to the meaning of messages and Arb, we parameterize both by a partial interpretation $\rho \in interp(Sym)$ whose interpretation of constants is expected to satisfy (5.5), (5.5), and (5.5). We define

$$\begin{aligned} [\![msg_0(kc, data, dest)]\!]^{\rho}_{SMsg} &= \textbf{if } \overline{\rho}(d) \notin Data \textbf{ then } \emptyset \\ &\quad\ \textbf{else } \{\rho(msg_0)(\rho(kc), \overline{\rho}(d), dest)\} \end{aligned} \tag{5.15}$$

$$\begin{aligned} [\![msg(kc, data, dest, m)]\!]^{\rho}_{SMsg} &= \textbf{if } \overline{\rho}(d) \notin Data \textbf{ then } \emptyset \\ &\quad\ \textbf{else } \textstyle\bigcup_{cv \in [\![m]\!]_{SMsg}} \{\rho(msg)(\rho(kc), \overline{\rho}(d), dest, cv)\} \end{aligned} \tag{5.16}$$

where $\overline{\rho}$ is given by (2.61).

It is convenient to extend $[\![]\!]_{SMsg}$ to all symbolic values, by defining, for $s \in SVal \setminus SMsg$,

$$[\![s]\!]_{SMsg} = [\![Msg]\!]_{AVal}. \tag{5.17}$$

This is a conservative approximation: symbolic values not in $SMsg$ are treated as a completely arbitrary messages. Note that we have elided the ρ in $[\![Msg]\!]^{\rho}_{AVal}$, since the meaning of Msg is independent of ρ; we use this notation for other abstract values as well.

Finally, for $kcs \subseteq KC$ and $ms \subseteq SVal$, the meaning of $Arb(kcs, ms)$ is the least set satisfying

$$\begin{aligned} [\![Arb(kcs, ms)]\!]^{\rho}_{AVal} = \textbf{let } & keys = (\textstyle\bigcup_{kc \in kcs}\{\rho(kc)\}) \cup (Key \setminus \bigcup_{kc \in kcs}\{\rho(kc)\}) \\ \textbf{in } & Name \cup [\![Data]\!]_{AVal} \cup Svc \cup keys \cup (\cup_{m \in ms}[\![m]\!]_{SMsg}) \\ & \cup (\cup_{k \in keys, x \in [\![Arb(kcs,ms)]\!]^{\rho}_{AVal}}\{sign(x, k)\}). \end{aligned} \tag{5.18}$$

Thus, the abstract values used in this analysis are

$$AVal = \{Data, Msg\} \cup Arbs \cup AMul \tag{5.19}$$

where

$$Arbs = \bigcup_{kcs \subseteq KC, ms \subseteq SVal} \{Arb(kcs, ms)\} \tag{5.20}$$

$$AMul = \{?, 1, *\} \tag{5.21}$$

5.2 Input-Output Functions

5.2.1 Input-Output Function for Servers

The input-output function $\mathit{server}(\mathit{kc}, \mathit{svc}, n, \mathit{next}, \mathit{nbrs})$ represents a server with a private key represented by $\mathit{kc} \in \mathit{KC}$ and that provides service $\mathit{svc} \in \mathit{Svc}$. Parameter $n \in \mathsf{N}$ is the (minimum) number of replicas of each service in the system: the server looks for a valid set of inputs of size $\lceil (n+1)/2 \rceil$ before processing a moving agent. To partially reflect the dependence of the remaining path of a moving agent on information stored by servers, the description of a server also specifies the next service $\mathit{next} \in \mathit{Svc}$ normally visited by moving agents that visit that server.[6] To further reflect this dependence and the possibility that messages from faulty components might "confuse" a server, if there is no (symbolic) majority among the input values corresponding to a moving agent, then we assume that the next destination for this moving agent might be any component in $\mathit{nbrs} \subseteq \mathit{Name}$. If the server is faulty, it sends arbitrary messages to and eavesdrops on the components in nbrs.

Inputs corresponding to moving agents initiated by different sources can be processed separately, so we write the input-output function for a server as

$$
\begin{array}{l}
\mathit{server}(\mathit{kc}, \mathit{svc}, n, \mathit{next}, \mathit{nbrs}) = \\
\quad (\lambda \mathit{fail}\!:\!\{\mathit{OK}, \mathit{ByzFail}\}.\ (\lambda h\!:\!\mathit{Hist}.\ (\lambda x\!:\!\mathit{Name}. \\
\qquad \textbf{let } \mathit{lbls} = \bigcup_{y \in \mathit{Name}} \pi_1(h(y)) \\
\qquad \textbf{in let } \mathit{evsdrprs} = \{x \in \mathit{Name} \mid \mathit{evsdrp} \in \bigcup_{v \in \overline{\pi_2}(\pi_1(h(x)))} \overline{\pi_2}(v)\} \\
\qquad \textbf{in if } \mathit{fail} = \mathit{OK} \textbf{ then } \langle \bigcup_{\mathit{src} \in \mathit{Src}} \mathit{server}_1(\mathit{kc}, \mathit{svc}, n, \mathit{next}, \mathit{src}, \mathit{lbls}, x, \mathit{evsdrprs}), \emptyset \rangle \\
\qquad\quad \textbf{else if } x \in (\mathit{evsdrprs} \setminus \mathit{nbrs}) \textbf{ then } \langle \{\langle *, \mathit{mkArb}(\{\mathit{kc}\}, \overline{\pi_2}(\mathit{lbls})), 0\rangle\}, \emptyset \rangle \\
\qquad\quad \textbf{else if } x \in \mathit{nbrs} \textbf{ then } \langle \{\langle *, \{\mathit{evsdrp}, \mathit{mkArb}(\{\mathit{kc}\}, \overline{\pi_2}(\mathit{lbls}))\}, 0\rangle\}, \emptyset \rangle \\
\qquad\quad \textbf{else } \langle \emptyset, \emptyset \rangle)))
\end{array}
\tag{5.22}
$$

where $\mathit{server}_1(\mathit{kc}, \mathit{svc}, n, \mathit{next}, \mathit{src}, \mathit{lbls}, x, \mathit{evsdrprs}) \in \mathit{Set}(L)$ is the set of output ms-atoms to component $x \in \mathit{Name}$ that represent moving agents initiated by source $\mathit{src} \in \mathit{Src}$, where $\mathit{evsdrprs} \subseteq \mathit{Name}$ is the set of eavesdroppers, $\mathit{lbls} \subseteq L$ is the set of input ms-atoms, and the first four parameters are as for server. The definition of server_1 appears in Figure 5.4 and is discussed in the following subsections. The function $\mathit{mkArb} \in \mathit{Set}(\mathit{KC}) \times \mathit{Set}(\mathit{Val}) \to \mathit{Val}$ returns an element of Arbs that represents all concrete values that can be generated from

[6] It is straightforward to let the next service normally visited depend on information carried by the moving agent, though abstracting from the identity of that service is difficult, as discussed in section 5.4.

the concrete values represented by the specified key constants and values:

$$\begin{aligned} mkArb(kcs, vs) = \ &\textbf{let } arbs = \textstyle\bigcup_{v \in vs} \overline{\pi_2}(v) \cap Arbs \\ &\textbf{in let } kcs_1 = kcs \cup (\textstyle\bigcup_{v \in vs} \overline{\pi_1}(v) \cap KC) \cup (\textstyle\bigcup_{Arb(kcs', ms) \in arbs} kcs') \\ &\textbf{in let } ms = (\textstyle\bigcup_{Arb(kcs', ms) \in arbs} ms) \cup (\textstyle\bigcup_{v \in vs} \overline{\pi_1}(v) \cap SMsg) \\ &\textbf{in } \{\_ \colon Arb(kcs_1, \overline{unpack}(ms))\}, \end{aligned} \tag{5.23}$$

where $unpack \in SMsg \to Set(SMsg)$ is

$$\begin{aligned} unpack(m) = \ &\textbf{match } m \textbf{ with} \\ &| msg(\_, \_, \_, m') \to \{m\} \cup unpack(m') \\ &| \_ \to \{m\}, \end{aligned} \tag{5.24}$$

and where $\overline{unpack} \in Set(SMsg) \to Set(SMsg)$ is defined by

$$\overline{unpack}(S) = \cup_{m \in S} unpack(S). \tag{5.25}$$

The use of *unpack* reflects a faulty component's ability to extract pieces of messages and incorporate them in its outputs.

Determining Set of Inputs that Contribute to the Output

In the definition of $server_1$, the value of *mmes* (mnemonic for "⟨multiplicity, ⟨message, extension⟩⟩'s") summarizes the sets of input messages that can contribute to the output of the server. Roughly, each element $\langle mul, mes \rangle$ of *mmes* corresponds to a possible quorum, i.e., a valid set of input messages of appropriate size. The first component, *mul*, is 1 if the inputs described by *mes* definitely represent a quorum (hence definitely cause an output) and is ? otherwise. Our definition of *mmes* has the property that if $\langle 1, mes \rangle \in mmes$, then also $\langle ?, mes \rangle \in mmes$; this is not necessary for correctness of the subsequent definitions, but it does no harm.

Second component *mes* describes a set of messages that might have been received by the server. If an input ms-atom contains $Arb(kcs, ms)$, a message in *ms* may be extended with signatures by keys in *kcs*. Such an *extended message* is represented by a pair $\langle m, ext \rangle$, where $m \in SMsg$ and where the extension $ext \in Seq(Name)$ is the additional sequence of sites by which m has been signed. As a special case, since a faulty source can generate valid messages from scratch, we allow extended messages of the form $\langle \perp, ext \rangle$ with $first(ext) \in Src$. It does no harm to allow extended messages of the form $\langle \perp, ext \rangle$ for arbitrary $ext \in Seq(Name)$, since they will not satisfy the test for validity (see the definition of $valid_M$ below). Thus, we

$$
\begin{array}{l}
\mathit{server}_1(\mathit{kc}, \mathit{svc}, n, \mathit{next}, \mathit{src}, \mathit{lbls}, x, \mathit{evsdrprs}) = \\
\quad \textbf{let } n = 1 + \max\left(\bigcup_{v \in \overline{\pi_2}(\mathit{lbls})} \bigcup_{ma \in \mathit{getMsgArb}(v)} \bigcup_{m \in \mathit{getMsgs}(ma)} \{|\pi_1(\mathit{getPath}(m))|\}\right) \\
\quad \textbf{in let } \mathit{mmes} = \{\langle \mathit{mul}, \mathit{mes}\rangle \in \{?, 1\} \times \mathit{Set}(\mathit{MsgExt}(n)) \mid \\
\qquad\qquad\qquad\qquad (\exists S \subseteq \mathit{lbls} : (\exists h \in \mathit{mes} \overset{\text{onto}}{\longrightarrow} S : \\
\qquad\qquad\qquad\qquad\qquad \wedge\ \vee\ |\mathit{mes}| = \lceil (n+1)/2 \rceil \wedge \overline{\mathit{getSigner}}(\mathit{mes}) \cap \mathit{Src} = \emptyset \\
\qquad\qquad\qquad\qquad\qquad\quad\ \ \vee\ |\mathit{mes}| = 1 \wedge \overline{\mathit{getSigner}}(\mathit{mes}) \cap \mathit{Src} \neq \emptyset \\
\qquad\qquad\qquad\qquad\qquad \wedge\ (\forall \mathit{me} \in \mathit{mes} : (\exists \mathit{ma} \in \mathit{getMsgArb}(\pi_2(h(\mathit{me}))) : \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \wedge\ \pi_1(\mathit{me}) \in \mathit{getMsgs}(\mathit{ma}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \wedge\ \pi_2(\mathit{me}) \in \mathit{Seq}(\overline{\mathit{prin}}(\mathit{getKeys}(\mathit{ma}))) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \wedge\ \mathit{ma} \in \mathit{Arbs} \Rightarrow \mathit{mul} = ?)) \\
\qquad\qquad\qquad\qquad\qquad \wedge\ (\forall \ell \in S : * \notin \overline{\pi_2}(\pi_1(\ell)) \Rightarrow |h^{\mathit{inv}}(\ell)| = 1) \\
\qquad\qquad\qquad\qquad\qquad \wedge\ \{?, *\} \cap \overline{\overline{\pi_2}}(\overline{\pi_1}(S)) \neq \emptyset \Rightarrow \mathit{mul} = ?)) \\
\qquad\qquad\qquad\qquad\qquad \wedge\ \mathit{valid}_M(\mathit{src}, \mathit{svc})(\mathit{mes})\} \\
\quad \textbf{in if } \mathit{mmes} = \emptyset \textbf{ then } \emptyset \\
\qquad \textbf{else let } \mathit{ds} = \bigcup_{\mathit{mes} \in \overline{\pi_2}(\mathit{mmes})} \textbf{if } \varepsilon \notin \overline{\pi_2}(\mathit{mes}) \textbf{ then } \{\bot\} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } \overline{\mathit{getData}}(\overline{\pi_1}(\{\langle m, \mathit{ext}\rangle \in \mathit{mes} \mid \mathit{ext} = \varepsilon\})) \\
\qquad\qquad \textbf{in if } |\mathit{ds}| = 1 \wedge \bot \notin \mathit{ds} \textbf{ then} \\
\qquad\qquad\qquad \textbf{let } d = \text{the element of } \mathit{ds} \\
\qquad\qquad\qquad \textbf{in let } \mathit{ms}_{\mathit{in}} = \bigcup_{\mathit{mes} \in \overline{\pi_2}(\mathit{mmes})} \mathit{MEtoMsg}(\mathit{next}, \mathit{mes}) \\
\qquad\qquad\qquad \textbf{in let } \mathit{ms} = \bigcup_{m \in\ \mathit{ms}_{\mathit{in}}} \{\mathit{apply}(\mathit{msg}, \langle\!\langle \mathit{kc}, \mathit{apply}(\widetilde{\mathit{svc}}, \langle\!\langle d \rangle\!\rangle), \mathit{next}, m \rangle\!\rangle)\} \\
\qquad\qquad\qquad \textbf{in let } \mathit{mul} = \textbf{if } 1 \in \overline{\pi_1}(\mathit{mmes}) \wedge \mathit{provides}(x) = \mathit{next} \textbf{ then } 1 \textbf{ else } ? \\
\qquad\qquad\qquad \textbf{in let } \ell = \langle \mathit{mul}, \mathit{ms} \times \{\mathit{Msg}\}, 0 \rangle \\
\qquad\qquad\qquad \textbf{in if } \mathit{provides}(x) = \mathit{next}\ \vee x \in \mathit{evsdrprs} \textbf{ then } \{\ell\} \textbf{ else } \emptyset \\
\qquad\qquad\quad \textbf{else } (\text{* symbolic output value not unique; approximate *}) \\
\qquad\qquad\qquad \textbf{if } x \in \mathit{nbrs} \cup \mathit{evsdrprs} \textbf{ then} \\
\qquad\qquad\qquad\quad \langle ?, \mathit{mkArb}(\{\mathit{kc}\}, \overline{\pi_2}(\mathit{lbls})), \mathit{tagOfSrc}(\mathit{src}) \rangle \\
\qquad\qquad\qquad \textbf{else } \emptyset
\end{array}
$$

Figure 5.4: Definition of server_1.

take *mes* to be a subset of

$$MsgExt = ((SMsg \cup \{\bot\}) \times Seq(Name)) \setminus \{\langle \bot, \varepsilon \rangle\}, \tag{5.26}$$

As argued below, it suffices to consider only extensions of at most a certain length n, so in Figure 5.4, we take *mes* to be a subset of

$$MsgExt(n) = ((SMsg \cup \{\bot\}) \times Seq(Name, n)) \setminus \{\langle \bot, \varepsilon \rangle\}, \tag{5.27}$$

where $Seq(S, n) \triangleq \{\sigma \in Seq(S) \mid |\sigma| \leq n\}$.

Inclusion of $\langle mul, mes \rangle$ in *mmes* is justified by the existence of a set $S \subseteq lbls$ and a correspondence h between *mes* and S that together satisfy the following five conditions, corresponding to the five conjuncts in the definition of *mmes*.

First Condition. The first condition is:

$$\begin{aligned} &\vee |mes| = \lceil (n+1)/2 \rceil \wedge \overline{getSigner}(mes) \cap Src = \emptyset \\ &\vee |mes| = 1 \wedge \overline{getSigner}(mes) \cap Src \neq \emptyset. \end{aligned} \tag{5.28}$$

This constrains the size of *mes*. There are two cases. A quorum comprising messages not from sources must be of size $\lceil (n+1)/2 \rceil$; this corresponds to the first disjunct. Sources are not replicated, so a single message signed by a source suffices. This corresponds to the second disjunct; it is formalized using $getSigner \in MsgExt \to Name$, which returns the name of the (last) signer of an extended message:

$$\begin{aligned} getSigner(\langle m, ext \rangle) = \ &\textbf{if } ext \neq \varepsilon \textbf{ then } last(ext) \\ &\textbf{else match } m \textbf{ with} \\ &\quad |msg_0(kc, \_, \_) \to prin(kc) \\ &\quad |msg(kc, \_, \_, \_) \to prin(kc) \end{aligned} \tag{5.29}$$

where *last* returns the last element of a sequence. $\overline{getSigner}$ is the pointwise extension of *getSigner* to sets of extended messages.

Second Condition. The second condition is:

$$\begin{aligned} (\forall me \in mes : (\exists ma \in getMsgArb(\pi_2(h(me))) :& \\ \wedge\ \pi_1(me) \in getMsgs(ma)& \\ \wedge\ \pi_2(me) \in Seq(\overline{prin}(getKeys(ma)))& \\ \wedge\ ma \in Arbs \Rightarrow mul = ?)).& \end{aligned} \tag{5.30}$$

This requires that the correspondence h be such that, for each extended message $\langle m, ext \rangle \in mes$,

(i) The message m appears in the value $\pi_2(h(me))$, i.e., m appears in some $ma \in \mathit{MsgArb}$ that appears in $\pi_2(h(me))$.

(ii) The extension ext is a sequence of names by which m can be extended. Specifically, if m appears in $\mathit{Arb}(\mathit{kcs}, \mathit{ms})$, then ext is a sequence of names in $\overline{\mathit{prin}}(\mathit{kcs})$; if m does not appear in an element of Arbs, ext is the empty sequence.

(iii) If m appears in $\pi_2(h(me))$ in an element of Arb, then $\mathit{mul} = ?$.

To check (i), we start by using the function $\mathit{getMsgArb} \in \mathit{Val} \rightarrow \mathit{Set}(\mathit{MsgArb})$, which extracts all elements of MsgArb that appear in a value, after applying the following conversion. If a value contains the pair $s : \mathit{Msg}$ with $s \notin \mathit{SMsg}$, we have no information about the signer, destination, *etc*, of the message it represents, so that pair is replaced with $\_ : \mathit{Arb}(\mathit{KC}, \mathit{SMsg}_0)$, which represents a completely arbitrary message. Let $\mathit{unknownToArb} \in \mathit{Val} \rightarrow \mathit{Val}$ denote this conversion. Then

$$\mathit{getMsgArb}(v) = (\overline{\pi_1}(v) \cap \mathit{SMsg}) \cup (\overline{\pi_1}(\mathit{unknownToArb}(v)) \cap \mathit{Arbs}). \tag{5.31}$$

After choosing $ma \in \mathit{getMsgArb}(\pi_2(h(me)))$, conditions (i) and (ii) are checked using the functions $\mathit{getMsgs} \in \mathit{MsgArb} \rightarrow \mathit{Set}(\mathit{SMsg})$ and $\mathit{getKeys} \in \mathit{MsgArb} \rightarrow \mathit{Set}(\mathit{KC})$, which extract the messages and key constants, respectively, that appear in an element of MsgArb:

$$\begin{aligned} \mathit{getMsgs}(ma) &= \textbf{match } ma \textbf{ with} && (5.32)\\ &\quad |\mathit{Arb}(\mathit{kcs}, \mathit{ms}) \rightarrow \mathit{ms} \\ &\quad |\_ \rightarrow \{ma\} \\ \mathit{getKeys}(ma) &= \textbf{match } ma \textbf{ with} && (5.33)\\ &\quad |\mathit{Arb}(\mathit{kcs}, \mathit{ms}) \rightarrow \mathit{kcs} \\ &\quad |\_ \rightarrow \emptyset \end{aligned}$$

Third Condition. The third condition is:

$$(\forall \ell \in S : * \notin \overline{\pi_2}(\pi_1(\ell)) \Rightarrow |h^{inv}(\ell)| = 1). \tag{5.34}$$

This says: if the multiplicity of a ms-atom isn't $*$, then that ms-atom corresponds to at most one element of mes.

Fourth Condition. The fourth condition is:

$$\{?, *\} \cap \overline{\overline{\pi_2}}(\overline{\pi_1}(S)) \neq \emptyset \Rightarrow \mathit{mul} = ?)). \tag{5.35}$$

This says: if the multiplicity of any ms-atom in S is not definite, then mul must be ?.

Fifth Condition. The fifth condition requires that the set *mes* of extended messages satisfy $valid_M(src, svc)(mes)$, where $valid_M(src, svc) \in Set(MsgExt) \to \mathbf{B}$. Function $valid_M$ uses $getPath \in SMsg \to ((Seq(Name) \times Svc) \cup \{\bot\})$ to extract the sequence of components visited by each message and the service to which the message is destined; while extracting this path, *getPath* checks that the signatures match the destinations, and returns $\bot$ if they don't match:

$$
\begin{aligned}
getPath(m) = \mathbf{match}\ m\ \mathbf{with}\ & \\
& |msg_0(kc, data, dest) \to \langle \langle\!\langle prin(kc) \rangle\!\rangle, dest \rangle \\
& |msg(kc, data, dest, m) \to \mathbf{match}\ getPath(m')\mathbf{with} \\
& \qquad |\bot \to \bot \\
& \qquad |\langle \sigma, svc \rangle \to \mathbf{if}\ svc = provides(prin(kc))\ \mathbf{then} \\
& \qquad\qquad \langle \sigma \cdot \langle\!\langle prin(kc) \rangle\!\rangle, dest \rangle \\
& \qquad\quad \mathbf{else}\ \bot.
\end{aligned}
\tag{5.36}
$$

$valid_M(src, svc)$ checks that the extended messages all (i) started from *src*, (ii) are destined for *svc*, (iii) visited the same sequence of services, and (iv) are signed by different replicas of the last visited service. For an extended message with a non-empty extension, we drop condition (ii), corresponding to the conservative approximation that the last destination contained in the extension may be arbitrary, even if the extension ends with the name of a non-faulty component. In other words, as mentioned above, although a non-faulty server ordinarily sends moving agents to *next*, if the server receives arbitrary inputs, we allow the possibility that it gets "confused" and sends the moving agent to an arbitrary service. Instead, for an extended message $\langle m, ext \rangle$ with non-empty extension, condition (ii) is replaced with the requirement that m is destined for the service provided by the first element of ext. Thus,

$$
\begin{aligned}
& valid_M(src, svc)(S) = \\
& \quad \textbf{if } (\exists \langle m, ext\rangle \in S : m \neq \bot \wedge getPath(m) = \bot) \textbf{ then } \text{false} \\
& \quad \textbf{else let } path = (\lambda \langle m, ext\rangle : S.\ \textbf{if } m = \bot \textbf{ then } ext \textbf{ else } \pi_1(getPath(m)) \cdot ext) \\
& \qquad \textbf{in } \wedge (\forall x \in S : first(path(x)) = src) \\
& \qquad\quad \wedge (\forall \langle m, ext\rangle \in S : \wedge\ ext = \varepsilon \Rightarrow \pi_2(getPath(m)) = svc \\
& \qquad\qquad\qquad\qquad\qquad\quad \wedge\ ext \neq \varepsilon \Rightarrow \vee\ m = \bot \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vee\ \pi_2(getPath(m)) = provides(first(ext))) \\
& \qquad\quad \wedge (\forall x_1, x_2 \in S : x_1 \neq x_2 \Rightarrow \\
& \qquad\qquad \wedge\ \overline{provides}(rest(path(x_1))) \in Seq(Svc) \\
& \qquad\qquad \wedge\ \overline{provides}(rest(path(x_1))) = \overline{provides}(rest(path(x_2))) \\
& \qquad\qquad \wedge\ |path(x_1)| > 1 \Rightarrow last(path(x_1)) \neq last(path(x_2))),
\end{aligned}
\tag{5.37}
$$

where $\overline{provides}$ is the pointwise extension of $provides$ to sequences of names, $rest$ returns the "rest" of a sequence (i.e., the sequence with its first element, if any, removed), and $first$ and $last$ return the first and last element of a sequence, respectively. It doesn't matter here what $first$ and $last$ return on the empty sequence.

Determining the Server's Outputs

The result of $server_1$ is determined from $mmes$ as follows. If $mmes$ is empty, then moving agents initiated by this source cause no outputs from this server. If $mmes$ is non-empty, then we gather a set ds of symbolic values corresponding to possible results of the majority vote on the data values in the inputs from the source. Recall that a server produces outputs only if it receives the *same* value from a majority of replicas of a service (one can think of a source as a service with one replica, so one value already forms a majority); thus, $mes \in Set(MsgExt)$ causes an output only if the data values in the concrete messages represented by the extended messages in mes are equal, in which case we can obtain a symbolic value representing that common data value by using $getData$ to extract the symbolic data value from any extended message in mes whose extension is empty. The function $getData \in SMsg \rightarrow SVal_0$ extracts the symbolic data from a message:

$$
\begin{aligned}
getData(s) = {} & \textbf{match } s \textbf{ with} \\
& |msg_0(\_, d, \_) \rightarrow d \\
& |msg(\_, d, \_, \_) \rightarrow d
\end{aligned}
\tag{5.38}
$$

On the other hand, if all extended messages in mes have non-empty extensions, then the common data value (if any) could be arbitrary, so $\bot$ is added to ds.

If ds contains a single symbolic value d, then d represents the result of the majority vote. The output message is signed by kc, contains as data the symbolic value $\widetilde{svc}(d)$, and is destined for the service $next$. The set of possibilities for the input message that is included in the output message is computed by transforming each extended message in $\bigcup_{mes \in \overline{\pi_2}(mmes)} mes$ into an element of $SMsg$ using the function $MEtoMsg \in Svc \times MsgExt \rightarrow SMsg$, defined by

$$\begin{aligned} MEtoMsg(next, \langle m, ext \rangle) = {} & \textbf{if } ext = \varepsilon \textbf{ then } m \\ & \textbf{else let } svc = provides(last(ext)) \\ & \quad \textbf{in let } m_1 = MEtoMsg(svc, \langle m, allbutlast(ext) \rangle) \\ & \quad \textbf{in let} d = apply(\widetilde{svc}, \langle\!\langle getData(m_1) \rangle\!\rangle) \\ & \quad \textbf{in } apply(msg, \langle\!\langle K_{last(ext)}, d, next, m_1 \rangle\!\rangle), \end{aligned} \tag{5.39}$$

where for a sequence s, $allbutlast(s)$ is s with the last element removed. Determining the multiplicity of the outgoing messages is straightforward. Note that, since we are assuming that each source initiates at most one moving agent, each server sends at most one "burst" of messages (i.e., a message to each provider of some service) corresponding to each source.

If ds contains contains $\bot$ or contains multiple symbolic values, then the result of the majority vote is not known exactly, so we adopt a coarse approximation, representing the output by the abstract value $mkArb(\{kc\}, \overline{\pi_2}(lbls))$. $tagOfSrc \in Src \rightarrow Tag$ returns a different tag for each source; this ensures that the ms-atoms produced by different calls to $server_1$ from a single call to $server$ are distinct. Note that it is safe to use a tag of 0 in case d is uniquely determined, since then the input message msg_{in} included in the output ms-atom ensures uniqueness of the output ms-atom.

Finally, we argue that restricting to extensions of length n does not affect the result of $server_1$; thus, the restriction is needed not for soundness but for termination. Informally, n is 1 more than the length of the longest sequence of signtures on any message that the server might have received. In the definition of n in Figure 5.4, note that max returns the maximal element of a set of natural numbers, and for convenience, we define $\pi_1(\bot) = \bot$ and $|\bot| = 0$. Suppose $\langle mul, mes \rangle$ would be added to $mmes$ if $MsgExt(n)$ were replaced with $MsgExt$ in the definition of $mmes$. We argue that this would not affect the result of $server_1$. Let $S \subseteq lbls$ be some set of ms-atoms that justifies the inclusion of $\langle mul, mes \rangle$ in $mmes$. Since $\langle mul, mes \rangle$ was not already in $mmes$, mes must contain an extension of length greater than n. Let i be the length of the shortest extension in mes. Since $valid_M(src, svc)(mes)$

holds, all elements of mes have the same "path length", i.e., for all $me \in mes$, $|path(me)|$ is the same. Since a message contributes at most $n-1$ to the length of $path(me)$, this common "path length" is at most $(n-1)+i$ and $i \geq 2$.

Let mes' be mes with the last $i-1$ elements of each extension removed. We claim that $\langle mul, mes' \rangle$ is in $mmes$. It is straightforward to show that $valid_M(src, svc)(mes')$ holds, and that the same set S of ms-atoms can be used to justify inclusion of $\langle mul, mes' \rangle$ in $mmes$, provided the extensions in mes' are of length at most n. The common "path length" for mes' is at most $(n-1)+i-(i-1)$, which simplifies to n, so the length of the longest extension in mes' is at most n. Thus, $\langle mul, mes' \rangle$ is in $mmes$.

Now consider the result of $server_1$. The length of the shortest extension in mes' is $i-(i-1)$, so every extension in mes' is non-empty, so ds contains $\perp$. So, adding $\langle mul, mes \rangle$ to $mmes$ only provides a redundant justification for the inclusion of $\perp$ in ds.

5.2.2 Other Input-Output Functions

Definition of *source***.** For $kc \in KC$, $data \in SVal$, and $dest \in Svc$, the input-output function $source(kc,data,dest)$ represents a source with private key kc that initiates a moving agent by sending a message containing symbolic data value $data$ to the replicas of service $dest$:

$$
\begin{aligned}
&source(kc, data, dest) = \\
&\quad (\lambda fail\!:\!\{OK\}.\ (\lambda h\!:\!Hist.\ (\lambda x\!:\!Name. \\
&\qquad \textbf{if } provides(x) = dest \textbf{ then } \langle \{\langle 1, msg_0(kc, data, dest)\!:\!Msg, 0\rangle\}, \emptyset\rangle \\
&\qquad \textbf{else } \langle \emptyset, \emptyset\rangle))).
\end{aligned}
\tag{5.40}
$$

Definition of *broker***.** The input-output function for a consolidator is almost the same as that for a server, except that a consolidator does not apply an operator to the outgoing data or sign its outgoing messages.

For $svc \in Svc$, $n \in \mathsf{N}$, $next \in Name$, and $nbrs \subseteq Name$, $broker(svc, n, next, nbrs)$ is given by the right side of (5.22) with $server_1(kc, svc, n, next, src, lbls, x, evsdrprs)$ replaced with $consolidator_1(svc, n, next, src, lbls, x, evsdrprs)$, and with $\{kc\}$ replaced with $\emptyset$ in both occurrences of $mkArb$.

In turn, $consolidator_1(n, next, src, lbls, x, evsdrprs)$ is given by the right side of the equation in Figure 5.4 with:

- $\langle mul, ms \times \{Msg\}, 0\rangle$ replaced with $\langle mul, d : Data, 0\rangle$,

- both occurrences of $provides(x) = next$ replaced with $x = next$ (since $next$ here is the name of the actuator, not a service it provides), and
- $mkArb(\ldots)$ replaced with $Data$ (since we assume a "confused" consolidator still sends only data values in $Data$ to the actuator).

Note that the bindings of S, ℓ_{in}, msg_{in}, and msg are dead code in $consolidator_1$ and can be eliminated.

5.3 Analysis of Perturbed Behavior

For the system in Figure 5.3, nf is given by

$$\begin{aligned}
nf(S) &= source(K_S, X, F)\\
nf(F_i) &= server(K_{F_i}, F, 3, G, nbrs \setminus \{F_i\})\\
nf(G_i) &= server(K_{G_i}, G, 3, B, nbrs \setminus \{G_i\})\\
nf(B) &= broker(B, 3, A, nbrs \setminus \{B\})\\
nf(A) &= (\lambda fail\colon\{OK\}.\ (\lambda h\colon Hist_{FC}.\ (\lambda x\colon Name.\ \langle\emptyset,\emptyset\rangle))).
\end{aligned}$$

where $nbrs = \{F_1, F_2, F_3, G_1, G_2, G_3, H_1, H_2, H_3, B\}$. For that system, $provides$ is given by

$$\begin{aligned}
provides(F_i) &= F\\
provides(G_i) &= G\\
provides(B) &= B.
\end{aligned}$$

We take $tagOfSrc(S) = 0$.

5.3.1 Failure of Visited Services.

Consider the failure scenario in which F_1 and G_2 fail. A straightforward calculation shows that the fixed-point is the same as the run in Figure 5.3, except that the outputs of the faulty components are different, and other components send messages to the faulty components as a result of eavesdropping. Specifically, for $x \in \{F_1, G_2\}$ and $y \in nbrs \setminus \{x\}$, the edge $\langle x, y\rangle$ is labeled with

$$\{evsdrp, Arb(\{K_{F_1}, K_{G_2}\}, ms_1)\}^*,$$

where $ms_1 = \{m^0, m^1_2, m^1_3, m^2_{2,1}, m^2_{3,1}, m^2_{2,3}, m^2_{3,3}\}$. Also, for $x \in \{F_1, G_2\}$ and $y \in nbrs \setminus \{F_1, G_2\}$, the edge $\langle y, x\rangle$ is labeled with all the output ms-atoms of component y in Figure 5.3, but with the multiplicities changed to $?$.

To help the reader verify this calculation, we give the values of *mmes* obtained in the evaluation of $server_1$ and $consolidator_1$ when the fixed-point has been reached; values of *mmes* in previous iterations of the fixed-point calculation are subsets of these. When evaluating $server_1$ for F_2 and F_3, the value of *mmes* is

$$\bigcup_{mul\in\{?,1\}} \{\langle mul, \{\langle m^0, \varepsilon\rangle\}\rangle\}.$$

Note that the inclusion of $\{\langle ?, \{\langle m^0, \varepsilon\rangle\}\rangle\}$ in *mmes* is justified by $S = \{\langle 1, m^0, 0\rangle\}$ and $S = \{\langle *, Arb(\{K_{F_1}, K_{G_2}\}, ms_1), 0\rangle\}$. The latter reflects the possibility that a faulty component obtains m^0 by eavesdropping and forwards m^0 to F_2 or F_3. Since the network is asynchronous, F_2 or F_3 might receive this forwarded copy before the original copy sent by the source.

When evaluating $server_1$ for G_1 and G_3, the value of *mmes* is

$$\begin{array}{c}\bigcup_{mul\in\{?,1\}}\{\langle mul, \{\langle m^1_2, \varepsilon\rangle, \langle m^1_3, \varepsilon\rangle\}\rangle\}\\ \cup\{\langle ?, \{\langle m^1_2, \varepsilon\rangle, \langle m^0, \langle\!\langle F_1\rangle\!\rangle\rangle\}\rangle, \langle ?, \{\langle m^1_3, \varepsilon\rangle, \langle m^0, \langle\!\langle F_1\rangle\!\rangle\rangle\}\rangle\}.\end{array}$$

The presence of $\{\langle m^1_2, \varepsilon\rangle, \langle m^0, \langle\!\langle F_1\rangle\!\rangle\rangle\}$ reflects the possibility that the faulty server F_1 sends the correct value to G_1 or G_3. In particular, if the extended message $\langle m^0, \langle\!\langle F_1\rangle\!\rangle\rangle$ contains the same data value as m^1_2 (namely, the value represented by $\widetilde{F}(X)$), then these two messages can cause an output; this is why they are included in *mmes* and thereby used to justify the inclusion of $\widetilde{F}(X)$ in *ds*. If $\langle m^0, \langle\!\langle F_1\rangle\!\rangle\rangle$ contains a different value than m^1_2, then these two messages do not cause an output, so these two messages do not justify adding any other data values to *ds*. The explanation for $\{\langle m^1_3, \varepsilon\rangle, \langle m^0, \langle\!\langle F_1\rangle\!\rangle\rangle\}$ is analogous.

When evaluating $consolidator_1$ for B, the value of *mmes* is

$$\begin{array}{l}\bigcup_{mul\in\{?,1\}}\bigcup_{i,i'\in\{2,3\}}\bigcup_{j\in\{1,3\}}\bigcup_{j'\in\{1,3\}\setminus\{j\}}\{\langle mul, \{\langle m^2_{i,j}, \varepsilon\rangle, \langle m^2_{i',j'}, \varepsilon\rangle\}\rangle\}\\ \cup\bigcup_{i\in\{2,3\},j\in\{1,3\}}\{\langle ?, \{\langle m^2_{i,j}, \varepsilon\rangle, \langle m^0, \langle\!\langle F_1, G_2\rangle\!\rangle\rangle\}\rangle\}\\ \cup\bigcup_{i,i'\in\{2,3\},j\in\{1,3\}}\{\langle ?, \{\langle m^2_{i,j}, \varepsilon\rangle, \langle m^1_{i'}, \langle\!\langle G_2\rangle\!\rangle\rangle\}\rangle\}.\end{array}$$

Extended messages with non-empty extensions are present here for analogous reasons.

Note that the voting in each step "heals" the effects of failure of a minority of the replicas in the previous step. For example, failure of F_1 no longer perturbs the output of G_1, so the system tolerates failure of F_1 and G_2.

5.3.2 Failure of Unvisited Services.

Consider the failure scenario in which H_1, H_2, and H_3 fail. A straightforward calculation shows that the fixed-point is the same as the run in Figure 5.3, except that the outputs of the faulty components are different, and other components send messages to the faulty components as a result of eavesdropping. Specifically, for $x \in \{H_1, H_2, H_3\}$ and $y \in nbrs \setminus \{x\}$, the edge $\langle x, y \rangle$ is labeled with

$$\{evsdrp, Arb(\{K_{H_1}, K_{H_2}, K_{H_3}\}, ms_2)\}^*,$$

where

$$ms_2 = \{m^0\} \cup (\bigcup_{i \in \{1,2,3\}} \{m_i^1\}) \cup (\bigcup_{i,j \in \{1,2,3\}} \{m_{i,j}^2\}).$$

Also, for $x \in \{H_1, H_2, H_3\}$ and $y \in nbrs \setminus \{H_1, H_2, H_3\}$, the edge $\langle y, x \rangle$ is labeled with all the output ms-atoms of component y in Figure 5.3, but with the multiplicities changed to ?.

5.3.3 Failure of Visited and Unvisited Services.

Consider the failure scenario in which F_1, H_1, and H_2 fail. As the reader may have suspected, the protocol described above does not tolerate this failure scenario. To help see why, we trace the fixed-point calculation. Let $faulty = \{F_1, H_1, H_2\}$. Let r_i denote the run obtained after i steps. Thus, $r_0 = \perp_{Run}$. Run r_1 is the same as r_0 except as follows:

x	y	$r_1(\langle x, y \rangle)$
$\{S\}$	$\{F_1, F_2, F_3\}$	m^0
$faulty$	$nbrs \setminus \{x\}$	$\{evsdrp, Arb(\{K_x\}, \emptyset)\}^*$

Run r_2 is the same as r_1 except as follows:

x	y	$r_2(\langle x, y \rangle)$
$\{F_2, F_3\}$	$\{G_1, G_2, G_3\}$	m_i^1
$\{F_2, F_3\}$	$faulty$	$(m_i^1)^?$
$\{F_1\}$	$nbrs \setminus \{F_1\}$	$\{evsdrp, Arb(\{K_{F_1}, K_{H_1}, K_{H_2}\}, \{m^0\})\}^*$
$\{H_1, H_2\}$	$nbrs \setminus \{x\}$	$\{evsdrp, Arb(\{K_{F_1}, K_{H_1}, K_{H_2}\}, \emptyset)\}^*$

Consider the evaluation of $server_1$ for F_2 in computing run r_3. The value of $mmes$ is $\{\langle 1, \{\langle m^0, \varepsilon \rangle\}\rangle, \langle ?, \{\langle m^0, \langle\langle F_1, H_1 \rangle\rangle \rangle, \langle m^0, \langle\langle F_1, H_2 \rangle\rangle \rangle\}\rangle\}$. The latter element corresponds to the possibility that H_1 and H_2 sent their private keys to F_1, which used its own key and those keys together to fool F_2. Thus, ds is $\{X, \perp\}$, so the output of F_2, sent to $nbrs \setminus \{F_2\}$, is an element of $Arbs$. The other non-faulty servers and the consolidator are also fooled. For $x \in \{F_2, F_3, G_1, G_2, G_3\}$ and $y \in nbrs \setminus \{x\}$, $r_3(\langle x, y \rangle) = Arb(\{K_{F_1}, K_{H_1}, K_{H_2}, K_x\}, \{m^0\})^*$. The consolidator sends $Data^?$ to the actuator (and to the eavesdroppers), so the fault-tolerance requirement is already violated. Informally, the protocol breaks down because each output message contains only one input message; a "quorum" of input messages should be included in each output message.

The fixed-point r is given by:

x	y	$r(\langle x, y \rangle)$
$\{S\}$	$\{F_1, F_2, F_3\}$	m^0
$\{F_2, F_3, G_1, G_2, G_3\}$	$nbrs \setminus \{x\}$	$Arb(kcs, \{m^0\})^?$
$faulty$	$nbrs \setminus \{x\}$	$\{evsdrp, Arb(kcs, \{m^0\})\}^*$
$\{B\}$	$\{A\} \cup faulty$	$Data^?$

where $kcs = \bigcup_{i \in \{1,2,3\}} \{K_{F_i}, K_{G_i}, K_{H_i}\}$, and where the other edges of r are labeled with the empty set.

5.4 Discussion

5.4.1 Symbolic *vs.* Abstract Values

In the above model, we introduced constants to represent names of services. Since symbolic values are intended primarily for representing relationships between values, one could argue that it would be more appropriate to use abstract values to represent names of services. For example, we could distinguish the destination service of a message in the abstract value instead of the symbolic value, just as the sender of a message is distinguished in the abstract value in the analysis of reliable broadcast in Section 4.1. We use constants here, instead of abstract values, only to improve the appearance of the formulas. Symbolic values are needed to track keys and data, so it is tidier to use symbolic values for all parts of the message, rather than splitting the information between the symbolic and abstract values. Finally, we note that this issue may be artificial, since it may be possible to construct a more flexible version of the framework in which symbolic and abstract values intermingle within terms.

5.4.2 Abstracting from Paths

One limitation of the above analysis is that it does not allow abstraction from the path traveled by the moving agent. For example, although data values are abstracted (since they are represented by variables), names are not abstracted, so a graph like the one in Figure 5.3 represents only moving agents that follow one specific path through the network. One way to eliminate this restriction is to introduce a distinction between abstract names and concrete names, and allow the correspondence to be chosen to "match" the path actually traveled by the moving agent. This is in the same spirit as our treatment of key constants: we require that the interpretation of key constants as keys satisfy some sanity conditions, but we do not fix the interpretation of key constants as specific keys.

However, abstracting from names poses obvious difficulties. In particular, when messages are sent to abstracted names, it's generally not clear which sets of messages form the input histories of which components. If the system is very homogeneous (i.e., the components whose names are abstracted all behave similarly, as do the servers in the previous section) and the processing of different messages by each component is sufficiently independent, these uncertainties are manageable, but in general, the difficulties are formidable.

5.4.3 Approximation of Message Extensions

Since elements of $Arbs$ contain a set (not a sequence) of key constants, the analysis does not keep track of the order in which keys in kcs might appear in message extensions. Some abstraction from the order of signatures in the extension is essential for the analysis to terminate, since extensions can be of unbounded length but contain only a finite number of different keys.

Chapter 6

Related and Future Work

This chapter puts the work described in this thesis in context. Section 6.1 looks to the past, discussing related work. Section 6.2 looks to the future, discussing several directions for future work.

6.1 Related Work

Abstract Interpretation

Abstract interpretation is an extremely general framework for program analysis [AH87]. Our analysis is distinguished from pure abstract interpretation by the use of symbolic values to track relationships between values; thus, our fixed-point analysis incorporates symbolic computation as well as abstract interpretation. To some extent, symbolic values can be simulated in the context of abstract interpretation, by introducing statically an abstract value corresponding to each symbolic value that will be needed in the course of the analysis; this approach and its limitations are discussed further below.

If one ignores for the moment our use of symbolic values, our analysis is a form of abstract interpretation. However, there is some qualitative difference between our analysis and most traditional uses of abstract interpretation, which typically deal with domains whose structure mirrors the structure of the *program* being analyzed. This is the case, for example, for analyses that infer types for *expressions*, or computing def-use chains between *occurrences of variables*, or determine whether certain *expressions* are constant. Such analyses are typically designed to be incorporated into compilers, so low computational cost is essential.

Fault-tolerance analysis focuses on properties of a system's behavior whose verification may require (in general) high computational complexity. So, the domains tend to be based less on the structure of the system and more on the structure of its executions. While there is no sharp distinction here, an example may help illustrate the difference. Consider a restricted form of fault-tolerance analysis that yields, for each pair of components, a single ms-atom describing the communication between those components. This restricted analysis has a static flavor, since the structure of the domains corresponds closely to the structure of the system, but it is sometimes inadequate. For example, in the analysis of FIFO reliable broadcast in Section 4.1, it was important to distinguish between the values and multiplicities of the two messages whose ordering is being checked. The restricted analysis would be too imprecise.

The abstraction mechanisms in our framework have been designed to provide great flexibility in the precision with which systems are modeled; for example, when writing an input-output function, one can choose to use at most one ms-atom in each poset, or one can choose to use many. In general, the framework supports rather than enforces approximations. With this flexibility comes a burden on the user to select appropriate approximations. This burden seems inevitable, since verification of asynchronous systems with channels of unbounded capacity is undecidable [BZ83].

Failure Propagation and Transformation Notation

In their work on applying HAZOP and FMECA to computer-based systems, McDermid *et al.* [FM93,FMNP94,MNPF95] have developed an approach to validation of fault-tolerance that shares with our work the idea of characterizing each component by how it generates and propagates "failures" (perturbations). Their approach is embodied in their *failure propagation and transformation notation* (FPTN). FPTN achieves simplicity at the expense of generality. One can choose for each system the relevant kinds of perturbations, but each kind of perturbation must be represented by a single boolean value. For example, one bit might indicate omission of an output; another bit might indicate arbitrary corruption of an output value.

Our framework is parameterized by the domains $AVal$ and $\Delta AVal$, so the representation can be customized for different application domains. For example, in the analysis of the cryptography-based protocol in Chapter 5, we introduce Arb and use it to represent (roughly speaking) the set of cryptographic information known to each faulty component.

Fault-Tolerance as Self-Similarity

Fault-tolerance as self-similarity [Web93] shares with our work the goals of separating the specification of fault-tolerance from other correctness requirements and developing specialized techniques for verification of fault-tolerance requirements. To achieve this, Weber adopts a rigid notion of fault-tolerance: he equates fault-tolerance with fault-masking. In other words, he defines a system to be fault-tolerant iff its visible behavior in the presence of faults is the same as in the absence of faults. This is attractive because fault-masking properties can be expressed in terms of bisimilarity:[1] a system masks a fault iff the fault causes transitions only between bisimilar states. Thus, this approach allows one to leverage work on checking bisimilarity [CS96]. This technique is interesting but limited in applicability to systems in which faults are completely masked.

Abstraction in Model Checking

Abstractions play an important role in our work. Clarke, Grumberg, and Long studied the use of the abstractions in conjunction with temporal-logic model-checking [CGL92,CGL94]. Their notion of abstraction corresponds roughly to abstract interpretation and to our notion of abstract values, though in their state-based approach, multiplicities are not explicit, so abstractions are used only for (data) values. They also propose so-called *symbolic abstractions*, which are just abbreviations for finite families of (non-symbolic) abstractions. Our symbolic values are closer to the technique they sketch in the last paragraph of [CGL94] for dealing with infinite-state systems.

In Kurshan's automata-based verification methodology, approximations are embodied in *reductions* between verifications [Kur89,Kur94]. A typical use of a reduction is to collapse multiple states of an automaton into a single state of a reduced automaton; this is analogous to introducing abstract values. Relationships between concrete values can be expressed in Kurshan's methodology using parameterized families of reductions, reminiscent of Clarke, Grumberg, and Long's so-called symbolic abstractions. For example, to verify that a bounded-length queue containing numbers in $[1..n]$ does not drop items, one can use a family of reductions that collapses the set $[1..n]$ of concrete data values to two abstract data values: the one being "focused on", specified as a parameter, and "everything else" [Kur94, Appendix D]. The relationship captured here is equality of each value with the concrete

[1]Roughly, states s and s' are bisimilar if the set of visible behaviors possible starting from state s equals the set of visible behaviors possible starting from state s'. For details, see [Mil89].

value being focused on.[2] The parameter of the reduction corresponds in our framework to use of a variable representing the focused-on value. For problems involving related values (e.g., X and $F(X)$), the reductions must introduce an abstract data value representing each such value. In effect, one must determine *in advance* all relevant symbolic values (e.g., all symbolic values that would arise during the fixed-point calculation in our framework) and introduce an abstract data value for each. Note that the resulting abstractions are not modular, since they may include abstract values corresponding to symbolic values that contain variables local to different processes.

In our framework, it is not necessary to require that the items in the queue come from a finite set. Using symbolic multiplicities, it is not even necessary to require that the queue have bounded length. In this case, the input-output function would need to incorporate non-trivial abstractions, which would need to be verified manually.

An attractive feature of Clarke and Long's work and Kurshan's work is that abstractions (or reductions) are specified as homomorphisms and applied to programs (or automata) automatically. We plan to look at mechanized support for applying abstractions in our framework.

Verification of Byzantine Agreement Algorithms

Our approach to fault-tolerance analysis is similar in spirit to the state-exploration technique of Gong, Lincoln, and Rushby [GLR95]. In both cases, an automated analysis is used to compute and evaluate the behavior of the system separately in each failure scenario of interest. However, the work described in [GLR95] apparently does not include the use of any form of abstraction.

6.2 Future Work

This section describes some possible extensions of and variations on our work.

Inter-channel orderings. Because our representation of runs contains no inter-channel orderings, our analysis suffers (in effect) from the merge anomaly [Kel78,Bro88]. Specifically,

[2]Although the method just described is one way to prove that the queue doesn't drop items, it is apparently not the method Kurshan has in mind, since in [Kur94, Appendix D], the reduction is not actually parameterized: the concrete value being focused on is fixed to be 1. Presumably Kurshan has in mind the following method. Let $Qred_i$ be the reduced automaton obtained from the reduction h_i that focuses on the concrete value i. It suffices to check that $Qred_i$ does not drop items, and that for each $i \in [2..n]$, $Qred_i$ is isomorphic to $Qred_1$. Note that this method still requires iterating over $O(n)$ reductions.

when modeling a non-strict component, one is forced to use a conservative approximation. One way to remedy this is to adopt at the abstract level some analogue of the approach used at the concrete level. However, this would lead to an inefficient analysis, since modeling each component by a set of functions causes an explosion in the number of combinations of functions that must be considered in the analysis of a system.

A more promising remedy is to add inter-channel orderings to the representation of runs. This is one of the ideas behind Brock and Ackermann's scenarios [BA81,Bro83] and Pratt's model of processes [Pra82]. In both of those models, a process is a (potentially infinite) set, each element of which represents a complete (and potentially infinite) behavior. This contrasts with Kahn's model, in which a process is a function that can be used to determine the behavior of a system incrementally *via* a fixed-point calculation. So, neither scenarios nor Pratt's model is directly suitable as the basis for an efficient fixed-point analysis. However, a related approach seems feasible. Roughly, one adds to the representation (2.29) of runs a partial order on the ms-atoms that appear in the run. For example, a run could be a pair $\langle r, \prec \rangle$, where $r \in \mathit{Name} \rightarrow \mathit{Hist}$ and $\prec \in \mathit{Order}(\bigcup_{\langle x,y \rangle \in \mathit{Name} \times \mathit{Name}} \{\langle x, y \rangle\} \times r(y)(x))$. Input-output functions must be extended to deal with these inter-channel orderings; in particular, the domain of input-output functions is extended with a partial order on the input ms-atoms (from all sources), and the range is extended with orderings between input and outputs (i.e., causal dependencies of outputs on inputs) and orderings between outputs. The details of this approach remain to be worked out (proving soundness of the analysis apparently becomes much more complicated).

Integrating symbolic and abstract values. As discussed in Section 5.4.1, the separation between symbolic and abstract values enforced in the current framework is sometimes awkward. It would be interesting to explore ways of allowing a tighter integration of the two. For example, we might label each subterm of a symbolic value with an abstract value. We should also allow the wildcard to appear within an expression. This would allow values like $\mathit{plus}(X : \mathbf{N}, \_ : ?) : \mathbf{N}$, which represents the set of numbers $\{\rho(X), \rho(X)+1\}$. We might want to refer to this value in other ms-atoms, so we should also allow a variable to be associated with this expression as a whole; thus, we might use a value of the form $(\mathit{plus}(X : \mathbf{N}, \_ : ?)$ as $Y) : \mathbf{N}$, where Y represents this value as a whole.

Dynamic creation of components. The frameworks described in this thesis cannot directly represent systems in which components are created dynamically. Such systems can be modeled by including a sufficient number of idle components and sending special messages to activate them when they are actually created. One difficulty with this approach is that it may be difficult to determine in advance what is a "sufficient number". Also, this approach is awkward if the input-output functions associated with new components are determined dynamically (i.e., when components are created), since our current frameworks use a fixed mapping nf from names to input-output functions. This dynamic behavior can be mimicked (albeit awkwardly) using input-output functions that are (in effect) interpreters for the language in which input-output functions are written.

Instead of developing techniques to simulate dynamic component creation in the current frameworks, we can extend the frameworks to represent dynamic component creation directly. One approach is to extend the range of input-output functions to include component creation events. A component creation event must contain the input-output function associated with the new component, so the type of input-output functions is of the form

$$IOF = InMsgs \to OutMsgs \times Set(Name \times IOF).$$

Making sense of recursive definitions of this form requires domain theory, as opposed to the set theory used in this thesis. An alternative approach is to give an operational semantics, in the form of a transition system for some specific agent language (e.g., lambda calculus extended with some primitives for agent creation and communication); this is the approach taken for actors in [AMST93].

Automated support for abstractions. As mentioned in Section 6.1, we plan to look at mechanized support for applying abstractions in our framework. This would reduce the burden of proving that input-output functions represent processes. To provide automated support, specific languages must be chosen for expressing processes and input-output functions. For the former, a language like PROMELA could be used [Hol91]; for the latter, a functional subset of CAML. An abstraction would be embodied as a transformation T that maps a process P to an input-output function $T(p)$ that represents P (and incorporates the abstraction). In general, correctness of the transformation would be verified manually; that is, one would prove that, for all processes P in a certain class, $T(P)$ represents P. The benefits of expressing an abstraction as a transformation are that the transformation can be applied automatically and the effort of verifying the transformation can be amortized by applying it to many systems.

State-based fault-tolerance analysis. The history-based model developed in this thesis makes multiplicities explicit and makes the entire history of a computation available to the input-output function at every step. This gives input-output functions fine control over the approximations used to represent the contents of the channels. Indeed, this is the primary benefit of using histories. On the other hand, a state-based approach would be simpler in some respects, so it is interesting to consider the possibilities for a state-based fault-tolerance analysis that can still cope with asynchronous distributed systems with unbounded communication channels. Naturally, the contents of the channels would be included as part of the state of the system. In order to represent indefinite and unbounded multiplicities (e.g., in the output of Byzantine-faulty components, or in the output of a component that repeatedly transmits "I'm alive" messages), the contents of channels could be approximated using an appropriate generalization of regular expressions. To use this state-based approach, one must still develop representations of the components that deal with this high-level representation of the contents of channels. Assuming the state of the system includes only the *current* contents of channels—not the entire history—less information is available to these representations of the components, so controlling the use of approximations (especially approximations of multiplicities) may be more difficult.

Applications. To test and refine the approach—and the tool described in Appendix B—we must apply them to more problems. Possible applications include efficient algorithms for asynchronous Byzantine Agreement [CR93], algorithms for the certified write-all problem [KMS95,BKRS96], secure protocols for group membership and reliable broadcast [Rei96, MR96], and cryptographic protocols for fault-tolerant moving agents [MvRSS96].

Appendix A

Index of Symbols

Symbol	Page	Description
$\triangleq$	11	equal by definition
Seq	11	finite and infinite sequences
$CHist$	11	history of concrete messages
		$CHist \triangleq Name \rightarrow Seq(CVal)$
$\rightarrow$	11	constructor for signature of functions
$\langle\!\langle \cdot \rangle\!\rangle$	10	sequence
ε	11	empty sequence
$\leq_{CHist}$	11	partial order on $CHist$
$DProcess$	12	determinate process
		$DProcess \triangleq CHist \twoheadrightarrow CHist$
$\twoheadrightarrow$	12	monotonic and continuous functions
$CRun$	12	concrete run
		$CRun \triangleq Name \rightarrow CHist$
$step$	12	step function
$\leq_{CRun}$	12	partial order on $CRun$
$crun$	13	concrete run of a determinate system
$\langle \cdot \rangle$	13	tuple
$Chain$	13	chains of a partial order
$\lvert \cdot \rvert$	13, 33	length of a sequence, or size of a set
dom	13, 34	domain of a sequence or function
$\perp_{CRun}$	14	least element of $CRun$

$\perp_{CHist}$	14	least element of $CHist$
$Process$	15	(non-determinate) process
		$Process \triangleq Set(IRProcess)$
$IRProcess$	15	input-restricted process
		$IRProcess \triangleq DProcess \times Set(CHist)$
$cruns$	15	concrete runs of a system
π_i	15	project ith component of a tuple
$\circ$	15	function composition
$enabled$	15	enabledness of input-restricted process
$\wedge$	16	conjunction (infix or bullet-style)
$\vee$	16	disjunction (infix or bullet-style)
$Order$	19	strict partial orderings
$POSet$	19	strictly-partially-ordered sets
$Hist$	19	history of messages
		$Hist \triangleq Name \rightarrow POSet(L)$
Run	19	run
		$Run \triangleq Name \rightarrow Hist$
L	20	labels
		$L \triangleq Mul \times Val \times Tag$
Val	20	values
		$Val \triangleq \mathcal{P}_{fin}(SVal \times AVal) \setminus \{\emptyset\}$
$\mathcal{P}_{fin}$	20	finite subsets
$\setminus$	20	set difference
$AVal$	21	abstract values
$Interp_{Set}$	21	interpretation (of abstract values)
		$Interp_{Set}(S) \triangleq S \rightarrow Set(CVal)$
Con	21	constant
Var	21	variable
$SVal_0$	22	symbolic values except wildcard
Sym	22	symbols (constants and variables)
		$Sym \triangleq Con \cup Var$
$SVal$	22	symbolic values
		$SVal \triangleq SVal_0 \cup \{\_\}$

Mul	24	multiplicities
		$\mathit{Mul} \triangleq \mathcal{P}_{\mathit{fin}}(\mathit{SVal} \times \mathit{AMul}) \setminus \{\emptyset\}$
AMul	24	abstract multiplicities
IOF	27	input-output functions
		$\mathit{IOF} \triangleq \{f \in \mathit{Hist} \to \mathit{Hist} \mid \mathit{tagUniform}(f)\}$
$\xrightarrow{\text{inj}}$	28	injections
$=_{\mathit{POSet}(L)}$	28	equality on $\mathit{POSet}(L)$
$=_{\mathit{Hist}}$	28	equality on Hist
$\mathit{tagUniform}$	28	output is uniform WRT tags in input
$\bot_{\mathit{Run}}$	28	least element of Run
$\bot_{\mathit{Hist}}$	28	least element of Hist
$=_{\mathit{Run}}$	28	equality on Run
interp	34	partial interpretation (of symbols)
$\rightharpoonup$	34	partial functions
$\leq_{\mathit{interp}}$	34	ordering on interp
$\xrightarrow{\text{onto}}$	34	surjective (onto) funtions
$\mathit{compat}_{\mathit{Val}}$	35	compatibility of an abstract value with a concrete value
$\mathit{compat}_{\mathit{POSet}(L)}$	35	compatibility of labels with a concrete history
g^{inv}	35	generalized inverse of function g
$[\![\,]\!]_{\mathit{POSet}(L)}$	36	meaning of a partial-order of labels
$[\![\,]\!]_{\mathit{Hist}}$	36	meaning of a history
$\sqsubset_{\mathit{IOF}}$	37	meaning of an input-output function
$\sqsubset_{\mathit{Sys}}$	37	meaning of a system
Interp	37	extensions of a partial interpretation (of symbols)
$[\![\,]\!]_{\mathit{Run}}$	37	meaning of a run
$\mathit{cruns}^{\mathit{fin}}$	38	finite-length concrete runs
$\downarrow$	39	restriction of an invariant to a set of names
$\leq_{\mathit{POSet}(L)}$	43	ordering on $\mathit{POSet}(L)$
$\leq_{\mathit{InHist}}$	43	ordering on histories regarded as inputs
$\leq_{\mathit{OutHist}}$	43	ordering on histories regarded as outputs
$\leq_{\mathit{Run}}$	43	ordering on runs
$\mathit{Process}_F$	50	failure-prone process
		$\mathit{Process}_F \triangleq \{p \in \mathit{Fail} \rightharpoonup \mathit{Process} \mid \mathit{OK} \in \mathit{dom}(p)\}$

IOF_F	50	input-output function for a failure-prone component $IOF_F \triangleq \{f \in Fail \rightharpoonup IOF \mid OK \in dom(f)\}$
FS	50	failure scenarios
fs_{OK}	50	failure-free failure scenario
$cruns_F$	50	concrete runs of a system
$step_F$	50	step function for a system in a given failure scenario
$\sqsubset_{IOF_F}$	50	meaning of an element of IOF_F
B	51	booleans
$origIndep_C$	63	sanity condition for $Process_{FC}$
$consistent$	64	sanity condition for $Process_{FC}$
$Process_{FC}$	64	failure-prone process, represented using changes
$cruns_{FC}$	64	concrete runs of a system, represented using changes
L_{FC}	66	original label with changes, or new label $L_{FC} \triangleq L_{per} \cup L_{new}$
L_{per}	66	original label with changes $L_{per} \triangleq Mul \times Val \times \Delta Mul \times \Delta Val \times Tag$
L_{new}	66	new label $L_{new} \triangleq Mul \times Val \times Tag$
$\Delta AVal$	66	change to abstract value
$\Delta AMul$	67	change to abstract multiplicity
ΔVal	67	change to value $\Delta Val \triangleq \mathcal{P}_{fin}(SVal \times \Delta AVal) \setminus \{\emptyset\}$
ΔMul	67	change to multiplicity $\Delta Mul \triangleq \mathcal{P}_{fin}(SVal \times \Delta AMul) \setminus \{\emptyset\}$
$Hist_{FC}$	67	history of messages, represented using changes $Hist_{FC} \triangleq Name \to POSet(L_{FC})$
Run_{FC}	67	run of a system, represented using changes $Run_{FC} \triangleq Name \to Hist_{FC}$
IOF_{FC}	69	input-output function for a failure-prone component represented using changes
$tagUniform_{FC}$	69	output is uniform WRT tags in input
$orig$	69	projection on original behavior
$origIndep$	69	sanity condition for IOF_{FC}

Appendix B

CRAFT: A Tool for Fault-Tolerance Analysis

The non-perturbational and perturbational analysis frameworks described in Chapter 3 are implemented in a prototype tool, called CRAFT (Change-Relations for Analysis of Fault-Tolerance). This appendix sketches the structure and use of CRAFT.

B.1 Overview

CRAFT is implemented in the functional programming language CAML Light [Ler97], a dialect of Standard ML [MTH90]. CRAFT provides a collection of CAML types and functions that implement the non-perturbational and perturbational analysis frameworks described in Chapter 3. CRAFT also provides a graphical interface to facilitate entry of systems and inspection of analysis results. This section gives an overview of the use of CRAFT; the remaining sections contain more detailed descriptions of the types and functions provided by CRAFT.

To use CRAFT to analyze a system, the first step is to express the input-output functions representing system components as CAML functions of appropriate type. Note that input-output functions are written directly in the CAML programming language. This allows the full power of CAML and its libraries to be used. However, automatically checking requirements (such as uniformity with respect to tags) on these functions is harder than it would be for a more restricted language; this is partly why CRAFT leaves enforcement of such requirements to the user. CRAFT provides CAML types corresponding to IOF and IOF_{FC}. Input-output functions should have one of these two types, depending on whether a non-perturbational or perturbational analysis is desired.

Once the input-output functions have been expressed in CAML, the remaining steps depend on whether the graphical interface is being used. Without the graphical interface, the next step is to define a mapping `sys` from names (represented in CAML as strings) to input-output functions that represents the system. Mappings are constructed using functions in the `map` module in the CAML standard library. Functions in the `map` module are also used to define failure scenarios, which are represented in CAML as mappings from names to failures. Given a system *sys* and a failure scenario `fs`, functions provided by CRAFT are used to compute a run `r` representing the behavior of system `sys` in that failure scenario. To determine whether such a run satisfies the fault-tolerance requirement, we express the fault-tolerance requirement as a CAML function `ftr`, which takes a failure scenario and a run as arguments and returns a Boolean, and compute `ftr fs r`. If this returns `false`, a textual representation of the run can be inspected to help ascertain the problem. CRAFT does not currently provide a special function to automatically repeat the analysis for all possible failure scenarios for a system, but this is trivial to implement, since CRAFT does provide a function that returns a list of all possible failure scenarios for a system.

CRAFT's graphical interface is implemented using CamlTk, a CAML interface to the Tk widget library. The graphical interface can be used by CAML-illiterates to access libraries of input-output functions already written in CAML.[1] A user clicks (button1)[2] on the canvas to add a component to the system. This creates a new node at the location of the click and pops up the "Create Node" window, which is used to specify the input-output function associated with the new node. Specifically, the "Create Node" window displays the names of all the input-output functions in the library and contains fields for entering the parameters of each input-output function.[3] The user selects one of those input-output functions and enters values for its parameters (if any). For example, if the function $Voter_{FC}$ defined in (3.53) is in the library, there would be fields to enter its three parameters (namely, *srcs*, *dest*, and *aval*). The user uses the same window to select the possible failures of the new component. Figure B.1 shows the "Create Node" window for a library containing input-output functions similar to those used in the running example in Chapter 3. If "Arbitrary Failure" is selected as a possible failure, the user enters in the "Dests" field below it the names of the neighbors of

[1] For historical reasons, the representations of runs and input-output functions used in the implementation of the graphical interface are slightly different than the representations described in Section B.2.

[2] By convention, mouse buttons are numbered from left to right.

[3] The writer of a library must include in the library a special value describing the names and types of the parameters of each input-output function. CRAFT uses this special value to generate a window with appropriate fields for entering parameters.

the new component; if the new component suffers an arbitrary failure, it will send arbitrary messages to its neighbors.

After entering a system by creating a node corresponding to each system component, the user can specify a failure scenario and execute the fixed-point analysis. When each component is created, failure OK is associated with that component; thus, the default failure scenario is fs_{OK}. To change the failure associated with a component, the user clicks (Control-button1) on the corresponding node. This pops up the "Set failure status" window, which is used to select one of the possible failures of that component. Nodes for which a failure other than OK is selected are displayed with a red border (recall that in the figures in this thesis, such nodes are displayed with dots on their circumference). By repeating this procedure, the user can select any failure scenario of interest.

When a failure scenario of interest has been selected, the user selects the "Analyze" command from the pull-down menu entitled "Analyze". The fixed-point is computed, and (if the computation terminates) the result is displayed. If the computed ms-atoms are not too long or too numerous, they are displayed directly on the edges of the graph, as in Figures B.1 and B.1. The ms-atoms are color-coded (unfortunately, this is not apparent from the black-and-white printouts in the figures): black and brown text are used for the value and multiplicity, respectively, in the original part of a ms-atom; red text is used for the perturbation; and blue text is used for new ms-atoms. The edges are also color-coded: black is used for edges labeled only with perturbed ms-atoms containing the identity perturbation; red, for edges labeled only with perturbed ms-atoms such that some ms-atom contains a perturbation other than the identity; blue, for edges labeled only with new ms-atoms or perturbed ms-atoms containing only the identity perturbation; and violet, for edges labeled with both new *mas*s and perturbed ms-atoms such that some ms-atom contains a perturbation other than the identity. If the textual representation of the ms-atoms does not fit on an edge, the edge is still color-coded, but the ms-atom is elided. The user can click (Shift-button2) on an edge to pop up a window showing all of the ms-atoms on that edge.

Figure B.1 contains a screen-dump of the result of the analysis in the absence of failures for the system used as the running example in Chapter 3. Figure B.1 shows the result of the analysis when component F1 suffers a value failure. Note that "TopΔ" corresponds to $\top_{\Delta V}$.

CRAFT supports miscellaneous other commands: saving and loading systems, showing intermediate results of the fixed-point calculation (i.e., "single-stepping" through the calculation), deleting and moving nodes, selecting different fonts and colors, *etc.*

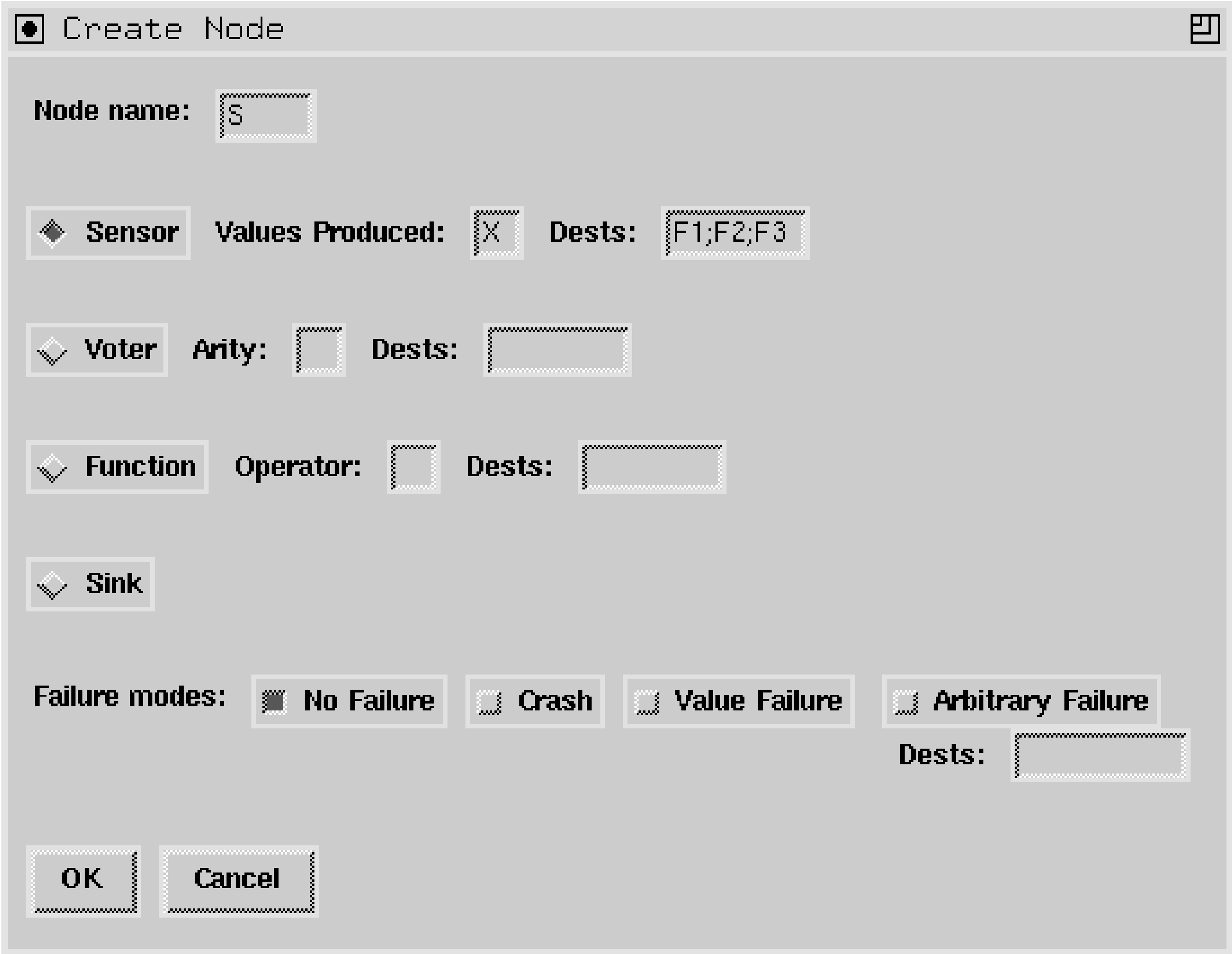

Figure B.1: Window for entering information about a new component.

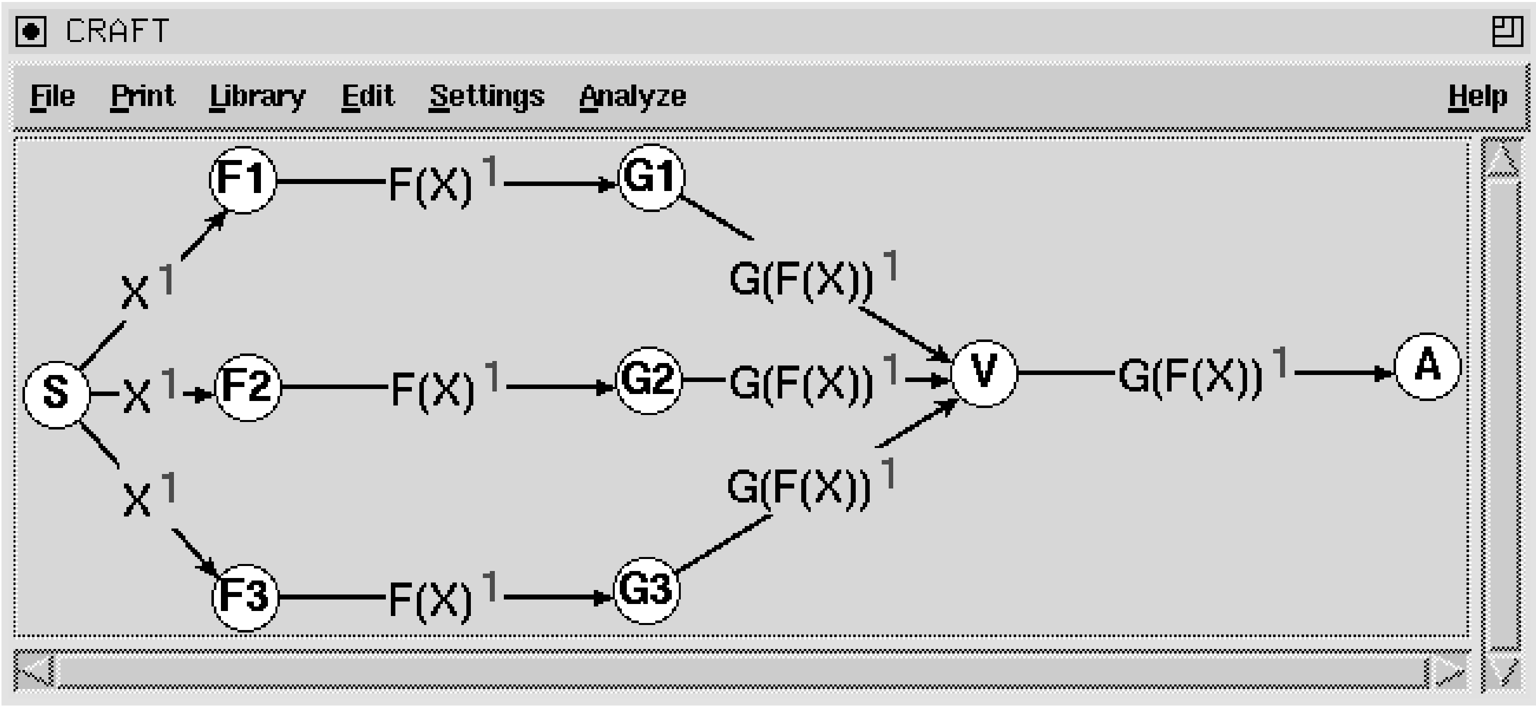

Figure B.2: Result of analysis in absence of failures.

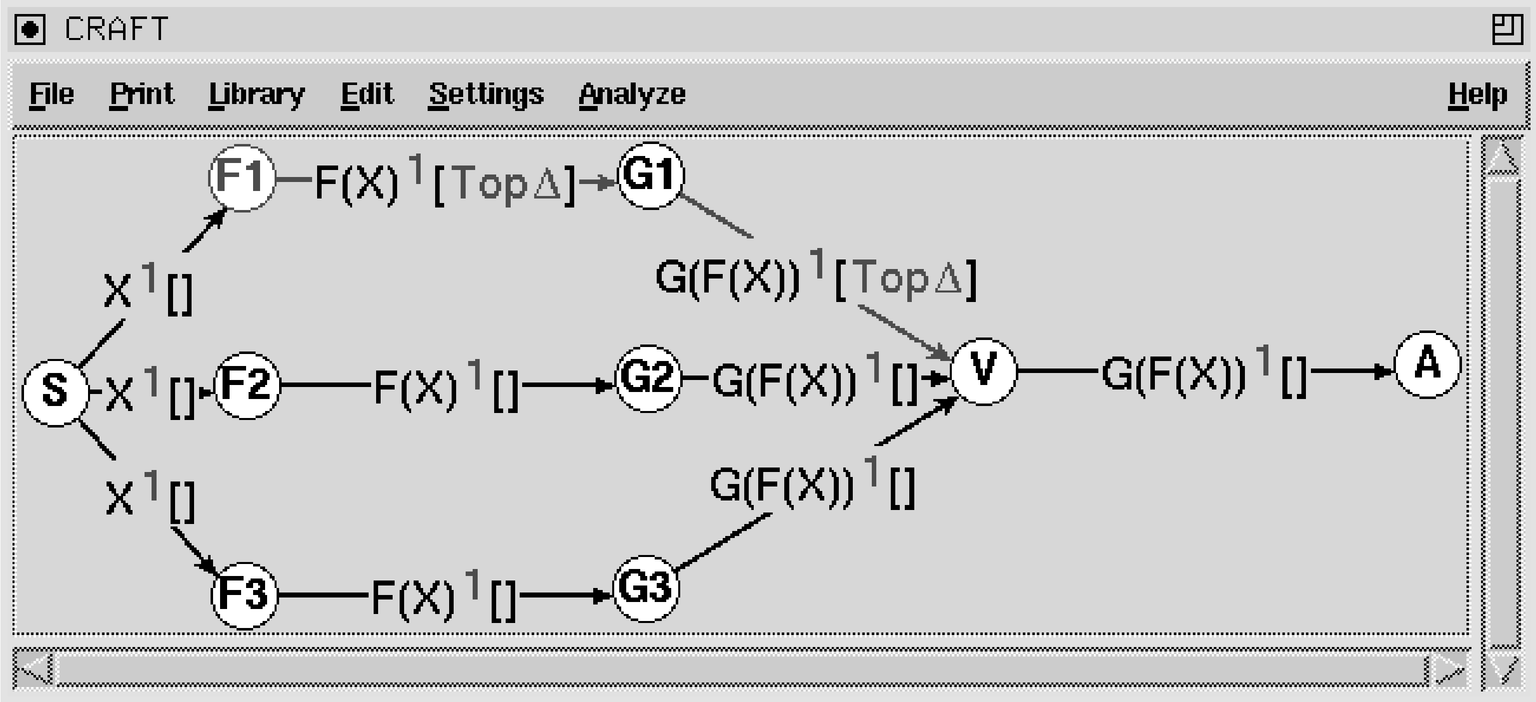

Figure B.3: Result of analysis when component F1 suffers a value failure.

B.2 Type Definitions and Function Declarations

The translation of the mathematical definitions into CAML is generally straightforward. Figure B.4 contains CAML type definitions corresponding to *Val* and *Mul*. Associated with each type is a function used to print values of that type. Declarations of those functions are omitted here.

The CAML type `sval` corresponding to *SVal* does not enforce two restrictions on the form of symbolic values: (1) the first argument to data constructor `Expr` should be a `Con` or `Var`, not an `Expr` or `Wild`; (2) the second argument to `Expr` should not contain `Wild`. One way to enforce these restrictions is to follow the approach taken in Section 2.2.2, i.e., to define `sval` in terms of auxiliary types `sym` and `sval_0`, corresponding to *Sym* and $SVal_0$, respectively. We would like these two auxiliary types to be subtypes of `sval`, but CAML does not support subtyping, so we would need to introduce data constructors to inject `sym` into `sval_0` and `sval_0` into `sval`. These extra constructors would be inconvenient. A better approach would be to introduce an abstract data type with two operations: a constructor function that checks these two requirements, and a destructor function that simply returns the symbolic value. With this approach, one needs only one extra constructor for each symbolic value, rather than an extra constructor for each symbol in the symbolic value. For convenience in development, the current implementation does not actually use such an abstract data type, but it would be trivial to do so using CAML's module system.

The type `aval` corresponding to *AVal* is generally straightforward. Note that the identifiers in the first argument of `Arb` are interpreted as constants (in *KC*). The treatment of equality for `aval` requires some care. One way to handle equality is to introduce, for each type, a function that tests equality of two elements of that type. However, it is more convenient if we can instead use CAML's built-in polymorphic structural equality function. Structural equality is simply a "pointwise extension" of equality on base types; for example, two lists of integers are structurally equal iff they have the same length and the elements in corresponding positions are equal. Is structural equality the desired equality on `aval`? In general, no. Recall from Chapter 5 that the arguments of *Arb* are sets, not sequences. Since sets are not a base type in CAML, we use lists as arguments of `Arb`.[4] Thus, in the desired equality on `aval`, the order of elements in the arguments of `Arb` should be irrelevant.

[4]CAML does provide a module that implements sets over ordered types using balanced trees. However, the built-in equality function does not have the desired meaning on elements of that type, because the same set can be represented by different balanced trees, depending on the history of operations used to construct that set. So, using the set module wouldn't help.

One way to achieve this is to write an equality function that deliberately ignores the order. As mentioned above, it is more convenient to use CAML's built-in equality function. This works provided the lists are maintained in a canonical form, i.e., sorted (according to some total order) and without duplicates. This is the approach taken in the current implementation. To mechanically enforce this invariant, we could introduce an abstract data type whose constructor function puts the given term into canonical form.

The type `amul` is the same as `aval`. Since CAML does not support subtyping, the only alternative would be to make `amul` an entirely distinct type, which would have prevented some code re-use. Of course, equating these two types introduces the possibility of input-output functions encountering in their input abstract values like `TopV` used as multiplicities. In this case, the input-output function should simply abort (e.g., by raising an exception).

The types `val` and `mul` are represented using lists, which should be regarded as sets. In other words, the list should be maintained in canonical form, as described above. Also, the empty list is prohibited. Again, it would be easy to mechanically enforce these restrictions using abstract data types.

The type `daval` corresponding to $\Delta AVal$ is straightforward; note that `Full`(a) corresponds to a_Δ. Type `damul` is the same as `daval`, for the same reasons that `amul` is the same as `aval`. Types `dval` and `dmul`, corresponding to ΔVal and ΔMul, respectively, are represented using lists, which should be kept in canonical form.

In order to re-use code for the perturbational and non-perturbational frameworks, we make the type of ms-atoms polymorphic in the type of "events", and use different types of events for the two frameworks. Specifically, making the tag part of the ms-atom allows re-use of the code that tests whether two graphs are equal up to renaming of tags. The types `event msatom` and `event_FC msatom` correspond to L and L_{FC}, respectively.

The type `'e poset` of posets with events of type `'e` is represented as a pair of a list of ms-atoms and an ordering. The list of ms-atoms is kept in canonical form (i.e., sorted and duplicate-free). The ordering is represented using the `order` module, which is part of CRAFT. Internally, the `order` module represents an ordering as a transitively-closed set of pairs.

The type `'e hist` of histories with events of type `'e` is represented using the `map` module, which is part of the CAML standard library and provides an implementation of finite maps with ordered domains using balanced trees. We allow partial maps, with the convention that elements not in the domain of the map are implicitly mapped to the empty poset. The use of finite maps rather than the function type `name -> 'e poset` allows efficient and convenient

```
(* component name *)
type name == string;;

(* identifiers are used to name constants and variables. *)
type identifier == string;;

(* SVal *)
type sval =
| Con of identifier
| Var of identifier
| Expr of sval*(sval list)
| Wild
;;

(* AVal *)
type aval =
| One                 (* denotes {1} *)
| ZeroOne             (* denotes {0,1}.  printed as "?". *)
| Nat                 (* denotes the natural numbers. *)
| Plus                (* denotes {1,2,...} *)
| TopV                (* denotes all concrete values *)
| MsgFrom of name     (* used in analysis of reliable bcast *)
| Data                (* used in analysis of Byz. agreement *)
| Msg                 (* used in analysis of Byz. agreement *)
| Arb of (identifier list * sval list)
                      (* used in analysis of Byz. agreement. *)
;;

(* AMul *)
type amul == aval;;

(* Val *)
type val == (sval * aval) list;;

(* Mul *)
type mul == (sval * amul) list;;
```

Figure B.4: CAML type definitions corresponding to *Val* and *Mul*.

```
(* Delta-AVal *)
type daval =
| Identity        (* denotes the identity relation on CVal *)
| Full of aval    (* denotes the full relation on aval *)
;;

(* Delta-AMul *)
type damul == daval;;

(* Delta-Val *)
type dval == (sval * daval) list;;

(* Delta-Mul *)
type dmul == (sval * damul) list;;

(* abstract event in non-perturbational framework *)
type event = mul * val;;

(* abstract event in perturbational framework *)
type event_FC =
| Pert of mul * val * dmul * dval
| New of mul * val
;;

(* tags used in ms-atoms *)
type tag == int;;

(* ms-atoms with events of type 'e.
   Type (event msatom) corresponds to L;
   type (event_FC msatom), to L_{FC}. *)
type 'e msatom == 'e * tag;;
```

Figure B.5: CAML type definitions corresponding to ΔVal, ΔMul, L, and L_{FC}.

iteration over the non-empty posets in a history.

The type `'e run` of runs with events of type `'e` is also represented using finite maps. By convention, a component name not in the domain of the map is implicitly mapped to the empty history.

The type `event iofn` corresponding to *IOF* is straightforward. The requirement that these functions be uniform with respect to tags is not mechanically enforced.

The type `(event_FC, 'f) iofn_F` corresponds to IOF_{FC}, with type `'f` corresponding to the set *Fail* of possible failures. Since elements of IOF_{FC} are partial functions, we represent them using a pair, whose first component specifies the domain of the partial function, and whose second component corresponds to the function itself.

A system with events of type `'e` and failures of type `'f` is represented by an element of type `('e, 'f) system_F`, i.e., by a finite mapping from names to input-output functions. Type `'f failure_scenario` is straightforward.

Recall that a fault-tolerance requirement is a function b such that for each failure scenario fs, $b(fs)$ is a predicate on runs. This signature corresponds directly to the type `('e,'f) ft_req` of fault-tolerance requirements for systems with events of `'e` and failures of type `'f`. The sanity condition that these functions be independent of tags cannot be expressed in the CAML type system; the user is responsible for ensuring that this condition is satisfied.

Finally, we are finished with the type definitions and come to the function declarations. The function application `step_F sys fs` corresponds to $step_F(nf, fs)$. Function `lfp` computes least fixed points of functions of type `('e run) -> ('e run)`. Function `failure_scenarios` returns a list containing all failure scenarios for a given system. The implementations of these functions are straightforward. The only non-trivial aspect is that `lfp` checks for termination of the fixed-point calculation using $=_{Run}$ (defined on page 28), so an implementation of this equality is needed. The current implementation of $r_1 =_{Run} r_2$ is as follows: (1) the tags in each run are "normalized" to be a prefix of the natural numbers; (2) if the runs contain different numbers of distinct tags, then $r_1 \neq_{Run} r_2$; (3) if the runs contain the same number n of distinct tags, then for each permutation σ of the natural numbers $0, 1, \ldots, n$, rename the tags in r_1 according to σ and check whether the resulting run equals r_2. If any permutation results in equality in step 3, then $r_1 =_{Run} r_2$; otherwise, $r_1 \neq_{Run} r_2$.

```
(* poset with events of type 'e. *)
type 'e poset == ('e msatom) list * ('e msatom) order__order;;

(* history.  use a map, not a function, so we can iterate over the
  non-empty edges in normalize_tags. *)
type 'e hist == (name, 'e poset) map__t;;

(* run with events of type 'e. Type (event graph) corresponds to Run;
  type (event_FC graph), to Run_{FC}. *)
type 'e run == (name, 'e hist) map__t;;

(* input-output function (no failures), with events of type 'e.
  Type (event iofn) corresponds to IOF. *)
type 'e iofn == ('e hist) -> ('e hist);;

(* input-output function with failures of type 'f.
  Type ((event, 'f) iofn_F) corresponds to IOF_F;
  type ((event_FC, 'f) iofn_F), to IOF_{FC}. *)
type ('e,'f) iofn_F == ('f list)*('f -> ('e iofn));;

type failure = OK | ValFail | ... ;;

(* system (with failures) *)
type ('e,'f) system_F == (name, ('e,'f) iofn_F) map__t;;

(* failure scenario *)
type 'f failure_scenario == (name, 'f) map__t;;

(* fault-tolerance requirement *)
type ('e,'f) ft_req ==  ('f failure_scenario) -> ('e run) -> bool;;

(* step function for a given system_F in a given failure scenario *)
value step_F : ('e,'f) system_F -> ('f failure_scenario)
  -> 'e run -> 'e run;;

(* least fixed point *)
value lfp : (('e run) -> ('e run)) -> ('e run);;

(* return a list containing all failure scenarios for a system_F. *)
value failure_scenarios : ('e,'f) system_F
  -> (('f failure_scenario) list);;
```

Figure B.6: CAML type definitions corresponding to *Run*, *IOF*, Run_{FC}, and IOF_{FC}, plus miscellaneous other CAML type definitions and function declarations.

B.3 Using CRAFT

This section describes in more detail how to use CRAFT without the graphical interface. To use CRAFT for a non-perturbational analysis:

1. Define appropriate types `aval` and `failure`.

2. Define input-output functions representing the components of the system. These functions should be elements of `(event, failure) iofn_F`.

3. Express the fault-tolerance requirement as a function of type `(event, failure) ft_req`.

4. Using those input-output functions, construct an element `sys` of type `(event, failure) system_F` representing the system. Functions in the `map` module in the CAML standard library are used to construct maps.

5. For each failure scenario `fs` of interest, call `lfp (step_F sys fs)` to compute a run `r` of type `event run` representing the behavior of the system in that failure scenario, and compute `ftr fs r` to check whether the fault-tolerance requirement is satisfied in that failure scenario. If the fault-tolerance requirement is violated in some failure scenarios, the corresponding runs can be inspected to help ascertain the problem.

To use CRAFT for a perturbational analysis, simply replace `event` with `event_FC` in the above.

Bibliography

[AH87] Samson Abramsky and Chris Hankin. An introduction to abstract interpretation. In Samson Abramsky and Chris Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 1. Ellis-Horwood, 1987.

[AMST93] Gul Agha, Ian A. Mason, Scott Smith, and Carolyn Talcott. A foundation for actor computation. *Journal of Functional Programming*, 1993.

[BA81] J. Dean Brock and William B. Ackerman. Scenarios: A model of non-determinate computation. In J. Diaz and I. Ramos, editors, *Formalisation of Programming Concepts*, volume 107 of *Lecture Notes in Computer Science*, pages 252–259. Springer-Verlag, 1981.

[BD92] Manfred Broy and Claus Dendorfer. Modelling operating system structures by timed stream processing functions. *Journal of Functional Programming*, 2:1–21, 1992.

[BKRS96] J. F. Buss, P.C. Kanellakis, P. L. Ragde, and A. Shvartsman. Parallel algorithms with processor failures and delays. *Journal of Algorithms*, 1996.

[BM82] R. J. R. Back and H. Mannila. A refinement of kahn's semantics to handle non-determinism and communication. In *Proc. First ACM Symposium on Principles of Distributed Computing*, pages 111–120, 1982.

[Bro83] J. Dean Brock. *A Formal Model of Non-Determinate Dataflow Computation.* Ph.D. dissertation, Massachusetts Institute of Technology, 1983. Available as MIT Laboratory for Computer Science Technical Report TR-309.

[Bro87] Manfred Broy. Semantics of finite and infinite networks of concurrent communicating agents. *Distributed Computing*, 2(1):13–31, 1987.

[Bro88] Manfred Broy. Nondeterministic data flow programs: how to avoid the merge anomaly. *Science of Computer Programming*, 10:65–85, 1988.

[Bro90] Manfred Broy. Functional specification of time sensitive communicating systems. In *Proceedings of the REX Workshop on Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*, pages 153–179. Springer-Verlag, 1990.

[BZ83] D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, April 1983.

[CdR93] Antonio Cau and Willem-Paul de Roever. Using relative refinement for fault tolerance. In J.C.P. Woodcock and P.G. Larsen, editors, *FME'93: Industrial-Strength Formal Methods. First International Symposium of Formal Methods Europe*, pages 19–41, 1993.

[CGL92] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 343–354, 1992.

[CGL94] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[CR93] Ran Canetti and Tal Rabin. Fast asynchronous Byzantine agreement with optimal resilience. In *25th Symposium on Theory of Computing*, pages 42–51. ACM Press, 1993.

[CS96] Rance Cleaveland and Steve Sims. The ncsu concurrency workbench. In Rajeev Alur and Tom Henzinger, editors, *Computer-Aided Verification (CAV '96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 394–397. Springer-Verlag, 1996.

[DBC91] Ben L. Di Vito, Ricky W. Butler, and James L. Caldwell. High level design proof of a reliable computing platform. In J. F. Meyer and R. D. Schlichting, editors, *Dependable Computing for Critical Applications 2*, volume 6 of *Dependable Computing and Fault-Tolerant Systems*, pages 279–306. Springer-Verlag, 1991.

[DS89] Peter Dybjer and Herbert P. Sander. A functional programming approach to the specification and verification of concurrent systems. *Formal Aspects of Computing*, 1:303–319, 1989.

[FM93] P. Fenelon and J. A. McDermid. An integrated toolset for software safety analysis. *Journal of Systems and Software*, 21(3):279–290, 1993.

[FMNP94] P. Fenelon, J. A. McDermid, M. Nicholson, and D. J. Pumfrey. Towards integrated safety analysis and design. *ACM Computing Reviews*, 2(1):21–32, 1994.

[GLR95] Li Gong, Patrick Lincoln, and John Rushby. Byzantine agreement with authentication: Observations and applications in tolerating hybrid and link faults. In *Dependable Computing for Critical Applications—5*, pages 79–90. IFIP WG 10.4, preliminary proceedings, 1995.

[Gun92] Carl A. Gunter. *Semantics of Programming Languages*. MIT Press, 1992.

[Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice–Hall, 1991.

[HT94] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR 94-1425, Cornell University, Department of Computer Science, 1994.

[Jon89] Bengt Jonsson. A fully abstract trace model for dataflow networks. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 155–165, 1989.

[Kah74] Gilles Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing 74: Proceedings of the IFIP Congress 74*, pages 471–475. North–Holland, 1974.

[Kel78] R. M. Keller. Denotational models for parallel programs with indeterminate operators. In E. J. Neuhold, editor, *Formal Description of Programming Concepts*, pages 337–366. North Holland, 1978.

[KMP94] Yonit Kesten, Zohar Manna, and Amir Pnueli. Temporal verification of simulation and refinement. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Proc. REX School/Symposium*, volume 803 of *Lecture Notes in Computer Science*, pages 273–346. Springer-Verlag, 1994.

[KMS95] P.C. Kanellakis, D. Michailidis, and A. Shvartsman. Controlling memory access in efficient fault-tolerant parallel algorithms. *Nordic Journal of Computing*, 1995.

[Kur89] R. P. Kurshan. Analysis of discrete event coordination. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Proc. of the REX Workshop on Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*, pages 414–453. Springer-Verlag, 1989.

[Kur94] Robert P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton Univesity Press, 1994.

[Lam93] Leslie Lamport. How to write a long formula. Technical Report SRC-119, Digital Equipment Corporation, Systems Research Center, 1993.

[Ler97] Xavier Leroy. *The Caml Light System*. INRIA, 1997. Available via http://pauillac.inria.fr/caml/.

[LJ92] Zhiming Liu and Mathai Joseph. Transformation of programs for fault-tolerance. *Formal Aspects of Computing*, 4:442–469, 1992.

[LM94] Leslie Lamport and Stephan Merz. Specifying and verifying fault-tolerant systems. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 863 of *Lecture Notes in Computer Science*, pages 41–76. Springer-Verlag, 1994.

[LR93] Patrick Lincoln and John Rushby. A formally verified algorithm for interactive consistency under a hybrid fault model. In *Proc. 23rd Intl. Symposium on Fault Tolerant Computing*, 1993.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[Mil89] R. Milner. *Communication and concurrency*. Prentice–Hall, 1989.

[MNPF95] J. A. McDermid, M. Nicholson, D. J. Pumfrey, and P. Fenelon. Experience with the application of HAZOP to computer-based systems. In *Proc. 10th Annual Conference on Computer Assurance*, pages 37–48, 1995.

[MR96] Dalia Malki and Michael Reiter. A high-throughput secure reliable multicast protocol. In *Proc. of the 9th IEEE Computer Security Foundations Workshop*, page 9=17. IEEE Computer Society Press, 1996.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[MvRSS96] Yaron Minsky, Robbert van Renesse, Fred B. Schneider, and Scott Stoller. Cryptographic support for fault-tolerant distributed computing. In *Proc. Seventh ACM SIGOPS European Workshop*, pages 109–114. ACM Press, September 1996.

[ORSvH95] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of pvs. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[PJ94] Doron Peled and Mathai Joseph. A compositional framework for fault-tolerance by specification transformation. *Theoretical Computer Science*, 128(1-2):99–125, 1994.

[Pra82] Vaughan R. Pratt. On the composition of processes. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 213–223. ACM Press, 1982.

[Rea89] Chris Reade. *Elements of Functional Programming*. Addison Wesley, 1989.

[Rei96] Michael K. Reiter. A secure group membership protocol. *IEEE Transactions on Software Engineering*, 22(1):31–42, 1996.

[RSA78] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[Rus90] James Robert Russell. *Full Abstraction and Fixed-Point Principles for Indeterminate Computation*. Ph.D. dissertation, Cornell University, Department of Computer Science, 1990.

[Sca62] James B. Scarborough. *Numerical Mathematical Analysis*. The Johns Hopkins Press, 5th edition, 1962.

[Sch94] Henk Schepers. A trace-based compositional proof theory for fault tolerant distributed systems. *Theoretical Computer Science*, 128(1–2):127–157, 1994.

[SN85] John Staples and V. L. Nguyen. A fixpoint semantics for nondeterministic data flow. *Journal of the ACM*, 32(2):411–444, April 1985.

[W+78] John H. Wensley et al. Sift: Design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of the IEEE*, 66(10):1240–1255, 1978.

[Web93] Doug G. Weber. Fault tolerance as self-similarity. In Jan Vytopil, editor, *Formal techniques in real-time and fault-tolerant systems*, pages 33–49. Kluwer Academic Publishers, 1993.