Automated Analysis of Fault-Tolerance in Distributed Systems*

Scott D. Stoller†
Computer Science Dept.
State University of New York at Stony Brook
Stony Brook, NY 11794
stoller@cs.sunysb.edu

Fred B. Schneider‡
Dept. of Computer Science
Cornell University
Ithaca, NY 14850
fbs@cs.cornell.edu

December 21, 2000

Abstract

A method for automated analysis of fault-tolerance of distributed systems is presented. It is based on a stream model of computation augmented with approximation constructs, and this facilitates efficient analysis. Analyses of a protocol for fault-tolerant moving agents and a reliable broadcast protocol illustrate the method.

1 Introduction

As computers become integrated into mission-critical systems, there is a growing need for techniques to establish that software systems satisfy their requirements. Fault-tolerance is likely to be one of these requirements. The following methodology can be used to establish fault-tolerance of a system:

0. Partition the system into components.
1. Identify possible failures of each component.
2. Posit correctness requirements on system behavior for each combination of possible failures.
3. Check whether the system satisfies these requirements.

This paper describes a method and automated tool for step 3.

A *failure* identified in step 1 is defined to be one way that a component's actual behavior might diverge from its normal (specified) behavior. For example, a *Byzantine failure* causes a processor

*Some of the material contained herein previously appeared in [SS98].

†Supported in part by NSF under Grant CCR-9876058 and ONR under Grants N00014-99-1-0358 and N00014-01-1-0109.

‡Supported in part by ARPA/RADC grant F30602-96-1-0317, AFOSR grant F49620-00-1-0198, Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory Air Force Material Command USAF under agreement number F30602-99-1-0533, National Science Foundation Grant 9703470, and a grant from Intel Corporation. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of these organizations or the U.S. Government.

to execute an arbitrary sequence of instructions, unrelated to the program it would have executed in the absence of the failure [LSP82]. A *failure scenario* for a system is an assignment of failures to a subset of the system's components.

In step 2 above, requirements on system behavior in various failure scenarios are posited. For example, an aircraft control system might be required to provide normal service despite the Byzantine failure of any single component. In our terminology, this requirement is: "for every failure scenario in which at most one component is faulty, signals sent by the aircraft control system to actuators should be the same as what would be sent if no failures had occurred."

Step 3 above is to check whether a system in the presence of failures satisfies some requirements. Experience has shown that informal arguments here tend to be error-prone and thus do not supply the desired level of assurance for mission-critical systems [ORSvH95]. For example, although replication and voting might seem easy to justify informally, the extensive literature on Byzantine agreement protocols and errors like the one described in [LR93] suggest that efficient coordination of non-faulty replicas in the presence of arbitrary behavior by faulty replicas is quite subtle.

The limitations of informal arguments has motivated the development of rigorous methods. One class of methods is based on formal logics. However, most people who design and validate fault-tolerant systems are not experts in mathematical logic or formal verification, so methods that require construction of proofs (even with support from a theorem-proving system) have a limited audience. Proof techniques designed specifically for verification of fault-tolerance (e.g., [SS83, CdR93, Web93, PJ94, Sch94]) facilitate the construction of these proofs but still require logical expertise of the user.

Another class of methods is based on exhaustive exploration of finite state spaces [Hol91, CGL94, Kur94, CS96]. These methods have enjoyed considerable success for verification of hardware and communication protocols. But for many fault-tolerant asynchronous distributed software systems, the time and memory required for state-space exploration are prohibitively large.

The method for analysis of fault-tolerance properties for distributed systems described in this paper uses a novel combination of stream-processing (or data-flow) models of networks of processes [Kah74, Bro87, Bro90] and abstract interpretation of programs [CC77, JN94]. An important feature of our method is its emphasis on communication (rather than state), motivated by the thesis that distributed systems often have natural descriptions in terms of communication. This emphasis shapes both the representation of system behavior and the method used to compute it. In our framework, system behavior is represented by *message flow graphs* (MFGs), which characterize the possible communication behaviors of the system. Each node of a MFG corresponds to a system component, and each edge is labeled with a description of the sequence of messages possibly sent from its source to its target. For simplicity, this paper considers systems in which components interact only by messages transmitted along unbounded FIFO channels; non-FIFO channels are easily accommodated, though.

Exact computation of all possible sequences of messages that might be sent during system execution is generally infeasible. To help make automated analysis feasible, our framework supports flexible and powerful abstractions (approximations). Traditionally, stream-processing models have been used as mathematical semantics and contained no abstractions. The abstractions in our method apply to *values* (the data transmitted in messages), *multiplicities* (the number of times each value is sent), and *message orderings* (the order in which values are sent). Values and multiplicities are approximated using abstract interpretation and symbolic computation. Message orderings are abstracted by allowing partial orders in place of total orders. Taken all together, the abstractions in our method enable compact representations of the highly non-deterministic behavior characteristic of failures and support abstraction from irrelevant aspects of a system's failure-free behavior.

We use only conservative abstractions, so the analysis never falsely implies that a system satisfies its fault-tolerance requirement. However, conservative abstractions do introduce the possibility of false negatives: an analysis might not establish that a system satisfies its fault-tolerance requirement, even though the system does.

Our analysis method is to compute, for each failure scenario of interest, an MFG representing the system's communication behaviors in that failure scenario. Each MFG is then checked to see whether the fault-tolerance requirement for that failure scenario is satisfied. A more common method (e.g., [SS83, LJ92, CdR93, Web93, PJ94, LM94]) is to model failures as events that occur non-deterministically during a computation; system behavior in all failure scenarios is analyzed together. We separate the analyses for different failure scenarios for two reasons: (*i*) this separation is convenient for failure scenarios with different requirements, and (*ii*) it helps keep the MFGs small and simple.

Our method for computing an MFG for a failure scenario is based on a stream-processing model of distributed systems. Each system component is represented by one or more *input-output functions* that describe its input/output behavior. An input-output function that represents a component takes as arguments sequences of messages received from different sources and returns the sequences of messages sent to different destinations by that component. Thus, an input-output function encapsulates (an abstraction of) the implementation of a component. One verifies independently for each component that the proposed input-output function faithfully (but perhaps approximately) represents the component's behavior. Given input-output functions representing components, the streams of messages sent on each channel during execution can be expressed and computed as a fixed-point.

Stream-processing models emphasize communication behavior, rather than the local state, of each component. Stream-processing models also admit compact representations for sequences of messages, and those representations can be used directly as inputs to the input-output function representing a component.

MFGs could be computed from conventional state-based models of components by constructing the graph of reachable states and then extracting an MFG, but this would typically be less efficient than what we do. Our analysis does not construct global states and therefore seems to circumvent the state-space explosion problem. At least, our experience has been that MFGs describing a system's behavior are often smaller than the full state graph for the system. Furthermore, MFGs are often smaller than reduced state graphs computed using partial-order methods [Pel98]. This is illustrated in Section 3.3.

One could regard our framework as having a powerful built-in partial-order method, because it does not consider interleavings of messages sent (or received) by a component on different channels. Such inter-channel orderings cannot be represented directly in MFGs, though in some cases such orderings can be inferred from properties of the input-output functions. Consequently, the behavior of systems that depend on such orderings cannot, in general, be analyzed exactly using our framework. It might be possible to augment MFGs to represent such orderings; this is discussed in

Section 5.2.

The rest of the paper is organized as follows. Section 2 describes the method. Section 3 illustrates the method with two examples. Section 4 describes a prototype implementation of the method. Section 5 discusses related and future work.

2 Communication-Based Analysis Framework

We first describe how data values are abstracted in our framework and then how sets and sequences of messages are abstracted. This leads directly to the definitions of input-output function and MFG that are the foundation of our analysis method.

2.1 Values

As in abstract interpretation, we introduce a set *AVal* of *abstract values*. Each abstract value represents a set of concrete values. For example, we use abstract value **N** to represent the set of 64-bit numbers, and in the analysis of the moving-agent protocol in Section 3.1, abstract value $Arb(keys, ms)$ represents concrete values that can be generated using encryption keys in the set *keys* and texts in the set *ms*.

For analysis, abstract values alone capture too little information about relationships between concrete values. The output of a majority voter, for example, depends on equality relationships among its inputs. If two voter inputs both have abstract value **N**, then there would be no way to tell whether they are equal. So, we introduce a set *SVal* of *symbolic values*, which are expressions composed of constants, variables, and a wildcard symbol. All occurrences of a constant or variable in a single MFG represent the same (concrete) value. The *wildcard* symbol "_" is used when a value is not known to have any interesting relationships to other values. Different occurrences of the wildcard in a MFG do *not* necessarily represent the same value. For example, if two inputs of a 3-way voter are denoted by the same (non-wildcard) symbolic value, then that symbolic value represents the majority value and therefore represents the voter's output.

A *constant* represents the same value in every execution of a system; most constants are typeset in a sans-serif font. For example, a constant maj might represent a majority function. A *variable* represents values that may be different in different executions of a system. Variables are useful for modeling outputs that are not completely determined by a component's inputs. Such outputs commonly arise with components that interact with an environment that is not being modeled explicitly. They also arise when a component's behavior is approximated. Each variable is *local* to (i.e., is associated with) one component and corresponds to a concrete value in that component's outputs [Sto97]. Associating each variable with one component makes it possible to check independently that each input-output function faithfully represents the behavior of the corresponding component.

A symbolic value and an abstract value together are often sufficient to characterize possible data values in a message. Analysis of a non-deterministic system might yield multiple such pairs, each representing some of the possible data values in a message. So, we use a set of such pairs to represent values, and define $Val \triangleq Set(SVal \times AVal) \setminus \{\emptyset\}$, where $Set(S)$ is the powerset of a set S. For example, $\{\langle X_1, Msg\rangle, \langle X_2, Msg\rangle\}$ denotes a value in Msg that is represented by symbolic value

X_1 or X_2. We usually write a pair $\langle s, a\rangle \in \mathit{SVal} \times \mathit{AVal}$ as $s\!:\!a$, and we usually omit curly braces around singleton sets. For example, $\{\langle \mathsf{F}(X), D\rangle\}$ may be written as $\{\mathsf{F}(X)\!:\!D\}$ or $\mathsf{F}(X)\!:\!D$. A wildcard is similar in meaning to omission of a symbolic value, so we usually elide wildcards. For example, $\{\langle \_, \mathbf{N}\rangle\} \in \mathit{Val}$ would be written as $\mathbf{N}$.

2.2 Multiplicities

Uncertainty in the number of messages sent during a computation may stem from intrinsic non-determinism of the original system, non-determinism introduced in a model when some aspects of the system or its environment are not modeled explicitly, non-determinism from failures, or from uncertainty caused by other abstractions (of data, multiplicity, or message ordering). For example, a component subject to Byzantine failures might emit outputs with an arbitrary multiplicity. To compactly represent these possibilities, we allow abstractions of the multiplicity (i.e., the number of occurrences) of each message.

We think of multiplicities as natural numbers and therefore represent them in the same way as data values. Thus, we define $\mathit{Mul} \triangleq \mathit{Set}(\mathit{SVal} \times \mathit{AMul}) \setminus \{\emptyset\}$, where the set AMul of *abstract multiplicities* is a subset of AVal and contains abstract values that represent subsets of the natural numbers, excluding $\emptyset$ and $\{0\}$ (these two subsets would correspond to something that did not happen.) We assume AVal contains the following: 1, denoting $\{1\}$; ?, denoting $\{0, 1\}$; and $*$, denoting the set of natural numbers. The notational conventions for Val also apply to Mul. For example, $\{\langle \_, ?\rangle\} \in \mathit{Mul}$ may be written as ?.

2.3 Sets and Posets of Messages

ms-atoms (mnemonic for "message-set atoms") are abstractions of sets of messages. Each ms-atom uses an element of Val to characterize the data in the messages and an element of Mul to characterize the number of messages in the set. Thus, the signature of ms-atoms is $\mathit{MSA} \triangleq \mathit{Val} \times \mathit{Mul}$. For example, an ms-atom $\langle X : \mathit{Msg}, *\rangle$ with value $X : \mathit{Msg}$ and multiplicity $*$ represents a set S of messages such that: (1) the data in each message is an element of (the set represented by) Msg, and the data is represented by variable X (hence all the messages in S contain the same data), and (2) the number of messages in S is arbitrary (but finite). As another example, an ms-atom $\langle \_ \!:\! \mathit{Msg}, *\rangle$ represents an arbitrary-sized set of messages, with each message containing a (possibly different) element of Msg. Similarly, $\langle X\!:\!\mathit{Msg}, Y\!:\!?\rangle$ represents Y occurrences of a value X in Msg, where the value of Y is zero or one. To promote the resemblance between ms-atoms and regular expressions, we write an ms-atom $\langle \mathit{val}, \mathit{mul}\rangle$ as $\mathit{val}^{\mathit{mul}}$, and if the multiplicity mul is 1, we usually elide it.

The set of sequences of messages possibly sent along a channel is represented by a partially-ordered set (poset) of ms-atoms. A (strict) partial order is represented as a pair $\langle S, \prec\rangle$, where S is a set and $\prec$ is an acyclic transitive binary relation on S. The meaning of the partial order is: for ms-atoms $x, y \in S$, if $x \prec y$, then the messages being represented by x are sent (and received, since channels are FIFO) before the messages being represented by y. If the exact order in which the messages are sent is known during the analysis, then the poset is totally-ordered, i.e., it is a sequence. Having posets of ms-atoms allows compact representation of the set of possible sequences of messages when orderings between some messages are uncertain. For example, consider a system in which a gateway forwards requests received from clients to another server. If the order in which the gateway receives requests from different clients is undetermined, then the set of forwarded requests would not be totally-ordered, and the gateway's outputs would be succinctly represented by a partially-ordered set of ms-atoms.

We could use a notation based on finite automata, rather than regular expressions, to represent sets of possible sequences of messages transmitted on a channel. Such a representation was explored in a state-based framework called queue-content decision diagrams (QDDs) [BG99]. However, to be as expressive as our posets of ms-atoms, the automata would have to be augmented with an analogue of symbolic multiplicities. Automata might provide a more efficient basis for the implementation, but using them would probably make input-output functions harder to write. This would not be an obstacle if automated support for generating input-output functions from state-based descriptions of components were available. Developing such support is an interesting open problem.

2.4 Input-Output Functions

Inputs to a component are represented in our framework by a *history*. Outputs of a component are also represented by a history. The signature *Hist* of histories is $\mathit{Hist} \triangleq \mathit{Name} \rightarrow \mathit{POSet}(\mathit{MSA})$. When a history h is used to represent the inputs to a component y, $h(x)$ represents the messages from x to y. When a history h is used to represent the outputs of a component y, $h(x)$ represents the messages from y to x. The behavior of a component y is represented (possibly with approximations) by an input-output function f_y such that, for every history h, $f_y(h)(x)$ represents the outputs of y to x for input history h. If we temporarily ignore failures, the signature *IOF* of input-output functions is $\mathit{IOF} \triangleq \mathit{Hist} \rightarrow \mathit{Hist}$.

Recall that we analyze different failure scenarios separately. To achieve this separation, we parameterize each input-output function by the possible failures of the corresponding component. Thus, input-output functions are elements of $\mathit{IOF}_F \triangleq \mathit{Fail} \rightharpoonup \mathit{IOF}$, where *Fail* is the set of all possible failures, and one-hooked arrow $\rightharpoonup$ indicates partial functions. For example, *Fail* might contain an element *Byz* corresponding to Byzantine failures. For $f \in \mathit{IOF}_F$, $\mathrm{domain}(f)$ is the set of failures that the component might suffer, and for each $\mathit{fail} \in \mathrm{domain}(f)$, $f(\mathit{fail})$ characterizes the component's behavior when failure *fail* occurs. By convention, *Fail* contains an element *OK* that indicates absence of failure. A failure scenario is a function in $\mathit{FS} \triangleq \mathit{Name} \rightarrow \mathit{Fail}$ that maps each component to one of its possible failures.

2.5 Message Flow Graphs

An MFG is formulated as a function: for an MFG g and components x and y, $g(x, y)$ is the label on the edge from x to y; thus, $g(x, y)$ characterizes the messages from x to y. We assume a system comprises a set of named components, with names from the set *Name*. The signature of MFGs is $\mathit{MFG} \triangleq (\mathit{Name} \times \mathit{Name}) \rightarrow \mathit{POSet}(\mathit{MSA})$, where $\mathit{POSet}(S)$ is the set of (strict) partial orders over a set S.

2.6 Computing *MFG*s

Recall that a system is a collection of named components. A system is represented by a function $nf \in Name \rightarrow IOF_F$, which gives the input-output function for each component. The behavior of a system nf in a failure scenario fs is computed using a function $step_{nf,fs} \in MFG \rightarrow MFG$, defined by

$$
\begin{array}{rll}
step_{nf,fs}(g) \triangleq & \text{the MFG } g', \text{ where} & \\
g'(x,y) = & \mathbf{let}\ f = nf(x)(fs(x)) & (*\ f \text{ is input-output function for } x\ *) \\
& \mathbf{and}\ h(z) = g(z,x) & (*\ h \text{ is input history of } x\ *) \\
& \mathbf{and}\ h' = f(h) & (*\ h' \text{ is output history of } x\ *) \\
& \mathbf{in}\ h'(y) & (*\ h'(y) \text{ represents messages from } x \text{ to } y\ *)
\end{array}
\tag{1}
$$

Informally, the MFG $step_{nf,fs}(g)$ represents the result of each component processing its inputs in the possibly-incomplete executions represented by the MFG g and producing possibly-extended outputs. An MFG representing the behavior of a system is computed by starting from the empty MFG *empty*, defined by $empty(x,y) = \langle \emptyset, \emptyset \rangle$, and repeatedly applying $step_{nf,fs}$ until a fixed-point is reached. Repeated application of $step_{nf,fs}$ corresponds to continued execution of the system being analyzed.

For a system with arbitrary input-output functions, iterative calculation of the fixed-point is not guaranteed to terminate. The possibility of non-termination is unavoidable—asynchronous distributed systems with unbounded channels are infinite-state and verification for them is (in general) undecidable. One possible cause of non-termination is that the number of ms-atoms labeling edges in the MFG being computed might grow without bound; this is possible because channels are unbounded queues, and our framework supports but does not enforce the use of abstractions in the representation of channel contents. Another possible cause is that the MFG might change after each iteration without the number of ms-atoms increasing; this situation is rare and, if monotonocity conditions are imposed to prohibit cyclic behavior [Sto97], is possible only if *AVal* or *SVal* is infinite.

In practice, if execution of the system always terminates, then use of reasonable input-output functions to represent the components should lead to termination of the fixed-point calculation. Our framework is not suitable for analyzing systems with infinite behaviors: each ms-atom represents a finite (though unbounded) number of messages, so the analysis would certainly diverge. The framework could be extended to analyze some classes of non-terminating systems, and this is discussed in Section 5.3.

2.7 Fault-tolerance Requirements

A fault-tolerance requirement stipulates conditions on the possible system behaviors in a specified failure scenario. These conditions are possible behaviors in a specified failure scenario. These conditions are expressed in our framework as a predicate on MFGs. A system nf satisfies a fault-tolerance requirement b for failure scenario fs if the fixed-point of $step_{nf,fs}$ satisfies b. Thus, to discharge step 3 of the methodology in Section 1, for each failure scenario fs, fixed-point of $step_{nf,fs}$ is computed and checked to determine whether it satisfies the fault-tolerance requirement for fs.

3 Examples

This section presents two examples—fault-tolerant moving agents [Sch97] and reliable broadcast [HT94]—and compares the computational cost of our analyses of them to that of state-space exploration optimized with partial-order methods.

The analysis of the protocol in [Sch97] for fault-tolerant moving agents shows how cryptographic primitives and moving agents can be modeled and analyzed in our framework. The exercise actually revealed a weakness in the first version of the protocol we discuss: that version tolerates fewer failures than the designers expected (and required), and it was information provided by our analysis that helped guide the development of a correct protocol.

Analysis of a reliable broadcast protocol illustrates how atomicity properties are handled in our framework. Atomicity properties are typically of the form: "All non-faulty components do *action*, or none of them do." In other words, atomicity properties correlate the multiplicities of actions at different sites. The reliable broadcast example demonstrates how symbolic multiplicities enable efficient analysis of atomicity properties.

3.1 Analysis of Fault-Tolerant Moving Agent Protocol

An interesting programming paradigm for distributed systems is that of *moving agents*. In this paradigm, agents move 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 must access a service located at site S_2 and move there. If the agent has gathered all of the information it requires, it might finish by moving to a final site A and delivering the result of the computation. The trajectory of sites visited by a moving agent is generally not known when the computation starts, since the trajectory might depend on information obtained as the computation proceeds. In this section, we consider protocols that enable moving agents to tolerate Byzantine failures of sites.

Replicated Two-Stage Moving Agent. To illustrate the fault-tolerance problems that arise with moving agents, consider a moving agent that visits two replicated services. See Figure 1. The moving agent starts at a source S, accesses service F, which is replicated at sites F_1, F_2, F_3, and then accesses service G, which is replicated at sites G_1, G_2, G_3. Since G is the last service it needs, the agent then moves to a *consolidator* B, which is responsible for delivering the result of the computation to actuator A. The consolidator computes the majority of the values it receives and sends that to the actuator; in addition, as discussed below, the consolidator uses an authentication mechanism to test validity of received values; invalid values are excluded from the vote.

The failure-free behavior of this moving agent (without any mechanism for fault-tolerance) is represented by the MFG in Figure 1. Constants F and G denote functions that represent the processing done by services F and G, respectively. For example, we see from the label on the edge from F_1 to G_1 in Figure 1 that the output of F_1 is in D and equals F applied to F_1's input. A typical moving agent accesses only some available services. To reflect this, the system shown in Figure 1 includes a service H, replicated at sites H_1, H_1, H_3 and not visited by this particular agent. The fault-tolerance requirement is:

MA-FTR. Inputs to actuator A are unaffected by Byzantine failure of a minority of the replicas of each service used by the moving agent and by Byzantine failure of any number of replicas of each service not used by the moving agent.

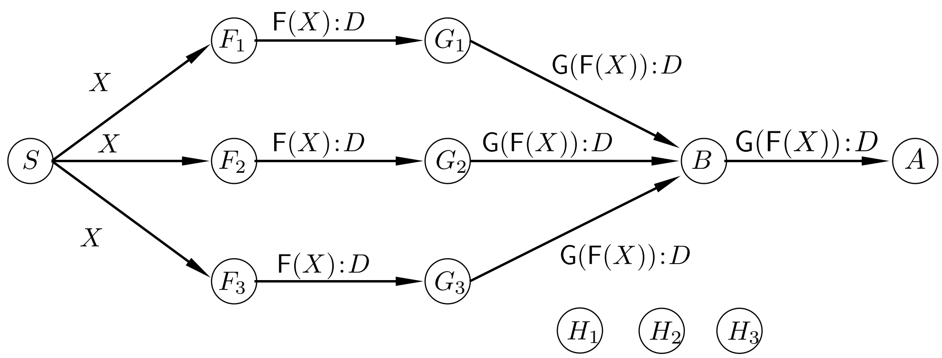

Figure 1: MFG for replicated two-stage moving agent.

Consider consolidator B in Figure 1. How does it decide which inputs are valid and therefore should be tallied? 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. However, that would assume a consolidator knows in advance that the last service visited by the moving agent will be service G—an unsound assumption, since the sequence of services visited by a moving agent is generally not known in advance.

The solution employed in the protocol we analyze is for validity of a message embodying a moving agent to be detemined by the consolidator based on the sequence of services that are visited. Digital signatures provide to the broker evidence of which services were visited; [MvRSS96] describes an alternative scheme based on shared secrets. Assume digital signatures are implemented using public-key cryptography, and assume that each component knows its own private key, the public key of every other component, and which service is provided by each server.

Each message sent by a source or server is signed, to foil faulty components that try to spoof (i.e., send messages that appear to be from other components). Each source or server includes in each outgoing message a declared destination, which is the name of the next service or consolidator to be visited by the moving agent embodied in that message. Each server x also includes in every outgoing message the incoming message that embodied the arrival of that moving agent at x; the consolidator uses this information when checking the validity of the message, as discussed below. Signatures on the recursively included messages provide a chain of evidence documenting the entire sequence of services actually visited by a moving agent that arrives at a consolidator.

A consolidator can test whether a message is valid by checking that the agent it embodies originated at a (legitimate) source, that the consolidator itself is the declared destination of the message, and that the sequence of declared next-destinations (obtained from the included messages) is consistent with the chain of signatures on the included messages. (The consolidator also verifies each of the signatures and considers the message invalid if any of those verifications fail.) In sum, a set S of messages is valid if: (1) each message in S is valid; (2) all messages in S contain the same sequence of declared destinations; (3) the final signatures on messages in S are (collectively) from a majority of the replicas of some service.

It is not difficult to describe the behavior of Byzantine faulty components and this protocol in our framework. Byzantine faulty components can spoof and eavesdrop (i.e., obtain copies of messages sent to other components). From the perspective of a message recipient, the possibility of spoofing causes uncertainty about the identity of the message sender. We model this uncertainty by using input-output functions that are independent of the names of the senders in the input history.

We model a faulty component (the eavesdropper) eavesdropping on a component x as the eavesdropper sending a distinguished value *evsdrp* to x, and we stipulate that the output history of a component that receives *evsdrp* allow the possibility of sending copies of all subsequent outputs to the eavesdropper.[1] Note that *evsdrp* messages are just modeling artifacts. A faulty server is assumed to be capable of eavesdropping on all components except actuators; actuators communicate only with consolidators.

To describe the protocol in our framework, we introduce some definitions. Let $D \in \mathit{AVal}$ describe the data carried by moving agents. Let $\mathit{Msg} \in \mathit{AVal}$ describe the messages sent by the protocol. Let Svc be a set of constants that are names of services. The processing done by a service $S \in \mathit{Svc}$ is represented by a constant operator S, as in Figure 1. We assume component names can be used as constants. Let $\mathit{Src} \subseteq \mathit{Name}$ be the set of names of (legitimate) sources. The set of private keys is $\mathit{Key} = \bigcup_{x \in \mathit{Name}} \mathsf{k}_x$, where $\mathsf{k}_x \in \mathit{Key}$ represents component x's unique private key (used to sign messages).

To conveniently represent messages sent by sources, we introduce a constant $\mathsf{msg_0}$ with the following interpretation: $\mathsf{msg_0}(k, \mathit{data}, \mathit{des})$ represents a message signed with private key $k \in \mathit{Key}$, carrying data represented by $\mathit{data} \in \mathit{SVal}$, and with destination (either a service or the name of a broker) represented by $\mathit{des} \in \mathit{SVal}$.

To conveniently represent messages sent by servers, we introduce a constant msg with the following interpretation: $\mathsf{msg}(k, \mathit{data}, \mathit{des}, \mathit{msg})$ represents a message signed with private key $k \in \mathit{Key}$, carrying data represented by $\mathit{data} \in \mathit{SVal}$, with destination (either a service or the name of a broker) represented by $\mathit{des} \in \mathit{SVal}$, and with $\mathit{msg} \in \mathit{SVal}$ representing a message that caused the server that received it to send this message.

The MFG in Figure 2 shows the behavior of this protocol for the replicated two-stage moving agent described above. The following abbreviations are used in the figure:

$$\begin{aligned} m0 &= \mathsf{msg_0}(\mathsf{k}_S, X, F) \\ m1_i &= \mathsf{msg}(\mathsf{k}_{F_i}, \mathsf{F}(X), G, m0) \\ m2_{i,j} &= \mathsf{msg}(\mathsf{k}_{G_j}, \mathsf{G}(\mathsf{F}(X)), B, m1_i). \end{aligned}$$

Tolerating Failure of Multiple Visited Services. The above protocol provides some fault-tolerance but does not satisfy MA-FTR. For example, it does not tolerate simultaneous failure of F_1 and G_2, because two of the consolidator's three inputs might be corrupted by these failures. To

[1] For this purpose, we allow an exception to the name-independence rule in the previous paragraph.

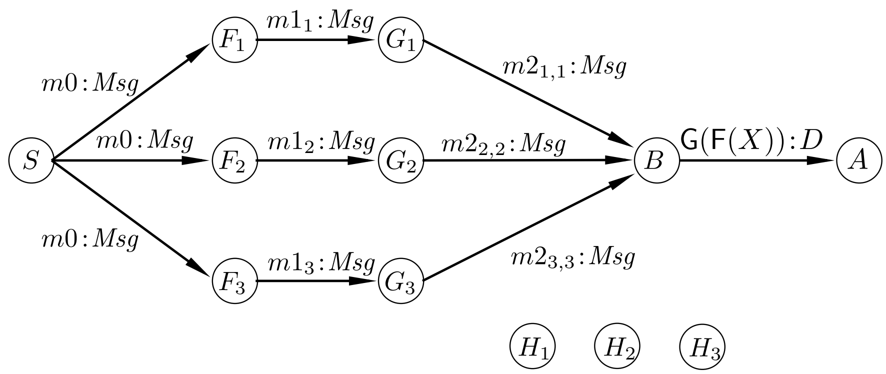

Figure 2: MFG for replicated two-stage moving agent, with authentication.

make the moving agent protocol more robust, we modify it so that (1) each server sends its outgoing messages to *all* replicas of the next service, and (2) validity tests and voting are incorporated into each stage of the computation after the first. The validity test and voting are just as described for consolidators.[2] Thus, a server sends messages only after receiving a valid set S of messages; to document the sequence of services visited by the moving agent, the server includes some message from S in the outgoing messages.[3] The behavior of the revised protocol is shown in the MFG in Figure 3. Each server G_j might include any one of its three input messages in its output, so the value in its outputs is a set of three possibilities; specifically, the value is $(ms_j \times \{Msg\}) \in Val$, where

$$ms_j = \{m2_{1,j}, m2_{2,j}, m2_{3,j}\}.$$

Detailed input-output functions for this protocol appear in [Sto97].

Analysis Results

MA-FTR requires that in each failure scenario in which a minority of the replicas of each service used by the moving agent fail and any number of replicas of each service not used by the moving agent fail, the input to the actuator is represented by symbolic value $\mathsf{G}(\mathsf{F}(X))$, as in the failure-free computation. We describe below the MFGs obtained for a few representative failure scenarios.

Failure of Visited Servers Only. Consider the failure scenario in which F_1 and G_2 fail. Let N be the set of servers and brokers—that is, those components to which servers can send messages; here, $N = \{F_1, F_2, F_3, G_1, G_2, G_3, H_1, H_2, H_3, B\}$. The fixed-point computed for this failure scenario is the same MFG as in Figure 3 except outputs of the faulty components are different and other

[2]The only remaining differences between a server and a consolidator are: (1) a consolidator does not perform application-specific computation, i.e., does not apply an operator to the data carried by the moving agent; (2) a consolidator does not include authentication information in its outputs, because the channel between the consolidator and the actuator is assumed to be secure.

[3]The reader who wonders whether multiple messages from S should be included in the outgoing messages is referred to Section 3.1.

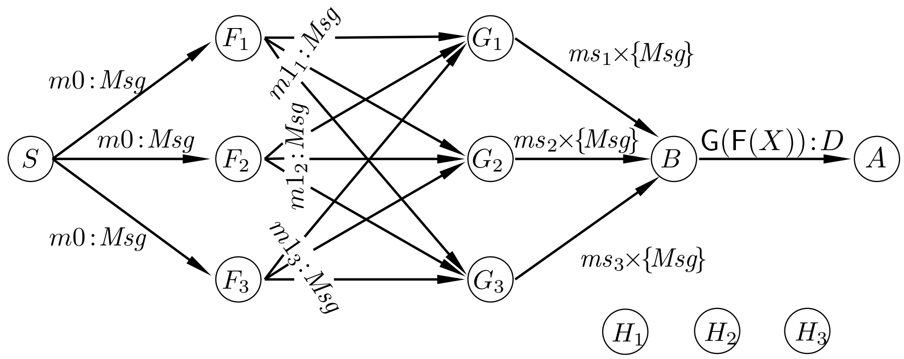

Figure 3: Run of replicated two-stage moving agent, with authentication and with voting after each stage. Each skewed ms-atom labels each of the three edges it crosses. Note that $ms_j \times \{Msg\} = \{m2_{1,j}\!:\!Msg,\, m2_{2,j}\!:\!Msg,\, m2_{3,j}\!:\!Msg\}$.

components send messages to the faulty components as a result of eavesdropping. Specifically, for $x \in \{F_1, G_2\}$, for $y \in N \setminus \{x\}$, edge $\langle x, y\rangle$ is labeled with the ms-atom

$$\{evsdrp, Arb(\{\mathsf{k}_{F_1}, \mathsf{k}_{G_2}\}, \{m0, m1_2, m1_3, m2_{2,1}, m2_{3,1}, m2_{2,3}, m2_{3,3}\})\}^*.$$

Arb was described in Section 2.1. Also, for $x \in \{F_1, G_2\}$ and $y \in N \setminus \{F_1, G_2\}$, edge $\langle y, x\rangle$ is labeled with all the output ms-atoms of component y in Figure 3 but with the multiplicities changed to ?. The input to the actuator is still represented by $\mathsf{G}(\mathsf{F}(X))$, so MA-FTR is satisfied in this failure scenario.

Failure of Unvisited Servers Only. Consider the failure scenario in which H_1, H_2, and H_3 all fail. The fixed-point computed for this failure scenario is the same MFG as in Figure 3 except outputs of the faulty components are different and other components send messages to the faulty components as a result of eavesdropping. Specifically, for $x \in \{H_1, H_2, H_3\}$, for $y \in N \setminus \{x\}$, edge $\langle x, y\rangle$ is labeled with the ms-atom

$$\{evsdrp, Arb(\{\mathsf{k}_{H_1}, \mathsf{k}_{H_2}, \mathsf{k}_{H_3}\}, \bigcup_{i,j\in\{1,2,3\}} \{m0, m1_i, m2_{i,j}\})\}^*.$$

Also, for $x \in \{H_1, H_2, H_3\}$ and $y \in N \setminus \{H_1, H_2, H_3\}$, edge $\langle y, x\rangle$ is labeled with all the output ms-atoms of component y in Figure 3 but with the multiplicities changed to ?. The input to the actuator is represented by $\mathsf{G}(\mathsf{F}(X))$, so MA-FTR is satisfied in this failure scenario.

Failure of Visited and Unvisited Servers. Finally, consider the failure scenario in which F_1, H_1, and H_2 fail. The protocol violates MA-FTR in this failure scenario. The problem is revealed by tracing the first three iterations of the fixed-point computation, which lead to the MFG in Figure 4. F_1 includes $m0$ in signed messages with all possible declared destinations, including H, and carrying arbitrary data. F_1 sends these messages to all components in N. H_1 and H_2 receive these messages and include them in signed messages carrying arbitrary data and with all possible declared destinations, including B. The messages from H_1 and H_2 might now pass the consolidator's validity test and cause the consolidator to send arbitrary data to the actuator, as indicated by the label on the edge from B to A in Figure 4. Thus, in this failure scenario, actuator A gets the wrong input. The symmetry of the MFG in Figure 4 reflects the symmetric behavior of faulty components: they send all of their outputs to every server and consolidator. One way to fix the protocol is to have servers include in each output message input messages from a majority of the replicas of some service instead of—as they now do—only a single input message that transported the agent.

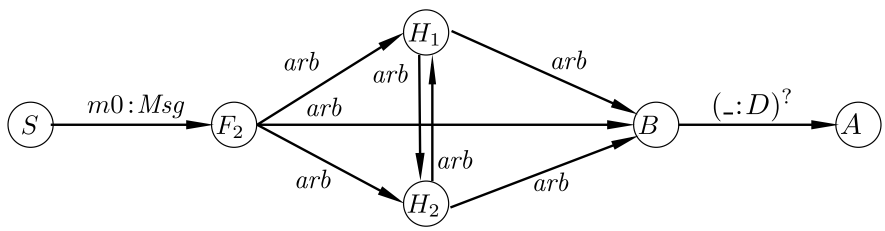

Figure 4: MFG for replicated two-stage moving agent after three iterations of fixed-point calculation, with F_1, H_1, and H_2 faulty. Here, $arb = Arb(\{\mathsf{k}_{F_1}, \mathsf{k}_{H_1}, \mathsf{k}_{H_2}\}, \{m0\})^*$. Components F_2, F_3, G_1, G_2, G_3 and H_3 and edges incident on them are elided, because the flaw in the protocol manifests itself without them. Labels that merely represent eavesdropping are also omitted, to avoid clutter.

3.2 Reliable Broadcast Example

In a reliable broadcast, components of the system correspond to processes. Clients $C_1, C_2, \ldots, C_N$ *broadcast* messages, and servers $S_1, S_2, \ldots, S_N$ *deliver* messages to clients. Each C_i communicates directly only with a corresponding server S_i. The correctness requirements for reliable broadcast are:

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

Integrity: For each message m, every non-faulty server delivers m at most once and does so only if m was previously broadcast by some client.

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

Send-omission failures cause a server to omit to send some (possibly all) messages that it would normally send [Had84, HT94]. The above Validity, Integrity, and Agreement requirements must be satisfied in failure scenarios in which the network remains connected (i.e., between each pair of non-faulty clients, there is a path in the connectivity graph containing only non-faulty servers), even if some servers suffer send-omission failures.

A reliable broadcast protocol is given in [HT94, section 6]. It relies on the assumption that each message broadcast is unique. This assumption is normally discharged by including the broadcaster's name and a sequence number (or timestamp) in each message. To model inclusion of the broadcaster's name, we use an abstract value $MF(C_i)$ (mnemonic for "message from C_i") to represent messages broadcast by client C_i. To model inclusion of a sequence number (or timestamp), we assert that, for each client, variables used to represent data broadcast by the client have unique values. In our framework, one can associate with each system component an assertion that constrains the values of that component's local variables. For example, we might use variables X and Y to represent the data in different messages broadcast by a client and assert that $X \neq Y$.

The protocol of [HT94, section 6] works as follows.

- Client C_i broadcasts a message m by sending m 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 relays the message to its neighboring servers and to its client.

The MFG in Figure 5 represents the failure-free behavior of this protocol in a system with three clients for executions where client C_1 broadcasts a single message. Variable X represents the data that is broadcast.

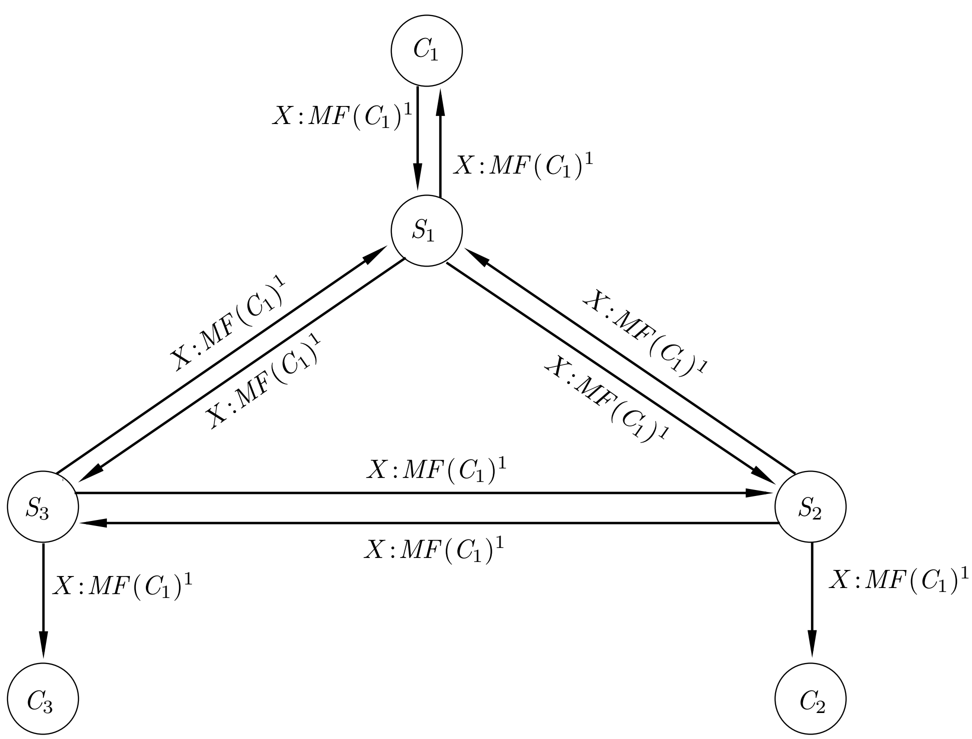

Figure 5: Failure-free behavior of reliable broadcast protocol.

Now, consider the failure scenario where S_1 suffers send-omission failures. To see why symbolic multiplicities are useful here, suppose they were not used. In this case, each of S_1's outputs has a possibility of occurring or not occurring and hence would have (abstract) multiplicity "?". And this uncertainty would propagate through the MFG during the fixed-point calculation. The result would be an MFG like the one shown in Figure 5, except with every multiplicity other than the one on the edge from C_1 to S_1 replaced with "?". We cannot conclude from this MFG that the Agreement requirement is satisfied, because this MFG represents executions in which S_2 delivers the message and S_3 does not, and *vice versa*. We know (from [HT94]) that such executions are not possible for this protocol.

We thus use symbolic multiplicities to model the protocol more accurately. Having the same non-wildcard symbolic value in the multiplicities on edges $\langle S_2, C_2 \rangle$ and $\langle S_3, C_3 \rangle$ implies that S_2 and S_3 deliver the message the same number of times. Since the multiplicities of those edges depend on which messages S_1 sends, the input-output functions use variables to represent the multiplicity with which S_1 sends each message. The following naming scheme is used for these variables: the value (zero or one) of $M_{x,y}^{C,i}$ indicates whether server x relays to component y the i'th message broadcast by client C. If a server receives the same message with multiplicities $X_1:?, \ldots, X_k:?$, then it relays that message with multiplicity $\mathsf{max}(X_1, \ldots, X_k):?$.

Consider the scenario in which client C_1 sends a single message and server S_1 is faulty. Let $M_{x,y}$ abbreviate $M_{x,y}^{C_1,0}$. The fixed-point calculation proceeds as follows.

1. Client C_1 initiates a broadcast by sending a message, represented by $X:MF(C_1)^1$, to S_1.

2. S_1 would normally relay the message to its neighbors. But a faulty S_1 might omit to do so. This is represented by S_1 sending $X:MF(C_1)^{M_{S_1,x}:?}$ to each neighbor $x \in \{C_1, S_2, S_3\}$.

3. If S_2 receives the message, then it relays the message to its neighbors. S_3 does the same. The resulting MFG appears in the top part of Figure 6.

4. If S_2 received the message from either of its neighboring servers, then it relays the message to its neighbors; this is reflected by the use of max in its outputs, as described above. S_3 does the same. The resulting MFG, which is the fixed-point, appears in the bottom part of Figure 6.

One can see from the final MFG that the correctness requirements for reliable broadcast are satisfied in this failure scenario. Validity is vacuous in this scenario, because C_1 is the only client that sends a message and S_1 is faulty. Integrity holds because all the ms-atoms on inedges of clients have symbolic value X, which was broadcast by C_1. Agreement holds because the same symbolic multiplicity appears in the ms-atoms on edges $\langle S_2, C_2 \rangle$ and $\langle S_3, C_3 \rangle$; S_1 is faulty in this scenario, so Agreement does not constrain the multiplicity with which S_1 delivers X.

If symbolic multiplicities were not used, accurate analysis of this protocol would be possible but more expensive. It could be done by regarding omissions of different subsets of S_1's failure-free outputs as different failures of S_1. This leads to a relatively large number of failure scenarios, which makes the analysis computationally more expensive.

3.3 Comparison to State-Space Exploration

It is instructive to compare the compactness of MFGs and the efficiency of our analysis to state-space exploration optimized with partial-order methods. For concreteness, we consider Spin [Hol97], which incorporates the partial-order method described in [HP94].

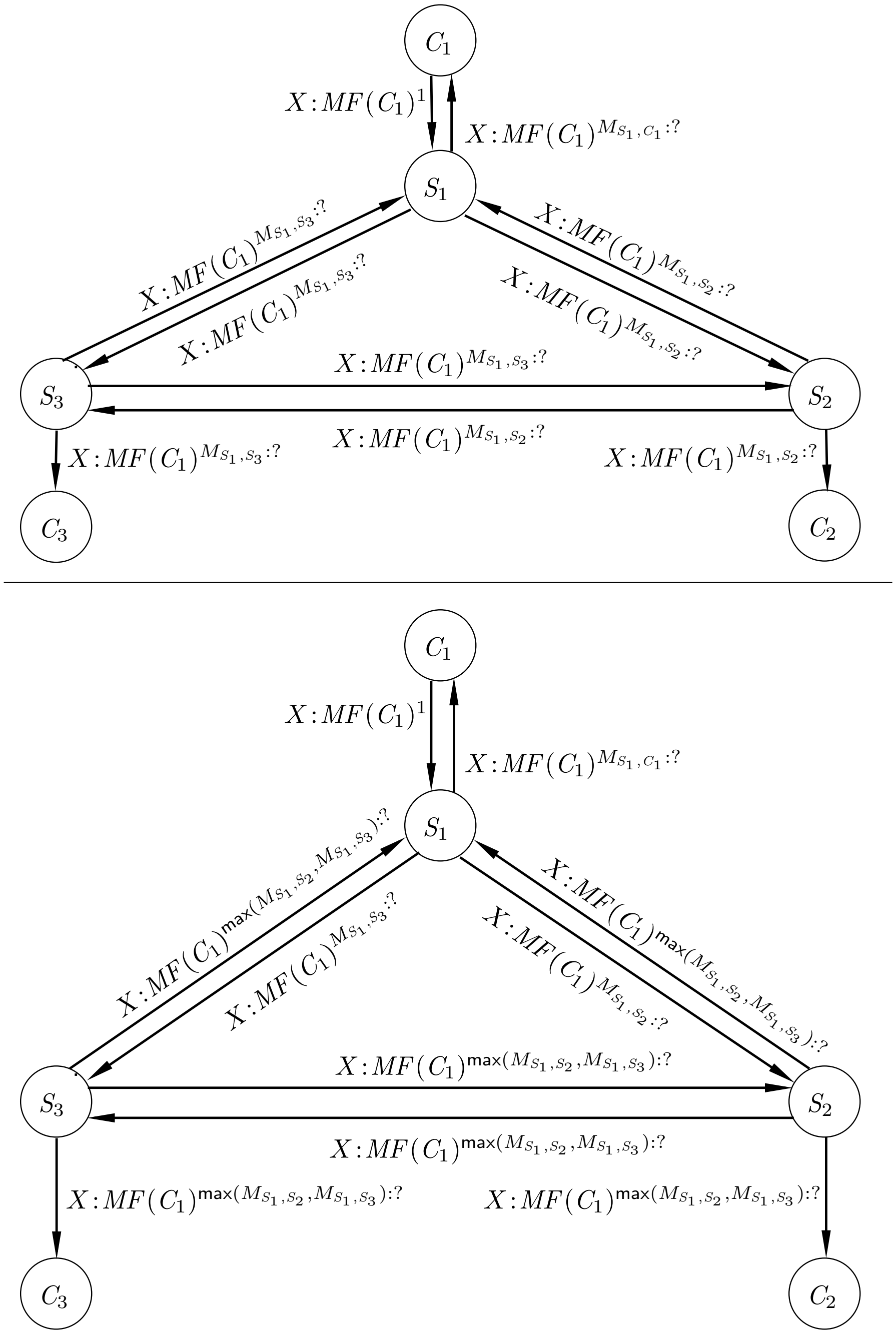

Figure 6: MFGs for reliable broadcast protocol when S_1 is faulty. Top: the MFG obtained after three applications of the *step* function. Bottom: the fixed-point.

Transitions of the same process are always dependent [HP94], due to control dependencies, so Spin would explore all reachable interleavings of the inputs to each component. As mentioned in Section 1, our framework does not directly represent inter-channel orderings, so it does not explore such interleavings. This difference often causes the size of the state space explored by Spin (measured by the number of states) to be larger than the size of the MFG (measured by the number of occurrences of constants and variables in ms-atoms). This occurs for both of the above examples. For the corrected protocol for moving agents in a system with N replicas of each service, the explored state space would contain states corresponding to different subsets of the $O(N)$ messages sent to each server being in the channels and would therefore be a factor of 2^N larger than the MFG. For reliable broadcast with N servers in a fully connected network, the explored state space would be a factor of 2^{N-1} larger than the MFG for scenarios involving a single broadcast. Since there are $(N-1)!$ interleavings of the inputs to each server, generating the state space takes $\Omega((N-1)!)$ time, whereas generating the MFG takes $O(N^2)$ time.

Symbolic multiplicities further improve the efficiency of the analysis of the reliable broadcast protocol in failure scenarios involving crash failures [LF82] or send-omission failures, because omissions of different sets of messages lead (at least temporarily) to different states, while symbolic multiplicities avoid explicit branching based on whether a message is sent or omitted. For example, with N servers in a fully connected network, for scenarios involving a single broadcast, this difference causes an exponential factor in the ratio of the size of the explored state space to the size of the MFG, in addition to the exponential factor described in the previous paragraph. Using queue-content decision diagrams (QDDs) [BG99] in the state-based approach would not provide this benefit.

To make the comparison concrete, we implemented the reliable broadcast example of Section 3.2 in Spin. As above, we consider scenarios involving one broadcast. It turns out that the number of explored states is smaller if each process has a single input channel, so we model the system that way. We used the largest possible atomic blocks when writing the protocol in Spin's input language; this also helps reduce the number of explored states. For $N = 3$ with no failures, the MFG in Figure 5 has size 30 (note that $M_{x,y}$ is a single constant), while Spin stores 224 states. For $N = 4$ with S_1 and S_2 having send-omission failures, the MFG in [Sto97, Figure 4.7], which is similar to the one on the bottom of Figure 6, has size 100, while Spin stores 3778 states.

4 An Implementation

We implemented our analysis method in a prototype tool called CRAFT (Cornell Rapid Analyzer for Fault Tolerance). CRAFT is implemented in CAML Light [Ler97], a dialect of Standard ML [MTH90]. The graphical interface is implemented using CamlTk, a CAML interface to the Tk widget library [Ous94]. CRAFT provides a collection of CAML types and functions used to express input-output functions and compute fixed-points, plus a graphical interface to facilitate entry of systems and inspection of analysis results. The screen-dump in Figure 7 shows an MFG similar to the one in Figure 1.

Allowing input-output functions representing system components to be written directly in CAML allows the full power of CAML and its libraries to be used. Users unfamiliar with CAML can use the graphical interface to describe systems whose components are represented by input-

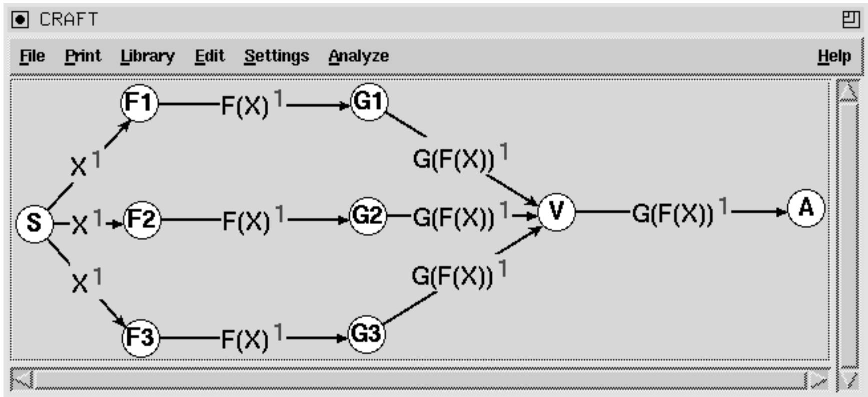

Figure 7: Screen-dump of tool showing MFG similar to the one in Figure 1. CRAFT has an option, used here, to elide abstract values in the *Val* part of ms-atoms.

output functions selected from libraries, specify failure scenarios, compute MFGs for the specified systems in the specified failure scenarios, and inspect the resulting MFGs. In displays produced by CRAFT, if the textual representation of the poset of ms-atoms labeling an edge does not fit on the line representing that edge on the screen, then the poset is elided. But a user can click on the edge to pop up a window detailing the ms-atoms on that edge.

5 Related Work and Discussion

A formal presentation of our analysis method, including a semantics and a proof of soundness, appears in [Sto97]. The semantics relates MFGs and input-output functions to objects in a more conventional system model (based on [Bro87, Bro90]), in which the behavior of a system of *processes* is characterized by a set of *executions* involving concrete (not abstract) values. Specifically, the semantics defines when an input-output function represents a process and when an MFG represents an execution. Such a semantics provides a rigorous framework for showing independently that each input-output function represents the corresponding process. Soundness is captured by a theorem that says (roughly):

> For all interpretations of constants, all systems of input-output functions, and all systems of processes, if each input-output function represents the corresponding process, then for each failure scenario, if the fixed-point calculation terminates with an MFG g, then all finite executions of the system of processes for that failure scenario are represented by g.

Infinite executions are excluded because MFGs (as currently defined) are not suitable for representing infinite executions; this issue is discussed in Section 5.3.

5.1 Abstraction

Our abstractions are, in some ways, similar to those proposed by Clarke, Grumberg, and Long [CGL94] and to those proposed by Kurshan [Kur89, Kur94].

Clarke *et al.* [CGL94] developed a method for using abstractions to reduce the complexity of temporal-logic model-checking. The class of abstractions they consider corresponds roughly to abstract interpretation and to our abstract values. They also propose so-called *symbolic abstractions*, which are convenient abbreviations for finite families of abstractions. Our symbolic values are closer to the technique they sketch at the conclusion of their paper for dealing with infinite-state systems than to their "symbolic abstractions".

In Kurshan's automata-based verification methodology, approximations are embodied in reductions between verification problems [Kur89, Kur94]. A typical reduction might collapse multiple states of an automaton to form a single state of some reduced automaton; this is analogous to introducing abstract values. Relationships between concrete values can be captured using (implicitly) parameterized families of reductions, reminiscent of Clarke, Grumberg, and Long's "symbolic abstractions".

For problems involving related values (e.g., X, $\mathsf{F}(X)$, and $\mathsf{G}(\mathsf{F}(X))$), the family of reductions must introduce an abstract value representing each of these values. In effect, the user must determine in advance all relevant symbolic values and introduce an abstract value for each. In contrast, with our method, the user need only determine for each component how constants and how its local variables are used to represent the computations performed by that component. Symbolic values are constructed dynamically by input-output functions as part of the fixed-point calculation. Our notion of local variables supports modular introduction of symbolic values. In Clarke *et al.*'s and Kurshan's methods, the abstract values that correspond to our symbolic values—and in particular those that correspond to variables associated with different components—must all be introduced together in the definition of the reduction (or the "abstraction", in the terminology of [CGL94]). In contrast, our framework often allows a user to introduce an input-output function representing a process (in other words, the input-output function is a reduced version of the process or an abstraction of the process) independently of the other processes and input-output functions, though sometimes information provided by an invariant constraining the values of non-local variables is needed.

An attractive feature of Clarke *et al.*'s work and Kurshan's work is that abstractions (or reductions) are specified as homomorphisms and applied to programs (or automata) automatically. Our framework does not currently provide such a convenient method for specifying abstractions; this is a direction for future work.

5.2 Inter-channel orderings

Because MFGs do not describe inter-channel orderings, our analysis suffers (in effect) from the merge anomaly [Kel78, Bro88]. Specifically, an input-output function representing a non-strict process cannot (in general) represent the process's behavior exactly; generally, the input-output function must be a conservative approximation. One way to remedy this would be to augment MFGs with a partial ordering that can express inter-channel orderings; this is reminiscent of Brock and Ackermann's scenarios [BA81, Bro83] and Pratt's model of processes [Pra82].

5.3 Infinite Executions

Multiplicities are defined to represent subsets of the natural numbers, so they can represent unbounded but not infinite sequences. One approach to analyzing systems with infinite executions is to generalize multiplicities along the lines of ω-regular-expressions. However, with this approach, termination of the analysis for most interesting systems will require approximations too coarse for checking whether the fault-tolerance requirement is satisfied.

Another approach is based on "factoring" of system behavior. Many fault-tolerant distributed systems are reactive systems that (in theory) can process unbounded streams of requests. For example, the reliable broadcast protocol described Section 3.2 can perform an arbitrary number of broadcasts. Termination of our analysis depends on factoring the system's behavior into subcomputations that can be analyzed separately. Such factoring is common: it can be seen in the analysis of the arithmetic pipeline in [CGL94] and in the analysis of the queue in [Kur94, Appendix D]. Our analysis in Section 3.2 of the reliable broadcast protocol considers computations involving only one broadcast, but (informally) this is sufficient, since it is easy to see that the protocol handles each broadcast independently.

Many fault-tolerant reactive systems have statically-determined periodic schedules, so it is natural to factor (decompose) the executions into periods and analyze one period. An approach along these lines to verification of aircraft control systems is described in [DBC91, Rus93]. Although that work uses a theorem prover, the same ideas could be used in our framework to verify whether a control system tolerates a specified rate of failures.

Acknowledgements. We would like to thank Yanhong Liu, Yaron Minsky, Robbert van Renesse, and Leena Unnikrishnan for helpful discussions.

References

[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.

[BG99] Bernard Boigelot and Patrice Godefroid. Symbolic verification of communication protocols with infinite state spaces using qdds. *Formal Methods in System Design*, 4(3):237–255, May 1999.

[Bro83] J. Dean Brock. *A Formal Model of Non-Determinate Dataflow Computation*. PhD thesis, 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 J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *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.

[CC77] Patrick Cousot and Radhia Cousot. A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[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.

[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.

[CS96] Rance Cleaveland and Steve Sims. The NCSU Concurrency Workbench. In R. Alur and T. Henzinger, editors, *Proc. 8th Int'l. Conference on Computer-Aided Verification (CAV)*, 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.

[Had84] Vassos Hadzilacos. *Issues of Fault Tolerance in Concurrent Computations*. PhD thesis, Harvard University, 1984. Also appeared as Department of Computer Science Technical Report 11-84.

[Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[Hol97] Gerard J. Holzmann. The Spin model checker,. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[HP94] Gerard J. Holzmann and Doron Peled. An improvement in formal verification. In *Proc. International Conference on Formal Description Techniques (FORTE)*, 1994.

[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.

[JN94] Neil D. Jones and Flemming Nielson. Abstract interpretation: A semantics-based tool for program analysis. In *Handbook of Logic in Computer Science*, pages 527–629. Oxford University Press, 1994.

[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.

[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 University Press, 1994.

[Ler97] Xavier Leroy. *The Caml Light System*. INRIA, 1997. Available via http://pauillac.inria.fr/caml/.

[LF82] Leslie Lamport and M. J. Fischer. Byzantine generals and transaction commit protocols. Technical Report 62, SRI International, 1982.

[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.

[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 D. 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.

[Ous94] John Ousterhout. *Tcl and the Tk Toolkit*. Addison Wesley, 1994.

[Pel98] Doron Peled. Ten years of partial order reduction. In Alan J. Hu and Moshe Y. Vardi, editors, *Proc. 10th Int'l. Conference on Computer-Aided Verification (CAV)*, volume 1427 of *Lecture Notes in Computer Science*, pages 17–28. Springer-Verlag, 1998.

[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.

[Rus93] John Rushby. A fault-masking and transient-recovery model for digital flight-control systems. In Jan Vytopil, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 109–136. Kluwer, 1993. Also appeared in SRI International Computer Science Laboratory Technical Report SRI-CSL-93-04.

[Sch94] Henk Schepers. A trace-based compositional proof theory for fault tolerant distributed systems. *Theoretical Computer Science*, 128(1–2):127–157, 1994.

[Sch97] Fred B. Schneider. Towards fault-tolerant and secure agentry. In Marios Mavronikolas, editor, *Proc. 11th International Workshop on Distributed Algorithms (WDAG '97)*, volume 1320 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[SS83] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, 1983.

[SS98] Scott D. Stoller and Fred B. Schneider. Automated stream-based analysis of fault-tolerance. In *Fifth International Symposium on Formal Techniques in Real Time and Fault Tolerant Systems (FTRTFT)*, volume 1486 of *Lecture Notes in Computer Science*, pages 113–122, Lyngby, Denmark, September 1998. Springer-Verlag.

[Sto97] Scott D. Stoller. *A Method and Tool for Analyzing Fault-Tolerance in Systems*. PhD thesis, Cornell University, May 1997.

[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, 1993.