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

March 5, 2003

Abstract

A method for automated analysis of fault-tolerance of distributed systems is presented. It is based on a stream (or data-flow) model of distributed computation. Temporal (ordering) relationships between messages received by a component on different

*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.

channels are not captured by this model. This makes the analysis more efficient and forces the use of conservative approximations in analysis of systems whose behavior depends on such inter-channel orderings. To further support efficient analysis, our framework includes abstractions for the contents, number, and ordering of messages sent on each channel. Analysis of a reliable broadcast protocol illustrates the method.

1 Introduction

As computers become integrated into critical systems, there is a growing need for techniques to establish that software systems satisfy their requirements. This paper describes a method and automated tool for checking whether a distributed system satisfies its requirements, with a focus on fault-tolerance requirements.

The method 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 could easily be accommodated, though.

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. Stream-processing models admit compact representations for sequences of messages, and those representations can be used directly in MFGs and as inputs to input-output functions.

A *failure scenario* for a system is an assignment of failures to a subset of the system's components. A fault-tolerance requirement is a condition that the system's behavior should satisfy in specified failure scenarios. Our analysis method is to compute, for each failure scenario for which the user specified a fault-tolerance requirement, an MFG representing the system's communication behaviors in that failure scenario. Each MFG is then checked to determine whether the fault-tolerance requirement for that failure scenario is satisfied. A more common method 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 to help keep the MFGs small and simple.

To reduce the computational cost of computing MFGs, our framework supports flexible and powerful abstractions (approximations). Traditionally, stream-processing models have been used as mathematical semantics and contained no abstractions. We use only conservative abstractions, so the analysis never falsely implies that a system satisfies its fault-tolerance requirement. 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.

Message sequence charts are a well-known notation for communication behavior [IT96]. A message sequence chart typically represents one possible communication behavior or a set of similar behaviors, while an MFG succinctly and perhaps approximately represents all possible communication behaviors of a system (in a specified failure scenario).

Our analysis does not construct global states of systems and does not consider interleavings of messages sent by a component on different channels (to different destinations) or interleavings of messages received by a component on different channels (from different sources). This sometimes forces the use of conservative approximations, but it makes our analysis more efficient than state-based or message-based analyses that explicitly consider interleavings of behavior of different components, because the number of possible interleavings is typically large. Section 3 illustrates this point with an example.

2 Communication-Based Analysis Framework

Section 2.1 introduces the running example used to illustrate our framework. Sections 2.2–2.4 describe the abstractions used in our framework to succinctly represent sets of sequences of messages that possibly flow along a channel. The abstractions 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). Sections 2.5 and 2.6 define input-output function and MFGs, which are the foundation of our analysis method. Section 2.6 also describes how, given input-output functions representing components, an MFG representing the streams of messages sent on each channel during execution is computed as a fixed-point. A detailed formal presentation of our analysis method appears in [Sto97].

2.1 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 client C_i communicates directly only with a corresponding server S_i. Following [HT94, section 3.1], we assume that each message broadcast by a client is unique. In an implementation, this assumption is normally discharged by including the broadcaster's name and a sequence number (or timestamp) in each message. The correctness requirements for reliable broadcast are [HT94]:

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. The above Validity, Integrity, and Agreement requirements must be satisfied in failure scenarios in which some servers suffer send-omission failures and the network remains connected (i.e., between each pair of clients, there is a path in the connectivity graph containing only non-faulty servers).

The reliable broadcast protocol in [HT94, section 6] serves as a running example. The protocol works as follows. A 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.

2.2 Abstraction for Values

To analyze streams of messages, we need a notation to describe the possible data values in each message. As in abstract interpretation, we introduce a set *AVal* of *abstract values*. Each abstract value represents a set of concrete values. For the reliable broadcast example, abstract value *Msg* represents all messages, and $MF(C_i)$ (mnemonic for "message from C_i") represents messages whose header indicates that they were broadcast by C_i.

For our analysis goals, these abstract values alone capture too little information about relationships between values. For example, to show that Integrity holds for a reliable broadcast protocol, the analysis must show that the messages delivered by two non-faulty servers are equal. If those messages were represented only by an abstract value like $MF(C_i)$, 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 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.

A *constant* represents the same value in every execution of a system. In the reliable broadcast example, the constant max represents the "maximum" 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 indeterminacy commonly stems 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, or non-determinism from failures. 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, so we define $\mathit{Val} \triangleq \mathit{SVal} \times \mathit{AVal}$. We usually write a pair $\langle s, a \rangle$ in Val as $s\!:\!a$. In the reliable broadcast example, $X\!:\!\mathit{MF}(C_1)$ denotes a value in $\mathit{MF}(C_1)$ that is represented by symbolic value X. A straightforward generalization, useful for analysis of some non-deterministic systems, is to represent the possible data values in a message by a set of such pairs, each representing some of the possible values [Sto97]. A wildcard is similar in meaning to omission of a symbolic value, so we usually elide wildcards. For example, $\langle \_, \mathit{Msg} \rangle$ would be written as Msg.

2.3 Abstraction for Multiplicities

We refer to the number of times a message is sent as its *multiplicity*. Our framework includes abstractions for multiplicities that allow compact descriptions of the possible behaviors of systems in which messages have different multiplicities in different executions. Such variation in multiplicities commonly stems from the same sources of non-determinism mentioned in

Section 2.2. In the reliable broadcast example, a server subject to send-omission failures might emit outputs with a multiplicity of zero or one. A component subject to Byzantine failures [LSP82] might emit outputs with an arbitrary multiplicity.

We think of multiplicities as natural numbers and therefore represent them in the same way as data values. Thus, we define $Mul \triangleq SVal \times AMul$, where the set $AMul$ of *abstract multiplicities* is a subset of $AVal$ and contains abstract values that represent subsets of the natural numbers. For the reliable broadcast example, we assume $AMul$ contains the following: 1, denoting $\{1\}$, and ?, denoting $\{0, 1\}$. The notational conventions for Val also apply to Mul. For example, $\langle \_, ? \rangle \in Mul$ may be written as ?.

Symbolic values in multiplicities naturally express correlations between multiplicities of different events. In the reliable broadcast example, the essence of the Agreement requirement is that the multiplicities with which any two non-faulty servers deliver a message should be equal. Send-omission failures by a server may cause the abstract values in those multiplicities to be ?. From that alone, we cannot determine whether the multiplicities are equal, but if the multiplicities contain the same non-wildcard symbolic value, then they are equal. More generally, symbolic multiplicities support efficient analysis of systems with atomicity requirements of the form: "All non-faulty components perform some action, or none of them do."

2.4 Abstraction for Sequences of Messages

The set of sequences of messages possibly sent along a channel is represented by a partially ordered set (poset) $\langle S, \prec \rangle$, where S is a set, and $\prec$ is an acyclic transitive binary relation on S. Each element of S represents a set of messages, as detailed below. The meaning of the partial order is: for $x, y \in S$, if $x \prec y$, then the messages represented by x are sent (and received, since channels are FIFO) before the messages 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.

Partial orders allow compact representation of the set of possible sequences of messages when orderings between some messages are uncertain. For the reliable broadcast example, consider a scenario in which a server receives two messages broadcast by different clients. If the order in which the server receives those two messages is undetermined, then the order in which it relays them is also undetermined, and the server's outputs are succinctly represented by a partially (not totally) ordered set.

Each element of the poset represents a set of messages. We call these elements *ms-atoms* (mnemonic for "message-set atoms") . 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 $\mathit{MSA} \triangleq \mathit{Val} \times \mathit{Mul}$. To promote the resemblance between ms-atoms and regular expressions, we write an ms-atom $\langle val, mul\rangle$ as val^{mul}, and if the multiplicity mul is 1, we usually elide it. For example, the ms-atom $X : \mathit{MF}(C_1)^{M:?}$ represents a set containing M messages with data represented by the value $X : \mathit{MF}(C_1)$, where the value of M is zero or one.

A notation based on finite automata, rather than regular expressions, could be used to represent sets of sequences of messages, as in [BG99, BH99]. 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.

2.5 Input-Output Functions

In our framework, inputs to a component are characterized by the possible sequences of messages received on its incoming channels. Outputs of a component are characterized by the possible sequences of messages sent on its outgoing channels. For simplicity, we assume a component has one incoming channel from (and hence one outgoing channel to) each component. We assume a system comprises a set of named components, with names from the set Name. Thus, inputs and outputs of a component are both represented by functions with signature $\mathit{Name} \to \mathit{POSet}(\mathit{MSA})$, where $\mathit{POSet}(\mathit{MSA})$ is the set of (strict) partial orders $\langle S, \prec \rangle$ such that $S \subseteq \mathit{MSA}$. We call such a function a *history* and define $\mathit{Hist} \triangleq \mathit{Name} \to \mathit{POSet}(\mathit{MSA})$.

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 of input-output functions is $\mathit{Hist} \to \mathit{Hist}$.

Recall from Section 1 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 $\mathit{IOF} \triangleq \mathit{Fail} \rightharpoonup (\mathit{Hist} \to \mathit{Hist})$, where Fail is the set of all possible failures, and one-hooked arrow $\rightharpoonup$ indicates partial functions. For the reliable broadcast example, Fail contains an element sendOm corresponding to send-omission failures. For $f \in \mathit{IOF}$, domain(f) is the set of failures that the component might suffer, and for each $\mathit{fail} \in$ 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 $\mathit{FS} \triangleq \mathit{Name} \to \mathit{Fail}$ that maps each component to one of its possible failures (possibly OK).

The input-output function f for a server in the reliable broadcast protocol works roughly as follows; details appear in [Sto97]. $f(\mathit{OK})(h)$ first collects all the data values in ms-atoms in h. Each such data value X corresponds to a broadcast message possibly received by the server. For each such data value X, $f(\mathit{OK})(h)$ symbolically computes the maximum of the multiplicities of all ms-atoms in h that contain X and then outputs X to all of its neighbors (i.e., its client and neighboring servers) with the resulting multiplicity M, because if the server receives X even once, it will relay X. A few straightforward rules are used to simplify M; for example, if any ms-atom in h that contains X has abstract multiplicity 1 (and the other ms-atoms that contain X have abstract multiplicity ?), then M has abstract multiplicity 1. $f(sendOm)(h)$ is similar except, for each of the server's neighbors y, the multiplicity with which the server sends X to y is the symbolic minimum of M (which is computed as above) and a unique variable V. Thus, V being equal to 0 or 1 corresponds to a send-omission failure occurring or not occurring, respectively, when X is relayed to y. The input-output function for server S uses the following naming scheme for these unique variables: the value of $M_{S,y}^{C,i}$ indicates whether a send-omission failure occurs when server S tries to relay to component y the i'th message broadcast by client C. Some straightforward simplifications are used; for example, the minimum of the multiplicities $\_ : 1$ and V :? is simplified to V:?.

In our framework, one can associate with each system component constraints on the values of the component's local variables. This is useful in the reliable broadcast example to model the assumption that each message broadcast by a client is unique. Specifically, inclusion of a sequence number (or timestamp) in each message broadcast by a client is modeled by associating with each client the constraint that variables used to represent messages broadcast by that client have unique values. For example, if variables X and Y represent the data in different messages broadcast by client C_1, then we have the constraint $X \neq Y$.

2.6 Message Flow Graphs and Fault-Tolerance Requirements

In our framework, system behavior is represented by *message flow graphs* (MFGs). Each node of a MFG corresponds to a system component, and each edge $\langle x, y \rangle$ is labeled with a description of the sequence of messages possibly sent from its source x to its target y. 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. The signature of MFGs is $MFG \triangleq (Name \times Name) \rightarrow POSet(MSA)$.

The MFG in Figure 1 represents the failure-free behavior of the reliable broadcast 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. We describe below how this MFG is

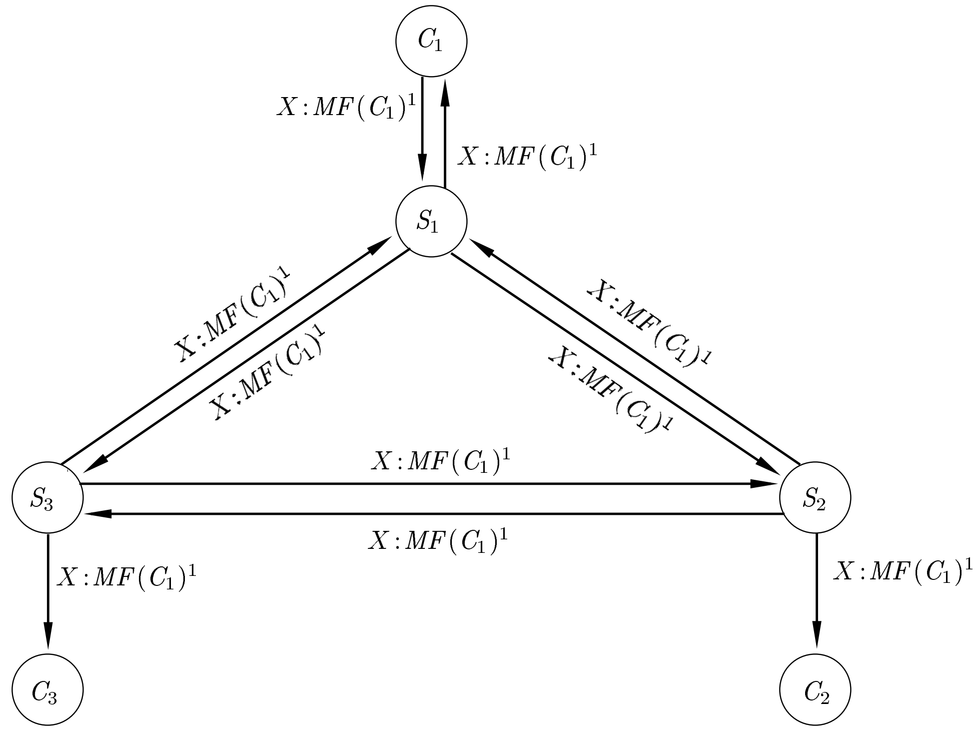

Figure 1: Failure-free behavior of reliable broadcast protocol.

computed.

A system is represented by a function $nf \in Name \rightarrow IOF$, 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

$$
\begin{array}{llll}
step_{nf,fs}(g) \triangleq & \multicolumn{3}{l}{\text{the MFG } g', \text{ where}} \\
& g'(x,y) = & \textbf{let } f = nf(x)(fs(x)) & (* \; f \text{ is input-output function for } x \; *) \\
& & \textbf{and } h(z) = g(z,x) & (* \; h \text{ is input history of } x \; *) \\
& & \textbf{and } h' = f(h) & (* \; h' \text{ is output history of } x \; *) \\
& & \textbf{in } h'(y) & (* \; h'(y) \text{ represents messages from } x \text{ to } y \; *)
\end{array} \tag{1}
$$

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. For each failure scenario fs for which the user specified a fault-tolerance requirement, 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. This fixed-point calculation is essentially the same as in other stream-processing frameworks, such as [Kah74, Bro87, Bro90].

To illustrate the fixed-point calculation, we consider the reliable broadcast protocol in the same scenario as for Figure 1 except with server S_1 being faulty with possible send-omission failures. Let $M_{S,y}$ abbreviate $M_{S,y}^{C_1,0}$. 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 2.

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 2.

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. In practice, for analysis of systems with finite executions, the fixed-point calculation terminates. Analysis of systems with infinite executions is discussed in Section 5.3.

A fault-tolerance requirement is expressed in our framework as a predicate b on MFGs, together with a set of failure scenarios for which the system's behavior should satisfy the

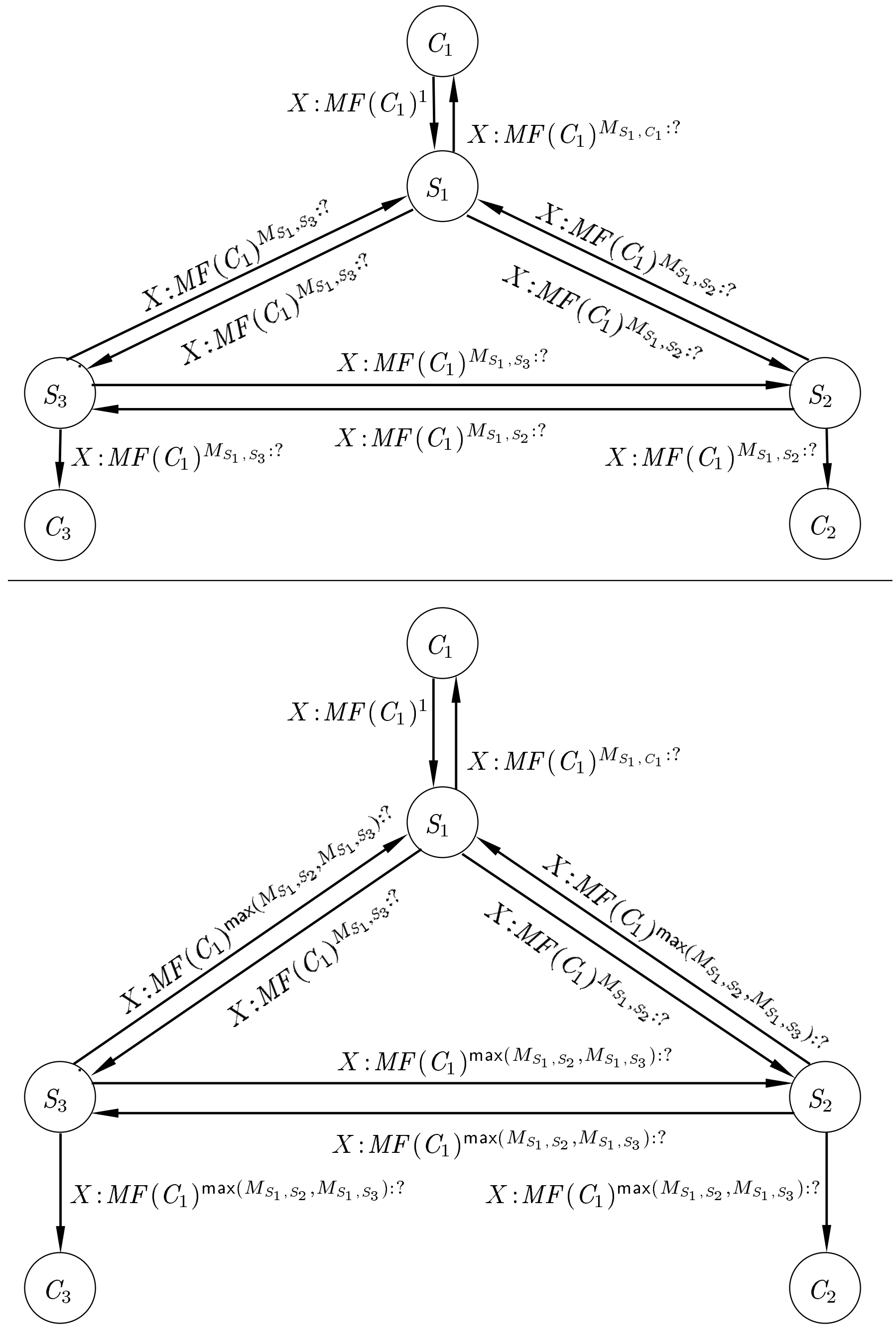

Figure 2: MFGs for reliable broadcast protocol when S_1 is faulty. Top: the MFG obtained after three applications of the *step* function. Bottom: the fixed-point.

predicate. For the reliable broadcast example, Agreement is expressed by the following predicate on MFGs: for each data value X broadcast by any client, either all channels from non-faulty servers to their clients contain an ms-atom with X as the data value and with abstract multiplicity 1, or all of those channels contain an ms-atom with X as the data value and with the same non-wildcard symbolic multiplicity. The system's behavior should satisfy this condition for failure scenarios in which some servers suffer send-omission failures and the network remains connected.

A system nf satisfies a fault-tolerance requirement b for failure scenario fs if the fixed-point of $step_{nf,fs}$ satisfies b. For the reliable broadcast example, one can see from the final MFG in Figure 2 that the correctness requirements 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 non-wildcard symbolic multiplicity appears in the ms-atoms with data value X on edges $\langle S_2, C_2\rangle$ and $\langle S_3, C_3\rangle$.

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 compare with Spin [HP95, Hol97]. Transitions of the same process are always dependent [HP95], due to control dependencies, so Spin explores all reachable interleavings of the inputs to each component. In contrast, our method does not consider interleavings of messages received by a component on different channels. This difference often causes the size of the state space explored by Spin (measured by the number of states) to be significantly larger than the size of the MFG (measured by the number of occurrences of constants and variables in ms-atoms). For the reliable broadcast example 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 for failure scenarios involving crash failures 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 the reliable broadcast protocol example with N servers in a fully connected network, 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. The arithmetic constraints in constrained queue-content decision diagrams [BH99] can express relationships between multiplicities, as our symbolic multiplicities do. The symbolic state-space exploration algorithm in [BH99] uses arithmetic constraints only to succinctly represent the effects of repeated execution of cycles in the control graph. For analysis of the reliable broadcast protocol, their algorithm does not provide the exponential savings achieved by our method, mainly because the protocol contains no control cycle that contains both send operations and receive operations. Similar comments apply to the symbolic state-space exploration algorithm based on queue-content decision diagrams in [BG99], which is a special case of the algorithm in [BH99].

We implemented the reliable broadcast example in Spin, using the largest possible atomic blocks and using a single input channel for each process; both of these choices reduce the number of explored states. For $N = 3$ with no failures, the MFG in Figure 1 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 MFG on the bottom of Figure 2, has size 100, while Spin stores 3778 states.

4 An Implementation

We implemented our analysis method in a prototype tool using CAML Light [Ler97]. The graphical interface is implemented using the Tk widget library [Ous94]. The tool provides a collection of CAML types and functions used to express input-output functions and compute fixed-points, libraries of pre-defined input-output functions, and a graphical interface to facilitate entry of systems and inspection of analysis results. Users familiar with CAML can define new input-output functions by writing them directly in CAML.

5 Related Work and Discussion

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, the family of reductions must introduce an abstract value representing each of these values. For example, in our analysis of the Oral Messages algorithm for Byzantine Agreement [LSP82], which uses majority voting, related values include X_1, X_2, X_3, and $\mathsf{maj}(X_1, X_2, X_3)$ [Sto97]. In effect, all relevant symbolic values must somehow be identified in advance, and an abstract value must be introduced for each of them. In contrast, with our method, the user need only determine for each component how constants and how its local variables are used to represent 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 expressions (such as $\mathsf{maj}(X_1, X_2, X_3)$) that contain 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

Our analysis does not construct global states or consider interleavings of messages sent (or received) by a component on different channels. Indeed, such inter-channel orderings cannot be represented directly in MFGs. In effect, our analysis suffers from the merge anomaly [Kel78, Bro88]. Specifically, an input-output function representing a component whose behavior depends on inter-channel orderings among its inputs cannot (in general) represent the component'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].

Many distributed systems can be analyzed precisely with our current framework. Examples include the three leader election algorithms for rings in [Lyn96, section 15.1], two-phase commit and three-phase commit [BHG87], the Timewheel atomic broadcast and group membership protocols [MFC97, MFC98] (real-time aspects can be modeled by introducing messages that represent passage of time, as in [BD92]), the Oral Messages and Signed Messages algorithms for Byzantine agreement [LSP82], and some protocols for Byzantine-fault-tolerant moving agents [Sch97]. Details of our analyses of the Oral Messages algorithm and a protocol for Byzantine-fault-tolerant moving agents appear in [Sto97].

Systems that cannot be analyzed precisely, because they depend on inter-channel orderings, include typical distributed spanning-tree algorithms and distributed mutual-exclusion algorithms.

5.3 Infinite Executions

Our framework is not suitable for analyzing systems with infinite behaviors, such as reliable communication protocols that may re-transmit a message infinitely often (note that our framework does not currently include fairness assumptions). 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 into finite subcomputations. Many fault-tolerant distributed systems perform (potentially infinite) sequences of independent or mostly independent finite subcomputations. For example, the reliable broadcast protocol described Section 2.1 can perform an arbitrary number of broadcasts, and each broadcast is handled independently. For such systems, it is sufficient to analyze computations containing only one (or a small number) of such subcomputations. This technique 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], as well as in our analysis of reliable broadcast, where we consider computations involving a single broadcast.

Many fault-tolerant 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.

Acknowledgments. We thank Y. Annie 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.

[BD92] Manfred Broy and Claus Dendorfer. Modelling operating system structures by timed stream processing functions. *Journal of Functional Programming*, 2:1–21, 1992.

[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.

[BH99] Ahmed Bouajjani and Peter Habermehl. Symbolic reachability analysis of fifo-channel systems with nonregular sets of configurations. *Theoretical Computer Science*, 221(1-2):211–250, 1999.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency control and recovery in database systems*. Addison Wesley, 1987.

[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.

[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.

[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.

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

[HP95] Gerard J. Holzmann and Doron Peled. An improvement in formal verification. In *Proc. 7th Int'l. Conference on Formal Description Techniques (FORTE '94)*, pages 197–211. Chapman & Hall, 1995.

[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.

[IT96] ITU-T. ITU-TS Recommendation Z.120: Message Sequence Chart (MSC), 1996.

[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:///caml.inria.fr/.

[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.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[MFC97] Shivakant Mishra, Christof Fetzer, and Flaviu Cristian. The timewheel asynchronous atomic broadcast protocol. In *Proceedings of the 1997 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1239–1248, June 1997.

[MFC98] Shivakant Mishra, Christof Fetzer, and Flaviu Cristian. The timewheel group membership protocol. In *Proceedings of the 3rd IEEE Workshop on Fault-tolerant Parallel and Distributed Systems*, April 1998.

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

[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.

[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. Springer-Verlag, September 1998.

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