Automated Analysis of Fault-Tolerance in Distributed Systems*

Scott D. Stoller
Dept. of Computer Science
Indiana University
Bloomington, IN 47405
stoller@cs.indiana.edu

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

November 18, 1996

1 Introduction

As computers are integrated into systems with stringent fault-tolerance requirements, there is a growing need for practical techniques to establish that these requirements are satisfied. Informal arguments do not supply the desired level of assurance for critical systems, while practitioners often lack the background needed to construct formal proofs. This paper describes an approach to automated analysis of fault-tolerance of distributed systems. The underlying principles are general, but specialized techniques are developed to deal efficiently with the types of systems and requirements that arise in this setting.

The first step of the verification process is to determine the *fault-tolerance requirements*, i.e., the conditions on the system's behavior that are required to hold for each *failure scenario*, i.e., each combination of failures of its components. For example, a typical fault-tolerance requirement for a distributed database system is that transactions are performed atomically (i.e., committed by all servers or none of them), despite crash failures. To achieve high assurance for safety-critical control systems, a more general failure mode, *Byzantine failure*, is usually considered. No assumptions are made about the possible behavior of a Byzantine-faulty component: it is considered arbitrarily non-deterministic. A typical requirement is that a control system operate normally despite, say, Byzantine failure of one component, and operate in a "safe" but possibly degraded fashion despite, say, Byzantine failure of one component and crash failure of another.

Second, these fault-tolerance requirements must be rigorously verified. One approach is to apply general-purpose proof-based verification techniques, typically with the support of a theorem-proving system [DB92, BVH94, ORSvH95]. This approach offers an attractive conceptual economy. However, people who design and validate fault-tolerant systems are generally not experts in mathematical logic or formal verification, so methods that require construction of large proofs (even with support from a theorem-proving system) have had a limited audience. Proof techniques designed specifically for verification of fault-tolerance (e.g., [CdR93, PJ94]) facilitate these proofs but still require considerable logical expertise of the user.

*This material is based on work supported in part by the NASA/ARPA grant NAG-2-893, AFOSR grant F49620-94-1-0198, and ARPA/RADC grant F30602-96-1-0317. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author and do not reflect the views of these agencies.

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

This paper explores a specialized approach to analysis of distributed systems, focusing on fault-tolerance properties. Our approach is not based on exhaustive state-space exploration. Instead, it is a novel hybrid of ideas from stream-processing (or data-flow) semantics of networks of processes [Kah74, Bro87, Bro90] and abstract interpretation of programs [AH87].

In stream-processing models, each component is represented by an *I/O function* describing its input/output behavior. The behavior of a system can be determined by a fixed-point calculation; this provides a clean algorithmic basis for the analysis. An I/O function encapsulates the implementation of a component, enabling a convenient separation of local and global analyses. Local analysis verifies independently for each component that the proposed I/O function represents it. Global analysis determines the system's behavior as a fixed-point.

The heart of our analysis is fixed-point calculations of graphs representing the computations of the system: each node corresponds to a component, and each edge is labeled with a "history" (stream) of messages sent from the source to the target. Exact computation of these graphs is generally infeasible, especially for the non-deterministic and non-terminating processes common in fault-tolerant distributed systems. To help make automated analysis feasible, our framework incorporates *abstraction* mechanisms that support flexible and powerful forms of approximation.[1] Traditionally, stream-processing models are used as mathematical semantics and incorporate no approximations.

Our analysis determines three kinds of information that together characterize the possible communication histories on an edge: *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 a form of abstract interpretation and, to more accurately track *relationships* between values, a form of symbolic computation. The latter is essential for analyzing systems that use replication and majority-voting to tolerate Byzantine failures, since the output of a voter depends on equalities between its inputs. Message orderings are approximated using partial (instead of total) orders. These approximation mechanisms together allow both compact representation of the highly non-deterministic behavior characteristic of severe failures and abstraction from irrelevant aspects of a system's failure-free behavior. The latter reflects a separation of concerns that is crucial for fault-tolerance analysis to be tractable.

Failures are often modeled as ordinary events that occur non-deterministically during a computation, but this makes it difficult to separate the effects of failures from other aspects of the system's behavior and hence to model the former more finely than the latter. One often wants to avoid case analysis corresponding to non-determinism in a system's failure-free behavior, while case analysis corresponding to different failure scenarios appears unavoidable in an automated approach.

[1] "Abstraction" is used here in the sense of "abstract interpretation", not in the sense of "abstract data types".

So, we parameterize systems by possible occurrences of failures, and analyze the system's behavior separately for different failure scenarios.

We propose a second model that further promotes this separation of concerns. In it, the effects of failures are represented as *perturbations* to the original outputs of a component. The fixed-point analysis propagates these perturbations to determine their effect on the subsequent execution of the system. The result of the analysis for a particular failure scenario is a graph describing both the system's original behavior and the global effects of those failures. This allows fault-tolerance requirements to be expressed as bounds on the acceptable perturbations to the system's behavior. For example, a typical fault-tolerance requirement is that the inputs to certain components are unchanged in certain failure scenarios. This condition is orthogonal to requirements on the system's failure-free behavior. Thus, these other requirements can be verified separately, using whatever methods are appropriate, while our method is used to analyze the fault-tolerance requirement, using coarse approximations of aspects of the system that are not directly relevant to its fault-tolerance. Explicit perturbations also allow direct expression of the sensitivity of a component to perturbations in its inputs; in some cases, representing this information in the abstract model is otherwise impossible.

2 Analyzing Failure-Free Systems

To build the reader's intuition, we briefly review a concrete stream-processing model before describing our framework. By *concrete*, we mean that no approximations are involved. Each component ("process") is represented by an I/O function that takes as argument the (history of) messages received by the component, and returns the (history of) messages sent by that component as a result of receiving those messages.

More formally, a system comprises a set of named components, with names from the set *Name*. Let *CVal* ("concrete values") be the set of values that can be transmitted in messages. A *concrete history* is an element of

$$CHist \triangleq Name \rightarrow Seq(CVal), \tag{1}$$

where $Seq(S)$ is the set of finite and infinite sequences of elements of a set S. When a history ch is regarded as the input to a component x, $ch(y)$ is the sequence of messages sent by y to x; when ch is regarded as the output of a component x, $ch(y)$ is the sequence of messages sent by x to y. The behavior of a system is represented by a *concrete run*, which is an element of

$$CRun \triangleq Name \rightarrow CHist. \tag{2}$$

For $cr \in CRun$, $cr(x)$ is the input history of component x in the run, i.e., $cr(x)(y)$ is the sequence of messages sent to x by y.

Following Kahn [Kah74], we consider first only *determinate* processes, i.e., processes that are (1) internally deterministic and (2) strict (i.e., at each instant, the process is willing to receive a message from at most one sender). Determinacy ensures that the input history of a process uniquely determines its output history. Thus, a determinate process corresponds to a monotonic and continuous (these are just sanity conditions) function in

$$DProcess \triangleq CHist \to CHist. \tag{3}$$

A system is represented by a function $np \in Name \to DProcess$ (np is mnemonic for "name → process"). The unique concrete run representing the behavior of a system np of determinate processes is the least fixed-point of $step(np) \in CRun \to CRun$, where

$$step(np)(cr) \triangleq (\lambda y\!:\!Name.\ (\lambda x\!:\!Name.\ np(x)(cr(x))(y))). \tag{4}$$

and the domain ordering on $CRun$ is the pointwise extension of the prefix ordering on sequences (of $CVal$'s). Informally, $step(np)(cr)$ represents the outcome of each component processing its inputs in the possibly-incomplete run cr and producing possibly-extended outputs. By a standard theorem [Gun92, chapter 4], this fixed-point exists and can be computed by starting with the empty run $\bot_{CRun} \triangleq (\lambda x : Name.\ (\lambda y : Name.\ \varepsilon))$, where ε is the empty sequence, and repeatedly applying $step(np)$ until a fixed-point is reached.

The restriction to determinate processes can be eliminated. A relatively straightforward approach, proposed by Broy [Bro87, Bro90], is (roughly) to represent a (possibly non-determinate) process as a *set* of determinate processes, each corresponding to one of the process's possible behaviors. The behavior of a non-determinate system is represented by the set of its possible concrete runs; roughly, for each possible choice of determinate processes from the sets representing the components, a concrete run is computed as a fixed-point, in the manner just described.

2.1 Abstract Model

At the abstract level, a (possibly non-determinate) system's behavior is approximated by a single "abstract" run. By analogy to definition (2) of $CRun$, a run is an element of

$$Run \triangleq Name \to Hist. \tag{5}$$

As before, for a run r and component x, the history $r(x)$ represents the inputs to x. The sequences of messages in concrete histories are approximated by partial orders of labels, in which each label approximates a set of messages. Thus, by analogy to definition (1) of $CHist$, we define

$$Hist \triangleq Name \to POSet(L), \tag{6}$$

where $POSet(S)$ is the set of strict partial orders over a set S, i.e., the set of pairs $\langle T, \prec \rangle$ where $T \subseteq S$ and $\prec$ is a strict partial order on T.

A run r can be interpreted as a labeled directed graph with nodes in $Name$ and with edge $\langle x, y \rangle$ labeled with the poset $r(y)(x)$, representing the messages sent from x to y. Each label in the poset approximates a set of messages. Labels are in

$$L \triangleq Mul \times Val \times Tag. \tag{7}$$

For a label $\langle mul, val, tag \rangle$, the multiplicity mul indicates how many messages may be represented by the label, and the value val describes the data sent in those messages. The tag is needed to allow multiple labels with the same value and multiplicity to appear on an edge. Multiplicities, values, and message orderings are discussed in more detail below.

The behavior of a process is approximated by an I/O function, which by analogy with (3) is in

$$IOF \triangleq \{f \in Hist \rightarrow Hist \mid tagUniform(f)\}, \tag{8}$$

where $tagUniform(f)$ asserts that renaming of tags in the input labels causes no change in the output labels except possibly renaming of tags. This requirement is sensible because tags do not appear in actual messages.

A system is represented by a function $nf \in Name \rightarrow IOF$ ("nf" is mnemonic for "name to I/O function"). A system's behavior is represented by the run $\mathrm{lfp}(step(nf))$, if it exists. In contrast to the concrete level, this fixed-point might not exist;[2] however, one can always search for a fixed-point by repeated application of $step(nf)$ starting from $\bot_{Run} \triangleq (\lambda x : Name.\ (\lambda y : Name.\ \langle\emptyset, \emptyset\rangle))$.

Values. What is the structure of *Val*? As in abstract interpretation, we introduce a set *AVal* of *abstract values*, each representing a set of concrete values. For example, consider messages that contain numerous header fields. If only the "From" field was relevant to the analysis, one could represent messages with abstract values of the form $MF(x)$ (mnemonic for "Message From x"), which represents all messages whose "From" field contains x.

Abstract values alone capture too little information about relationships between concrete values. For example, consider a system containing a majority voter. The voter's outputs depend on equality relationships among its inputs. If two inputs both have abstract value $\mathbf{N}$, denoting the natural numbers, there is no way to tell from this whether they are equal. To more accurately track relationships between values, we introduce a set *SVal* of *symbolic values*, which are expressions composed of constants and variables. A *constant* represents the same value in every concrete run of the system; for example, a constant *maj* might represent a majority function.

A *variable* represents values that may be different in different concrete runs of the 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 modeled explicitly; they also arise when a component's behavior is approximated. Each variable is associated with ("local to") a single component, whose behavior in a given concrete run determines the value of that variable. This allows independent proofs that each I/O function represents the behavior of the corresponding process.

For convenience, we include in *SVal* a special *wildcard* symbol "_", which can always represent any value. Finally, we allow a value to contain a set of possibilities; thus, we define

$$Val \triangleq Set(SVal \times AVal) \setminus \{\emptyset\}, \tag{9}$$

where $Set(S)$ is the powerset of a set S. Note that I/O functions incorporate a form of symbolic computation, since their inputs and outputs contain symbolic values.

[2] Roughly, the reason is that runs do not have canonical forms [Sto97].

Notation. Since abstract values are analogous to types, we sometimes write $\langle s, a\rangle \in SVal \times AVal$ as $s : a$. We often omit braces around singleton sets; for example, $\{\langle X, \mathbf{N}\rangle\} \in Val$ may be written $X : \mathbf{N}$. We sometimes elide the wildcard; thus, $\{\langle \_, \mathbf{N}\rangle\} \in Val$ may be written $\mathbf{N}$.

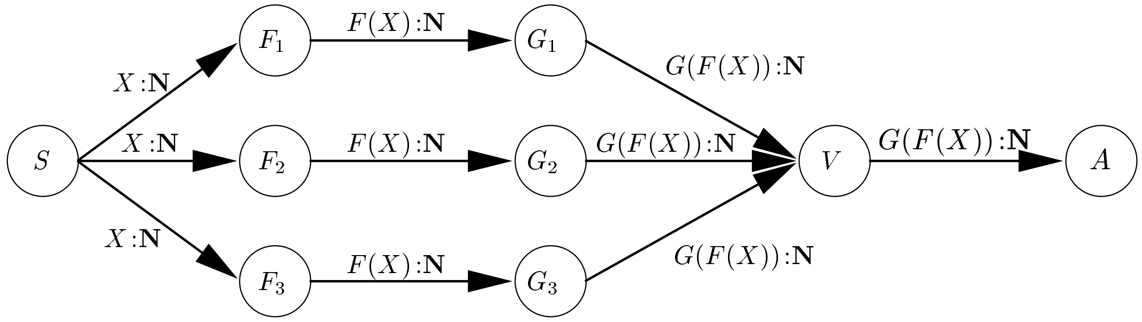

Figure 1: Run for majority-voter example, without failures.

Example. Consider a two-stage replicated pipeline. The system contains a source S, which sends a value to three components F_1, F_2, F_3, which each apply a function represented by the constant F to their input and send the result to the next stage in the pipeline. The components G_1, G_2, G_3 in the next stage each apply a function represented by the constant G to their input and send the result to a voter. The 3-way voter V waits for an input from each G_i, applies a 3-way majority function, represented by the constant maj, to those inputs, and sends the result to an actuator A. More precisely, maj represents any function of 3 arguments that, when any two of its arguments are equal, returns that repeated value.

A run representing the behavior of this system appears in Figure 1. Here, X is a local variable of the source. The run is obtained as a fixed-point from I/O functions for the components. Due to space limitations, we only discuss the I/O function for the voter. Briefly, if the voter receives 3 inputs containing symbolic values s_1, s_2, s_3, then in the general case, the voter's output contains the symbolic value $maj(s_1, s_2, s_3)$. If, in addition, two inputs contain the same symbolic value s (other than the wildcard), then a step of symbolic simplification can be done, yielding the symbolic value s. For example, $maj(X, X, Y)$ simplifies to X.

Multiplicities. Multiplicities (i.e., numbers of messages) also need to be approximated. Uncertainty in the number of messages sent during a computation may stem from various sources, including non-determinism of components (especially faulty components), non-determinism of message arrival order, approximation of values, and approximation of "loops" (i.e., cycles of messages). For example, a component subject to omission failures might emit outputs with a multiplicity of zero or one, while a component subject to Byzantine failures might emit outputs with an arbitrary multiplicity. Thinking of multiplicities as natural numbers suggests representing them in the same way as data values. Thus, we define

$$Mul \triangleq Set(SVal \times AMul) \setminus \{\emptyset\}, \tag{10}$$

where the set $AMul \subseteq AVal$ of *abstract multiplicities* contains abstract values whose meanings are subsets of the natural numbers, excluding the subsets $\emptyset$ and $\{0\}$.

Abstract multiplicities are analogous to the superscripts in regular expressions. To promote the resemblance, we assume $AVal$ contains: 1, denoting $\{1\}$; ?, denoting $\{0,1\}$; and $*$, denoting all natural numbers. The notational conventions for values apply to multiplicities as well. Furthermore, we sometimes write the label $\langle mul, val, 0\rangle$ as val^{mul}, and, if the multiplicity mul is 1, we elide it, as in Figure 1.

Message Ordering. Message ordering plays little role in the examples in this paper, so we discuss it very briefly. The partial ordering on the labels in a history approximates the orderings between the messages represented by those labels: if label ℓ_1 is ordered before ℓ_2, then all messages represented by ℓ_1 were sent before (and, since message delivery is assumed for convenience to be FIFO, are received before) all messages represented by ℓ_2. Our use of partial orders to represent message ordering supports similar benefits to partial-order methods in model-checking [WG93]. We are exploring extensions to the framework to reflect orderings between messages sent by different components.

3 Analyzing Systems with Failures

Since each component's behavior depends on what failures it suffers, we parameterize each component (i.e., each I/O function) by its possible failures. Thus, I/O functions are now elements of $IOF_F \triangleq Fail \rightharpoonup IOF$, where $Fail$ is the set of all possible failures and the one-hooked arrow indicates partial functions. For $f \in IOF_F$, $dom(f)$ is the set of failures that the component might suffer, and for each $fail \in dom(f)$, $f(fail)$ describes the component's behavior when failure $fail$ occurs. A failure scenario is a function in $FS \triangleq Name \to Fail$ that maps each component to one of its possible failures. By convention, $OK \in Fail$ represents absence of failure.

A fault-tolerance requirement is formalized as a set $\{\langle S_1, b_1\rangle, \langle S_2, b_2\rangle, \ldots\}$, where S_i is a set of failure scenarios, and b_i is a predicate on $FS \times Run$. For each i and each $fs \in S_i$, $b_i(fs, r)$ should hold for the run r representing the system's behavior in failure scenario fs. For the above majority-voter example, consider a "value failure", represented by $valFail \in Fail$, that causes a component to output incorrect values. Informally, the fault-tolerance requirement is that the inputs to the actuator are unchanged (compared to the failure-free run) in failure scenarios in which at most 1 of the components F_i or G_i fail. Expressing "unchanged" (compared to a given run) in the current framework is a bit subtle. Let r_0 denote the failure-free run, i.e., the run computed for failure scenario $(\lambda x\!:\!Name.\ OK)$. A first attempt is simply to require that the inputs to the actuator be the same: $b(fs, r)$ is $r(A) = r_0(A)$. However, this is too weak: it does not ensure that the variable X represents the same concrete value in the two runs. Since X is local to the source, this follows if the source's outputs are unchanged, which follows if the source itself does not fail and (recursively) the source's inputs are unchanged. In general, we can formulate a recursive predicate on runs that ensures that a given component's inputs are unchanged. In this example, the source has no inputs, so $b(fs, r)$ is: $r(A) = r_0(A) \wedge fs(S) = OK \wedge r(S) = r_0(S)$.

Example. Consider, for example, failure of F_1. F_1's arbitrary output could be represented by the value $Y_1 : \mathbf{N}$, where Y_1 is a local variable of F_1, or by $\_ : \mathbf{N}$. When the voter receives one changed input, its output to the actuator is unchanged, so the above fault-tolerance requirement is satisfied for this failure scenario. A similar analysis for each of the 6 possible failure scenarios shows that the system satisfies its fault-tolerance requirement.[3]

3.1 Perturbations

The above style of expressing that part of a system's behavior is unchanged is awkward and unnecessarily restrictive. For example, consider a failure mode of F_1 in which it sends arbitrary values to the source as well as G_1. Assuming the source is designed to ignore these unexpected inputs, the system tolerates this failure. However, there is no way to express in the above framework that these unexpected inputs to the source do not affect its outputs, i.e., that they do not change the output value represented by X.

The above framework also has limited power to express non-trivial relationships between (concrete) values in the original and perturbed computations. For example, consider a system in which failures may perturb the readings of replicated sensors by at most ε, causing the inputs and therefore also the outputs of a controller to change by at most ε'. The relationship "changed by at most ε" can't be expressed in the above framework.

The root of these limitations is the inability to express correlations between a component's original and perturbed behaviors. This problem arises also in concrete models; it is not an artifact of the abstraction mechanisms. We overcome these limitations by making such correlations explicit. Instead of describing the faulty behavior separately, we describe it in terms of *perturbations* to the original behavior. At the abstract level, we augment labels to have the form

$$L_{chng} = \mathit{Mul} \times \mathit{Val} \times \Delta \mathit{Mul} \times \Delta \mathit{Val} \times \mathit{Tag} \tag{11}$$

where Mul and Val reflect the failure-free (or "original") behavior, and $\Delta\mathit{Mul}$ and $\Delta\mathit{Val}$ are the perturbations to the multiplicity and value, respectively. In order to track relationships between perturbations, we take $\Delta\mathit{Mul}$ and $\Delta\mathit{Val}$ to have the same structure as values, i.e., to contain symbolic values from SVal paired with new kinds of abstract values $\Delta\mathit{AMul}$ and $\Delta\mathit{AVal}$, respectively. Semantically, changes are binary relations: thus, elements of $\Delta\mathit{AMul}$ and $\Delta\mathit{AVal}$ denote binary relations over $\mathbf{N}$ and CVal, respectively.

The perturbed behavior may also involve messages with no analogue in the original behavior—for example, messages sent between components that didn't communicate in the original computation. These are represented by a second kind of label:

$$L_{new} = \mathit{Mul} \times \mathit{Val} \times \mathit{Tag}. \tag{12}$$

Now, labels are elements of $L_{FC} = L_{chng} \cup L_{new}$.

Notation. We use the same notational conventions for $\Delta\mathit{Val}$ and $\Delta\mathit{Mul}$ as for Val and Mul. We sometimes write a label $\langle mul, val, 0\rangle \in L_{new}$ as val^{mul}. Similarly, we sometimes write a label

[3] It suffices to analyze 2 failure scenarios, if a symmetry argument is used.

$\langle val, mul, \delta mul, \delta val, 0\rangle \in L_{chng}$ as $val^{mul}[\delta val^{\delta mul}]$. We sometimes elide a change of id, where id is the identity relation; for example, the label $\langle val, mul, id, id, 0\rangle \in L_{chng}$ may be written $val^{mul}[]$. The empty brackets are retained to distinguish this from the shorthand for labels in L_{new}.

I/O functions have the same type as before, but over the new set of labels. I/O functions now describe how each component propagates its original inputs and how it propagates changes in its inputs. For an I/O function f, if one ignores the perturbations and "new" labels (i.e., labels in L_{new}) in the input and output, $f(\mathit{fail})$ describes the original behavior of the component; the perturbations and new labels in the output reflect the effects of the failure *fail* and the effects of perturbations and new labels in the inputs. We have given semantics for the abstract models with and without explicit perturbations by relating them to variants of Broy's concrete model [Bro87, Bro90].

Example. Consider again the system of Figure 1. The fault-tolerance requirement is that the change to the actuator's original inputs is the identity, and that the actuator has no new inputs. Figure 2 shows the result of the fixed-point analysis for the failure scenario in which F_1 suffers a value failure. The failure introduces perturbations in F_1's output. G_1 propagates the perturbation from its input to its output. Thus, the analysis shows that the system satisfies its fault-tolerance requirement in this failure scenario.

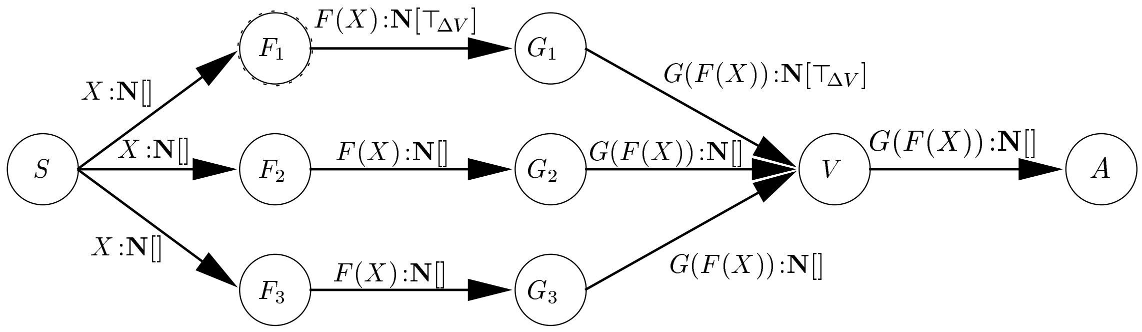

Figure 2: Run for majority-voter example, in the failure scenario in which F_1 suffers a value failure *valFail*, analyzed using explicit perturbations. When a single label ℓ appears on an edge, it abbreviates the poset $\langle\{\ell\}, \emptyset\rangle$. Faulty components are distinguished by dots on their circumference. $\top_{\Delta V} \in \Delta AVal$ denotes an arbitrary change, i.e., the full relation on $CVal$.

Now consider instead Byzantine failures, in which a faulty component sends arbitrarily many arbitrary values to every process with which it can communicate. This behavior can be compactly represented using the label $\top^*_{AVal}$, where $\top_{AVal} \in AVal$ represents all concrete values. Assume F_1 can communicate directly with S and G_1 and that the source ignores "unexpected" inputs. The run for this failure scenario is shown in Figure 3. Thus, the analysis shows that the system satisfies its fault-tolerance requirement in this failure scenario.

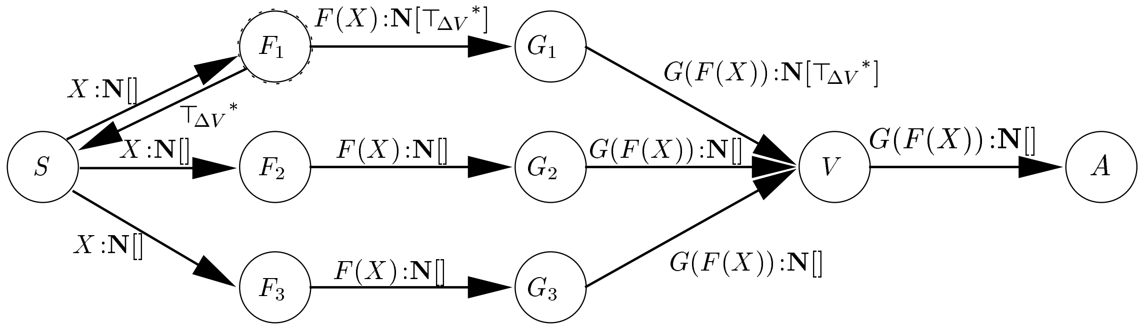

Figure 3: Run for majority-voter example, in the failure scenario in which F_1 suffers a Byzantine failure, analyzed using explicit perturbations. The notation is the same as in Figure 2.

4 Examples

4.1 Byzantine Agreement

A seminal paper by Lamport, Shostak, and Pease defines the problem of Byzantine Agreement and presents two solutions [LSP82]. Both can be analyzed in our framework. The Oral Messages algorithm is essentially a recursive application of majority voting, so the analysis is similar in style to the simple example in Section 3. The Signed Messages algorithm is more efficient but requires digital signatures, which can be modeled in our framework [Sto97]. Both algorithms assume synchronous communication. Although our framework embodies asynchronous communication, synchronous communication can be encoded by sending special values to represent passage of time [Bro90, BD92].

4.2 Reliable Broadcast

The power of symbolic multiplicities to track relationships between multiplicities is useful in analysis of atomicity properties, which are typically of the form: "All non-faulty components do *action*, or none of them do." Thus, atomicity properties correlate the multiplicities of actions at different sites. We use a reliable broadcast protocol to illustrate our treatment of crashes and atomicity.

The system comprises clients $C_1, \ldots, C_n$ with corresponding servers $S_1, \ldots, S_n$. For brevity, we briefly describe the protocol but omit the I/O functions. A client C_i broadcasts a message by sending it to its server S_i. When a server receives a message, it checks whether it has received that message before. If so, it ignores the message; if not, it sends the message to all its neighbors (including its client). We use the abstract value $MF(C)$ to denote messages broadcast by client C.

For example, consider the system with $n = 3$ and with each server having the other two servers as neighbors. Suppose C_1 broadcasts a message $X : MF(C_1)$. The run in Figure 4 represents the failure-free behavior of this system.

Now consider the effects of failures. For brevity, we consider here only the requirement of *agreement*: if a client of a non-faulty server delivers a message m, then all clients of non-faulty servers eventually deliver m [HT94, section 3].

One way to analyze systems subject to crash failures is to analyze separately the behavior

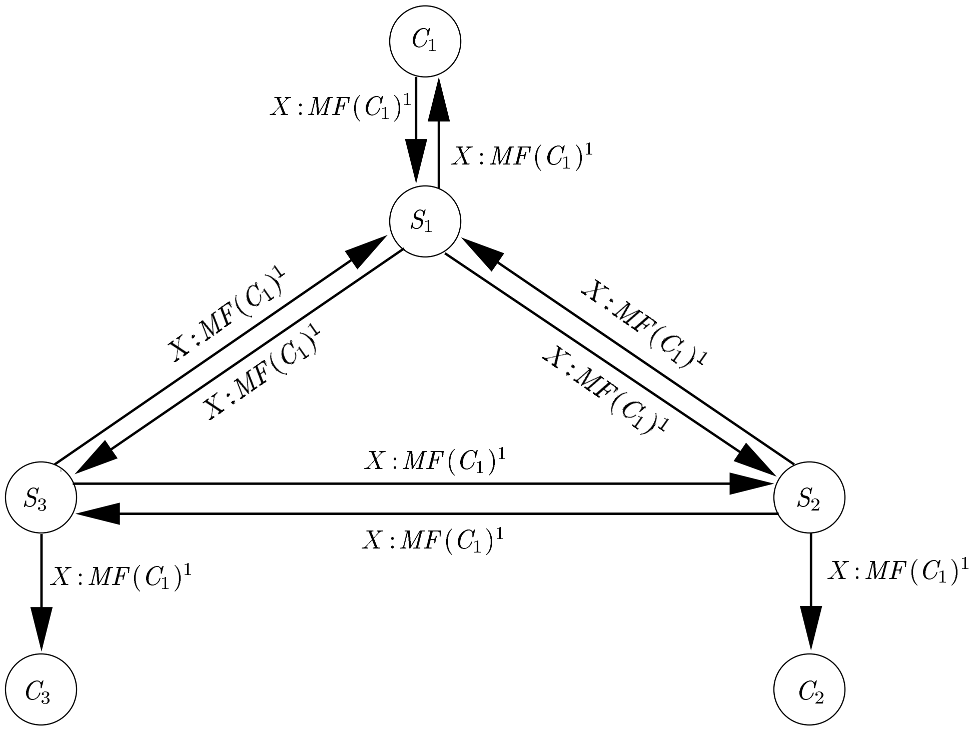

Figure 4: Failure-free behavior of the reliable broadcast protocol.

resulting from crashes that occur at different times during the execution; in our framework, this could be done by representing crashes at different logical times with different elements of *Fail*. A more efficient approach is to introduce $crash \in Fail$, which indicates only that a component crashes at some unspecified time during execution, in which case each of the component's outputs has a possibility of not occurring.[4] For example, consider a failure scenario in which S_1 crashes. Each of its outputs gets multiplicity ? instead of 1, and this uncertainty is propagated through the rest of the computation, resulting in a run like the one in Figure 4 but with every multiplicity replaced with ?. We cannot conclude from this run that agreement holds; too much information about correlations between multiplicities has been lost.

To express those correlations using symbolic multiplicities, we modify the I/O function for a server to use variables with the following interpretation: the value (zero or one) of variable $c.i.x.y$ indicates whether server x crashes before it relays to component y the i'th message[5] broadcast by client c. Note that $c.i.x.y$ is local to server x. If a server receives the same message twice, with multiplicities X :? and Y :?, then it relays that message with multiplicity $\max(X, Y)$:?, where the constant max has the obvious meaning. The run obtained using these modified I/O functions appears in Figure 5. Since the symbolic multiplicities of the inputs to C_2 and C_3 are the same, agreement is satisfied.

[4] More precisely, a *suffix* of its outputs might not occur; reflecting this requires more care but is necessary to verify FIFO delivery.

[5] We use 0-based indexing, so the first message corresponds to $i = 0$.

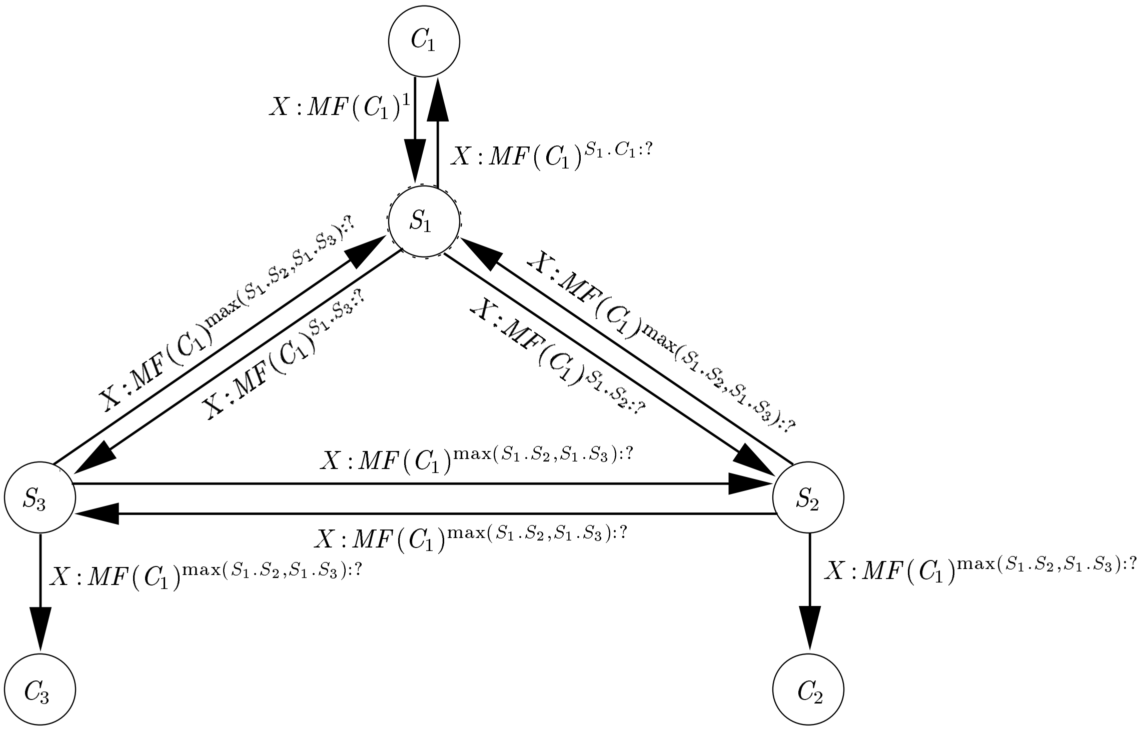

Figure 5: Behavior of the reliable broadcast protocol when S_1 crashes. In the figure, $x.y$ abbreviates $C_1.0.x.y$.

5 Discussion and Related Work

Abstractions. Abstractions (i.e., approximations) play an important role in our work. Clarke and Long studied the use of the abstractions in conjunction with temporal-logic model-checking [CGL94]. Their notion of abstraction corresponds roughly to abstract interpretation and to our abstract values, though in their state-based approach, multiplicities are not explicit, so abstractions are used only for (data) values. They also propose so-called *symbolic abstractions*, which are just abbreviations for finite families of (non-symbolic) abstractions. Our symbolic values are closer to the technique they sketch in the last paragraph of their paper for dealing with infinite-state systems. An attractive feature of their work is that abstractions are specified as homomorphisms and applied to programs automatically. We plan to look at mechanized support for applying abstractions in our framework.

In Kurshan's automata-based verification methodology, approximations are embodied in *reductions* between verifications [Kur89, Kur92, Kur94]. Relationships between concrete values can be captured using parameterized families of reductions, reminiscent of Clarke and Long's "symbolic abstractions". For example, a bounded-length queue can be proven not to drop items using a family of reductions that collapse a set of (concrete) data values to 2 "abstract" data values: the one being focused on, specified as a parameter, and "everything else" [Kur92, Appendix D] (the parameterization is not explicit in Kurshan's presentation). For problems involving related values (e.g., X and $F(X)$), the reductions must introduce an "abstract" data value representing each such value. In effect, one must determine in advance all relevant symbolic values and introduce an "abstract" data value for each.

Explicit Perturbations. McDermid *et al.*'s work on validation of fault-tolerance shares with our work the idea of characterizing each component by how it generates and propagates failures [FM93, FMNP94, MNPF95]. Their work emphasizes simplicity over generality by incorporating a somewhat restricted representation of changes.

Our explicit perturbations are related to error analysis for numerical computations and to incremental computation [RR93, Liu96], which studies how changes to the input of a computation are propagated to its output, so that new outputs can be computed efficiently by updating the old output. In all three areas, the goal is a separation of concerns, though these other techniques do not address the abstractions needed for fault-tolerance analysis.

Completeness. Asynchronous distributed systems with communication channels of unbounded capacity are infinite-state and therefore cannot be handled with finite-state methods. Such systems can be modeled in our framework, though their verification is, in general, undecidable. More generally, since we do not restrict attention to finite-state systems, we are forced to use conservative approximations in the analysis, introducing the possibility of false negatives.

Termination. Since our analysis constructs histories, approximations of values and multiplicities are generally needed for termination. Currently, the framework *supports* but does not *enforce* the approximations needed for termination; for maximum flexibility, the choice of when and how to approximate is left to the I/O functions. This is feasible because the entire input history is available to the I/O function. For example, an I/O function might produce several outputs, each with multiplicity 1, up to some threshold (possibly dependent on the particular inputs), then, if it receives more inputs, approximate by using a multiplicity $*$ in its output. In practice, it seems to be relatively easy to determine whether specific abstractions will provide termination.

Many fault-tolerant distributed systems are reactive systems that process unbounded streams of requests. Termination of the analysis depends on factoring the system's behavior into sub-computations that can be analyzed separately. For example, the reliable broadcast protocol described Section 4.2 can perform an arbitrary number of broadcasts, while the above analysis considers an execution involving only one broadcast. The protocol can be verified more rigorously using a standard approach (see, for example, the analysis of the arithmetic pipeline in [CGL94], or the analysis of the 4-element queue in [Kur92]).

The problem of dealing with infinite runs also arises in verification of control systems. An approach to verification of aircraft control systems is described in [DBC91, Rus93]. Although the work described there uses a theorem-prover, the same ideas can be used in our framework to automatically check whether a given control system tolerates a specified rate of failures.

Future Work. Success of this approach for analyzing particular classes of systems depends on the ability to find abstractions that are precise enough to avoid false negatives and coarse enough to be tractable, and on mechanized support for applying those abstractions to programs. We have implemented the analysis in a prototype tool, *CRAFT*. We plan to test the approach and tool by applying them to more problems, e.g., efficient Byzantine-agreement algorithms, algorithms for the certified write-all problem [KMS95, BKRS96], secure protocols for group membership and reliable broadcast [Rei96, MR96], and cryptographic protocols for fault-tolerant moving agents [MvRSS96].

References

[AH87] Samson Abramsky and Chris Hankin. An introduction to abstract interpretation. In Samson Abramsky and Chris Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 1. Ellis-Horwood, 1987.

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

[BKRS96] J. F. Buss, P.C. Kanellakis, P. L. Ragde, and A. Shvartsman. Parallel algorithms with processor failures and delays. *Journal of Algorithms*, 1996.

[Bro87] Manfred Broy. Semantics of finite and infinite networks of concurrent communicating agents. *Distributed Computing*, 2(1):13–31, 1987.

[Bro90] Manfred Broy. Functional specification of time sensitive communicating systems. In *Proceedings of the REX Workshop on Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*, pages 153–179. Springer-Verlag, 1990.

[BVH94] Ricky W. Butler, Ben L. Di Vito, and C. Michael Holloway. Formal design and verification of a reliable computing platform for real-time control (phase 3 results). NASA Technical Memorandum 109140, NASA Langley Research Center, 1994.

[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 Rajeev Alur and Tom Henzinger, editors, *Computer-Aided Verification (CAV '96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 394–397. Springer-Verlag, 1996.

[DB92] Ben L. Di Vito and Ricky W. Butler. Provable transient recovery for frame-based, fault-tolerant computing systems. In *Proc. 13th IEEE Real-Time Systems Symposium*, pages 275–278. IEEE, 1992.

[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.

[FM93] P. Fenelon and J. A. McDermid. An integrated toolset for software safety analysis. *Journal of Systems and Software*, 21(3):279–290, 1993.

[FMNP94] P. Fenelon, J. A. McDermid, M. Nicholson, and D. J. Pumfrey. Towards integrated safety analysis and design. *ACM Computing Reviews*, 2(1):21–32, 1994.

[Gun92] Carl A. Gunter. *Semantics of Programming Languages.* MIT Press, 1992.

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

[HT94] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR 94-1425, Cornell University, Department of Computer Science, 1994.

[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.

[KMS95] P.C. Kanellakis, D. Michailidis, and A. Shvartsman. Controlling memory access in efficient fault-tolerant parallel algorithms. *Nordic Journal of Computing*, 1995.

[Kur89] R. P. Kurshan. Analysis of discrete event coordination. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Proc. of the REX Workshop on Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*, pages 414–453. Springer-Verlag, 1989.

[Kur92] Robert P. Kurshan. Automata-theoretic verification of coordinating processes. Available from ftp://ftp.research.att.com/dist/COSPAN/notes.PS.Z, August 1992.

[Kur94] Robert P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach.* Princeton Univesity Press, 1994.

[Liu96] Yanhong A. Liu. *Incremental Computation: A Semantics-Based Systematic Transformational Approach.* PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, January 1996.

[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.

[MNPF95] J. A. McDermid, M. Nicholson, D. J. Pumfrey, and P. Fenelon. Experience with the application of HAZOP to computer-based systems. In *Proc. 10th Annual Conference on Computer Assurance*, pages 37–48, 1995.

[MR96] Dalia Malki and Michael Reiter. A high-throughput secure reliable multicast protocol. In *Proc. of the 9th IEEE Computer Security Foundations Workshop*, page 9=17. IEEE Computer Society Press, 1996.

[MvRSS96] Yaron Minsky, Robbert van Renesse, Fred B. Schneider, and Scott Stoller. Cryptographic support for fault-tolerant distributed computing. In *Proc. Seventh ACM SIGOPS European Workshop*, pages 109–114. ACM Press, September 1996.

[ORSvH95] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of pvs. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[PJ94] Doron Peled and Mathai Joseph. A compositional framework for fault-tolerance by specification transformation. *Theoretical Computer Science*, 128(1-2):99–125, 1994.

[Rei96] Michael K. Reiter. A secure group membership protocol. *IEEE Transactions on Software Engineering*, 22(1):31–42, 1996.

[RR93] G. Ramalingam and Thomas Reps. A categorized bibliography on incremental computation. In *Conference Record of the 20th Annual ACM Symposium on POPL*, pages 502–510, Charleston, South Carolina, January 1993.

[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 Academic Publishers, 1993. Also appeared in SRI International Computer Science Laboratory Technical Report SRI-CSL-93-04.

[Sto97] Scott D. Stoller. *Tools for Evaluating Fault-Tolerance in Systems.* PhD thesis, Cornell University, Department of Computer Science, forthcoming in 1997.

[WG93] Pierre Wolper and Patrice Godefroid. Partial-order methods for temporal verification. In *CONCUR '93: 4th International Conference on Concurrency Theory*, pages 233–246, 1993.