Specifications for Fault Tolerance: A Comedy of Failures

Felix C. Gärtner
Darmstadt University of Technology

Department of Computer Science
Technical Report TUD-BS-1998-03

October 21, 1998

Abstract

A substantial difficulty in rigorously reasoning about fault tolerant distributed algorithms is the necessity to formally describe faulty behavior. In this paper, we present a unified and formal approach to specify such behavior. It is based on the observation that faulty behavior can be regarded as a special form of (programmable) system behavior. Consequently, a failure model is defined to be a program transformation which can be used to evaluate the correctness properties of fault tolerant algorithms. We re-formulate several failure models which are pervasive in the literature in terms of our approach and show some interesting relations between them. In order to show the feasibility of this approach, we apply our methodology to the problem of reliable broadcast.

Categories and Subject Descriptors: C.4 [**Performance of Systems**]: Fault tolerance; modeling techniques; F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: mechanical verification; specification techniques

General Terms: Algorithms, Design, Reliability, Verification

Additional Key Words and Phrases: Byzantine faults, case study, crash faults, fail-stop, failure model, fault modeling, omission faults, reliable broadcast

This work was supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the "Graduiertenkolleg ISIA" at Darmstadt University of Technology.
Address: Department of Computer Science, Alexanderstr. 10, D-64283 Darmstadt, Germany, Email: `fcg@acm.org`

1. INTRODUCTION

Despite many research efforts, the formal verification of fault tolerant distributed systems is still a tedious undertaking. While hand-written correctness arguments are commonplace and sometimes easy to obtain, they often lack the necessary rigor and are error prone. The alternative to a hand-written proof is to mechanically develop or check correctness proofs using systems like PVS [Owre et al. 1996]. However, while these systems maintain the required rigor, the need to formally deal with *every* relevant aspect of the problem and the underlying system model has restrained this method from yet becoming a common technique. These problems are especially obvious in dependable systems like flight control software where correctness proofs are obviously extremely desirable but the possible occurence of faults influences the specification and the underlying system model in intricate and surprising ways.

A substantial difficulty in rigorously reasoning about fault tolerant distributed algorithms is the necessity to formally describe faulty behavior. In this paper, we present a unified and formal approach to specify such behavior. It is based on the observation that faulty behavior can be regarded as just some other kind of (programmable) system behavior. For example, a system crash can be modeled by adding an additional boolean variable *up* to the state of the system and inhibiting all program actions if $\neg up$ holds. The state transition from *up* to $\neg up$ can be viewed as a crash fault. Consequently, a failure model like *crash* [Hadzilacos and Toueg 1994] can be defined to be a program transformation. This opens the field of reasoning about correctness properties of fault tolerant algorithms to the large body of standard methods for mechanical correctness evaluation. We re-formulate several failure models which are pervasive in the literature in terms of our approach and show some interesting relations between them. As an example, we apply our methodology to the problem of reliable broadcast and thus show that our method is feasible and can faciliate fully automatic reasoning about any form of fault tolerant algorithms. The method of specifying program properties and reasoning about them is based on the UNITY theory by Chandy and Misra [1988], which we assume the reader is fairly familiar with (we will briefly recall those parts of their theory which are necessary for the understanding of this paper in Section 2).

This paper is structured as follows: In Section 2, we define our system model and some important terms used in this paper. Section 3 then presents formalizations of well-known failure models from the literature. As an example we use the problem of *reliable broadcast* in point-to-point networks (Section 4): first, we give a specification of the problem and an algorithm for an ideal, failure free environment; subsequently, we transform the algorithm into one which is exposed to faulty behavior and use the presented methodology to mechanically prove the correctness of the protocol under the given failure model. In Section 5, we briefly review the related work of other authors that we build upon. Section 6 contains some conclusions.

2. PROGRAMS, FAULTS, SPECIFICATIONS, AND FAULT TOLERANCE

In this section we recall the standard definitions of programs [Chandy and Misra 1988], faults [Arora and Gouda 1993], specifications [Alpern and Schneider 1985], and fault tolerance [Arora and Kulkarni 1998a].

2.1 Programs

We model a distributed system as a finite set of processes $\{p_1, \dots, p_n\}$ which communicate over reliable unidirectional channels. A process runs a *local algorithm* which consists of a set of *local variables* and a finite set of actions. The value of these variables form the *local state* of the process. An *action* has a unique name, and is of the form:

$$\langle \text{name} \rangle :: \langle \text{guard} \rangle \longrightarrow \langle \text{command} \rangle$$

The guard is a boolean expression over the local state and the command is an atomic and instantaneous update of zero or more local variables. An action is *enabled* if its guard evaluates to *true*. We write the command **skip** for an assignment to zero variables.

The channel between processes p_i and p_j is modeled by a shared variable $C_{i,j}$ that has some domain over a fixed message alphabet M.[1] To send a message $m \in M$ from p_i to p_j, process p_i assigns $C_{i,j} := \mathbf{snd}(C_{i,j}, m)$. Process p_j (on the other end of the channel) can test via the predicate $\mathbf{arr}(C_{i,j}, m)$ whether message m has arrived and is ready to be received. To actually receive this message, p_j invokes $b, C_{i,j} := \mathbf{rcv}(C_{i,j}, m), \mathbf{rmv}(C_{i,j}, m)$. The function **rcv** assigns m to a local variable b, wheras **rmv** removes m from the channel (**rcv** and **rmv** may only be invoked together). The functions **snd**, **arr**, **rcv** and **rmv** are such that the *asynchrony condition* [Chandy and Misra 1988] for all channel variables holds. This means that communication is implementable using asynchronous message passing. We assume that the directed communication graph over processes defined by the channels is at least connected, i.e., there exists a path between every pair of processes.

For convenience, we introduce a notation to express that one action is a restriction of another. We write

$$\langle \text{name}' \rangle :: \langle \text{guard}' \rangle \wedge \langle \text{name} \rangle$$

to define a new action $\langle \text{name}' \rangle$ whose guard is obtained by adding the guard $\langle \text{guard}' \rangle$ as a conjunct to the original guard and leaving the command unchanged.

A *distributed algorithm* (or simply *program* for short) is the union of a finite set of local algorithms. The *global state* (or *configuration*) of a program consists of all the local states together with the state of the communication channels.

An *execution* of a program (also called a *distributed computation*) is an infinite sequence of global states $s_0, s_1, \dots$ where for every s_i, s_{i+1} is obtained by executing a single enabled action of a local algorithm in state s_i. Finite executions are technically turned into infinite executions by infinitely repeating the final state. In places where we explicitly refer to finite executions we will call them *partial executions.* We assume a weak notion of fairness, meaning that a local action which is continuously enabled along the states of an execution is eventually executed.

2.2 Faults

The formal definition of faults is based on the observation that a processes' faulty behavior ist just another kind of (programmable) behavior. Thus a fault can be modeled as an unwanted but nevertheless possible state transition. By using addi-

[1]For simplicity, we do not count a channel as being part of the local state of a process.

tional (virtual) variables all types of faulty behavior can be modeled. So formally, a *fault* is just the occurence of an additional virtual program action.

On the other hand, a failure model describes the manner in which components of a system may exhibit faulty behavior. Because we model faults as program actions, a failure model can be seen as adding behavior to an existing program. Formally, a *failure model* is a program transformation, i.e., a function that maps a program A to a program A'. Program A is the original program, which by itself runs in an ideal fault free environment; A' is program A that may be subject to faults. Note that A' will never be explicitly implemented; the transformation process is just a means to be able to reason about programs which are subject to faults. Concrete examples of failure models will appear in Section 3.

2.3 Specifications

A *property* of a program is a set of executions. A program always defines in itself a property which is the set of executions which are possible starting from an initial configuration. A specific property p is said to *hold* for a program, if the set of executions defined by the program is a subset of p.

There are two main types of properties called *safety* and *liveness*. A *safety property* informally states that a program never leaves a fixed set of "safe" states. Formally, it is a set of executions that meet the following condition: for every execution σ not in that set, there exists a prefix α of σ, such that for all executions β, the concatenation of α and β is not in that set. Thus a safety property rules out a set of "unwanted" computation prefixes.

On the other hand, a *liveness property* informally states that any partial execution is not lost (i.e., there is still hope). Formally, it is a set of executions that meet the following condition: for every partial execution α in that set, there exists an execution β such that the concatenation of α and β is in that set. Thus, a liveness property rules out a set of "unwanted" computation suffixes. Note that both safety and liveness are static properties of an algorithm, i.e., at any time either they hold or they are violated.

A *problem specification* consists of a safety and a liveness property. A program is said to be *correct* regarding a problem specification, iff both the safety and the liveness property hold for that program. In the UNITY logic [Chandy and Misra 1988], safety properties are usually proved by an invariance argument. Liveness properties are specified using the operators *ensures* and *leads-to* (denoted $\mapsto$). Let p and q be two state predicates. Then p *ensures* q means that if predicate p is true at some point in the execution, p remains true as long as q is false, and eventually q becomes true. The predicate $p \mapsto q$ states that if p is true at some point in the execution, q will eventually become true (no matter if q remains true). (For a more detailed discussion of these operators and the associated proof rules, confer Chandy and Misra [1988].)

2.4 Fault Tolerance

Given a failure model F and a program A which is correct regarding a problem specification S, we can now apply the program transformation F to A, yielding a program A'. If A' is still correct regarding S then we say that A is *(masking) fault tolerant to F for S*.

3. A HIERARCHY OF FAILURE MODELS

In this section we discuss the common failure models that prevail in the literature and attempt to provide formal definitions of them. Then we show that these fault classes form a strict hierarchy, which can formally be proven. The failure models considered here are fail-stop, crash, send omission, receive omission, general omission and Byzantine. These models are restricted to affect only a fraction of all processes (usually half or one third). Let $\mathcal{F}$ denote this subset. As usual in these failure models, we assume that the communication links between processes not in $\mathcal{F}$ form a connected graph and are reliable.

3.1 Fail-stop

In the fail-stop model a process fails by halting [Schlichting and Schneider 1983]. The fact that this process has failed is however detectable by other processes. Formally, the program transformation *fail-stop* is defined as follows:

For every process $i \in \mathcal{F}$, do the following:

FS1 Add a boolean variable $\mathit{stopped}_i$ to the local state.

FS2 Transform every action x into the action

$$\langle \text{fail-stop } x \rangle :: \neg \mathit{stopped}_i \wedge \langle x \rangle$$

FS3 Add the action

$$\langle \text{fail-stop} \rangle :: \mathit{true} \longrightarrow \mathit{stopped}_i := \mathit{true}$$

FS4 Add the action

$$\langle \text{inform} \rangle :: \mathit{stopped}_i \longrightarrow C_{i,j} := \mathbf{snd}(C_{i,j}, m_i)$$

where m_i is a special message indicating that process i has fail-stopped.

Rule FS4 makes the fail-stopped state of a process detectable.

3.2 Crash

In the crash model a process fails by halting [Hadzilacos and Toueg 1994]. Formally, the program transformation *crash* is defined as follows:

For every process $i \in \mathcal{F}$, do the following:

C1 Add a boolean variable down_i to the local state.

C2 Transform every action x into the action

$$\langle \text{crash } x \rangle :: \neg \mathit{down}_i \wedge \langle x \rangle$$

C3 Add the action

$$\langle \text{crash} \rangle :: \mathit{true} \longrightarrow \mathit{down}_i := \mathit{true}$$

3.3 Send Omission

In the send omission model a process may exhibit the same behavior as in the crash model. Additionally, a process may fail by transmitting only a subset of messages that it actually attempts to send [Hadzilacos 1984; Hadzilacos and Toueg 1994]. Formally, the program transformation *send-omission* is defined as follows:

For every process $i \in \mathcal{F}$ do the following:

S1 For all actions of type

$$\langle x \rangle :: \langle \text{guard}_x \rangle \longrightarrow x_1, \ldots, x_n, C_{i,j} := y_1, \ldots, y_n, \mathbf{snd}(C_{i,j}, m)$$

add the action

$$\langle \text{s-omit } x \rangle :: \langle \text{guard}_x \rangle \longrightarrow x_1, \ldots, x_n := y_1, \ldots, y_n$$

S2–S4 Apply rules C1–C3.

Rule S1 models the fact that a process has the "choice" either to send a message or not to send the message.

3.4 Receive Omission

Consequently, in the receive omission model a process may also exhibit the crash model behavior, but additionally a process may fail by receiving only a subset of messages that it actually attempts to receive [Hadzilacos and Toueg 1994]. Formally, the program transformation *receive-omission* is defined as follows:

For every process $i \in \mathcal{F}$ do the following:

R1 For all actions of type

$$\langle x \rangle :: \langle \text{guard}_x \rangle \wedge \mathbf{arr}(C_{j,i}, m) \longrightarrow x_1, \ldots, x_n, z, C_{j,i} := y_1, \ldots, y_n, \mathbf{rcv}(C_{j,i}, m), \mathbf{rmv}(C_{j,i}, m)$$

add the action

$$\langle \text{r-omit } x \rangle :: \langle \text{guard}_x \rangle \wedge \mathbf{arr}(C_{j,i}, m) \longrightarrow x_1, \ldots, x_n, C_{j,i} := y_1, \ldots, y_n, \mathbf{rmv}(C_{j,i}, m)$$

R2–R4 Apply rules C1–C3.

Note that rule R1 applies to any action that attempts to receive a message. Hence, by executing the additional action, a message which is about to be received can simply vanish from the channel.

3.5 General Omission

In the general omission model a process may experience both send- and receive-omission failures [Perry and Toueg 1986]. Formally, the program transformation *general-omission* is defined as follows:

For every process $i \in \mathcal{F}$ do the following:

G1 Apply rule S1.

G2 Apply rule R1.

G3–G5 Apply rules C1–C3.

3.6 Byzantine

The Byzantine failure model [Lamport et al. 1982] allows the nodes to act in any way allowed by the system model. This encompasses arbitrary, even malicious behavior. Formally, the program transformation *Byzantine* is defined as follows:

For every process $i \in \mathcal{F}$ do the following:

B1 Add the following action:

$$\langle \text{loop} \rangle :: \mathit{true} \longrightarrow \mathbf{skip}$$

B2 For all $m \in M$ add the action

$$\langle \text{send } m \rangle :: \mathit{true} \longrightarrow C_{i,j} := \mathbf{snd}(C_{i,j}, m)$$

B3 For all $m \in M$ add the action

$$\langle \text{receive } m \rangle :: \mathbf{arr}(C_{j,i}, m) \longrightarrow C_{j,i} := \mathbf{rmv}(C_{j,i}, m)$$

It is important to not underestimate the simplicity of the resulting local algorithm. It is capable of both acting according to the original local algorithm and also of behaving arbitrarily as seen by an outside observer. The way of scheduling the additional actions is the source of a possible malevolence; however, as we are concerned with reasoning about static properties of programs, scheduling is of no concern here (as long as it is fair). The mere possibility of arbitrary behavior suffices for this failure model to be useful.

3.7 Relations to the System Model

The presented failure models can be seen as a kind of low level failure specifications: their semantics depend heavily on the system model which is used. For example, the distinction between *fail-stop* and *crash* can be made only in fully asynchronous systems, because in asynchronous systems it is impossible to distinguish a crashed processor from one which is merely very slow [Chandy and Misra 1986]. In synchronous or partially synchronous systems the information message sent by a fail-stopped process is equivalent to the passing of a certain time period. Thus *crash* and *fail-stop* burn down to the same. The problem of how to specify faults in a more abstract way will be briefly discussed in the conclusions.

3.8 Defining a Hierarchy of Failure Models

It is often desirable to compare failure models with eachother and have some order of severity between them. Note that the measure of severity only makes sense if it is related to some particular program A. Let $\mathit{Prop}(A)$ denote the set of possible executions of program A. Then formally, a failure model F_2 is *more severe* than a failure model F_1 regarding a program A (denoted $F1 <_A F_2$), iff

$$\mathit{Prop}(F_1(A)) \subset \mathit{Prop}(F_2(A))$$

Informally, this means that the behavior of A under failure model F_1 is more restricted than under F_2 (it must be a *proper* subset). Such a notion is useful to make the following statements: if A tolerates the failures of F_2 and if $F_1 <_A F_2$, then A tolerates the failures of F_1.

The severity order $<_A$ is a transitive partial order. Figure 1 shows the failure models of Section 3 put into relation regarding $<_A$, where A is some program that sends and receives messages and also does some internal computations. Note that if A were a program which never received any messages, then $\mathit{Prop}(\mathit{receive\text{-}omission}(A))$ would be equal to $\mathit{Prop}(\mathit{crash}(A))$ and so $\mathit{crash} <_A$ *receive-omission* would not hold.

4. EXAMPLE: FROM BROADCAST TO RELIABLE BROADCAST

In this section we show how the formal failure models of Section 3 faciliate the correctness proofs of fault tolerant algorithms. We use the basic broadcast algorithm

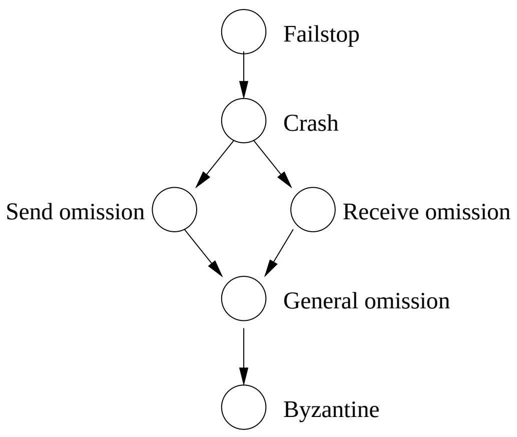

Fig. 1. Hierarchy of failure models using $<_A$ in asynchronous systems.

of Hadzilacos and Toueg [1994] and the crash failure model as an example.

4.1 Specification of Broadcast

Informally, a broadcast algorithm ensures that any message which was broadcast by a process is eventually delivered at all processes (liveness) and that the set of delivered messages contains no spurious messages (safety) [Hadzilacos and Toueg 1994]. Formally, a broadcast is defined in terms of two primitives: **broadcast** and **deliver**. A process p_i wanting to broadcast a message m invokes $\mathbf{broadcast}_i(m)$. This message is delivered at some process p_j if this process invokes $\mathbf{deliver}_j(m)$. Let B denote the multiset of messages over M that have been broadcast, and let D_i stand for the multiset of messages over M that have been delivered at p_i. Multisets are just like usual sets, except that elements can be contained more than once. For any multiset X there is a function $\#_X(y)$ which denotes the number of occurences of element y in X.

The correctness specification can now be formally stated as:

—(safety)

$$B \supseteq D_i \wedge \forall x \in D_i : \#_{D_i}(x) \leq 1$$

—(liveness)

$$\begin{gathered}[\mathbf{broadcast}_i(m) \wedge m \notin B \mapsto \mathbf{deliver}_i(m)] \wedge \\ [\mathbf{deliver}_i(m) \mapsto \forall j : m \notin D_j \Rightarrow \mathbf{deliver}_j(m)]\end{gathered}$$

The safety property ensures that every message is delivered at most once and that every delivered message was actually broadcast (Integrity). The liveness property ensures that every message which is broadcast at p_i for the first time will eventually be delivered locally (Validity) and that delivery at p_i will lead to delivery at all other processes (Agreement).

4.2 Broadcast Algorithm

Figure 2 shows the local broadcast algorithm of a single process. It uses the paradigm of message diffusion to reach its broadcast objective. A process p_i wanting to $\mathbf{broadcast}_i(m)$ places m into its own broadcast buffer $\tilde{b}$ (Action B1; g is some local guard that does not mention $\tilde{b}$). This message is transferred into the incoming buffer b through action B1', and in B4 assigned to the delivery buffer d. This is equivalent to an invocation of $\mathbf{deliver}_i(m)$. (The variables $\tilde{b}$, b, and d can take a special "empty" value denoted $-$.) Also in B4, the message is subsequently sent to all other neighbors. A process receiving such a message transfers it into its incoming buffer (B2) and relays it to all of its neighbors only if it has not been delivered locally before (B3 and B4).

We assume that every message is broadcast only once. This is ensured by postulating the existence of a global set B which contains all messages which have been previously broadcast by all processes. The guard $m \notin B$ of B1 ensures the uniqueness of broadcasted messages. (Note however that in practice one must tag messages with sequence numbers and process identities to achieve the same goal.)

From the algorithm we can see that the statement $\mathbf{broadcast}_i(m)$ is equivalent to the state where $\tilde{b} = m$ at process p_i, and that $\mathbf{deliver}_j(m)$ is equivalent to $d = m$ at p_j. So the liveness property can be rewritten as

$$(\tilde{b}_i = m \wedge m \notin B \mapsto d_i = m) \wedge (d_i = m \mapsto \forall j : d_j = m)$$

where $\tilde{b}_i$ and d_i denote variables $\tilde{b}$ and d at p_i. In cases where the index is clear from the context we will omit it.

Local Variables:

$\tilde{b} \in M \cup \{\perp\}$: broadcast buffer (initially $\perp$)
$b \in M \cup \{\perp\}$: incoming buffer (initially $\perp$)
$d \in M \cup \{\perp\}$: delivery buffer (initially $\perp$)
$D \in \mathcal{M}(M)$: multiset of delivered messages (initially empty)

Global Specification Variable:

$B \in \mathcal{M}(M)$: multiset of broadcasted messages (initially empty)

Algorithm:

```
B1::   $\tilde{b} = \perp \wedge m \notin B \wedge g$                $\rightarrow$   $\tilde{b}, B := m, B \cup \{m\}$
B1'::  $b = \perp \wedge \tilde{b} \neq \perp$                      $\rightarrow$   $b, \tilde{b} := \tilde{b}, \perp$
B2::   $\mathbf{arr}(C_{j,i}, m) \wedge b = \perp$             $\rightarrow$   $b, C_{j,i} := \mathbf{rcv}(C_{j,i}, m), \mathbf{rmv}(C_{j,i}, m)$
B3::   $d = \perp \wedge b \neq \perp \wedge b \in D$          $\rightarrow$   $b := \perp$
B4::   $d = \perp \wedge b \neq \perp \wedge b \notin D$       $\rightarrow$   $b, d, D, C_{i,k} := \perp, b, D \cup \{b\}, \mathbf{snd}(C_{i,k}, b)$
B5::   $d \neq \perp$                                   $\rightarrow$   $d := \perp$
```

Fig. 2. Local algorithm for reliable broadcast in point-to-point networks.

4.3 Specification for Reliable Broadcast

A reliable broadcast is a broadcast that achieves the broadcast objective despite of some sort of failure in the system. The failure model used in this example is the crash failure model, i.e., there exists a set of *faulty* processes that crash at some point during the execution. All non-faulty processes are called *correct*. We must adapt the specification of reliable broadcast to deal with this new situation. A simple way to do this is to restrict the specification to all processes that do not crash. Formally, this yields:

—(safety)

$$\mathbf{invariant}(down_i = \mathit{false}) \Rightarrow B \supseteq D_i \wedge \forall x \in D_i : \#_{D_i}(x) \leq 1$$

—(liveness)

$$\begin{aligned}&\mathbf{invariant}(down_i = \mathit{false}) \Rightarrow\\ &\quad [(\tilde{b}_i = m \wedge m \notin B \mapsto d_i = m) \wedge\\ &\quad (d_i = m \mapsto \forall j : \mathbf{invariant}(down_j = \mathit{false}) \wedge m \notin D_j \Rightarrow d_j = m)]\end{aligned}$$

The safety property reads: every message that is delivered at a correct process was previously broadcast, and delivery at correct processes occurs at most once. Liveness reads: every message broadcast by a correct process p_i will be eventually delivered at p_i, and if a correct process delivers a message then eventually all correct processes will deliver the message.

Note that there can be different notions of the correctness of a reliable broadcast. They differ in the restrictions imposed on the behavior of faulty processes (see for example the *uniformity* property [Gopal and Toueg 1990; Hadzilacos and Toueg 1994], which requires that messages delivered at faulty processes must be delivered at all correct processes too).

4.4 Proof of Correctness

We now prove that the protocol from Figure 2 is masking fault tolerant to the crash failure model for the refined specification from the previous subsection. This is done by reasoning about the protocol after applying the program transformation *crash* to it.

4.4.1 *Safety.* To prove that the safety property holds, it suffices to show that it is an invariant of the protocol at all correct processes. Note however that in general it is necessary to prove that the faulty processes from $\mathcal{F}$ also preserve the invariant. However, because the precondition $\mathbf{invariant}(down_i = \mathit{false})$ does not hold for faulty nodes, they cannot violate the safety predicate. For correct processes, we will in fact show, that the following stronger property S is an invariant:

$$\begin{aligned}S \equiv\ & B \supseteq D \wedge \forall x \in D : \#_D(x) \leq 1 \wedge\\ & (b = - \ \vee b \in B) \wedge (\tilde{b} = - \ \vee \tilde{b} \in B) \wedge\\ & (\neg\mathbf{arr}(C_{i,j}, y) \vee \mathbf{rcv}(C_{i,j}, y) \in B)\end{aligned}$$

We first show that the initial configuration I implies S: In the initial configuration $B = D = \emptyset$, $\tilde{b} = b = d = -$ and $\neg\mathbf{arr}(C_{i,j}, m)$. From $\emptyset \supseteq \emptyset$ we can deduce $B \supseteq D$, which is the first conjunct of S. From $\forall x \in \emptyset : \#_D(x) = 0$ we can derive the second conjunct, and $b = -$, $\tilde{b} = -$, $\neg\mathbf{arr}(C_{i,j}, m)$ cater for the last three conjuncts. So $I \Rightarrow S$.

Now we must show that each action of the algorithm preserves S.

4.4.1.1 *B1:.* In B1, a new broadcast message is placed into $\tilde{b}$ and added to B. Because $B \cup \{m\} \supseteq B$ the first conjunct is preserved. The conjunct $(\tilde{b} = - \vee \tilde{b} \in B)$ stays valid because $\tilde{b} = m \in B \cup \{m\}$. And any property of type $x \in B$ implies $x \in B \cup \{y\}$. So B1 preserves S.

4.4.1.2 *B1':.* In B1', the value of $\tilde{b} \neq -$ is transferred to b if $b = -$. This affects only the two conjuncts $(b = - \vee b \in B)$ and $(\tilde{b} = - \vee \tilde{b} \in B)$. From S and $\tilde{b} \neq -$ we have that $\tilde{b} \in B$. Hence, after executing the assignment both $b \in B$ and $\tilde{b} = -$ will hold. So B1' preserves S.

4.4.1.3 *B2:.* In B2, an incoming message is placed into b. From S and the fact that $\mathbf{arr}(C_{j,i}, m)$ we deduce that $\mathbf{rcv}(C_{j,i}, m) \in B$ and so that after the assignment $b \in B$ holds. So the conjunct $(b = - \vee b \in B)$ is preserved. The semantics of the communication primitives give $\mathbf{arr}(\mathbf{rmv}(C_{i,j}, m), m) = \mathit{false}$ and so the final conjunct of S is preserved as well. So B2 preserves S.

4.4.1.4 *B3:.* In B3, an incoming message is dismissed if it has already previously been delivered. Only the third conjunct of S is affected and it is preserved because $b = -$ holds after the assignment. So B3 preserves S.

4.4.1.5 *B4:.* This is maybe the most difficult passage of this part of the proof. Action B4 delivers a previously non-delivered message by placing it into d and relays it to all its neighbors. Consider the first conjunct $B \supseteq D$: from S and $b \neq -$ we follow that the value of b must be in B. But because $b \in B$ and $b \notin D$ the predicate $B \supseteq D \cup \{b\}$ holds. So, because b is added to D by the assignment, the first conjunct is preserved. Because $b \notin D$ it must be the case that $\#_D(b) = 1$ after executing the statement. So the conjunct $\forall x \in D : \#_D(x) \leq 1$ is preserved too. As $b = -$ after the assignment, the third conjunct is trivially preserved. Finally, the sending of a message m either does not influence the last conjunct (because $\mathbf{arr}(C_{i,k})$ remains unaffected) or m is receivable. But in the latter case S gives that $m = \mathbf{rcv}(C_{i,k}, m) \in B$, so the final conjunct is preserved as well. Hence, B4 preserves S.

4.4.1.6 *B5:.* B5 trivially preserves S since the value of d does not appear in S.

4.4.2 *Liveness.* For reasons outlined above, we can restrict our attention again to all correct processes when proving that the liveness property holds. For this we show two things:

—(Validity)

$$\mathbf{invariant}(\mathit{down}_i = \mathit{false}) \Rightarrow (\tilde{b}_i = m \wedge m \notin B \mapsto d_i = m)$$

—(Agreement)

$$\begin{aligned}&\mathbf{invariant}(\mathit{down}_i = \mathit{false}) \Rightarrow \\ &\quad (d_i = m \mapsto \forall j : \mathbf{invariant}(\mathit{down}_j = \mathit{false}) \wedge m \notin D_j \Rightarrow d_j = m)\end{aligned}$$

4.4.2.1 *Validity.*. Assume $down_i = \mathit{false}$ is invariant (so p_i is a correct process), $m \notin B$ and $\tilde{b}_i = m$. From the fact that $B \supseteq D_i$ always holds, $m \notin B$ implies $m \notin D_i$. So the following *ensures* relations hold for the protocol:

$$\begin{array}{rl} \tilde{b}_i = m \ \ \mathit{ensures} \ \ b_i = m & \{\text{from action B1'}\} \\ b_i = m \wedge m \notin D_i \ \ \mathit{ensures} \ \ d_i = m & \{\text{from action B4}\} \end{array}$$

As p *ensures* q implies $p \mapsto q$ and by transitivity of $\mapsto$ we finally have $\tilde{b}_i = m \mapsto s_i = m$ under the given assumptions. So Validity holds.

4.4.2.2 *Agreement.*. As a preliminary, observe that if b_i or d_i have some value different from $-$, eventually they will take on the value $-$ again (formally: $b_j = x \mapsto b_j =-$ and $d_j = x \mapsto d_j =-$).

Now, assume again that p_i is a correct process and $d_i = m$. From the fact that $d_i = m$ and the algorithm, we can deduce that $\mathbf{snd}(C_{i,j}, m)$ has been assigned to every outgoing channel and thus $m \in C_{i,j}$. The channel properties ensure that eventually $\mathbf{arr}(C_{i,j}, m)$ will hold at the destination processes, and once $b_j =-$ holds, this will eventually lead to $b_j = m$ (action B2).

Now assume that $m \notin D_j$. Action B4 gives the following fact:

$$d_j =- \ \wedge b_j = m \wedge m \notin D_j \ \ \mathit{ensures} \ \ d_j = m$$

Because $\Rightarrow$ implies $\mapsto$, *ensures* implies $\mapsto$ and by transitivity of $\mapsto$, we have that $d_i = m \mapsto d_j = m$ for all neighbors p_j of p_i. This argument can be generalized by induction over the structure of the graph connecting correct processes. Because the failure model guarantees that this graph is connected, eventually $d_j = m$ will hold for all correct processes p_j. So Agreement holds.

5. RELATED WORK

The failure models considered here and their interrelations were studied by Schneider [1993], Barborak, Dahbura, and Malek [1993] and Hadzilacos and Toueg [1993, 1994]. While their exposition is precise, it does not have the necessary level of detail to comply with the rigor of mechanical verification systems like PVS [Owre et al. 1996].

The idea to represent faults as program actions goes back to a seminal paper by Cristian [1985]. It was further developed in a series of papers by Arora et al. [Arora and Gouda 1993; Arora and Kulkarni 1998a; Arora and Kulkarni 1998b], who stress that every form of failure can be modeled by this method. However, they do not attempt to uniformly re-formulate the standard failure models. Specific system components like the faulty channels of Chandy and Misra [1988] or of Lynch [1996] have been used to model faulty parts of a system. But to the best of the author's knowledge the notion of a failure model being a program transformation is novel, although some preliminary ideas regarding crash [Kekkonen 1998; Lynch 1996] and Byzantine failures [Lynch 1996] have appeared in the literature. The work closest to ours in both aim and scope is a recent paper by Völzer [1998] which formally specifies *crash* and *omission* failures using a Petri-net-based modeling approach. Völzer [1998] distinguishes between a *fault impact model* (which specifies the local fault behavior, e.g., a crash) and a *rely specification* (stating global system assumptions, e.g., no more than t processes crash). A *fault impact model* is added to a system by superimposing a specific Petri net. However, neither *fail-stop* nor *Byzantine* faults are discussed.

6. CONCLUSIONS

In this paper we have presented a unified and formal approach to specify failure models which can be used to formally prove the correctness of fault tolerant algorithms. The method is based on the notion of a failure model being a program transformation and reasoning about transformed programs to evaluate fault tolerance properties. As an example, we applied our method to the problem of reliable broadcast in the crash failure model and showed, how this approach can be a useful step towards the formal rigor necessary in mechanical verification of such algorithms.

To conclude, we would like to mention several points that are not yet fully understood and thus could be rewarding as a subject of future work.

Firstly, it is interesting to see the correlation between parts of the send and receive omission failure models and the specifications of unreliable links presented in a paper by Basu, Charron-Bost, and Toueg [1996]. It seems as if the semantics of *fair lossy links* closely resembles the behavior obtained from the general omission failure model without crash semantics. Modeling link failures with our method could provide insight into the relations between both models.

Secondly, the recent work of Völzer [1998] in Petri-net-based fault modeling suggests to decompose formal fault models into a local and a global part. This seems promising since local fault actions differ fundamentally from global assumptions in a decisive way: they "add" behavior, while the latter restricts it. Investigating the relations between the Petri-net approach and ours in this point seems rewarding. Joint work is planned.

Thirdly, note that the presented algorithm is also correct in general omission environments since processes which experience send or receive omission failures are considered faulty and do not influence the refined specification for reliable broadcast. This observation leads to the final fourth point:

It is quite obvious that the correctness proof of Section 4.4 is relatively simple because of the nature of the augmented specification: it was restricted to all non-faulty processes, i.e., the behavior of faulty processes was not taken into account. However, as mentioned earlier, there are also other possible ways to qualify specifications for failure models. Future work must investigate these ways to be able to derive augmented specifications, because they are a prerequisite for applying mechanichal verification. Finding a specification formalism for failure models which is more abstract and thus more adequate for this task than the "implementation approach" presented here will be the focus of our ongoing work.

ACKNOWLEDGMENTS

Thanks to Anish Arora, Hagen Völzer, Franck Petit and Henning Pagnia for discussing this topic with me, and to William Shakespeare for inspiring the title.

REFERENCES

Alpern, B. and Schneider, F. B. 1985. Defining liveness. *Information Processing Let-*

ters 21, 181–185.

ARORA, A. AND GOUDA, M. 1993. Closure and convergence: a foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering 19*, 11, 1015–1027.

ARORA, A. AND KULKARNI, S. 1998a. Detectors and correctors: A theory of fault-tolerance components. In *Proceedings of the 18th International Conference on Distributed Computing Systems* (1998).

ARORA, A. AND KULKARNI, S. S. 1998b. Component based design of multitolerance. *IEEE Transactions on Software Engineering 24*, 1, 63–78.

BARBORAK, M., DAHBURA, A., AND MALEK, M. 1993. The consensus problem in fault-tolerant computing. *ACM Computing Surveys 25*, 2 (June), 171–220.

BASU, A., CHARRON-BOST, B., AND TOUEG, S. 1996. Simulating reliable links with unreliable links in the presence of process crashes. In *WDAG96 Distributed Algorithms 10th International Workshop Proceedings, Springer-Verlag LNCS:1151* (1996), pp. 105–122.

CHANDY, K. M. AND MISRA, J. 1986. How processes learn. *Distributed Computing 1*, 40–52.

CHANDY, K. M. AND MISRA, J. 1988. *Parallel Program Design : a Foundation.* Addison-Wesley, Reading, Mass.

CRISTIAN, F. 1985. A rigorous approach to fault-tolerant programming. *IEEE Transactions on Software Engineering 11*, 1 (Jan.), 23–31.

GOPAL, A. AND TOUEG, S. 1990. On the specification of fault-tolerant broadcast. In *Proc. Int. Workshop on Future Trends of Distributed Computing Systems* (Cairo, Egypt, 1990), pp. 54–56. IEEE Computer Society Press.

HADZILACOS, V. 1984. *Issues of Fault Tolerance in Concurrent Computations.* Ph. D. thesis, Harvard University. also published as Technical Report TR11-84.

HADZILACOS, V. AND TOUEG, S. 1993. Fault-tolerant broadcasts and related problems. In S. MULLENDER Ed., *Distributed Systems* (Second ed.)., Chapter 5, pp. 97–145. Addison-Wesley.

HADZILACOS, V. AND TOUEG, S. 1994. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425 (May), Cornell University, Computer Science Department. This a revised version of [Hadzilacos and Toueg 1993].

KEKKONEN, S. 1998. *Résistance aux Fautes dans les Algorithmes Répartis: Auto-Stabilisation et Tolérance aux Fautes.* Ph. D. thesis, Université de Paris-Sud, France.

LAMPORT, L., SHOSTAK, R., AND PEASE, M. 1982. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems 4*, 3 (July), 382–401.

LYNCH, N. 1996. *Distributed Algorithms.* Morgan Kaufmann Publishers, Inc., San Francisco, CA.

MULLENDER, S. Ed. 1993. *Distributed Systems* (Second ed.). Addison-Wesley.

OWRE, S., RAJAN, S., RUSHBY, J., SHANKAR, N., AND SRIVAS, M. 1996. PVS: Combining specification, proof checking, and model checking. In R. ALUR AND T. A. HENZINGER Eds., *Computer-Aided Verification, CAV '96*, Number 1102 in Lecture Notes in Computer Science (New Brunswick, NJ, July/August 1996), pp. 411–414. Springer-Verlag.

PERRY, K. J. AND TOUEG, S. 1986. Distributed agreement in the presence of processor and communication faults. *IEEE Transactions on Software Engineering 12*, 3 (March), 477–482.

SCHLICHTING, R. D. AND SCHNEIDER, F. B. 1983. Fail stop processors: an approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems 1*, 3 (Aug.), 222–238.

SCHNEIDER, F. B. 1993. What good are models and what models are good? In S. MULLENDER Ed., *Distributed Systems* (Second ed.)., Chapter 2, pp. 17–26. Addison-Wesley.

VÖLZER, H. 1998. Verifying fault tolerance of distributed algorithms formally: An example. In *Proceedings of the 1998 International Conference on Application of Concurrency to System Design (CSD98)* (Fukushima, Japan, March 1998), pp. 187–197. IEEE Computer Society Press.