Formal Specification of Fault Tolerance and its Relation to Computer Security

D.G. Weber

Odyssey Research Associates, Inc.
301A Harris B. Dates Drive
Ithaca, NY 14850-1313
607 277 2020

Abstract

The techniques of formal verification are one means for gaining greater assurance of the correctness of software. These techniques require precise specification of the properties to be assured. This paper formulates precise specifications corresponding to the intuitive notions of "fault tolerance" and of "graceful degradation". An analogy is constructed between these fault-tolerance specifications and a particular class of specifications for computer security. On the basis of this analogy, it is argued that formal verification of fault tolerance will face some of the same problems, and benefit from some of the same solutions, as verification of security.

1 Introduction

In this paper, we will be concerned with specification and verification of fault-tolerance properties. We will be seeking a precise definition of the term "fault tolerance" and asking what steps must be taken to prove that a system design is in fact fault tolerant according to the definition. We will be only minimally concerned with strategies, designs, and algorithms used to implement fault-tolerant systems, and only then as examples to show why a particular definition of "fault tolerance" is relevant.

One previous effort toward verifying fault tolerance can be found in the SIFT project (Software Implemented Fault Tolerance) [11]. SIFT was an ultra-reliable fault-tolerant computer designed for aircraft flight control. A precise model of this system was developed, and constraints (specifications) on the model which implied the correctness of the system were written down. However, the SIFT approach differs from ours, in that the property of "fault tolerance" was never considered in isolation, but was always implicitly subsumed in the other system specifications. One could not prove, or even state, that SIFT was fault tolerant without additionally stating and proving that it satisfied many other correctness properties as well. We intend to find a definition of fault tolerance that can stand by itself.

The remainder of this section discusses factors that should be excluded from a formal specification of computer system properties. Section 2 develops specifications for fault tolerance and graceful degradation. In section 3 we consider verification of systems against these specifications. This entire paper is condensed from [9]. A more complete discussion, development of fault-tolerance properties, and examples, can be found there.

1.1 Fault Scenarios and MTTF

A system's fault tolerance is sometimes expressed as a mean time to failure (MTTF). One goal of a verification methodology for fault-tolerant systems might be to prove that a system's MTTF exceeds some value. However, the MTTF is not just a property of a computer system, but involves the environment of that system as well. A measure of fault tolerance that depends only on the system design, and not on its environment, would be preferable.

A **fault scenario** is a history of a system's interaction with its environment, and which includes not only the system's inputs and outputs but also a description of faults, including which components failed, when they failed, and how each failure is expected to manifest itself in the future. We suppose that a system's environment determines

[0] This work was supported by the Air Force Systems Command at Rome Air Development Center under Contract No. F30602-86-C-0263. The views and conclusions contained in this paper are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Air Force or the U.S. Government.

the likelihood of each fault scenario.

If we can now decide, for each scenario, whether the system design will fail, and at what time it fails, then we can (in principle) calculate the MTTF by averaging over all the fault scenarios. This average, however, is usually quite complicated and involves many approximations and assumptions about the environment.

Instead of calculating MTTF, we will consider verification of fault tolerance to be proof that a system design will not fail for a given fault scenario or some set of fault scenarios. For example, one might verify that system failure does not happen in any fault scenario in which at most one fault occurs. This meaning of "fault tolerance" depends only on the system design, and is independent of environmental factors.

2 Formal Specification of Fault Tolerance

2.1 Specifying Faults

Before defining "fault tolerance", it is first necessary to define "fault". Abstractly, a system or system component is faulty when it no longer performs according to its specification.

The literature of fault tolerance identifies various "specifications" for components after they have failed. These include Byzantine (a failed component exhibits arbitrary behavior), fail-stop (a failed component halts and its failure can be detected), and others.

2.2 Non-Interference

The simplest way to define "fault tolerance" is converse to our previous definition of "fault": a system is fault tolerant if it performs according to its specification. Thus, it must behave as though it were non-faulty, even in the presence of faulty behavior of its components.

Denote by N the set of fault scenarios under which no faults occur. Let C be a set of fault scenarios under which we desire fault tolerance; no loss of generality will result if we require that $N \subseteq C$. Suppose that D is a system design that exhibits the behavior desired, and that FTD is a fault-tolerant version of D. There are now three methods by which we may show that FTD is fault tolerant under the given set of fault scenarios, C:

1. We may show that the behavior of D, under scenarios N, is identical to the behavior of FTD under C.

2. We may characterize the behavior of D under N by some specification, S. Then we may show that FTD implements S under C.

3. We may show that the behavior of FTD under N is equivalent to the behavior of FTD under C.

The first of these methods is usually impractical. We would need to construct two separate versions of one system.

The second method is the one used in the SIFT project. The specification S describes the correct behavior both of D and of FTD. We would then require simply that FTD behave correctly.

The third method captures the notion of fault tolerance as a comparison of behaviors with and without occurrences of faults. However, it does so without referring either to design D or to its specification S. The relevant aspects of each can be derived from FTD alone. The behavior of FTD under scenarios with no faults should be equivalent to behavior of D under the same scenarios. Therefore, the third method has a clear advantage over the others: the property of fault tolerance becomes entirely a property of the behavior of FTD, and does not involve extra correctness constraints that may be required by S. It is this third method we will proceed to develop.

Our statement of fault tolerance is now a relation on the behavior of the system under different sets of fault scenarios. Let the possible ways a system may interact with its environment be called "events". Sequences of events will be called "histories", and a history that is possible for a system will be called a "trace" of that system. One way to characterize a design is to give the set of its traces. This is a simplification of the approach of CSP [2]. "Behavior" will be defined by the sequence of events of a trace that are visible to a system's users.

Simple fault tolerance is then: allowing certain fault scenarios does not change the visible aspects of the set of traces of a fault-tolerant system. We may say that the occurrence of fault events does not *interfere* with a system's behavior.

2.2.1 Formal Definition

Let a system be characterized by a tuple $\langle E, F, T \rangle$, where E is the set of possible events, $F \subseteq E$ is the set of possible fault events, and T is the set of traces of events chosen from E.

For any event sequence h and set of events S, the notation $h \uparrow S$ will denote the sequence derived from h but with all events not in S removed and the ordering of the remaining events retained. For any trace h, what was called "behavior" above will now be denoted by the sequence of non-fault events, $h \uparrow \overline{F}$, where $\overline{F} = (E - F)$ is the complement of F with respect to E. Two sets of traces exhibit the same behavior if, for any trace in one, there is a trace in the other which has the same sequence of events in $\overline{F}$.

We characterize each fault scenario as a sequence of events. Each scenario is simply a history, whether or not it is a possible history for a given system.

We will now formalize the above definition of fault tolerance for the system $A = \langle E, F, T \rangle$. Let C be the set of fault scenarios for which A is to be fault tolerant. The definition says that adding fault scenarios in C will not enlarge the set of behaviors. This is equivalent to the following statement in our notation:

$$(FT1) \qquad \forall \beta \in T, \beta \in C \ \rightarrow \ \beta \uparrow \overline{F} \in T$$

This says that from any given possible history that is also a fault scenario in C, we can construct a second possible history simply by removing all fault events from the first. This definition of simple fault tolerance can be augmented in various ways.

2.2.2 Example: File System

As an example, consider a fault-tolerant file system. An external user or other client of the file system interacts with it through operations such as "read-file", "write-file", and so on. These operations, along with any of their associated parameters and return values, will be taken as the events from which system histories are constructed. Certain observable behaviors, or traces, are expected, e.g., if a user writes a file and then reads the same file, the contents returned should be the same as the contents written. So the history $h_1 = \langle$ write-file 'contents', read-file 'contents' $\rangle$ should be a trace, while the history $h_2 = \langle$ write-file 'contents', read-file 'garbage' $\rangle$ where 'contents' and 'garbage' are different, should not be.

Fault events in the hardware supporting the file system may make various bizarre behaviors possible. For example, if the sequence $h_3 = \langle$ write-file 'contents', fault, read-file 'garbage' $\rangle$ is a trace, then the system may appear as though it had actually executed the illegal history h_2. If, however, the file system is to be fault tolerant with respect to fault scenarios including h_2, then our property (FT1) demands that h_3 won't exist if h_2 doesn't.

Why isn't property (FT1) equivalent to requiring that the file system work correctly? In fact, (FT1) is weaker. Suppose, for some reason, that when each file is written, the file system alters the contents with some arbitrary function Mod; when a file is read, the altered contents are returned. The traces of this new file system, if it is fault tolerant, will include:

$\langle$ write-file 'contents', read-file Mod('contents') $\rangle$

$\langle$ write-file 'contents', fault, read-file Mod('contents') $\rangle$

and possibly many others. Like our previous file system that did not alter the content of files, this one satisfies (FT1) because the alterations are made to files regardless of the presence of a fault. But, if we have specified a file system that does not modify the content of files, this implementation is incorrect even though it is fault tolerant.

2.3 Analogy with Multi-Level Security

Property (FT1) is often called a "non-interference" property. Non-interference properties have been explored and used extensively in the context of multi-level computer security (MLS) [3] [5] [6] [8]. It is reasonable to ask, then, what is the relation between fault tolerance and multi-level security? We begin with a brief description of MLS.

In a secure computer system, it is desirable to prevent sensitive information from flowing to users who are not authorized access to it. In military systems, the sensitivity of information and the authorization of users can both be labeled by a partially-ordered set of levels. Highly sensitive information is brought into the system by the inputs of system users who have high levels of authorization. The multi-level security problem is the prevention of information transfer from those high-level inputs to outputs that can be seen by users who are not so highly authorized.

The key to defining security in this way is the definition of "information flow". We suppose that the system's traces are known. One definition of information flow is then the ability of a particular user to use the observed behavior of the system, plus knowledge of its traces, to make deductions about its unseen behavior. Information will flow from unseen inputs to observed behavior if more can be deduced about those inputs than could be deduced if the system's behavior were not observed.

Non-interference can be taken (loosely) to mean that high-level inputs do not interfere with or influence processing, and hence outputs, on lower levels. Stated more precisely, the existence of high-level inputs cannot be deduced from observing a particular history of lower-level inputs and outputs.

An analogy between fault tolerance and multi-level security properties can now be drawn. A non-interference property for multi-level security can be converted to a non-interference property for fault tolerance by translating from "highly sensitive" inputs to "fault events", and from "less sensitive inputs and outputs" to "non-fault events", i.e., ordinary system behavior. A fault event is thus considered a type of input, although not from any user. The non-interference property for MLS can then be translated into the language of fault tolerance: the existence of fault events cannot be deduced from particular histories of ordinary system behavior. This is the property (FT1).

The analogy can be posed in another way if we consider the work of Biba [1] in extending MLS to handle some aspects of information integrity. His work added integrity levels to the security levels already used for marking the sensitivity of information and the authorizations of users. The Biba integrity property, in effect, allows information to flow only from inputs to outputs of the same or lower integrity level. Given this property, high-integrity users would be prohibited from deducing (and hence from being corrupted by) information about inputs at lower integrity levels. The analogy between fault tolerance and MLS can be recast in terms of integrity, in which case a fault event is seen to be analogous to an input of low integrity level. This becomes an appropriate analogy if we consider that fault events are not usually a high-integrity source of information.

The analogy we have constructed, between fault-tolerance properties on one hand and multi-level security properties on the other, is not perfect and breaks down in several ways:

- Unlike the inputs from users, which are the ultimate source of information in MLS systems, fault events are usually not external and not observable.

- Systems are never, in practice, tolerant to all fault scenarios: some possible sequence of faults will cause the system to fail. This differs from the analogous multi-level security case, in which one desires to build systems that are secure under all possible histories of sensitive inputs. Users of an insecure system may conspire to transmit information by concocting unusual or unlikely sequences of inputs; fault events are assumed not to do this.

Because fault-tolerance non-interference properties and MLS non-interference properties are formally similar, can the same kinds of implementation mechanisms be used for both? The differences listed above indicate why this will not work. Faults are not external events, and therefore it is not possible for a system to decide, without further processing, whether they are fault events or not. A fault-detection mechanism may be needed. In secure systems, however, inputs are associated with the authorization level of the user who causes them. The difference in brief: faults don't log in! Thus, even though there is a formal similarity between fault tolerance and MLS properties, the designs used to implement them will be different.

While fault tolerance and MLS need different designs and implementations, the methods used to *verify* an implementation either fault tolerant or secure should be similar. See section 3.

2.4 Graceful Degradation

The problem of specifying graceful degradation of a system's service in response to faults will have much in common with the previous discussion of specifying fault tolerance. We expect that many systems will be fault tolerant for some fault scenarios, gracefully degrade for others, and be chaotic for the rest. As a result, specifications for graceful degradation may be merely modifications or generalizations which weaken those we have already discussed for pure fault tolerance. In fact, the specifications we will arrive at in this section can be seen simply as a more comprehensive way to define fault tolerance itself.

2.4.1 Limited Interference

If fault tolerance is to be expressed as a non-interference property, as discussed in section 2.2, then graceful degradation may be expressible as some limited interference of faults with external behavior. A specification of limited interference should be a generalization of a specification for non-interference. The form of the specification must then show the way in which interference of fault events with normal behavior is limited.

When we developed property (FT1), we required that system behavior with and without faults be *identical*. This prevents an observer from deducing that *any* faults have occurred. Unlike the MLS case, though, it may not be a problem that one can deduce the existence of faults, so long as the system behavior in response to those faults is "good enough". Thus, we need not demand that behaviors be identical, but only that they be *acceptably equivalent*. If α and β are behaviors (sequences in which no fault events occur) then let $\alpha \equiv \beta$ mean that the two behaviors are acceptably similar. The relation '$\equiv$', called the **tolerance relation**, will be an equivalence relation on behaviors.

Adding fault scenarios to a system satisfying (FT1) will not enlarge the set of behaviors. We now want a second property, (FT2), such that adding fault scenarios will not enlarge the set of possible equivalence classes of behaviors, where a "possible equivalence class" is one that contains at least one possible behavior. Repeating the analysis that led to (FT1) but demanding only equivalent instead of identical behavior, we find that

$$(FT2) \qquad \forall \beta \in T, \beta \in C \;\rightarrow\; \exists \gamma \in T, \gamma \upharpoonright F = \langle\rangle \; and \; \gamma \equiv \beta \upharpoonright \overline{F}$$

is the generalized fault tolerance, or graceful degradation, property that results. This property says: for any fault scenario, we must be able to find an alternate possible history that is fault-free and is acceptably equivalent. If, for some reason, an observer of the system could not distinguish the behavior γ from $\beta \upharpoonright \overline{F}$, then, just as for (FT1), the existence of faults could not be deduced.

As an example, consider a system that must perform two tasks, A and B. Suppose that fail-stop processors A_1 and A_2 are dedicated to simultaneous execution of task A, while fail-stop processors B_1 and B_2 are dedicated to task B. Ignoring the amounts of time needed for processors to compare final results, this system will be perfectly fault tolerant (FT1) for a fault scenario in which processors A_1 and B_1 fail: both tasks will complete, and they will complete in the same amount of time they would have taken if no faults had occurred. However, it will not be fault tolerant for a fault scenario in which processor A_2 fails in addition to A_1 and B_1. In this case, processing of task A is interrupted, and will not be completed unless the system uses processor B_2 to finish task A, and even then task A will not be finished in the same amount of time as in a fault-free scenario.

For this example, we would like "graceful degradation" to mean that for any fault scenario in which three or fewer processors fail, both tasks A and B will eventually complete. To implement this specification will require the system to reconfigure in some fault scenarios: interrupted processing of task A will need to be continued or restarted on a processor originally dedicated to task B, or vice versa. The processing power of the system will then be degraded, because a single processor will take longer to complete both tasks than either one separately, but the response to faults will be graceful because at least both tasks will be finished. To describe this type of graceful degradation, we would choose a tolerance relation that ignores timing: two histories are equivalent if they involve the same sequence of events, but at possibly different times.

Note that (FT2) reduces to (FT1) in the case that the tolerance relation is equality. It is the choice of tolerance relation that determines how faults interfere with behavior, and therefore the meaning of "graceful degradation". As one chooses larger sets of fault scenarios, C, the tolerance relation must be chosen to treat more behaviors as equivalent.

This form of limited interference can be viewed in terms of the analogy with multi-level security. Systems that meet (FT2) but not (FT1) are analogous to systems that leak some information from high security levels to lower ones, and are thus not perfectly secure.

3 Verification of Fault Tolerance

We have argued that non-interference specifications can be used to capture the intuitive notion of fault tolerance. How can a system be verified, in practice, to implement this sort of specification?

Once the constructs of "event" and "trace" are related to features of the implementation, one might appeal directly to the definition in constructing a proof. However, for all but the simplest system, this approach becomes very complicated.

Many of existing verification tools [7][4] provide little help either. Typically, these tools are designed for proof of invariants, or more generally, of *embedded assertions*: conditions that hold at a particular point in an execution history. Unfortunately, an embedded assertion expresses a condition that applies to each history independently, whereas a non-interference specification applies to the entire set of possible histories at once. The help provided by these tools is not sufficient.

Because fault-tolerance specifications are formally similar to specifications for multi-level security, this same problem arises in the verification of MLS. In that domain, specialized techniques can be applied to analyze special cases. For example, the technique of [10] is one in which the existence of some traces is shown by modifying other traces in appropriate ways. Demonstrating the existence of one trace, given another trace, is exactly what is needed in both (FT1) and (FT2), and this technique is in fact one that can be applied either to designs in the MLS or fault-tolerance domains.

References

[1] K. Biba. Integrity considerations for secure computer systems. Technical Report MTR-3153, MITRE Corp., Bedford, MA, Apr. 1977.

[2] S. Brookes, C. Hoare, and A. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3), 1984.

[3] J. Goguen and J. Meseguer. Security policy and security models. In *IEEE Symp. Security and Privacy*, 1982.

[4] D. Good et al. Report on the Gypsy language, version 2.0. Technical Report ICSCA-CMP-10, Institute of Computer Science, Univ. of Texas, Austin, Sept. 1978.

[5] J. Haigh et al. An experience using two covert channel analysis techniques on a real system design. In *IEEE Symp. Security and Privacy*, 1986.

[6] D. McCullough. Specifications for multi-level security and a hook-up property. In *IEEE Symp. Security and Privacy*, 1987.

[7] B. Silverberg et al. The HDM handbook, volume ii. Technical Report Deliverable A006, project 4828, SRI, Menlo Park, CA, June 1979.

[8] S. T. Vinter, D. Weber, et al. A secure distributed operating system. In *IEEE Symp. Security and Privacy*, 1988.

[9] D. Weber. Specifications for fault tolerance. Technical Report 19-3, ORA Corp., Jan. 1988.

[10] D. Weber and B. Lubarsky. The SDOS project – verifying hook-up security. In *Comp. Security Applications Conf.*, pages 7–15, 1987.

[11] J. Wensley et al. SIFT: Design and analysis of a fault-tolerant computer for aircraft control. *Proc. IEEE*, 66(10), Oct. 1978.