An Exercise in Systematically Deriving Fault-Tolerance Specifications*

Felix C. Gärtner
Computer Science Department
Darmstadt University of Technology
D-64283 Darmstadt, Germany
Email: felix@informatik.tu-darmstadt.de

Abstract

We perform a case study on the interrelations between problem specifications in ideal environments and in faulty ones. The problem considered is that of consensus *and the failure model used is* crash. *The goal of this research is to uncover the influences that specific failure models have on problem specifications so that fault-tolerance specifications can be systematically derived.*

Keywords: specification, verification, fault tolerance, failure models, consensus problem.

1. Introduction

A system is usually defined as a "thing" that interacts with its environment in a discrete fashion across a well-defined boundary (called an *interface*) [12]. Specifications for systems identify the intended behavior at the interface. If a system S exhibits only the kind of behavior permitted by some specification M, then we say that "S is correct regarding M". In ideal and fault-free environments we call M the *ideal problem specification*.

There are different notions of correctness when fault-tolerance issues are involved. A necessary starting point when exploring any of these notions is always an assumption about the faults that may happen within or outside of the system. A description of faulty behavior of a class of systems is commonly referred to as a *failure model*. When "applied" to a system, a failure model completely determines the system's behavior in faulty environments. In some cases, a system can still satisfy its ideal problem specification even when subcomponents follow some failure model. This leads to a first notion of correctness in faulty environments, i.e., that of *masking fault-tolerance*, meaning that the effects of faults are transparently handled at the system interface. It is however obvious that the ideal problem specification cannot be satisfied if a sufficiently hostile failure model is chosen. So different notions of correctness in faulty environments refer to a different and necessarily weaker problem specification which we call a *fault-tolerance specification* (or simply *tolerance specification*).

A system can be thought of as a hardware or software module (or *program* for short) which is supposed to implement a given specification. We take the view that programs and specifications are both formulas of some temporal logic, so programs are also specifications, yet very low level ones. At the program level, the effects of a failure model are concrete and have been well-studied. At this level, some specific failure model can be "implemented" by augmenting the source code in such ways that the resulting program mimics the behavior of the failure model. Thus, generally a failure model can be viewed as a program transformation F, i.e., a function that maps programs to programs [7]. Given some program S, then $S' = F(S)$ is the program under failure model F.

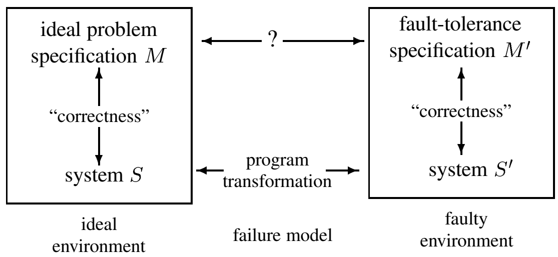

Figure 1. The relationship between ideal and fault-tolerance specifications.

While these topics are largely understood at the program level, the effects of failure models at the abstract specification level remain largely uncharted. This is depicted in Figure 1. Given a specific ideal problem specification and a failure model there are usually many (weaker) tolerance specifications for that problem. In this paper we take a first attempt to study the effects of specific failure models on ideal problem specifications. A better understanding of the issues involved may enable the systematic derivation of fault-tolerance specifications, thus supporting modular and formal verification of fault-tolerant systems. Our study is based on an unusual way to describe failure models at the specification level and is arranged along the lines of an example. We constrain our attention to the well-known *crash* failure model and discuss how it may influence system properties.

*This work was supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the "Graduiertenkolleg ISIA" at Darmstadt University of Technology. For an extended version of this paper the reader is refered to a previously published technical report [8].

2. What Exactly is a (Fault-Tolerance) Specification?

When specifying reactive systems it is necessary to distinguish between the system and its environment. The environment consists of all "things" that have access to the interface of the system. These may be other systems or simply a (human) user. A specification of a system S asserts that S will guarantee a property M under the assumption that the environment guarantees some property E.

Of course, systems may be constructed using many subsystems, each of which having its own interface and its own specification. For example, in a distributed system consisting of n processes $p_1, \ldots, p_n$ that communicate via a communication subsystem each process as well as the communication mechanism form systems in the sense defined above. Each system is part of the environment of the other. In the specification of a process p_i for example, E will define what p_i can expect from the communication subsystem (i.e., the type and order of "messages" received) and M will describe how p_i reacts (i.e., what "messages" it will send). Taking all processes and the communication subsystem together yields a larger system. Specifications for such larger distributed systems define what task should be solved by coordinated actions of its subcomponents. These types of specifications are those which interest us most. However, reasoning about them involves reasoning about the properties of subcomponents.

But what exactly is a property? Here we take a state-based approach to define the corresponding terms. A *state* is some assignment of values to the "variables" of an interface. We assume that a state completely describes the state of the interface at some instant. An *execution* of a system is a sequence of states. A *property* is a set of executions. A system by itself defines a property, which is the set of all executions possibly observable at the interface. We say that a property P *holds* for a system S if the set of executions defined by S is contained in P. We sometimes call a state change at the interface an *action*. State changes at the interface are either initiated by the system or by the environment. This can be viewed as the distinction between input and output variables (i.e., input variables are written by the environment and only read by the system, output variables are defined analogously).

Lamport identified two main types of properties called *safety* and *liveness* [11]. A safety property informally states that some bad thing will never happen. Intuitively, if a property P is finitely refutable then P is a safety property. Partial correctness or diagrams of finite state automata are examples of safety properties. On the other hand, a liveness property informally states that some good thing will eventually happen. Termination and guaranteed service are examples of liveness properties.[1]

Recalling the informal exposition above, a specification formally is the property $E \Rightarrow M$, which is the set of all executions that are in M or not in E. A fault-tolerance specification is usually obtained by weakening either E or M (or both).[2] Weakening E resembles the anticipation of abnormal behavior of the environment, whereas weakening M indicates that the system itself will sometimes deviate from its original correctness specification.

3. High-Level Failure Specifications

As mentioned in the introduction, a *failure model* describes the manner in which system or subcomponents of a system may fail. It is always tied to a specific system (or class of systems). In this section, we will use as an example the *crash* failure model, which is usually tied to systems resembling concurrent processes. In the crash failure model, the system fails simply by halting [9].

Völzer [13] observed that failure models formally consist of two distinct parts: (1) the *impact model*, a specification of the additional faulty behavior of the system, and (2) the *rely specification*, a specification of assumptions that restrict the set of possible system executions (e.g., "no more that t nodes may crash"). The first part is responsible for weakening the ideal specification of some system: through the added behavior more executions become possible. Usually, this involves augmenting both safety and liveness properties of the given system. The second part can be seen as bounding the additional behavior from becoming too bad. This can be seen as requiring a safety property for the entire system. It seems as if both parts are always needed. However, the rely specification usually is a global assumption on a composed system.

3.1. Characterizing Crash in Terms of Safety and Liveness

Now consider the crash failure model, let p_i be a process that may crash and let $E_i \Rightarrow M_i$ be the correctness specification that p_i satisfies in fault free operations. Process p_i has some fixed but unknown set of states. Now, we add a special "crashed" state c to this set and characterize the additional behavior in terms of safety and liveness as follows:

- (safety) p_i operates according to $E_i \Rightarrow M_i$ unless it has reached c, and p_i cannot leave c (i.e., the successor state of c is again c), and
- (liveness) eventually p_i will reach c.

The "safety" properties of the crash failure model ensures that the crash "does nothing bad", i.e., it does not cause the system to behave in more benign or more malevolent ways. The liveness property ensures that the crash will eventually occur and is important, since the safety property alone would

[1] We omit formal definitions of safety and liveness for brevity. The interested reader is referred to the work of Alpern and Schneider [2].

[2] With the given definition of a property, the term 'weakening' refers to adding executions to a property. Consequently, the value false (designating the weakest property) is equivalent to the set of *all* executions and true is the empty set.

be meaningless. We call this interpretation of crash in terms of safety and liveness "unusual" because for example the usual informal understanding of, for example, liveness as "something good will eventually happen" needs a little twist of mind if the good thing is a crash. Note however, that the safety and liveness conditions are not "stand-alone" properties in the actual sense defined above because they obviously refer to existing properties of a given system (here p_i).

This type of description captures only the impact model and not the rely specification of crash. The rely specification is an assertion about a system composed of crashing processes. For the crash failure model it usually asserts that the number of processes allowed to crash does not exceed some given value $t < n$.

3.2. Effects on Original Specifications

What effects can a failure model like the one defined in the previous section have on the specification $E_i \Rightarrow M_i$ of process p_i? As shown by Alpern and Schneider [2], it is possible to write M_i as the intersection of a safety property M_S and a liveness property M_L. Let's look at the safety property first.

The crash failure model asserts that p_i never leaves the crash state c if it ever enters it. This weakens the original safety property to yield a larger set M'_S which additionally contains all executions that also play to this new rule. However, observe that the original property M_S cannot be violated since the safety condition of crash prohibits anything from happening. (A violation of M_S would only be possible if M'_S would contain a reference to c. But c is added after M_S is specified.)

Now consider p_i's liveness property M_L. It is weakened to additionally allow all runs in which the crash action eventually happens. For example, if M_L requires p_i to eventually perform some action a, then the modified liveness property M'_L contains all runs in which eventually a happens *and* eventually the crash action happens. The intersection of M'_S and M'_L results in the "desired" behavior of a crashing node.

Obviously, the crash failure model applied to p_i cannot directly influence E_i. The only way in which it can take effect on E_i is by viewing the environment of p_i as consisting of equally constructed processes that themselves are subject to crash failures. Hence, the environment assumption of p_i is related to the system guarantees M_j of the neighboring processes in the distributed system. At the moment we do not know what form E_i has, apart from the fact that it should be a safety property.[3]

4. Deriving Fault-Tolerance Specifications

In this section we study how tolerance specifications can be derived from an ideal problem specification and a failure model. We use the well-studied problem of *consensus* [4] as an example.

4.1. Consensus

As we are in a state-based environment, we will define the consensus problem using two process variables π_i and δ_i where π_i is the *propose variable* at process p_i and δ_i is the *decision variable* at p_i. Both can take any value v from a finite set V of decision values. If some value v is placed into π_i, we say that p_i proposes v. Dually, if p_i places some value v in δ_i then we say that p_i decides v. We assume that processes propose and decide at most once and that they must propose a value before they can decide.

An algorithm that solves consensus must fulfill the following three correctness conditions:

T1 (termination) eventually every process decides,

A1 (agreement) no two processes decide different values, and

V1 (validity) the decision value must have been proposed by at least one process.

In contrast to termination (which is a liveness property), agreement and validity are both safety properties; agreement ensures the actual consensus while the validity condition rules out trivial algorithms where processes do not communicate.

4.2. Degraded Consensus Specifications

In terms of the framework described in Section 2 the consensus problem of the previous section is a specification $E \Rightarrow M$ of a system composed of multiple processes. The interface between this system and its environment consists of the input variables π_i and the output variables δ_i. The environment assumption E therefore contains the assumption that, if some value v is written to some variable π_i then $v \in V$. The system property M contains a formal description of the three correctness conditions of consensus from Section 4.1.

How may $E \Rightarrow M$ be modified to account for the fact that subcomponents may crash? A first approach would be to compose the specifications of the subsystems to a specification of the entire system and study the differences to the ideal problem specification. However, this requires knowledge of the exact subcomponent specifications. But as we are not concerned with a particular protocol here, we do not see how this approach can be of use to us.

Another approach starts with the ideal problem specification. By means of some heuristics we can then weaken the ideal safety and liveness properties to yield a set of degraded problem requirements that can be composed to weaker problem specifications.

Termination. Termination is a liveness property and requires *all* processes to eventually decide. However, the interplay between the modified process properties M'_S and M'_L

[3] A result of Abadi and Lamport states that under "normal" circumstances environmental assumptions consist merely of safety properties [1, Theorem 1].

may lead to a situation where processes may never decide. Thus, T1 must be weakened.

The rely specification of crash restricts the number of crashing processes to be at most t (where $t < n$). This is a safety requirement on the composed system because we can tell in finite time when it is violated. The rely specification rules out all executions in which more than t processes enter the crashed state. So the intersection between the rely specification and all of the processes's system guarantees yields exactly the following liveness condition:

T2 eventually every process that does not crash decides.

Agreement and Validity. We have seen that the crash failure model does not augment the safety properties of the processes. As the conjunction of these safety properties is again a safety property we can follow that safety properties that were satisfied before are still satisfied under the crash failure model. Because agreement and validity are both safety properties we can deduce that they are still valid and need no weakening.

A degraded specification of consensus under the crash failure model therefore reduces to the conditions T2, A1 and V1. Note that this version of the consensus problem is similar to *uniform consensus* since processes are required to decide uniformly (i.e., a restriction is placed even on those processes which eventually crash) [9]. We conjecture that this is the strongest form of consensus achievable under the crash failure model and it seems as though the methodology presented in this paper allows to prove this (future work must investigate this problem further.) The *non-uniform* version of consensus (where processes that eventually crash are allowed to decide differently from correct processes) [9] is weaker than uniform consensus.

5. Discussion

A first reference to work in the area of producing degraded specifications is a paper by Echtle [6]. He reduces the environment assumptions of a process to the predicate false, thus modeling worst case (Byzantine) behavior. Later, Cristian advocated the precise formulation of failure semantics in a widely circulated article appearing in *Communications of the ACM* [5]. Independently, Herlihy and Wing pioneered the area of degraded specifications [10]; in their *relaxation lattice method* [10] they show how to obtain a set of "degraded" specifications by weakening the requirements (or as they call *constraints*) that constitute the ideal problem specification. This resulting set of specifications was shown to form a lattice with the ideal specification at the top and the "empty" specification at the bottom. However, it was not clear how to generally derive these constraints. This is what interests us.

Based on the division of system properties into safety and liveness, Arora and Kulkarni have suggested to derive these constraints by either disregarding liveness but preserving safety, or by "temporarily" disregarding safety but preserving liveness [3]. Here, we have tried to combine both approaches in a case study. An interesting point that led to the derivation of termination requirement T2 was that it was the result of conjoining properties of subcomponents with the rely specification of the failure model. This shows that derivation recipies for degraded specifications can go beyond the use of application-dependent heuristics and even have some potential of automation.

References

[1] M. Abadi and L. Lamport. Composing specifications. *ACM Trans. on Progr. Lang. and Systems*, 15(1):73–132, Jan. 1993.

[2] B. Alpern and F. B. Schneider. Defining liveness. *Inf. Proc. Letters*, 21:181–185, 1985.

[3] A. Arora and S. S. Kulkarni. Component based design of multitolerant systems. *IEEE Trans. on Softw. Eng.*, 24(1):63–78, Jan. 1998.

[4] M. Barborak, A. Dahbura, and M. Malek. The consensus problem in fault-tolerant computing. *ACM Computing Surveys*, 25(2):171–220, June 1993.

[5] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, Feb. 1991.

[6] K. Echtle. Fehlermodellierung bei Simulation und Verifikation von Fehlertoleranz-Algorithmen für Verteilte Systeme. In F. Belli, S. Pfleger, and M. Seifert, editors, *Software-Fehlertoleranz und -Zuverlässigkeit*, number 83 in Informatik-Fachberichte, pages 73–88. Springer-Verlag, 1984.

[7] F. C. Gärtner. Specifications for fault tolerance: A comedy of failures. Technical Report TUD-BS-1998-03, Dept. of Comp. Science, Darmstadt University of Technology, Darmstadt, Germany, Oct. 1998.

[8] F. C. Gärtner. An exercise in systematically deriving fault-tolerance specifications. Technical Report TUD-BS-1999-01, Dept. of Comp. Science, Darmstadt University of Technology, Darmstadt, Germany, Mar. 1999.

[9] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Cornell University, Comp. Science Dept., May 1994.

[10] M. P. Herlihy and J. M. Wing. Specifying graceful degradation. *IEEE Trans. on Par. and Distr. Systems*, 2(1):93–104, Jan. 1991.

[11] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. on Softw. Eng.*, 3(2):125–143, Mar. 1977.

[12] L. Lamport. A simple approach to specifying concurrent systems. *Communications of the ACM*, 32(1):32–45, Jan. 1989.

[13] H. Völzer. Verifying fault tolerance of distributed algorithms formally: An example. In *Proc. of the Int. Conf. on Application of Concurrency to System Design (CSD98)*, pages 187–197, Fukushima, Japan, Mar. 1998. IEEE Comp. Soc. Press.