Redundancy in Space in Fault-Tolerant Systems

Felix C. Gärtner
Technische Universität Darmstadt
Fachbereich Informatik
D-64283 Darmstadt, Germany
felix@informatik.tu-darmstadt.de

Hagen Völzer
Humboldt-Universität zu Berlin
Institut für Informatik
D-10099 Berlin, Germany
voelzer@informatik.hu-berlin.de

Darmstadt University of Technology
Department of Computer Science
Technical Report TUD-BS-2000-06

July 20, 2000

Abstract

A formal definition of *redundancy in space* is proposed which captures intuitive concepts such as hardware and software redundancy. We then consider the problem of constructing a fault-tolerant program p' from a fault-intolerant program p. We prove that p' is fault-tolerant with respect to a safety specification only if it is redundant in space. We discuss the assumptions of our result and the consequences to the design of fault-tolerant systems.

Keywords:

fault-tolerance, safety, redundancy, transition systems, theory

Contents

1 Introduction

While the methodologies for achieving fault tolerance are quite diverse, researchers have achieved an unspoken consensus that fault tolerance can only be achieved by employing some form of *redundancy*. What redundancy is and in what forms it manifests itself in computing systems is however not uniformly accepted.

For example, consider the standard fault-tolerance mechanism of *triple modular redundancy* (TMR). In TMR, a critical component is triplicated and all three units work in parallel. The outputs of the three components are fed into a voter which takes the majority of its three inputs (see Figure 1) [9]. This mechanism allows to tolerate transient and permanent faults in at most one of the three components. The redundancy employed here is obvious and usually termed *hardware redundancy* [5].

But a similar technique can be applied in software too. For example, in the ISIS group communication system [6], a critical process is spawned several times forming a process group. Before responding to a request, the group tries to reach a consensus on what the reply should be. Thus, transient or permanent faults in a subset of processes can be tolerated. This is usually termed *software redundancy*. Schneider's state machine approach [13] employs similar methods. Obviously, both notions of hardware and software redundancy have the basic mechanism in common. But what exactly constitutes this mechanism?

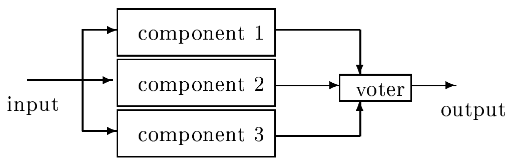

Figure 1: Triple Modular Redundancy (TMR).

Consider a different example from the area of data communications. Here, a parity bit is often added to a transmitted byte in order to detect single bit errors during transmission. Upon a mismatch in parity, the system retransmits the previous message. This technique helps to tolerate transient faults on the communication line, e.g., caused by excessive noise. The type of redundancy employed here is usually called *redundancy in time* [5]. It usually means that a fault-tolerant computation may take longer if faults occur. Redundancy in time seems to be fundamentally different than hardware and software redundancy, but what are exactly the differences between them? Moreover, the parity bit which was employed in the retransmission example, is a standard error detection method from coding theory, and coding theory has its own set of redundancy definitions. How do all these forms of redundancy exactly relate?

The present work tackles some of these questions. We propose a definition of redundancy which is based on a general formal system model and which arguably captures the essence behind the notions of hardware and software redundancy. More precisely, we study the following question: Given a fault-intolerant program p_1, i.e., one which violates some correctness specification *SPEC* in the presence of faults; what form of redundancy is inherent in a "fault-tolerant version" p_2 of p_1? Our result tackles the case where *SPEC* is a safety specification. In this case we show that the fault-tolerant version p_2 contains *redundant states*. We define redundant state to be a state which is not reachable if no faults occur. Although this result is easy to believe, a formal proof depends on quite a few assumptions. By discussing these assumptions we hope to contribute to a theory of redundancy which we believe to be central in a theory of fault-tolerant computing. And although we do not do this here, with our definition it is possible to define measures of the "redundancy-ness" of programs and compare two programs in this respect.

Some of the ideas presented here have appeared in prior work by the first author [7] who discusses them within a general survey of fault tolerance methodologies. However, the presentation in [7] is informal so that some definitions remain imprecise and misleading. In other related work, Arora and Kulkarni [3, 4] have proposed a technique for achieving fault tolerance which operates under similar preconditions. They show that to be fault tolerant with respect to a safety property it is necessary to use a program component which they call a *detector*. Combined with our result follows that detectors are redundant in space. Hence, our work can be seen as an explanation as to *why* detectors achieve their goal. To the best of our knowledge we know of no other related work which has attempted to provide formal definitions of redundancy in the context of fault-tolerant computing.

The paper is structured as follows: We start by defining the system model used throughout this paper in Sect. 2. In Sect. 3 we develop a sequence of definitions and lemmas which lead to our definition of redundancy in space and allow to prove that redundancy in space is necessary to build fault-tolerant programs which maintain a safety property. We conclude in Sect. 4.

2 Formal Preliminaries

In this section we define the formal system model used throughout this paper. It is partly based on the system model of the fault-tolerance theory of Arora and Kulkarni [3].

2.1 States, Traces and Properties

The state space of a program is modeled as a countable nonempty set C of states. A *state predicate over some state set* C is a boolean predicate over C. An equivalent way of specifying a state predicate over C is giving a subset of C. A *state transition over some state set* C is a pair (r, s) of states from C. A *state transition set over* C is a set of state transitions over C.

In the following, let C be a state set and T be a state transition set. We define a *trace over* C to be a non-empty sequence $s_1, s_2, s_3, \ldots$ of states over C. We sometimes use the notation s_i to refer to the i-th element of a trace. Note that traces can be finite or infinite. A trace is *finite* if its length is finite.

We will always use greek letters to denote traces and normal lowercase letters to denote states. For a trace $\sigma = s_1, s_2, \ldots$ we use $|\sigma|$ to denote the length of σ and $\sigma|_i$ to denote the prefix of length i of σ (i.e., the trace $s_1, s_2, \ldots, s_i$). For two traces α and β, we write $\alpha \cdot \beta$ to mean the concatenation of the two traces. We say that a transition t *occurs* in some trace σ if there exists an i such that $(s_i, s_{i+1}) = t$.

We define a *property over* C to be a set of traces over C. A trace σ *satisfies* a property P iff $\sigma \in P$. If σ does not satisfy P we say that σ *violates* P. There are two important types of properties called *safety* and *liveness* [2]. Informally spoken, a safety property demands that "something bad never happens" [10], i.e., they rule out a set of unwanted trace prefixes. Mutual exclusion and deadlock freedom are two prominent examples of safety properties. A liveness property on the other hand demands that "something good will eventually happen" [10] and can be used to formalize, e.g., notions of termination. Safety properties are defined as follows.

Definition 1 (safety property over C) *A* safety property S over C *is a property over* C *for which the following holds: For each trace* σ *which violates* S *there exists a prefix* α *of* σ *such that for all traces* β*,* $\alpha \cdot \beta$ *violates* S*.*

In this paper we are mainly concerned with safety properties and not with liveness. In the context of fault tolerance liveness properties are often treated by help of safety properties in the following sense: the set of states is subdivided into a set of "good" states and a set of "bad" states. If the system is in a good state liveness is guaranteed. However, the property of being in a good state is a safety property.

2.2 Programs and Programming Notation

Programs are often defined as transition systems by giving a state set C, a set of initial states I and a transition relation T over C.

Definition 2 (program) *A* program p *is a tuple* (C, I, T)*, where* C *is a state set,* I *is a non-empty subset of* C *(the set of initial states of the program) and* T *is a state transition set over* C*.*

A program p in itself describes a property, i.e., set of traces over C. By slight abuse of notation we sometimes write $\sigma \in p$, i.e., we use the same symbol for the program and the property which it represents. The state predicate I together with the state transition set T describe a safety property S, i.e., all traces which are constructable by starting in a state in I and using only state transitions from T. Often, the definition of a program also contains an additional property which states that the program eventually makes progress. Because we are only interested in the safety properties of a program we omit this item here.

We use a state-chart like notation to represent programs, i.e., states are drawn as circles and transitions are arrows between states. Initial states are attributed with an asterisk.

2.3 Specifications and Correctness

Following Arora and Kulkarni [4], we assume that problem specifications are fusion closed. Informally spoken, fusion closure states that the entire history of every trace is present in every state of the trace. This allows us to simplify the proofs by reasoning about states rather than state sequences. Fusion closure is defined as follows:

Let C be a state set, $s \in C$, X be a set of sequences of states over C, α, γ finite state sequences, and β, δ, σ be state sequences over C.

Definition 3 (fusion closure) *The set X is* fusion closed *if the following holds: If $\alpha \cdot s \cdot \beta$ and $\gamma \cdot s \cdot \delta$ are in X then $\alpha \cdot s \cdot \delta$ and $\gamma \cdot s \cdot \beta$ are also in X.*

Following Arora and Kulkarni [4, p. 75] we also argue that fusion closed specifications are not restrictive since every specification which is not fusion closed can be transformed into an equivalent fusion closed specification by using *history variables*. We will study the precise consequences of this assumption below.

Note that Arora and Kulkarni [4] also assume that their specifications are suffix closed. With this assumption they can define a program as consisting only of a state space, a transition relation, and a fairness property (i.e., they omit the set of initial states). They argue that this eases many proofs of program correctness [4, p. 65]. Since we prefer proofs involving initial states of programs, we do not require specifications to be suffix closed.

Definition 4 (specification over C) *A* specification over C *is a property over C which is fusion closed.*

A safety specification is a specification which is a safety property.

Let *SPEC* be a specification and p be a program over C. We say that p *satisfies SPEC* iff all traces in p satisfy *SPEC*. Consequently, we say that p *violates SPEC* iff there exists a trace $\sigma \in p$ which violates *SPEC*.

Definition 5 (maintains) *Let α be the prefix of a trace in p and let SPEC be a specification. We say that α* maintains *SPEC iff there exists a trace β such that $\alpha \cdot \beta \in$ SPEC.*

The notion of maintaining a specification can be used to define a safety property in the following way.

Proposition 1 *S is a safety property iff $\sigma \notin S \Leftrightarrow$ there exists a prefix α of σ such that α does not maintain S.*

A trace σ maintains a specification iff all its prefixes maintain the specification.

Proposition 2 *Let σ be a finite trace and SPEC be a specification. The following two statements are equivalent:*

1. σ maintains SPEC.

2. All prefixes of σ maintain SPEC.

Proofs are written in a structured style similar to proof trees of interactive theorem proving environments. This approach is advocated by Lamport who promises that this style "makes it much harder to prove things that are not true" [11]. The proof is a sequence of numbered steps at different levels. Every step has a proof which may be refined at lower levels by additional steps. For example, step $\langle 1 \rangle 2.$ is the second step on level 1. Proofs may also be read in a structured way, for example, by reading only the top level steps and going into sublevels only when necessary.

Proof sketch: The proof mainly follows from the definition of "maintains". Note that all steps in the proof are equivalences.

1 $\langle 1 \rangle 1.$ σ maintains *SPEC*.
Proof: From item 1. □

2 $\langle 1 \rangle 2.$ $\exists \delta . \sigma \cdot \delta \in \mathit{SPEC}$
Proof: From Definition of "maintains". □

3 $\langle 1 \rangle 3.$ $\exists \delta . \alpha \cdot \beta \cdot \delta \in \mathit{SPEC}$
Proof: Write σ as $\alpha \cdot \beta$. □

4 $\langle 1 \rangle 4.$ $\exists \gamma . \alpha \cdot \gamma \in \mathit{SPEC}$
Proof: Write $\beta \cdot \delta$ as γ. □

5 $\langle 1 \rangle 5.$ α maintains *SPEC*.
Proof: From definition of "maintains". □

6 $\langle 1 \rangle 6.$ Q.E.D.
Proof: Since α can be any prefix, all prefixes of σ maintain *SPEC*. □

We now describe the consequences of assuming that specifications are fusion closed. If a specification is fusion closed, it is possible to identify a set of "bad" transitions, i.e., transitions which inevitably lead to a violation of the specification. This is formalized in Lemma 1.

If *SSPEC* is a safety property, due to Proposition 1 every trace not in *SSPEC* has a prefix which does not maintain *SSPEC*. From the definition of maintains we have that there must be a transition where σ switches from "good" to "bad", i.e., σ can be written as $\alpha \cdot d \cdot b \cdot \beta$ such that $\alpha \cdot d$ maintains *SPEC* and all "longer" prefixes (starting with $\alpha \cdot d \cdot b$) do not maintain *SPEC*. We now show that (d, b) is a transition which will cause any trace in which it occurs to violate *SPEC*. (A similar result was proven by Arora and Kulkarni [3, "Only-if" part of Lemma 3.2].)

Lemma 1 *Let $p = (C, I, T)$ be a program, SSPEC be safety property which is fusion closed and assume that p violates SSPEC and that for all $x \in I$ holds that x maintains SSPEC. Then there exists a transition $(d, b) \in T$ such that for all traces σ of p: if (d, b) occurs in σ then $\sigma \notin$ SSPEC.*

Assume: 1. *SSPEC* is a safety property which is fusion closed,

2. $p = (C, I, T)$ is a program which violates *SSPEC*, and

3. $\forall x \in I$ holds that x maintains *SSPEC*.

PROVE: There exists a transition $(d, b) \in T$ such that for all traces $\sigma \in p$: if (d, b) occurs in σ then $\sigma \notin SSPEC$.

PROOF SKETCH: For a given trace, the existence of a transition where that trace turns from good to bad follows from the fact that *SSPEC* is a safety property. It remains to show that whenever such a transition occurs in a trace σ, then σ does not maintain *SSPEC*.

1 $\langle 1 \rangle 1$. There exists a trace $\sigma \in p$ which is not in *SSPEC*.
PROOF: From assumption 2 and definition of "violates". $\Box$

2 $\langle 1 \rangle 2$. There exists a prefix $\tilde{\alpha}$ of σ which does not maintain *SSPEC*.
PROOF: From step $\langle 1 \rangle 1$, the fact that *SSPEC* is a safety property, and Proposition 1. $\Box$

3 $\langle 1 \rangle 3$. $\tilde{\alpha}$ can be written as $\alpha \cdot d \cdot b \cdot \beta$ such that $\alpha \cdot d$ maintains *SSPEC* and all prefixes of $\tilde{\alpha}$ starting with $\alpha \cdot d \cdot b$ do not maintain *SSPEC*.
PROOF: Assumption 3 implies that the initial state of a trace maintains *SSPEC*. The remainder of this step follows from step $\langle 1 \rangle 2$ and the fact that *SSPEC* is a safety property. $\Box$

4 $\langle 1 \rangle 4$. Take some other trace ρ of p in which (d, b) occurs, i.e., $\rho = \hat{\alpha} \cdot d \cdot b \cdot \hat{\beta}$. Trace ρ is not in *SSPEC*.

4.1 $\langle 2 \rangle 1$. ASSUME: ρ is in *SSPEC*.
PROVE: false

4.1.1 $\langle 3 \rangle 1$. All prefixes of ρ maintain *SSPEC*.
PROOF: From the assumption that *SSPEC* is a safety property and Proposition 1. $\Box$

4.1.2 $\langle 3 \rangle 2$. $\hat{\alpha} \cdot d \cdot b$ maintains *SSPEC*.
PROOF: From step $\langle 3 \rangle 1$ and the fact that $\hat{\alpha} \cdot d \cdot b$ is a prefix of ρ. $\Box$

4.1.3 $\langle 3 \rangle 3$. $\exists \hat{\delta} : \hat{\alpha} \cdot d \cdot b \cdot \hat{\delta} \in SSPEC$.
PROOF: From step $\langle 3 \rangle 2$ and the definition of "maintains". $\Box$

4.1.4 $\langle 3 \rangle 4$. $\exists \delta : \alpha \cdot d \cdot \delta \in SSPEC$.
PROOF: From step $\langle 1 \rangle 3$ ($\alpha \cdot d$ maintains *SSPEC*) and definition of "maintains". $\Box$

4.1.5 $\langle 3 \rangle 5$. $\alpha \cdot d \cdot b \cdot \hat{\delta} \in SSPEC$.
PROOF: From steps $\langle 3 \rangle 3$ and $\langle 3 \rangle 4$ and fusion closure of *SSPEC*. $\Box$

4.1.6 $\langle 3 \rangle 6$. $\alpha \cdot d \cdot b$ maintains *SSPEC*.
PROOF: From step $\langle 3 \rangle 5$ and the definition of "maintains". $\Box$

4.1.7 $\langle 3 \rangle 7$. Q.E.D.
PROOF: The result of step $\langle 3 \rangle 6$ contradicts step $\langle 1 \rangle 3$ (that $\alpha \cdot d \cdot b$ does not maintain *SSPEC*). $\Box$

4.2 $\langle 2 \rangle 2$. Q.E.D.
PROOF: The step follows from step $\langle 2 \rangle 1$ and proof by contradiction. $\Box$

5 $\langle 1 \rangle 5$. Q.E.D.
PROOF: The Lemma follows directly from step $\langle 1 \rangle 4$. $\Box$

3 Fault-tolerant Systems and Redundancy in Space

Programs with trace semantics and fusion closed specifications as introduced in the previous section provide the environment in which we study the question of what type of redundancy is necessary to maintain safety specifications in the presence of faults. In this section we formalize the notions involved.

In the previous section we have stated what it means for a program to satisfy or violate a specification. Given some program p_1 our goal is to define the notion of a fault-tolerant version p_2 of p_1 meaning that p_2 does exactly what p_1 does in fault-free scenarios and has additional fault-tolerance abilities which p_1 lacks. To state that some program p_2 has the same behavior as another program p_1 we now introduce the notion of an *extension.*

3.1 Extensions

Let C_1 and C_2 be two state sets. We call a mapping $\varphi : C_2 \to C_1$ a *state projection.* If $c_1 \in C_1$, $c_2 \in C_2$ and $\varphi(c_2) = c_1$, we say that c_1 and c_2 *φ-correspond.* The definition of a state projection can be easily extended to traces in the following way: Let $\sigma = s_0, s_1, s_2, \ldots$ be a trace, then $\varphi(\sigma) = \varphi(s_0), \varphi(s_1), \varphi(s_2), \ldots$. A state projection applied to a property p is taken to yield the set of all φ-corresponding traces of p. Analogously, for a property P we let $\varphi(P)$ mean the set $\{\varphi(\sigma) | \sigma \in P\}$.

Definition 6 (extends) *Let* $p_1 = (C_1, I_1, T_1)$ *and* $p_2 = (C_2, I_2, T_2)$ *be two programs. Program* p_2 extends *program* p_1 *through state projection* $\varphi : C_2 \to C_1$ *iff the following two conditions hold:*

1. $\varphi(I_2) = I_1$
2. $\varphi(T_2) = T_1$

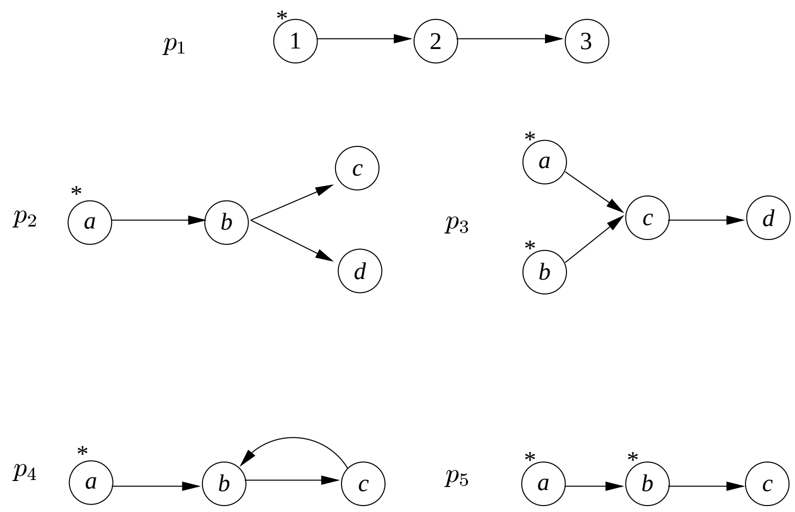

Figure 2: Examples of programs which extend and do not extend p_1

Figure 2 shows at the top a simple program p_1 and several other programs. The state projection is simply a vertical projection, i.e., in p_2 the states c and d both map to the state 3 of p_1 and in p_3 the states a and b both map to state

1 of p_1. While programs p_2 and p_3 both extend p_1, programs p_4 and p_5 do not extend p_1.

Note that our notion of extension is similar to the notion of *refinement* [1]: p_2 refines p_1 iff $\varphi(p_2) \subseteq p_1$. Choosing equality instead of the subset relation can be viewed as "refinement at the same level of abstraction".

Since p_1 and p_2 can also be used to denote the safety property represented by the respective programs it is relatively easy to prove the following proposition:

Proposition 3 *If p_2 extends p_1 through φ then $\varphi(p_2) = p_1$.*

It is important that the implication from Proposition 3 cannot be strengthened to an equivalence. This is because $\varphi(S_2) = S_1$ refers only to those states which are reachable from initial states and our definition of extends also includes possible non-reachable states.

3.2 Failure Models

Since we are concerned with fault tolerant systems we must have a way of modeling faulty behavior. We define a failure model F as being a program transformation [8], i.e., a mapping F from programs to programs. The resulting program is called the *fault-affected version* and, in our notation, is always primed, i.e., F maps a program $p = (C, I, T)$ to a program $p' = (C', I', T')$. The behavior of p' is defined by a state projection π. As there can be more than one projection for the same type of behavioral change, we make π part of the definition of F. To be reasonable, F must retain the full original behavior of p, i.e., $p \subset \pi(p')$. We call this the *enlargening behavior property* of a failure model. Also, we require that a failure model does not tamper with the set of initial states, i.e., we rule out "immediate" faults that occur before the system is switched on. We call this the *initial-state preservation property.* Since it is common to assume that faults take effect somewhere within the computation of a program this is not a big restriction.

Let $\mathcal{P}_C$ be the set of all programs over C and $\Pi_{C' \to C}$ be the set of all state projections from C' to C.

Definition 7 (failure model) *A* failure model *F is a mapping $F : \mathcal{P}_C \to \mathcal{P}'_C \times \Pi_{C' \to C}$ such that the following two conditions hold:*

1. *(enlargening behavior) if $F(p) = (p', \pi)$ then $p \subset \pi(p')$, and*
2. *(initial-state preservation) if $F(p) = (p', \pi)$ then $I = \pi(I')$.*

Restricting ourselves to safety specifications alone has an important implication: it also allows us to restrict our attention to a class of faults which we call *added transitions.*

Definition 8 (added transition) *Let F be a failure model, $p = (C, I, T)$ and $p' = (C', I', T')$ be programs and $F(p) = (p', \pi)$. We say that F* adds transition *$t' \in T'$* to *p iff $\pi(t') \notin T$.*

If the fault-affected version p' of a program p violates a safety property which originally was satisfied by p, then this must be due to added transitions.

Lemma 2 *Let F be a failure model, $p = (C, I, T)$ and $p' = (C', I', T')$ be programs, $F(p) = (p', \pi)$ and SSPEC be a safety specification. If p satisfies SSPEC and p' violates SSPEC then F adds at least one transition to p.*

ASSUME: 1. F is a failure model, $p = (C, I, T)$ and $p' = (C', I', T')$ are programs such that $F(p) = (p', \pi)$.

2. *SSPEC* is a safety specification.

3. p satisfies *SSPEC*.

4. p' violates *SSPEC*.

PROVE: There exists a transition $t' \in T'$ such that F adds t' to p.

PROOF SKETCH: The proof is straightforward: We assume that F adds no transitions to p and derive a contradiction by showing that every transition which is possible in p' is equally possible in p.

1 $\langle 1 \rangle 1$. ASSUME: F adds no transition, i.e., for all transitions $t' \in T'$ holds that $\pi(t') \in T$.
PROVE: False

1.1 $\langle 2 \rangle 1$. There exists a finite trace $\sigma' \in p'$ such that $\pi(\sigma') \notin SSPEC$.
PROOF: From the fact that p' violates *SSPEC* and that *SSPEC* is a safety property. $\square$

1.2 $\langle 2 \rangle 2$. For all transitions t' occurring in σ' there exists a transition $t \in T$ such that $\pi(t') = t$.
PROOF: From step $\langle 2 \rangle 1$ and the assumption of $\langle 1 \rangle 1$. $\square$

1.3 $\langle 2 \rangle 3$. $\sigma = \pi(\sigma')$ is a trace of p.
PROOF: From step $\langle 2 \rangle 2$. $\square$

1.4 $\langle 2 \rangle 4$. p violates *SSPEC*.
PROOF: From step $\langle 2 \rangle 3$. $\square$

1.5 $\langle 2 \rangle 5$. Q.E.D.
PROOF: Step $\langle 2 \rangle 4$ contradicts the fact that p satisfies *SSPEC*. $\square$

2 $\langle 1 \rangle 2$. Q.E.D.
PROOF: Follows from step $\langle 1 \rangle 1$ and proof by contradiction. $\square$

In our notation we will depict added transitions as dashed arrows.

3.3 Fault-Tolerant Versions and Conservative Failure Models

Now we are able to define a central concept of this paper, that of a *fault-tolerant version*. It captures the idea of starting with some program p_1 which is not fault tolerant regarding a specification *SPEC* and some failure model F. A fault-tolerant version p_2 of p_1 is a program which has the same behavior as p_1 if no faults occur, but additionally satisfies *SPEC* in the presence of faults.

Definition 9 (fault-tolerant version) *Let F be a failure model, SPEC be a specification and p_1 and p_2 be programs. Let $F(p_1) = (p'_1, \pi_1)$ and $F(p_2) = (p'_2, \pi_2)$ and assume that p_1 satisfies SPEC but $\pi_1(p'_1)$ violates SPEC. We call* *a program p_2 the* F-tolerant version *of program p_1 for SPEC iff the following conditions hold:*

1. *p_2 extends p_1 through φ,*
2. *$\varphi(\pi_2(p'_2))$ satisfies SPEC.*

Let $p_1 = (C_1, I_1, T_1)$ and $p_2 = (C_2, I_2, T_2)$ be two programs and let p_2 extend p_1. Now consider a failure model F which can be applied to p_1 and p_2 resulting in fault-affected versions p'_1 and p'_2. Does p'_2 also extend p'_1? In general, this is not the case since F can treat p_1 and p_2 differently (i.e., it can introduce different faulty behavior into both programs). However, realistic failure models are a little more restricted. For example, the failure model of Arora and Kulkarni [4] introduces additional transitions into the program by adding a set of additional actions. These transitions take effect in the same way in all extensions of the original program.

Postulating that if p_2 extends p_1 then p'_2 extends p'_1 is a very strong assumption, as we now show. In fact, this assumption conflicts with our definition of fault-tolerant version.[1]

Proposition 4 *Let $p_1 = (C_1, I_1, T_1)$ and $p_2 = (C_2, I_2, T_2)$ be programs, F be a failure model, $F(p_1) = (p'_1, \pi_1)$ and $F(p_2) = (p'_2, \pi_2)$. Additionally, let SPEC be a specification and p_2 be the F-tolerant version of p_1 for SPEC. Then p'_2 does not extend p'_1.*

ASSUME: p_2 is an F-tolerant version of p_1 for *SPEC*.
PROVE: p'_2 does not extend p'_1.

PROOF SKETCH: The proof is indirect: We assume that p'_2 extends p'_1 and then show that p'_2 must violate *SPEC*. This is a contradiction to the assumption that p'_2 is an F-tolerant version of p_1.

1 $\langle 1\rangle 1$. ASSUME: p'_2 extends p'_1 through φ.
PROVE: false

1.1 $\langle 2\rangle 1$. Let S'_1 and S'_2 denote the safety properties of p'_1 and p'_2 respectively. Then $\varphi(\pi_2(S'_2)) = \pi_1(S'_1)$.
PROOF: From the assumption that p'_2 extends p'_1, the definition of "extends" (Definition 6) and Proposition 1. □

1.2 $\langle 2\rangle 2$. p'_1 violates *SPEC*.
PROOF: From the assumption that p_2 is an F-tolerant version of p_1 and the definition of "fault-tolerant version" (Definition 9). □

1.3 $\langle 2\rangle 3$. There exists a trace $\sigma_1 \in \pi_1(p'_1)$ such that $\sigma_1 \notin SPEC$.
PROOF: From step $\langle 2\rangle 2$ and the definition of "violates". □

1.4 $\langle 2\rangle 4$. $\sigma_1 \in \pi_1(S'_1)$.
PROOF: From step $\langle 2\rangle 3$. □

1.5 $\langle 2\rangle 5$. There exists a trace $\sigma_2 \in \pi_2(p'_2)$ such that $\varphi(\sigma_2) = \sigma_1$.

[1] Postulating that if from p_2 extends p_1 follows that p'_2 extends p'_1 was called *monotonicity* by Peled and Joseph [12, p. 103]. However, their definition of "extends" is that of the usual *refinement* concept [1].

PROOF: From step $\langle 2\rangle 4$ and step $\langle 2\rangle 1$. □

1.6 $\langle 2\rangle 6$. There exists a trace $\sigma_2 \in \pi_2(p'_2)$ such that $\varphi(\sigma_2) \notin SPEC$.

PROOF: From step $\langle 2\rangle 5$ and step $\langle 2\rangle 3$. □

1.7 $\langle 2\rangle 7$. Q.E.D.

PROOF: Step $\langle 2\rangle 6$ implies that p'_2 violates *SPEC*. This is a contradiction to the assumption that p_2 is an F-tolerant version of p_1 for *SPEC*. □

2 $\langle 1\rangle 2$. Q.E.D.

PROOF: Follows directly from step $\langle 1\rangle 1$. □

Proposition 4 states that, if F leads to p'_2 extending p'_1 then it is not possible to build fault-tolerant versions. Thus, we must seek for weaker definitions that still restrict the generality of F in a certain way.

Note that we run into the same problems as above if we define our restrictions on F in the following way: If there exists a trace $\sigma_1 \in \pi_1(p'_1)$ which was introduced by F (i.e., $\sigma_1 \notin p_1$), then there must exist a trace $\sigma_2 \in \pi_2(p'_2)$ such that $\varphi(\sigma_2) = \sigma_1$. This assumption can be viewed as "extension in one direction": p'_2 might not extend p'_1 anymore, but any trace in p'_1 must be present in p'_2. So the validity of Proposition 4 still holds under this weaker assumption. This is because it is the idea of a fault-tolerant version that it must *exclude* precisely those faulty executions from its set of traces which violate *SPEC*.

What we want from a realistic failure model (which is compatible with our definition of fault-tolerant version) is that fault transitions must be able to happen at the same places in both p'_1 and p'_2. This means that fault-tolerant versions are not able to avoid faults by "sailing around" certain states. To see why this makes sense, consider the two programs p_1 and p_2 depicted as graphs in Fig. 3. The failure model F is such that it introduces a single fault transition indicated by a dashed line. As usual, the fault-affected versions of p_1 and p_2 (i.e., p_1 and p_2 together with the dashed transition) are denoted p'_1 and p'_2. Let the correctness specification *SPEC* be "it is never the case that state 5 is reached unless previously state 3 has been reached".

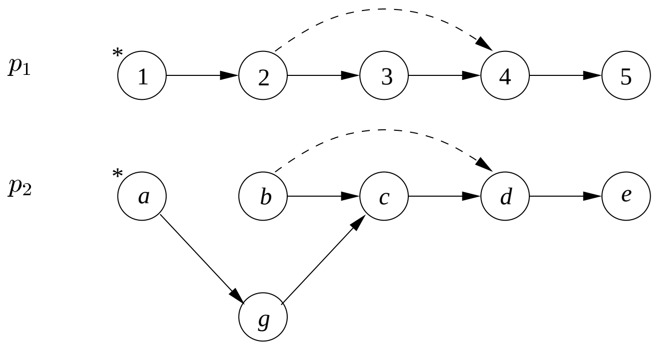

Figure 3: Sailing around faults.

Obviously, p'_1 violates *SPEC* since it allows the trace $1 \cdot 2 \cdot 4 \cdot 5$. Observe that using a projection φ which maps state a of p_2 to state 1 of p_1, states b and g to state 2 and so on, p_2 extends p_1. In fact, since even p'_2 satisfies *SPEC*, p_2

is an F-tolerant version of p_1.

The restriction on F underlying Fig. 3 is that F introduces the "same" transitions into p_1 and p_2, i.e., if there exists a fault transition in p_1 then there must exist a φ-corresponding transition in p_2 and vice versa. Since F is fixed, p_2 can be built to sail around dangerous states. We think that this should not be termed fault tolerance since in p_2' no faults can ever occur (it is rather something like fault avoidance). We now restrict our attention to failure models which ensure that faults cannot be circumvented; they must be able to occur in structurally the same places. This leads to the concept of *conservativeness*. Conservativeness means that if a fault transition can occur in some trace of p_1', then there must be a corresponding trace and a corresponding fault transition possible in p_2'.

Definition 10 (conservative failure model) *Let $p_1 = (C_1, I_1, T_1, L_1)$ and $p_2 = (C_2, I_2, T_2, L_2)$ be two programs, let p_2 extend p_1 through φ and let F be a failure model where $F(p_1) = (p_1', \pi_1)$ and $F(p_2) = (p_2', \pi_2)$. A failure model F is* conservative *iff the following holds: For all traces $\alpha_1 \cdot s_1 \cdot r_1 \in \pi_1(p_1')$ where $(s_1, r_1) \in (\pi_1(T_1') \setminus T_1)$ there exists $\alpha_2 \cdot s_2 \cdot r_2 \in \pi_2(p_2')$ such that $\varphi(\alpha_2 \cdot s_2 \cdot r_2) = \alpha_1 \cdot s_1 \cdot r_1$.*

As an example, consider the two programs in Fig. 3 again. The failure model there is not conservative since there is a trace $1 \cdot 2 \cdot 4$ (where the final transition is a fault transition) which has no corresponding trace in p_2'. To make it conservative it would be sufficient to introduce an additional fault transition from state g to d in p_2'.

The definition of *conservativeness* has interesting consequences. For example, if there is a "bad" state transition (i.e., one which causes the violation of a safety property) then we can show that this transition must be a program transition and cannot be a transition introduced by F.

Lemma 3 *Let F be a conservative failure model, SPEC be a specification and p_2 be an F-tolerant version of $p_1 = (C_1, I_1, T_1)$ for SPEC. If*

- *$\alpha_1 \cdot s_1 \cdot r_1$ is a trace prefix of a trace in $\pi_1(p_1')$ where*
- *$\alpha_1 \cdot s_1$ maintains SPEC but*
- *$\alpha_1 \cdot s_1 \cdot r_1$ does not maintain SPEC*

then $(s_1, r_1) \in T_1$.

ASSUME:
1. F is a conservative failure model,
2. *SPEC* is a specification,
3. p_2 is an F-tolerant version of p_1,
4. $\alpha \cdot s_1 \cdot r_1$ is a trace prefix of a trace in $\pi_1(p_1')$,
5. $\alpha_1 \cdot s_1$ maintains *SPEC*, and
6. $\alpha_1 \cdot s_1 \cdot r_1$ does not maintain *SPEC*.

PROVE: $(s_1, r_1) \in T_1$

PROOF SKETCH: We assume that $(s_1, r_1) \notin T_1$ and derive a contradiction. This is possible because if $(s_1, r_1) \notin T_1$, then (s_1, r_1) must have been introduced by F. However, due to the conservativeness of F, F must introduce some φ-corresponding transition to the transition set T_2'. This leads to p_2' violating *SPEC*. However, p_2' satisfies *SPEC* from assumption 3, a contradiction.

1 $\langle 1 \rangle 1$. ASSUME: $(s_1, r_1) \notin T_1$ (i.e., $(s_1, r_1) \in (\pi_1(T_1') \setminus T_1)$
PROVE: False

1.1 $\langle 2 \rangle 1$. There exists a trace $\sigma_2 = \alpha_2 \cdot s_2 \cdot r_2 \cdot \delta_2 \in \pi_2(p_2')$ such that $\varphi(\alpha_2 \cdot s_2 \cdot r_2) = \alpha_1 \cdot s_1 \cdot r_1$.
PROOF: From the assumption 3 follows that p_2 extends p_1. The proof of the step then follows from the assumption from step $\langle 1 \rangle 1$ and the definition of conservativeness (Definition 10). □

1.2 $\langle 2 \rangle 2$. $\varphi(\sigma_2) = \alpha_1 \cdot s_1 \cdot r_1 \cdot \delta_1 \in SPEC$.
PROOF: Follows from step $\langle 2 \rangle 1$ and the assumption that $\varphi(\pi_2(p_2'))$ satisfies *SPEC* (the latter is part of assumption 3). □

1.3 $\langle 2 \rangle 3$. For all traces β_1 the trace $\alpha_1 \cdot s_1 \cdot r_1 \cdot \beta_1 \notin SPEC$.
PROOF: This follows from assumption 6 and the definition of "maintains" (Definition 5). □

1.4 $\langle 2 \rangle 4$. $\alpha_1 \cdot s_1 \cdot r_1 \cdot \delta_1 \notin SPEC$.
PROOF: From step $\langle 2 \rangle 3$ by choosing β_1 as δ_1 from step $\langle 2 \rangle 2$. □

1.5 $\langle 2 \rangle 5$. Q.E.D.
PROOF: Step $\langle 2 \rangle 4$ contradicts step $\langle 2 \rangle 2$. □

2 $\langle 1 \rangle 2$. Q.E.D.
PROOF: The conclusion follows directly from step $\langle 1 \rangle 1$. □

Another consequence of conservativeness in conjunction with fault-tolerant versions that maintain a specification *SPEC* is the following: a fault-tolerant version can be used to replay an "unsafe" trace (i.e., one that does not maintain *SPEC*) as long as it actually maintains *SPEC*.

Lemma 4 *Let $p_1 = (C_1, I_1, T_1)$ and $p_2 = (C_2, I_2, T_2)$ be two programs, F be a failure model, $F(p_1) = (p_1', \pi_1)$ and $F(p_2) = (p_2', \pi_2)$. If SPEC is a specification, F is a conservative failure model, and p_2 is the F-tolerant version of p_1. Then all traces $\sigma_1 \in \pi_1(p_1')$ which maintain SPEC have a φ-corresponding trace $\sigma_2 \in \pi_2(p_2')$.*

ASSUME:
1. *SPEC* is a specification,
2. F is a conservative failure model,
3. $p_1 = (C_1, I_1, T_1, L_1)$ and $p_2 = (C_2, I_2, T_2, L_2)$ are two programs and $F(p_1) = (p_1', \pi_1)$ and $F(p_2) = (p_2', \pi_2)$, and
4. p_2 is an F-tolerant version of p_1 for *SPEC*.

PROVE: For all $\sigma_1 \in \pi_1(p_1')$ holds: If σ_1 maintains *SPEC* then there exists a $\sigma_2 \in \pi_2(p_2')$ such that $\varphi(\sigma_2) = \sigma_1$.

PROOF SKETCH: The proof is by induction over the length of σ_1. The base case follows from the initial-state preservation of F and the fact that p_2 extends p_1.

The induction step must make a case analysis of the added transition. If it is a program transition, the proof is easy. If it is a transition introduced by F, we argue using the conservativeness property of F.

1 $\langle 1 \rangle 1$. The conclusion holds for all σ_1 with $|\sigma_1| = 1$.

1.1 $\langle 2 \rangle 1$. σ_1 consists of only one (initial) state $s'_1 \in I'_1$. There exists an $s_1 \in I_1$ such that $\pi_1(s'_1) = s_1$.

PROOF: From the fact that F is initial-state preserving. □

1.2 $\langle 2 \rangle 2$. There exists an $s_2 \in I_2$ such that $\varphi(s_2) = s_1$.

PROOF: From step $\langle 2 \rangle 1$ and the fact that p_2 extends p_1. □

1.3 $\langle 2 \rangle 3$. There exists an $s'_2 \in I'_2$ such that $\pi_2(s'_2) = s_2$.

PROOF: From the fact that F is initial-state preserving. □

1.4 $\langle 2 \rangle 4$. Q.E.D.

PROOF: Take σ_2 of the conclusion to consist only of s'_2 from step $\langle 2 \rangle 3$. □

2 $\langle 1 \rangle 2$. ASSUME: The conclusion holds for all traces σ_1 with $|\sigma_1| \leq k$.
PROVE: The conclusion holds for all traces ρ_1 with $|\rho_1| = k + 1$.

2.1 $\langle 2 \rangle 1$. Let $\rho_1 = \alpha_1 \cdot q_1 \cdot r_1$ such that $|\rho_1| = k + 1$ and ρ_1 maintains *SPEC*. Then $\alpha_1 \cdot q_1$ maintains *SPEC*.

PROOF: Follows from Proposition 2. □

2.2 $\langle 2 \rangle 2$. There exists a trace $\alpha_2 \cdot q_2 \in \pi_2(p'_2)$ such that $\varphi(\alpha_2 \cdot q_2) = \alpha_1 \cdot q_1$.

PROOF: From step $\langle 2 \rangle 1$ and induction hypothesis. □

2.3 $\langle 2 \rangle 3$. Q.E.D.

2.3.1 $\langle 3 \rangle 1$. CASE: $(q_1, r_1) \in T_1$
PROVE: Q.E.D.

PROOF: If $(q_1, r_1) \in T_1$ then there must be a $(q_2, r_2) \in T_2$ because p_2 extends p_1. Thus, we can concatenate the trace from step $\langle 2 \rangle 2$ with this transition, yielding a trace $\sigma_2 = \alpha_2 \cdot q_2 \cdot r_2 \in \pi_2(p'_2)$ that meets the conclusion. □

2.3.2 $\langle 3 \rangle 2$. CASE: $(q_1, r_1) \in (\pi_1(T'_1) \setminus T_1)$
PROVE: Q.E.D.

PROOF: From step $\langle 2 \rangle 2$ and the case assumption we have that $\alpha_1 \cdot q_1 \cdot r_1 \in \pi_1(p'_1)$ and that $(q_1, r_1) \in (\pi_1(T'_1) \setminus T_1)$. This meets the prerequisites for the definition of conservativeness. Since F is conservative, there must be a σ_2 such that $\sigma_2 = \alpha_2 \cdot q_2 \cdot r_2 \in \pi'_2(p'_2)$ such that $\varphi(\sigma_2) = \sigma_1$. □

2.3.3 $\langle 3 \rangle 3$. Q.E.D.

PROOF: Step follows from the fact that steps $\langle 3 \rangle 1$ and $\langle 3 \rangle 2$ cover all cases. □

3 $\langle 1 \rangle 3$. Q.E.D.

PROOF: The conclusion follows from steps $\langle 1 \rangle 1$ and $\langle 1 \rangle 2$ by induction over the length of σ_1. □

3.4 Redundancy in Space

Now we turn to the central part of our paper, the definition of redundancy in space and its relation to safety specifications.

A redundant program contains parts which — under normal circumstances — are not necessary. (In the case of fault-tolerant systems the redundant parts come into play when faults occur.) In the case of safety specifications these parts are non-reachable states. This is what we call *redundancy in space*.

Definition 11 (non-reachable state) *Let $p = (C, I, T)$ be a program. A state $s \in C$ is a* non-reachable state *iff for all traces $\sigma \in p$, $\sigma = s_1 \cdot s_2 \cdot s_3 \cdots$, and for all i, $s_i \neq s$.*

Definition 12 (redundancy in space) *A program p employs* redundancy in space *iff it contains non-reachable states.*

In the following we present the central result of this paper: If p_2 is the fault-tolerant version of some fault-intolerant program p_1, then p_2 must have non-reachable states. To give an intuitive example, consider Figure 4. It shows two programs p_1 and p_2, where the transitions which are added by F are shown as dashed arrows and in which the state projection again maps vertically aligned states onto each other. Assume the safety specification *SPEC* is "state 5 is only reached if state 3 has been reached before". Obviously, program p_2 is the F-tolerant version of p_1 for *SPEC* because it simply stops in state d if state c has been omitted.

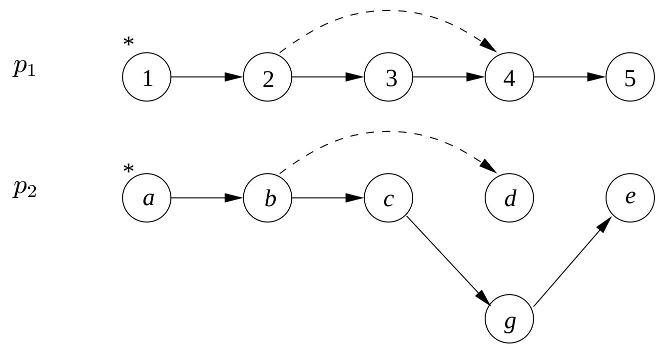

Figure 4: Example accompanying Theorem 1.

Theorem 1 (redundancy in space necessary for safety) *Let $p_1 = (C_1, I_1, T_1)$ and $p_2 = (C_2, I_2, T_2)$ be two programs, F be a conservative failure model, $F(p_1) = (p'_1, \pi_1)$ and $F(p_2) = (p'_2, \pi_2)$. If p_2 is an F-tolerant version of p_1 for a safety specification SSPEC then p_2 employs redundancy in space.*

ASSUME: 1. p_2 is an F-tolerant version of p_1 for a safety specification *SSPEC*,

2. F is an initial-state preserving, conservative failure model.

PROVE: p_2 employs redundancy in space.

PROOF SKETCH: The idea of the proof is to take a trace σ_1 from p'_1 which violates *SSPEC* and derive a trace ρ_2 from p'_2 which is a "maximum replay" of σ_1. The existence of ρ_2 is guaranteed by Lemma 4.
Then we can show that the final state d_2 of ρ_2 is a non-reachable state. If d_2 were reachable we could construct a trace of p_1 (not p'_1) that doesn't maintain *SSPEC*. This contradicts the assumption that p_1 satisfies *SSPEC*.

1 $\langle 1 \rangle 1.$ There exists a trace σ_1 of $\pi_1(p'_1)$ such that $\sigma_1 \notin \mathit{SSPEC}$.
PROOF: By the assumption, p_2 is an F-tolerant version of p_2 for SSPEC. This implies that $\pi_1(p'_1)$ violates SSPEC. The proof follows from the definition of "violates". □

2 $\langle 1 \rangle 2.$ There exists a prefix ρ_1 of σ_1 such that ρ_1 does not maintain SPEC.
PROOF: Follows from step $\langle 1 \rangle 1$ and the fact that SSPEC is a safety property. □

3 $\langle 1 \rangle 3.$ Trace ρ_1 can be written as $\rho_1 = \alpha_1 \cdot d_1 \cdot b_1 \cdot \beta_1$ such that the following three properties hold:

1. $\alpha_1 \cdot d_1$ maintains SSPEC and all longer prefixes of σ_1 do not maintain SSPEC,
2. (d_1, b_1) is a "bad" transition, i.e., all traces in which (d_1, b_1) occurs do not maintain SSPEC, and
3. $(d_1, b_1) \in T_1$.

PROOF: Items 1 and 2 follow from Lemma 1. Item 3 follows from Lemma 3. □

4 $\langle 1 \rangle 4.$ There exists a trace $\rho_2 = \alpha_2 \cdot d_2 \in \pi_2(p'_2)$ such that $\varphi(\rho_2) = \alpha_1 \cdot d_1$.
PROOF: Follows from step $\langle 1 \rangle 3$ and Lemma 4. □

5 $\langle 1 \rangle 5.$ d_2 is a non-reachable state of p_2.

5.1 $\langle 2 \rangle 1.$ ASSUME: d_2 is reachable, i.e., there exists a trace $\delta_2 \cdot d_2 \in p_2$.
PROVE: False

5.1.1 $\langle 3 \rangle 1.$ There exists a trace $\delta_1 \cdot d_1 \in p_1$ such that $\varphi(\delta_2 \cdot d_2) = \delta_1 \cdot d_1$.
PROOF: From the fact that p_2 extends p_1. □

5.1.2 $\langle 3 \rangle 2.$ The trace $\delta_1 \cdot d_1 \cdot b_1$ is a trace of p_1.
PROOF: From step $\langle 3 \rangle 1$ and item 3 of step $\langle 1 \rangle 3$. □

5.1.3 $\langle 3 \rangle 3.$ $\delta_1 \cdot d_1 \cdot b_1 \notin \mathit{SSPEC}$
PROOF: Follows from the fact that (d_1, b_1) is a "bad" transition (item 2 of step $\langle 1 \rangle 3$). □

5.1.4 $\langle 3 \rangle 4.$ Q.E.D.
PROOF: From step $\langle 3 \rangle 3$ we have a trace of p_1 which is not in SSPEC, i.e., p_1 violates SSPEC, a contradiction to the assumption that p_1 satisfies SSPEC (because p_2 is an F-tolerant version of p_1). □

5.2 $\langle 2 \rangle 2.$ Q.E.D.
PROOF: Follows directly from step $\langle 2 \rangle 1$. □

6 $\langle 1 \rangle 6.$ Q.E.D.
PROOF: From step $\langle 1 \rangle 5$ and the definition of redundancy in space. □

4 Discussion

It is commonly agreed that redundancy is a key concept in fault tolerance. We proposed a formal definition of redundancy in space that we believe to capture a common intuition of hardware and software redundancy. Using the definition, we have shown that this form of redundancy is necessary to add fault-tolerance to a given system with respect to a safety property.

As mentioned above, there is a second type of redundancy which appears quite often in the literature: redundancy in time. A natural definition of redundancy in time within our system model would be that a program employs redundancy in time iff there are transitions which never occur in any fault-free trace of that program. The definition is simple, but proving certain consequences, like a possible affinity to liveness properties, depends on many additional system parameters and turns out to be more difficult than the safety case. This will be investigated in our continuing work.

Acknowledgments

Work by the first author was supported by the German Research Association (DFG) as part of the "Graduiertenkolleg ISIA" at Darmstadt University of Technology. Work of the second author was supported by the DFG project "Konsensalgorithmen". We also wish to thank Sandeep Kulkarni for helpful discussions.

References

[1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[2] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[3] Anish Arora and Sandeep Kulkarni. Detectors and correctors: A theory of fault-tolerance components. In *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems (ICDCS98)*, May 1998.

[4] Anish Arora and Sandeep S. Kulkarni. Component based design of multitolerant systems. *IEEE Transactions on Software Engineering*, 24(1):63–78, January 1998.

[5] Algirdas Avižienis. Fault-tolerant systems. *IEEE Transactions on Computers*, 25(12):1304–1312, December 1976.

[6] K. P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):36–53, 1993.

[7] Felix C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys*, 31(1):1–26, March 1999.

[8] Felix C. Gärtner. Transformational approaches to the specification and verification of fault-tolerant systems: Formal background and classification. *Journal of Universal Computer Science (J.UCS)*, 5(10):668–692, October 1999. Special Issue on Dependability Evaluation and Assessment.

[9] Pankaj Jalote. *Fault tolerance in distributed systems.* Prentice-Hall, Englewood Cliffs, NJ, USA, 1994.

[10] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, March 1977.

[11] Leslie Lamport. How to write a proof. *American Mathematical Monthly*, 102(7):600–608, August/September 1995.

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

[13] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.