Automating the Addition of Fail-Safe Fault-Tolerance: Beyond Fusion-Closed Specifications

Felix C. Gärtner
École Polytechnique Fédérale de Lausanne (EPFL)
Departement de Systèmes de Communications
Laboratoire de Programmation Distribuée
CH-1015 Lausanne, Switzerland
fcg@acm.org

Arshad Jhumka
Technische Universität Darmstadt
Fachbereich Informatik
D-64283 Darmstadt, Germany
arshad@informatik.tu-darmstadt.de

April 11, 2003

Swiss Federal Institute of Technology (EPFL)
School of Computer and Communication Sciences
Technical Report IC/2003/23

Abstract

The fault tolerance theories of Arora and Kulkarni [3] and of Jhumka *et al.* [11] view a fault-tolerant program as the result of composing a fault-intolerant program with fault tolerance components called *detectors* and *correctors*. At their core, the theories assume that the correctness specifications under consideration are *fusion closed*. In general, fusion closure of specifications can be achieved by adding *history variables* to the program. However, addition of history variables causes an exponential growth of the state space of the program, causing addition of fault tolerance to be expensive. To redress this problem, we present a method which can be used to add history information to a program in a way that (in a certain sense) minimizes the additional states. Hence, automated methods that add fault tolerance can now be efficiently applied in environments where specifications are not necessarily fusion closed.

Keywords:

fault-tolerance, safety, fusion closure, specifications, transition systems, theory, extension

1 Introduction

It is an established engineering method in computer science to generate complicated things from simpler things. The most obvious example for this is a compiler for a programming language (like C). The compiler takes a high-level programming instruction in form of a C program and generates a sequence of machine code instructions that perform the specified task. Of course, the original C program might be complicated too, but it is at least easier to understand than the generated assembly code since it abstracts away from the machine architecture and supports a more natural formulation of control structures etc.

Another area in which this technique has been applied is the area of fault-tolerant systems. The goal is to start off with a system which is not fault-tolerant for certain kinds of faults and use a sound procedure to transform it into a program which is fault-tolerant. The approaches which have been proposed range from practical proposals like Schneider's state machine approach [17] to theoretical studies like the one by Basu et al. [5]. The former approach can be used to tolerate permanent faults in a certain number of replicated processes while the latter approach studies tolerance against certain types of transient communication faults. Although these methods can be combined, in general they seem a little oversized since they cannot be easily adapted to other types of faults with finer granularity like a stuck-at-0 register.

To this end, Arora and Kulkarni [3] initially presented a method which can be used to combat finer grained fault assumptions. Fault tolerance is achieved by composing a fault-intolerant program with two types of fault-tolerance components called *detectors* and *correctors*. Briefly spoken, a detector is used to detect a certain (error) condition on the system state and a corrector is used to bring the system into a valid state again. Since common fault-tolerance methods like triple modular redundancy or error correcting codes can be modeled by using detectors and correctors, the theory can be viewed as an abstraction of many existing fault tolerance techniques, including the state machine approach.

Kulkarni and Arora [12] and more recently Jhumka *et al.* [11] proposed methods to automate the addition of detectors and correctors to a fault-intolerant program. The basic idea of these methods is to perform a state space analysis of the fault-affected program and change its transition relation in such a way that it still satisfies its specification in the presence of faults. These changes result in either the removal of transitions to satisfy a safety specification or the addition of transitions to satisfy a liveness specification. Gärtner and Völzer [9] analyzed the assumptions behind the original Kulkarni-Arora method and argued that it is based on two distinct forms of redundancy: *redundancy in space* and *redundancy in time*. The former refers to non-reachable states of the program while the latter refers to non-reachable transitions. However, the detector/corrector method cannot be viewed as a method which "adds redundancy" (like for example the state machine approach) because the redundancy is already present in the fault intolerant program. This stems from the fact that Arora and Kulkarni [3] assume that their correctness specifications are *fusion closed*.

Basically, fusion closure means that the next step of a program merely depends on the current state and not on the previous history of the execution. For example, given a program with a single variable $x \in \mathbb{N}$, then the specification "never $x = 1$" is fusion closed while the specification "$x = 4$ implies that previously $x = 2$" is

not. Specifications written in the popular Unity Logic [6] are fusion closed [10], as are specifications consisting of state transition systems (like C programs). But general temporal logic formulas which are usually used in the area of fault-tolerant program synthesis and refinement [15, 16] are not. Arora and Kulkarni [3, p. 75] originally argued that this assumption is not restrictive in the sense that for every non-fusion closed specification there exists an "equivalent" specification which is fusion closed if it is allowed to add *history variables* to the program. History variables are additional control variables which are used to record the previous state sequence of an execution and hence can be used to answer the question of, e.g., "has the program been in state $x = 2$?". Using such a history variable h the example above which was not fusion closed can be rephrased in a fusion-closed fashion as:

$$\text{"never } (x = 4 \text{ and } (x = 2) \notin h)\text{"}$$

However, these history variables add states to the program and in effect add the necessary redundancy to be fault-tolerant.

There are obvious "brute force" approaches on how to add history information like the one sketched above where the history variable remembers the entire previous state sequence of an execution. However, since history variables must be implemented, they exponentially enlarge the state space of the fault-intolerant program. Rephrasing this in the redundancy terminology of Gärtner and Völzer [9], history variables add redundancy in space. Specifically, the history variables add exponential redundancy in space, which is costly. So, we are interested in adding as little redundancy (i.e., as little additional states) as possible. Intuitively, the minimal amount of redundancy which is necessary to tolerate a certain class of faults depends on the kind and nature of the faults.

In this paper, we present a method to add history states to a program in a way which (in general) avoids exponential growth of the state space. More specifically, we start with a problem specification SPEC_1 which is *not* fusion closed, a program Σ_1 which satisfies SPEC_1 and a class of faults F. Depending on F we show how to transform SPEC_1 and Σ_1 into SPEC_2 and Σ_2 in such a way that (a) SPEC_2 is fusion closed, (b) Σ_2 can be made fault tolerant for SPEC_2 iff Σ_1 can be made fault tolerant for SPEC_1, and (c) Σ_2 is (in a certain sense) minimal with respect to the added states. We restrict our attention to cases where *SPEC* is a safety property and therefore are only concerned with what Arora and Kulkarni call *fail-safe fault-tolerance* [3].

The benefit of the proposed method is the following: Firstly, it makes the methods which automatically add detectors [11, 12] amendable to specifications which are not fusion closed and closes a gap in the applicability of the detector/corrector theory [3]. And secondly, the presented method offers further insight into the efficiency of the basic mechanisms which are applied in fault tolerance.

The paper is structured as follows: We first present some preliminary definitions in Section 2 and then relate the assumption of fusion closure to the notion of state space redundancy in Section 3. In Section 4 we study specifications which are not fusion closed and present a method which makes these types of specifications efficiently manageable in the context of automated methods which add fault tolerance. Finally, Section 5 presents some open problems and directions for future work.

2 Formal Preliminaries

In this section we define the formal system model used throughout this paper.

2.1 States, Traces and Properties

The *state space* of a program is an unstructured finite nonempty set C of states. A *state predicate over* C is a boolean predicate over C. A *state transition over* C is a pair (r, s) of states from 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 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, 13]. Informally spoken, a safety property demands that "something bad never happens" [13], i.e., it rules 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" [13] and can be used to formalize, e.g., notions of termination. Since we are only concerned with safety properties we omit a formal definition of liveness. Safety properties are formally 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.*

2.2 Programs, Specifications and Correctness

We define programs as state transition systems consisting of a state set C, a set of initial states $I \subseteq C$ and a transition relation T over C, i.e., a *program* (sometimes also called *system*) is a triple $\Sigma = (C, I, T)$. 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. We denote this property by *safety-prop*(Σ). For brevity, we sometimes write Σ instead of *safety-prop*(Σ). A state $s \in C$ of a program Σ is *reachable* iff there exists a trace $\sigma \in \Sigma$ such that s occurs in σ. Otherwise s is *non-reachable*. Sometimes we will call a non-reachable state a *redundant*.

We define specifications to be properties, i.e., a *specification over* C is a property over C. A *safety specification* is a specification which is a safety property. Unlike Arora and Kulkarni [3], we do *not* assume that problem specifications are fusion closed. Fusion closure is defined as follows: Let C be a state set, $s \in C$, X be property over C, α, γ finite state sequences, and β, δ, σ be state sequences over C.

Definition 2 (fusion closed set) *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.*

It is easy to see that for every program Σ holds that *safety-prop*(Σ) is fusion closed. Intuitively, fusion closure means that the entire history of every trace is present in every state of the trace. We will give examples for fusion closed and not fusion closed specifications later.

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

2.3 Extensions

Given some program $\Sigma_1 = (C_1, I_1, T_1)$ our goal is to define the notion of a fault-tolerant version Σ_2 of Σ_1 meaning that Σ_2 does exactly what Σ_1 does in fault-free scenarios and has additional fault-tolerance abilities which Σ_1 lacks. Sometimes, $\Sigma_2 = (C_2, I_2, T_2)$ will have additional states (i.e., $C_2 \supset C_1$) and for this case we must define what these states "mean" with respect to the original program Σ_1. This is done using a *state projection function* $\pi : C_2 \mapsto C_1$ which tells which states of Σ_2 are "the same" with respect to states of Σ_1. A state projection function can be naturally extended to traces and properties, e.g., for a trace $s_1, s_2, \ldots$ over C_2 holds that $\pi(s_1, s_2, \ldots) = \pi(s_1), \pi(s_2), \ldots$

Definition 3 (extends) *Let $\Sigma_1 = (C_1, I_1, T_1)$ and $\Sigma_2 = (C_2, I_2, T_2)$ be two programs. Program* Σ_2 extends program Σ_1 using state projection π *iff the following conditions hold:*

1. $C_2 \supseteq C_1$,
2. *π is a total mapping from C_2 to C_1 (for simplicity we assume that for any $s \in C_1$ holds that $\pi(s) = s$), and*
3. $\pi($*safety-prop*$(\Sigma_2)) =$ *safety-prop*(Σ_1).

Note that the concept of extension is related to the notion of *refinement* [1]. Extensions are refinements with the additional property that the original state space is preserved and that there is no notion of *stuttering* [1].

If Σ_2 extends Σ_1 using π and Σ_1 satisfies *SPEC* then obviously $\pi(\Sigma_2)$ satisfies *SPEC*. When it is clear from the context that Σ_2 extends Σ_1 we will simply say that Σ_2 satisfies *SPEC* instead of "$\pi(\Sigma_2)$ satisfies *SPEC*".

2.4 Fault Models and Fault-Tolerant Versions

Since we are concerned with fault tolerant systems we must have a way of modeling faulty behavior. We define a fault 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*. For a given program Σ, $F(\Sigma)$ is also called *program Σ in the presence of faults F*.

We require that a fault 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 also restrict ourselves to the case where F "adds" transitions, since this is the only way to violate a safety specification.

Definition 4 (fault model) *A* fault model F *maps a program* $\Sigma = (C, I, T)$ *to a program* $F(\Sigma) = (F(C), F(I), F(T))$ *such that the following conditions hold:*

1. $F(C) = C$
2. $F(I) = I$
3. $F(T) \supset T$

For a given fault model F and a specification *SPEC*, we say that a program Σ is F*-intolerant with respect to SPEC* if Σ satisfies *SPEC* but $F(\Sigma)$ violates *SPEC*.

Given two programs Σ_1 and Σ_2 such that Σ_2 extends Σ_1 and a fault model F, it makes sense to assume that F treats Σ_1 and Σ_2 in a "similar way". Basically, this means that F should at least add the same transitions to Σ_1 and Σ_2. But with respect to the possible new states of Σ_2 it can possibly add new fault transitions. This models faults which occur within the fault-detection and correction mechanisms.

Definition 5 (fault extension monotonicity) *A fault model* F *is* extension monotonic *iff for any two programs* $\Sigma_1 = (C_1, I_1, T_1)$ *and* $\Sigma_2 = (C_2, I_2, T_2)$ *such that* Σ_2 *extends* Σ_1 *using* π *holds:*

$$F(T_1) \setminus T_1 \subseteq F(T_2) \setminus T_2$$

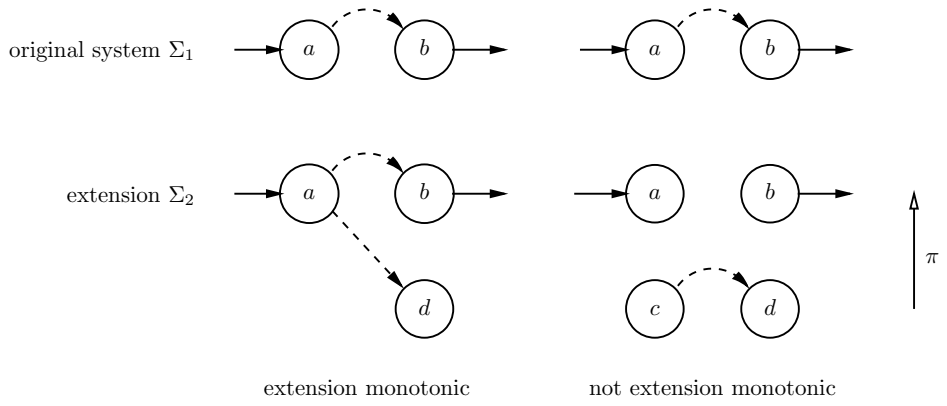

Figure 1: Examples for extension monotonic and not extension monotonic fault models.

An example is given in Fig. 1. The original system is given at the top and the extension is given below (the state projection is implied by vertical orientation, i.e., states which are vertically aligned are mapped to the same state by π). In the left example the fault model is extension monotonic since all fault transitions in Σ_1 are also in Σ_2. The right example is not extension monotonic. Intuitively, an extension monotonic fault model maintains at least its original transitions over extensions.

The extension monotonicity requirement does not restrict faulty behavior on the new states of the extension. However, we have to restrict this type of behavior since it would be impossible to build fault-tolerant versions otherwise. In this paper we assume a very general type of restriction: it basically states that in any infinite sequence of extensions of the original program there is always some point where F does not introduce new fault transitions anymore.

Definition 6 (finite fault model) *A extension monotonic fault model F is* finite *iff for any infinite sequence of programs $\Sigma_1, \Sigma_2, \ldots$ such that for all i, Σ_{i+1} extends Σ_i holds that there exists a j such that for all $k \geq j$ no new fault transition is introduced in Σ_k, i.e., $F(T_{k+1}) \setminus T_{k+1} = F(T_k) \setminus T_k$.*

Finite fault models retain the fault transitions in the original program (i.e., they are extension monotonic for each pair of extensions). They do not restrict the additional faulty behavior introduced in the new states of an extension. However, they exclude fault models for which infinite redundancy is necessary to tolerate them. The engineering process is as follows: Given a program Σ_1 and a fault model F, we extend Σ_1 to Σ_2 to make F tolerable. Then we look at the new states introduced in this process and consider faults which might happen there. Regarding these new faults we construct a new extension Σ_3 of Σ_2 to potentially tolerate these faults. This process is repeated. In theory, this process might never terminate, namely if F forever adds certain kinds of faults to the new states. A finite fault model guarantees that this process must eventually terminate. In this paper, we assume our fault model to be finite and extension monotonic.

Now we are able to define a *fault-tolerant version*. It captures the idea of starting with some program Σ_1 which is fault-intolerant regarding a specification *SPEC* and some fault model F. A fault-tolerant version Σ_2 of Σ_1 is a program which has the same behavior as Σ_1 if no faults occur, but additionally satisfies *SPEC* in the presence of faults.

Definition 7 (fault-tolerant version) *Let F be a fault model, SPEC be a specification and Σ_1 and Σ_2 be programs. Assume that Σ_1 satisfies SPEC but $F(\Sigma_1)$ violates SPEC. We call a program Σ_2 the* F-tolerant version *of program Σ_1 for SPEC using state projection π iff the following conditions hold:*

1. *Σ_2 extends Σ_1 using π,*
2. *$F(\Sigma_2)$ satisfies SPEC.*

3 Problem Statement

The basic task we would like to solve is to construct a fault-tolerant version for a given program and a safety specification.

Definition 8 (general fail-safe transformation problem) *Given a fault model F and a program Σ_1 which is F-intolerant with respect to a general safety specification $SPEC_1$. The* general fail-safe transformation problem *consists of finding a fault-tolerant version of Σ_1, i.e., a program Σ_2 such that Σ_2 extends Σ_1 and $F(\Sigma_2)$ satisfies $SPEC_1$.*

The case where *SPEC* is fusion closed has been studied by Kulkarni and Arora [12] and Jhumka *et al.* [11], i.e., they solve a restricted transformation problem.

Definition 9 (fusion-closed fail-safe transformation problem) *The* fusion-closed fail-safe transformation problem *consists of solving the general fail-safe transformation problem where SPEC$_1$ is fusion closed.*

In the remainder of this section we briefly recall the approaches used by Kulkarni and Arora [12] and Jhumka *et al.* [11] to solve the latter problem.

3.1 Adding Fail-Safe Fault Tolerance to Fusion-Closed Specifications

The basic mechanism which Kulkarni and Arora [12] and Jhumka *et al.* [11] apply is the creation of non-reachable states. The fact that specifications are fusion closed implies that safety specifications can be concisely represented by a set of "bad" transitions, transitions which causes a violation of the specification [3, 10].

Definition 10 (maintains) *Let Σ be a program, SPEC be a specification and α be a finite computation of Σ. We say that α* maintains *SPEC iff there exists a sequence of states β such that $\alpha \cdot \beta \in$ SPEC.*

If *SPEC* is a safety property, every trace not in *SPEC* has a prefix which does not maintain *SPEC*. From the definition of maintains, we have that there must be a transition where a given trace 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*. Arora and Kulkarni have shown [4, "Only-if" part of Lemma 3.2] that (d, b) is a transition which will cause any trace in which it occurs to violate *SPEC*. We rephrase this result as follows:

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

The known automated procedures [11, 12] which are based on the concept of non-reachable states use the following approach for addition of fail-safe fault tolerance: Since $F(\Sigma_1)$ violates *SPEC*, there must exist executions in which a specified bad transition occurs. Inevitably, we must prevent the occurrence of such a transition. So, for all bad transitions $t = (d, b)$ we must make either state d or state b unreachable in $F(\Sigma_2)$. If t is a program transition then it depends on whether or not t is reachable in Σ_1 or not.

- If t is a reachable program transition, then a violation of *SPEC* can occur even if no faults occur, so, obviously, no fault-tolerant version exists since we would have to change the behavior of the original program.
- If t is a redundant (i.e., non-reachable) program transition, then we can remove it resulting in a smaller transition set T_2 of Σ_2.

If t is a transition which has been introduced by F, then we cannot remove it directly. The best we can do is make the starting state d of t unreachable. But this can only be done if there exists a non-reachable program transition on the path to d. If such a transition exists, we can safely remove it. If not, then again no fault-tolerant version exists.

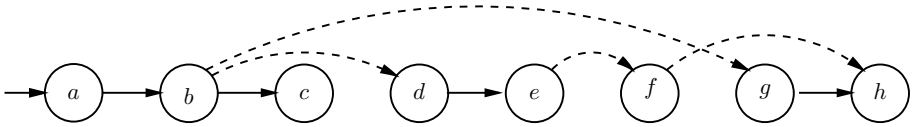

Figure 2: Illustration for the Kulkarni-Arora method. The specification is "never h".

As an illustration of the method consider Figure 2 which shows a program Σ_1 in a state-chart like notation. Again, states are drawn as circles and transitions are arrows between states. Initial states are identified using arrows without starting states. Transitions which are introduced by F are shown as dashed arrows.

Assume the correctness specification *SPEC* for Σ_1 is that it never reaches state h. Obviously, the system satisfies *SPEC* in the absence of faults but it violates *SPEC* in the presence of faults. The bad transitions which we must prevent in $F(\Sigma_2)$ are all transitions which have state h as destination state. We can remove the transition (g, h) easily from T_2 because its removal does not change the fault-free behavior of Σ_2. But we cannot remove transition (f, h) since it is a fault transition. But luckily, there exists a redundant transition (d, e) on the path leading to f which can be removed in T_2. So Σ_2 is constructed from Σ_1 by removing (g, h) and (d, e) from T_2.

Figure 2 also helps to illustrate the cases where no fault-tolerant version exists. For example, if there were a transition $(c, h) \in T_1$ then h is reachable in Σ_1 and, hence, Σ_1 does not satisfy the specification anyway. The other case arises for example, if there were a fault transition $(b, h) \in F(T_1)$, i.e., h is reachable along a path with only fault transitions and reachable program transitions. Again, such an F is not tolerable. However, the fact that F is not tolerable is not a drawback of the transformation method; it is simply states that generally the chosen fault assumption is too severe to be tolerated.

This concludes the recapitulation of the known approaches to automatically make a program fail-safe fault tolerant. Recall that specifications are required to be fusion closed. The above illustrations show that fusion closure together with the assumption that a fault assumption is tolerable implies that state space redundancy (e.g., states d and g) is already available in the fault-intolerant system Σ_1. This type of redundancy allows to formulate *detection predicates* in the language of guarded commands [6, 7] which is the basis of the Arora-Kulkarni theory [3]. These detection predicates are conjoined to the guards of certain actions and hence have the effect of removing transitions.

3.2 Handling Specifications which are Not Fusion Closed

Programs are presented in a guarded command notation [6, 7]. The state space of a program is defined by a set of variables and the state transitions by a set of actions. An

process Σ_1
var $x \in \{0,1,2,3,4\}$ **init** 0
begin

$$\begin{array}{lll} x=0 & \longrightarrow & x := 1 \\ [\!] \quad x=1 & \longrightarrow & x := 2 \\ [\!] \quad x=2 & \longrightarrow & x := 3 \\ [\!] \quad x=3 & \longrightarrow & x := 4 \\ [\!] \quad f: x=1 & \longrightarrow & x := 3 \end{array}$$

end

process Σ_2
var $x \in \{0,1,2,3,4\}$ **init** 0
h **sequence of** $\{0,1,2,3,4\}$ **init** $\langle\rangle$
begin

$$\begin{array}{lll} x=0 & \longrightarrow & x := 1;\ h := \langle 1\rangle \\ [\!] \quad x=1 & \longrightarrow & x := 2;\ h := \langle 1,2\rangle \\ [\!] \quad x=2 & \longrightarrow & x := 3;\ h := \langle 1,2,3\rangle \\ [\!] \quad x=3 & \longrightarrow & x := 4;\ h := \langle 1,2,3,4\rangle \\ [\!] \quad f: x=1 & \longrightarrow & x := 3 \end{array}$$

end

Figure 3: Two programs in guarded command notation.

action of a program has the form

$$\langle \text{guard} \rangle \rightarrow \langle \text{statement} \rangle$$

in which the guard is a boolean expression over the program variables and the statement is either the empty statement or an instantaneous assignment to one or more variables. An execution is constructed by repeatedly and non-deterministically choosing any action where the guard evaluates to true and executing the corresponding action.

Consider the program on the left side of Figure 3. The program has a variable x which can take five different values (0–4) and simply proceeds from state $x = 0$ to $x = 4$ through all intermediate states. The fault assumption F has added one transition from $x = 1$ to $x = 3$ to the transition relation (the action is marked with an 'f').

Consider the correctness specification

$$\textit{SPEC} = \text{“always } (x = 4 \text{ implies that previously } x = 2)\text{”}$$

Note that $F(\Sigma_1)$ does not satisfy *SPEC* (i.e., $F(\Sigma_1)$ can reach state $x = 4$ without having been in state $x = 2$), and that *SPEC* is not fusion closed. To see the latter, consider the two traces $0, 3, 2, 4$ and $2, 3, 4$ from *SPEC*. The fusion at state $x = 3$ yields trace $0, 3, 4$ which is not in *SPEC*. Since *SPEC* is not fusion closed, we cannot apply the known transformation methods [11, 12].

The specification can be made fusion closed by adding a history variable h which records the entire state history. Such a variable has been added to the program on the right side of Figure 3. Now *SPEC* can be rephrased as

$$\textit{SPEC} = \text{“always } (x = 4 \text{ implies } \langle 2\rangle \in h)\text{”}$$

or equivalently

$$\textit{SPEC} = \text{“never } (x = 4 \text{ and } \langle 2\rangle \notin h)\text{”}$$

Now we can identify a set of bad transitions which must be prevented, e.g.:

$$x = 3 \wedge h = \langle 1\rangle \rightarrow x = 4 \wedge h = \langle 1,2,3,4\rangle$$

The precondition for the transition to a state where $x = 4$ must be strengthened by the detection predicate $h \neq \langle 1\rangle$, i.e., the fourth guarded command of Σ_2 must be changed to:

$$x = 3 \wedge h \neq \langle 1\rangle \quad \longrightarrow \quad x := 4;\ h := \langle 1,2,3,4\rangle$$

Hence, bad transitions are prevented and the modified system satisfies *SPEC* in the presence fault f.

3.3 State Space Redundancy Through History Variables

Adding a history variable h in the previous example adds states to the state space of the system. In fact, defining the domain of h as the set of all sequences over $\{0, 1, 2, 3, 4\}$ adds infinitely many states. Clearly this can be reduced by the observation that if faults do not corrupt h, then h will only take on five different values ($\langle\rangle$, $\langle 1\rangle$, $\langle 1, 2\rangle$, $\langle 1, 2, 3\rangle$, and $\langle 1, 2, 3, 4\rangle$). But still, the state space has been increased from five states to $5^2 = 25$ states.

Note that Σ_2 has redundant states and Σ_1 is not redundant at all. So the redundancy is due to the history variable h. But even if the domain of h has cardinality 5, the redundancy is in a certain sense not minimal, as we now explain.

Consider the program Σ_3 on the left side of Figure 4. It tolerates the fault f by adding only *one* state to the state space of Σ_1 (namely, $x = 5$). The state space together with the transitions is depicted on the right side of the figure. Note that Σ_3 has only one redundant state, so Σ_3 can be regarded as redundancy-minimal with respect to *SPEC*. The metric used for minimality is the number of redundant states. We want to exploit this observation to deal with the general case.

process Σ_3
var $x \in \{0, 1, 2, 3, 4, 5\}$ **init** 0
begin

	$x = 0$	$\longrightarrow$	$x := 1$
$[\!]$	$x = 1$	$\longrightarrow$	$x := 2$
$[\!]$	$x = 2$	$\longrightarrow$	$x := 5$
$[\!]$	$x = 5$	$\longrightarrow$	$x := 4$
$[\!]$	$f : x = 1$	$\longrightarrow$	$x := 3$

end

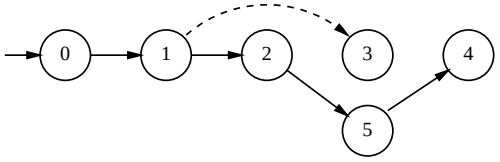

Figure 4: A redundancy-minimal version of the program in Figure 3 in guarded commands (left) and a state chart notation (right). The specification is "always ($x = 4$ implies that previously $x = 2$)".

4 Beyond Fusion Closure

Although the automated procedures of [11, 12] were developed for fusion-closed specifications, they (may) still work for specifications which are not fusion closed only if the fault model has a certain pleasant form. For example, consider the system in Figure 5 and the specification

$$\textit{SPEC} = \text{"}(e \text{ implies previously } c) \text{ and (never } g)\text{"}$$

Obviously, the fault model F can be tolerated using the known transformation methods because F does not "exploit" the part of the specification which is not fusion closed.

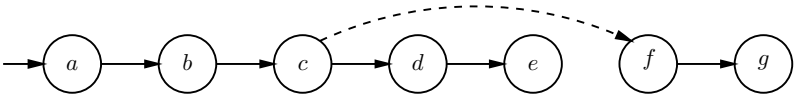

Figure 5: The fail-safe transformation can be successful even if the specification is not fusion closed. The specification in this case is "(e implies previously c) and (never g)".

4.1 Exploiting Non-Fusion Closure

Now we formalize what it means for a fault model to "exploit" the fact that a specification is not fusion-closed (we call this property *non-fusion closure*). First we define what it means for a trace to be the fusion of two other traces.

Definition 11 (fusion and fusion point of traces) *Let s be a state and $\alpha = \alpha_{pre} \cdot s \cdot \alpha_{post}$ and $\beta = \beta_{pre} \cdot s \cdot \beta_{post}$ be two traces in which s occurs. Then we define*

$$\mathit{fusion}(\alpha, s, \beta) = \alpha_{pre} \cdot s \cdot \beta_{post}$$

If $\mathit{fusion}(\alpha, s, \beta) \neq \alpha$ and $\mathit{fusion}(\alpha, s, \beta) \neq \beta$ we call s a fusion point *of α and β.*

Lemma 2 *For the fusion of three traces α, β, γ holds: If s occurs before s' in β then*

$$\mathit{fusion}(\alpha, s, \mathit{fusion}(\beta, s', \gamma)) = \mathit{fusion}(\mathit{fusion}(\alpha, s, \beta), s', \gamma)$$

and

$$\mathit{fusion}(\gamma, s', \mathit{fusion}(\alpha, s, \beta)) = \mathit{fusion}(\gamma, s', \beta)$$

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" [14]. 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: Assume that s occurs before s' in β. The proof is by direct calculation. Let $\alpha = \alpha_{pre} \cdot s \cdot \alpha_{post}$, $\beta = \beta_{pre} \cdot s \cdot \beta_{mid} \cdot s' \cdot \beta_{post}$, and $\gamma = \gamma_{pre} \cdot s' \cdot \gamma_{post}$. Then

$$\begin{aligned}
\mathit{fusion}(\alpha, s, \mathit{fusion}(\beta, s', \gamma)) &= \mathit{fusion}(\alpha, s, \beta_{pre} \cdot s \cdot \beta_{mid} \cdot s' \gamma_{post}) \\
&= \alpha_{pre} \cdot s \cdot \beta_{mid} \cdot s' \cdot \gamma_{post} \\
&= \mathit{fusion}(\alpha_{pre} \cdot \cdot s \cdot \beta_{mid} \cdot s' \cdot \beta_{post}, s', \gamma_{post}) \\
&= \mathit{fusion}(\mathit{fusion}(\alpha, s, \beta), s', \gamma)
\end{aligned}$$

proves the first equation and

$$\begin{aligned}
\mathit{fusion}(\gamma, s', \mathit{fusion}(\alpha, s, \beta)) &= \mathit{fusion}(\gamma, s', \alpha_{pre} \cdot s \cdot \beta_{mid} \cdot s' \cdot \beta_{post}) \\
&= \gamma_{pre} \cdot s' \cdot \beta_{post} \\
&= \mathit{fusion}(\gamma, s', \beta)
\end{aligned}$$

proves the second equation. □

If *SPEC* is a set of traces, we recursively define the fusion closure of *SPEC*, denoted by *fusion-closure*(*SPEC*), as the set which is closed under finite applications of the *fusion* operator.

Definition 12 (fusion closure) *Given a specification SPEC, a trace σ is in fusion-closure$(SPEC)$ iff*

1. *σ is in SPEC, or*
2. *σ = fusion(α, s, β) for traces $\alpha, \beta \in$ fusion-closure$(SPEC)$ and a state s that occurs in α and β.*

Lemma 2 guarantees that every trace in *fusion-closure*$(SPEC)$ which is not in *SPEC* has a "normal form", i.e., it can be represented uniquely as the sequence of fusions of traces in *SPEC*. This is shown in the following theorem.

Theorem 1 *For every trace $\sigma \in$ fusion-closure$(SPEC)$ which is not in SPEC there exists a sequence of traces $\alpha_0, \alpha_1, \alpha_2, \ldots$ and a sequence of states $s_1, s_2, s_3, \ldots$ such that*

1. *for all $i \geq 0$, $\alpha_i \in$ SPEC,*
2. *for all $i \geq 1$, s_i is a fusion point of α_{i-1} and α_i, and*
3. *σ can be written as:*

$$\sigma = \textit{fusion}(\textit{fusion}(\ldots \textit{fusion}(\alpha_0, s_1, \alpha_1), s_2, \alpha_2), s_3, \alpha_3), \ldots)$$

PROOF SKETCH: The proof is by induction on the structure of how σ evolved from traces in *SPEC*. Basically this means an induction on the number of fusion points in $sigma$. The induction step assumes that σ is the fusion of two traces which have at most n fusion points and depending on their relative positions uses the rules of Lemma 2 to construct the normal form for $sigma$.

1 $\langle 1 \rangle 1$. The theorem holds for all traces which have one fusion point.
PROOF: Since σ has one fusion point, it can be written as $\sigma =$ *fusion*$(\alpha_0, s_1, \alpha_1)$ with α_0 and α_1 from *SPEC* and s_1 a fusion point of α_0 and α_1. □

2 $\langle 1 \rangle 2$. ASSUME: The theorem holds for all traces with at most n fusion points.
PROVE: The theorem holds for all traces σ which are fusions of traces with at most n fusion points.
PROOF SKETCH: Take two traces τ and τ' which have at most n fusion points and which share an additional common fusion point s (see Fig. 6). The new fusion point s divides the fusion points in τ and τ' into two groups of k and m fusion points in τ (and k' and m' fusion points in τ' respectively). The fusion of both traces will maintain the k fusion points of τ and the m' fusion points of τ'. This follows from the second equation of Lemma 2. Because of the ordering of the fusion points we can use the first equation of Lemma 2 to construct the normal form. In general, the resulting trace can have more than n fusion points.

2.1 $\langle 2 \rangle 1$. σ can be written as $\sigma =$ *fusion*(τ, s, τ') where τ and τ' have at most n fusion points.
PROOF: Follows from the fact that σ is the fusion of two traces with at most n fusion points. □

2.2 $\langle 2 \rangle 2$. σ can be written as

$$\sigma = \textit{fusion}(\textit{fusion}(\ldots \textit{fusion}(\alpha_0, s_1, \alpha_1), s_2, \alpha_2) \ldots), s,$$
$$\textit{fusion}(\ldots \textit{fusion}(\alpha'_0, s'_1, \alpha'_1), s'_2, \alpha'_2) \ldots))$$

PROOF: Follows from the induction hypothesis and by replacing τ and τ' with their normal forms in the formula of step $\langle 2\rangle 1$. □

2.3 $\langle 2\rangle 3$. Let k, m, k' and m' denote the number of fusion points to the left and right of s in τ and τ' (see Fig. 6). Then σ can be written as

$$\textit{fusion}(\ldots \textit{fusion}(\alpha_0, s_1, \alpha_1), s_2, \alpha_2)\ldots), s,$$
$$\textit{fusion}(\ldots \textit{fusion}(\alpha'_{k'}, s'_{k'+1}, \alpha'_{k'+1}), s'_{k'+2}, \alpha'_{k'+2})\ldots))$$

PROOF: The first k' fusion points of τ' precede s and so by repeatedly applying the second equation of Lemma 2 we can remove the k' first applications of fusions from the formula of step $\langle 2\rangle 2$. □

2.4 $\langle 2\rangle 4$. σ can be written as

$$\textit{fusion}(\ldots \textit{fusion}(\alpha_0, s_1, \alpha_1), s_2, \alpha_2)\ldots),$$
$$s_k, \alpha_k), s, \alpha'_{k'}), s'_{k'+1}, \alpha'_{k'+1}), s'_{k'+2}, \alpha'_{k'+2})\ldots)$$

PROOF: From the definition of fusion, we can ignore the final m fusion points of τ. The formula follows by repeatedly applying the first equation of Lemma 2 to the formula of step $\langle 2\rangle 3$ (shifting the fusion operator to the left). □

2.5 $\langle 2\rangle 5$. Q.E.D.

PROOF: The formula of step $\langle 2\rangle 4$ has the required normal form because all α_i and α'_j are in *SPEC* and all s_i and s'_j are fusion points of consecutive elements in the formula. □

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

PROOF: Follows from induction. □

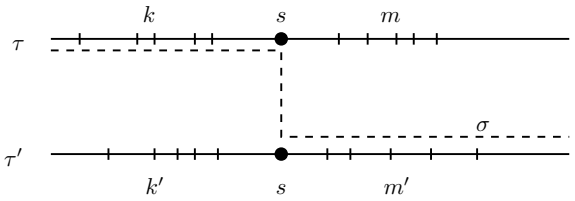

Figure 6: Diagram accompanying the proof of Theorem 1.

Now consider the system depicted in Figure 7. The corresponding specification is:

$$\textit{SPEC} = \text{“}f \text{ implies previously } d\text{”}$$

The system may exhibit the following two traces in the absence of faults, namely $\alpha = a \cdot b \cdot c$ and $\beta = a \cdot d \cdot e \cdot f$. In the presence of faults, a new trace is possible, namely $\gamma = a \cdot b \cdot e \cdot f$. Observe that γ violates *SPEC* and that γ is the fusion of two traces $\alpha, \beta \in$ *SPEC* (the state which plays the role of s in Definition 11 is state e). In such a case we say that fault model F exploits the non-fusion closure of *SPEC*.

We now formally define what is meant by exploiting the non-fusion closure of a specification.

Definition 13 (exploiting non-fusion closure) *Let Σ be a system, F be a fault model and SPEC be a specification which is satisfied by Σ. Then $F(\Sigma)$* exploits the non-fusion closure of *SPEC iff there exists a trace $\sigma \in F(\Sigma)$ such that $\sigma \notin$ SPEC and $\sigma \in$ fusion-closure(SPEC).*

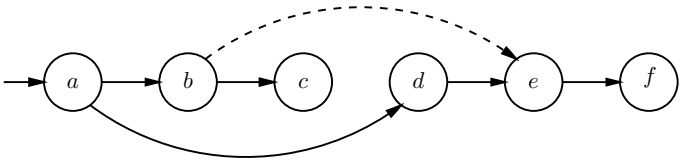

Figure 7: Example where the non-fusion closure of a specification is exploited by a fault model. The specification is "f implies previously d".

Intuitively, exploiting the non-fusion closure means that there exists a bad computation ($\sigma \notin$ *SPEC*) that can potentially "impersonate" a good computation ($\sigma \in$ *fusion-closure*(*SPEC*)). Definition 13 states that F causes a violation of *SPEC* by constructing a fusion of two (allowed) traces.

Given a fault model F such that $F(\Sigma)$ exploits the non-fusion closure of *SPEC*, then also we say that *the non-fusion closure of SPEC is exploited for* Σ *in the presence of* F.

Obviously, if for some specification *SPEC* and system Σ such an F exists, then *SPEC* is not fusion closed. Similarly trivial to prove is the observation that no fault model F can exploit the non-fusion closure of a specification which is fusion closed.

On the other hand, if the non-fusion closure of *SPEC* cannot be exploited, this does not necessarily mean that *SPEC* is fusion closed. To see this consider Figure 8. The correctness specification *SPEC* of the program is "c implies previously a". Obviously, a fault model can only generate traces that begin with a. Since a is an initial state and we assume initial state preservance, no F can exploit the non-fusion closure. But *SPEC* is not fusion closed.

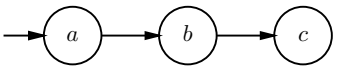

Figure 8: Example where the non-fusion closure cannot be exploited but the specification is not fusion closed. The specification is "c implies previously a".

4.2 Preventing the Exploitation of Non-Fusion Closure

The fact that a fault model may not exploit the non-fusion closure of a specification will be important in our approach to solve the general fail-safe transformation problem (Def. 8). A method to solve this problem, i.e., that of finding a fault-tolerant version Σ_2, should be a generally applicable method, which constructs Σ_2 from Σ_1 (this is depicted in the top part of Figure 9). Instead of devising such a method from scratch, our aim is to reuse the existing transformations to add fail-safe fault tolerance which are based on fusion-closed specifications [11, 12]. This approach is shown in the bottom part of Figure 9. Starting from Σ_1, we construct some intermediate program Σ_2' and some intermediate fusion-closed specification *SPEC*$_2$ to which we apply one of the above mentioned methods for fusion-closed specifications [11, 12]. The construction of Σ_2' and *SPEC*$_2$ must be done in such a way that the resulting program satisfies the properties of the general transformation problem stated in Definition 8. How can this

be done?

The idea of our approach is the following: First, choose SPEC_2 to be the fusion closure of SPEC_1, i.e., choose

$$\mathit{SPEC}_2 = \mathit{fusion\text{-}closure}(\mathit{SPEC}_1)$$

and construct Σ_2' from Σ_1 in such a way that $F(\Sigma_2')$ does not exploit the non-fusion closure of SPEC_1. More precisely, Σ_2' results from applying a constructive method (which we give below) which ensures that

- Σ_2' extends Σ_1 using some state projection π and
- $F(\Sigma_2')$ does not exploit the non-fusion closure of SPEC_1.

Our claim, which we formally prove later, is that the program Σ_2 resulting from applying (for example) the algorithms of [11, 12] to Σ_2' with respect to SPEC_2 in fact satisfies the requirements of Definition 8, i.e., Σ_2 is in fact an F-tolerant version of Σ_1 with respect to SPEC_1.

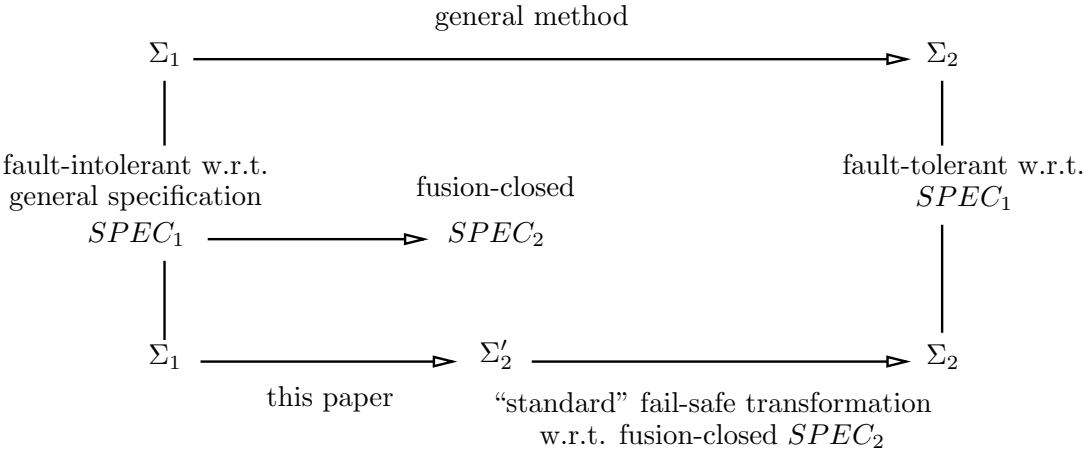

Figure 9: Overview of transformation problem (top) and our approach (bottom). The constructive method described in Section 4.3 offers a solution to the first step (i.e., $\Sigma_1 \rightarrow \Sigma_2'$).

4.3 Bad Fusion Points

For a given system Σ and a specification *SPEC*, how can we tell whether or not the nature of *SPEC* is exploitable by a fault model? For the negative case (where it can be exploited), we give a sufficient criterion. It is based on the notion of a *bad fusion point*.

Definition 14 (bad fusion point) *Let SPEC be a specification, Σ be a system satisfying SPEC, s be a state of Σ, and F a fault model such that $F(\Sigma)$ violates SPEC. State s is a* bad fusion point of Σ for *SPEC* in the presence of F *iff there exist traces $\alpha, \beta \in$ SPEC such that*

1. *s is a fusion point of α and β,*

2. *fusion$(\alpha, s, \beta) \in F(\Sigma)$, and*

3. *fusion$(\alpha, s, \beta) \notin$ SPEC.*

Intuitively, a bad fusion point is a state in which "multiple pasts" may have happened, i.e., there may be two different execution paths passing through s, and from the point of view of the specification it is important to tell the difference. We now give several examples of bad fusion points.

As an example, consider Fig. 7 where e is a bad fusion point. To instantiate the definition, take $\alpha = a \cdot b \cdot e \in F(\Sigma)$ and $\beta = a \cdot d \cdot e \cdot f \in F(\Sigma)$. The fusion at e yields the trace $a \cdot b \cdot e \cdot f$ which is not in *SPEC*.

Theorem 2 (bad fusion point criterion) *Let SPEC be a specification, Σ be a system satisfying SPEC and F be a fault model. The following two statements are equivalent:*

1. *Σ has no bad fusion point for SPEC in the presence of F.*

2. *$F(\Sigma)$ does not exploit the non-fusion closure of SPEC.*

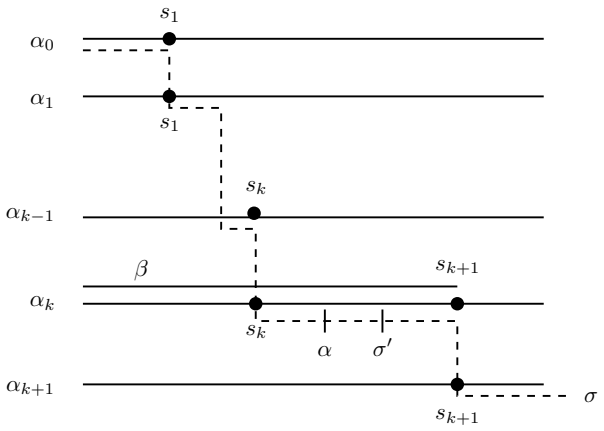

Figure 10: Diagram accompanying the proof of Theorem 2.

PROOF SKETCH: We prove the contraposition of the theorem in both directions. First we assume that $F(\Sigma)$ exploits the non-fusion closure and use Theorem 1 to construct a bad fusion point. Second we prove that if there exists a bad fusion point then $F(\Sigma)$ exploits the non-fusion closure.

1 $\langle 1 \rangle 1$. ASSUME: Σ has no bad fusion point for *SPEC* in the presence of F.
PROVE: $F(\Sigma)$ does not exploit the non-fusion closure of *SPEC*.

1.1 $\langle 2 \rangle 1$. ASSUME: $F(\Sigma)$ exploits the non-fusion closure of *SPEC*.
PROVE: False

1.1.1 $\langle 3 \rangle 1$. There exists a minimal prefix σ' of σ which violates *SPEC*.
PROOF: Follows from the fact that $\sigma \notin$ *SPEC* and that *SPEC* is a safety property. □

1.1.2 $\langle 3 \rangle 2$. σ' contains at least one fusion point.

PROOF: Since $\sigma \notin$ *SPEC* but $\sigma \in$ *fusion-closure*(*SPEC*) we can apply Theorem 1 and write σ as the fusion of traces $\alpha_i \in$ *SPEC* (see Fig. 10. If there were no fusion point within σ', then σ' would be a prefix of α_0, a contradiction to the fact that $\alpha_0 \in$ *SPEC*. □

1.1.3 $\langle 3 \rangle 3$. Let s denote the rightmost fusion point s_k in σ' and let α denote the prefix of σ' up to and including state s (see Fig. 10). Then $\alpha \in$ *SPEC*.

PROOF: Follows from the fact that σ' is minimal (i.e., prefixes of σ' satisfy *SPEC*, shown in step $\langle 3 \rangle 1$) and the fact that α is a prefix of σ'. □

1.1.4 $\langle 3 \rangle 4$. If there exists a fusion point s_{k+1} after s_k in α_k, let β be the trace α_k up to and including s_k (see Fig. 10). Otherwise let β be the trace α_k. Then $\beta \in$ *SPEC*.

PROOF: In both cases β is a prefix of α_k, which is in *SPEC* and so $\beta \in$ *SPEC* too. □

1.1.5 $\langle 3 \rangle 5$. *fusion*$(\alpha, s, \beta) \notin$ *SPEC*

PROOF: Follows from the fact that σ' is a prefix of *fusion*(α, s, β) and *SPEC* is a safety property (any extension of σ' is not in *SPEC*). □

1.1.6 $\langle 3 \rangle 6$. *fusion*$(\alpha, s, \beta) \in F(\Sigma)$

PROOF: Follows from the construction of α and s (in step $\langle 3 \rangle 3$) and β (in step $\langle 3 \rangle 4$) and the fact that *fusion*(α, s, β) is a prefix of σ which is in $F(\Sigma)$. □

1.1.7 $\langle 3 \rangle 7$. s is a bad fusion point for Σ in the presence of F.

PROOF: Steps $\langle 3 \rangle 3$ and $\langle 3 \rangle 4$ exhibit traces α and β which are both in *SPEC*. Step $\langle 3 \rangle 6$ shows that their fusion at state s is in $F(\Sigma)$. Finally, step $\langle 3 \rangle 5$ shows that this fusion is not in *SPEC*. From Definition 14 follows that s is a bad fusion point for Σ in the presence of F. □

1.1.8 $\langle 3 \rangle 8$. Q.E.D.

PROOF: Step $\langle 3 \rangle 7$ contradicts the assumption that Σ has no bad fusion point in the presence of F. □

1.2 $\langle 2 \rangle 2$. Q.E.D.

PROOF: Follows indirectly from step $\langle 2 \rangle 1$. □

2 $\langle 1 \rangle 2$. ASSUME: $F(\Sigma)$ does not exploit the non-fusion closure of *SPEC*.
PROVE: *SPEC* has no bad fusion point for Σ in the presence of F.

2.1 $\langle 2 \rangle 1$. ASSUME: *SPEC* has a bad fusion point for Σ in the presence of F.
PROVE: False

2.1.1 $\langle 3 \rangle 1$. There exists a trace σ in $F(\Sigma)$ such that $\sigma \notin$ *SPEC* and σ is the fusion of two traces α and β in *SPEC* at some state s.

PROOF: From assumption. □

2.1.2 $\langle 3 \rangle 2$. The non-fusion closure of *SPEC* can be exploited for Σ

PROOF: From step $\langle 3 \rangle 1$ and the definition of exploits (Definition 13) □

2.1.3 $\langle 3 \rangle 3$. Q.E.D.

PROOF: Step $\langle 3 \rangle 2$ contradicts the assumption of the theorem. □

2.2 $\langle 2 \rangle 2$. Q.E.D.

PROOF: Follows indirectly from step $\langle 2 \rangle 1$. □

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

PROOF: The two top level steps show both directions of the equivalence. □

4.4 Removal of Bad Fusion Points

Theorem 2 states that it is both necessary and sufficient to remove all bad fusion points from Σ to make its structure robust against fault models that exploit the non-fusion closure of *SPEC*. So how can we get rid of bad fusion points?

Recall that a bad fusion point is one which has multiple pasts, and from the point of view of the specification, it is necessary to distinguish between those pasts. Thus, the basic idea of our method is to introduce additional states which split the fusion paths. This is sketched in Figure 11. Let $\Sigma_1 = (C_1, I_1, T_1)$ be a system. If s is a bad fusion point of Σ_1 for *SPEC*, there exists a trace $\beta \in$ *SPEC* and a trace $\alpha \in F(\Sigma)$ which both go through s.

Constructive Method to Remove Bad Fusion Points: To remove bad fusion points, we now construct an extension $\Sigma_2 = (C_2, I_2, T_2)$ of Σ_1 in the following way:

- $C_2 = C_1 \cup \{s'\}$ where s' is a "new" state,
- $I_2 = I_1$, and
- T_2 results from T_1 by "diverting" the transitions of β to and from s' instead of s.

The extension is completed by defining the state projection function π to map s' to s. Observe that s is not a bad fusion point regarding α and β anymore because α now contains s and β a different state s' which cannot be fused. So this procedure gets rid of one bad fusion point. Also, it does not by itself introduce a new one, since s' is an extension state which cannot be referenced in *SPEC*. So we can repeatedly apply the procedure and incrementally build a sequence of extensions $\Sigma_1, \Sigma_2, \ldots$ where in every step one bad fusion point is removed and an additional state is added. However, F may cause new bad fusion points to be created during this process by introducing new faults transitions defined on the newly added states. But since the fault model is finite it will do this only finitely often. Hence, repeating this construction for every bad fusion point will terminate unless there are infinitely many bad fusion points. This, however, is impossible if the state space is finite.

Note that in the extension process, certain states can be extended multiple times because they might be bad fusion points for different combinations of traces.

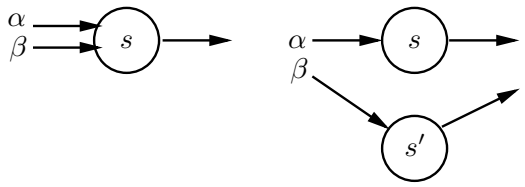

Figure 11: Splitting fusion paths.

We now prove that the above method results in a program with the desired properties.

Lemma 3 *Let F be a fault model, SPEC$_1$ be a non-fusion closed specification, and Σ_1 be a program such that Σ_1 satisfies SPEC$_1$ but $F(\Sigma_1)$ violates SPEC$_1$. The program*

Σ_2' which results from applying the constructive method described above satisfies the following properties:

1. *Σ_2' extends Σ_1 using some state projection π and*
2. *$F(\Sigma_2')$ does not exploit the non-fusion closure of SPEC$_1$.*

PROOF SKETCH: To show the first point we argue that there exists a projection function π (which is induced by our method) such that every fault-free execution of Σ_2' is an execution of Σ_1. To show the second point, we argue that the method removes all bad fusion points and apply the bad fusion point criterion proved as Theorem 2.

1 $\langle 1 \rangle 1$. The induced projection function π of the constructive method above is such that Σ_2' extends Σ_1 using π.

1.1 $\langle 2 \rangle 1$. For every state s of Σ_1 exists a π-image s' in the state space of Σ_2'.
PROOF: The constructive method starts off with the the state space of Σ_2' being equal to the state space of Σ_1 and any subsequent changes to π do not affect this initial mapping. □

1.2 $\langle 2 \rangle 2$. Consider an arbitrary fault-free execution $\sigma' = s_1', s_2', \ldots$ of Σ_2'. Then $\pi(\sigma')$ is an execution of Σ_1.
PROOF: Looking at Figure 11, every execution σ' of Σ_2' evolves from an execution of Σ_1 by splitting fusion paths and adapting π appropriately. Therefore, under the projection function π both executions look the same. Formally, this is proved using an induction on the length of the execution. □

1.3 $\langle 2 \rangle 3$. Q.E.D.
PROOF: Steps $\langle 2 \rangle 1$ and $\langle 2 \rangle 2$ prove the two conditions of Definition 3 (extension) with respect to the projection function π. Hence, Σ_2' extends Σ_1 using π. □

2 $\langle 1 \rangle 2$. $F(\Sigma_2')$ does not exploit the non-fusion closure of *SPEC*$_1$.

2.1 $\langle 2 \rangle 1$. Σ_2' has no bad fusion point in the presence of F.
PROOF: This is a result from applying the constructive method. Because all fusion paths are split, no fusion points remain. □

2.2 $\langle 2 \rangle 2$. Q.E.D.
PROOF: Because of step $\langle 2 \rangle 1$ we can apply the bad fusion point criterion (Theorem 2) which shows that the non-fusion-closure of *SPEC* cannot be exploited for Σ_2' in the presence of F. □

3 $\langle 1 \rangle 3$. Q.E.D.
PROOF: The above two steps show the two consequents of the lemma. □

4.5 Correctness of the Combined Method

Starting from a program Σ_1, Lemma 3 shows that the program Σ_2' resulting from the constructive method for removing bad fusion points enjoys certain properties (see Fig. 9). We now prove that starting off from these properties and choosing *SPEC*$_2$ as the fusion closure of *SPEC*$_1$, the program Σ_2, which results from applying the algorithms of [11, 12] on Σ_2', has the desired properties of the transformation problem (Definition 8).

Lemma 4 *Given F, SPEC$_1$, and Σ_1 as in Lemma 3, let SPEC$_2$ = fusion-closure(SPEC$_1$) and let Σ_2 be the result of applying any of the known methods that solve the fusion-closed transformation problem of Definition 9 to Σ_2' with respect to F and SPEC$_2$,*

where Σ_2' results from Σ_1 through the application of the constructive method. Then the following statements hold:

1. *Σ_2 extends Σ_1 using some state projection π.*
2. *If $F(\Sigma_2)$ satisfies **SPEC**$_2$ then $F(\Sigma_2)$ satisfies **SPEC**$_1$.*

PROOF SKETCH: To prove the first point we argue that a fault tolerance addition procedure only removes non-reachable transitions. Hence, every fault-free execution of Σ_2' is also an execution of Σ_2. But since Σ_2' extends Σ_1 so must Σ_2. To show the second point we first observe that $F(\Sigma_2')$ does not necessarily satisfy ***SPEC***$_1$ but not all traces for this are in $F(\Sigma_2)$ anymore (due to the removal of bad transitions during addition of fault tolerance). Next we show that any trace of $F(\Sigma_2)$ which violates ***SPEC***$_1$ must exploit the non-fusion closure of ***SPEC***$_1$. But this must also be a trace of $F(\Sigma')$ and so is ruled out by assumption.

1 $\langle 1\rangle 1$. If Σ_2' extends Σ_1 using state projection π' then Σ_2 extends Σ_1 using state projection π

1.1 $\langle 2\rangle 1$. Application of the known methods to add fail-safe fault tolerance according to Definition 9 does not change the fault-free behavior of that system.
PROOF: For the methods of Kulkarni and Arora [12] and Jhumka *et al.* [11] this has been discussed in Section 3. □

1.2 $\langle 2\rangle 2$. Every (fault-free) execution of Σ_2' is also a (fault-free) execution of Σ_2 and vice versa.
PROOF: Follows from step $\langle 2\rangle 1$ and the fact that Σ_2 results from Σ_2' by applying the fail-safe-tolerance transformation (see Fig. 9). □

1.3 $\langle 2\rangle 3$. Every execution of Σ_2' under π' is an execution of Σ_1.
PROOF: Follows from the assumption that Σ_2' extends Σ_1 using π'. □

1.4 $\langle 2\rangle 4$. Every execution of Σ_2 is also an execution of Σ_1 under π' and vice versa.
PROOF: Starting with an arbitrary execution σ of Σ_2, step $\langle 2\rangle 2$ allows to find an equivalent execution σ' of Σ_2'. Then for σ', step $\langle 2\rangle 3$ allows to find an equivalent execution σ'' of Σ_1. □

1.5 $\langle 2\rangle 5$. Q.E.D.
PROOF: Step $\langle 2\rangle 4$ allows to construct a state projection function such that the safety properties of Σ_1 and Σ_2 are identical. Hence, Σ_2 extends Σ_1. □

2 $\langle 1\rangle 2$. ASSUME: 1. $F(\Sigma_2')$ does not exploit the non-fusion closure of ***SPEC***$_1$.
2. $F(\Sigma_2)$ satisfies ***SPEC***$_2$.
PROVE: $F(\Sigma_2)$ satisfies ***SPEC***$_1$.

2.1 $\langle 2\rangle 1$. All executions σ of $F(\Sigma_2')$ that violate ***SPEC***$_1$ are not in $F(\Sigma_2)$.
PROOF: This follows from applying a fail-safe tolerance transformation procedure, such as those in [11, 12]. Since these procedures are proved to be sound, i.e., the resulting programs are indeed fail-safe fault-tolerant, then no execution can violate the specification. □

2.2 $\langle 2\rangle 2$. $\forall \sigma \in F(\Sigma_2) : \sigma \in$ ***SPEC***$_2$
PROOF: Follows directly from second assumption, i.e., $F(\Sigma_2)$ satisfies ***SPEC***$_2$. □

2.3 $\langle 2\rangle 3$. $\forall \sigma \in F(\Sigma_2) : \sigma \in F(\Sigma_2')$
PROOF: The known fail-safe tolerance transformation procedures that solve Definition 9 guarantee that $F(\Sigma_2) \subseteq F(\Sigma_2')$, from which this step follows. □

2.4 $\langle 2\rangle 4$. $F(\Sigma_2)$ does not exploit non-fusion closure of ***SPEC***$_1$.

PROOF: For a contradiction, assume that there is an execution $\tau \in F(\Sigma_2)$ that exploits non-fusion closure of ***SPEC***$_1$. Since $\tau \in F(\Sigma_2)$, from step $\langle 2 \rangle 3$ we have that $\tau \in F(\Sigma_2')$. Hence, $F(\Sigma_2')$ also exploits the non-fusion closure of ***SPEC***$_1$, a contradiction to assumption 2. □

2.5 $\langle 2 \rangle 5$. $\forall \sigma \in F(\Sigma_2) : \sigma \in$ ***SPEC***$_1$

2.5.1 $\langle 3 \rangle 1$. ASSUME: $\sigma \in \Sigma_2$
PROVE: QED
PROOF: Since $\sigma \in \Sigma_2$ and Σ_2 extends Σ_1 we have that $\sigma \in \Sigma_1$. But since Σ_1 satisfies ***SPEC***$_1$ we conclude that $\sigma \in$ ***SPEC***$_1$. □

2.5.2 $\langle 3 \rangle 2$. ASSUME: $\sigma \in F(\Sigma_2) \setminus \Sigma_2$
PROVE: QED
PROOF: First note that σ cannot be in *fusion-closure*(***SPEC***$_1$)$\setminus$***SPEC***$_1$ (follows from step $\langle 2 \rangle 4$). But since *fusion-closure*(***SPEC***$_1$) = ***SPEC***$_2$ and since $F(\Sigma_2)$ satisfies ***SPEC***$_2$ we have that σ must be in ***SPEC***$_1$. □

2.5.3 $\langle 3 \rangle 3$. Q.E.D.
PROOF: Follows from steps $\langle 3 \rangle 1$ and $\langle 3 \rangle 2$ and the fact that they cover all cases. □

2.6 $\langle 2 \rangle 6$. Q.E.D.
PROOF: Step $\langle 2 \rangle 5$ shows that $F(\Sigma_2)$ satisfies ***SPEC***$_1$ which is what we wanted to prove. □

3 $\langle 1 \rangle 3$. Q.E.D.
PROOF: Steps $\langle 1 \rangle 1$ and $\langle 1 \rangle 2$ prove the first and second point of the lemma, respectively. □

Lemmas 3 and 4 together guarantee that the composition of the method described in Section 4.3 and the fail-safe transformation methods for fusion-closed specifications in fact solves the transformation problem for non-fusion closed specifications of Definition 8.

Theorem 3 *Given a fault model F and a program Σ_1 which is F-intolerant with respect to a non-fusion closed specification* ***SPEC***$_1$. *The composition of the constructive method described in Section 4.3 and the fail-safe transformation methods for fusion-closed specifications solves the general transformation problem of Definition 8, i.e., constructs a program Σ_2 such that Σ_2 extends Σ_1 and $F(\Sigma_2)$ satisfies* ***SPEC***$_1$.

4.6 Examples

Finally, we present two examples of the application of our method. The top of Figure 12 (system 1) shows the original system. The augmented system is depicted at the bottom (system 4). The correctness specification for the system is "(d implies previously b) and (e implies previously c)". There are only two bad fusion points, namely c and d which have to be extended. In the first step, c is "removed" by splitting the fusion path which is indicated using two short lines. This results in system 2. Subsequently, d is refined, resulting in system 3. Note that d has to be refined twice because there are two sets of fusion paths. This results in system 4, which can be subject to the standard fail-safe transformation methods, which will remove the transitions (c, d'') and (d, e).

A similar, yet more complex example is shown in Figure 13. The correctness specification for the system 1 at the top is "g implies previously (b or c)". The figure

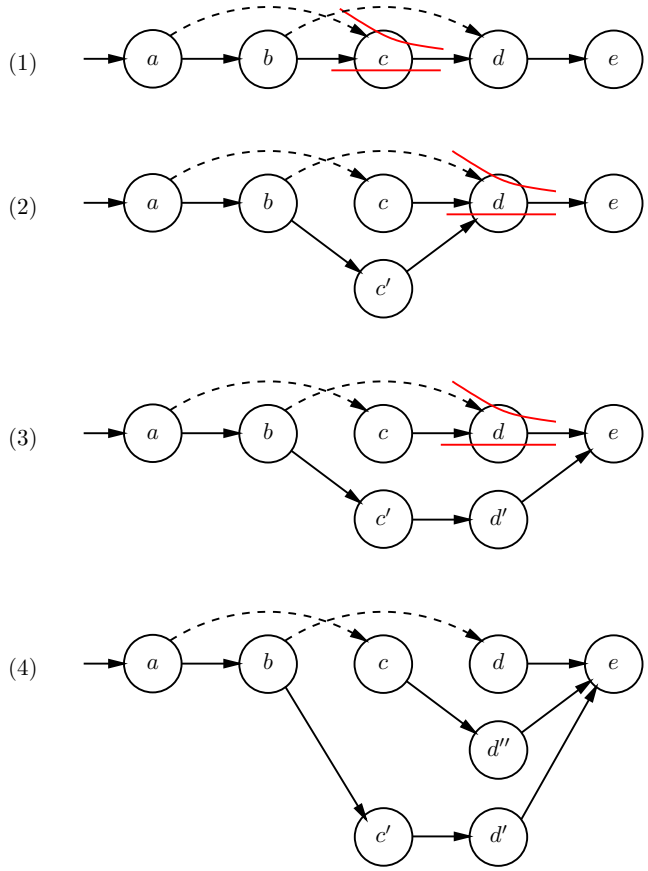

Figure 12: Removing bad fusion points. The specification is "(d implies previously b) and (e implies previously c)".

shows that again a "two level" extension is necessary here, since the only execution which must be prevented is the one which uses *both* fault transitions. This means that state f is a bad fusion point for multiple execution paths and hence must be refined twice (note that the fault transition (d, f) is a new fault added to the system in the extension).

4.7 Discussion

The complexity of our method directly depends on the number of bad fusion points which have to be removed. Bad fusion points are not hard to find if the specification is given as a temporal logic formula in the spirit of those used throughout this paper. For example, if specifications are given in the form "x only if previously y" then only states which occur in traces between x and y can be fusion points. Candidates for *bad* fusion points are all states where two execution paths merge.

Our method requires to check every one of these states whether it is a bad fusion point. So obviously, applying our method induces a larger overhead than directly adding history variables. But as can be seen in Figs. 12 and 13, the number of states is significantly less than adding a general history variable. For example, a clever addition of history variables to the system im Fig. 12 would require two bits, one to record the visit to state b and one to record the visit to c. Overall this would result in $2 \times 2 \times 5 = 20$ states. Our methods achieves the same result with a total of 8 states. The system in Fig. 13 could employ a boolean history variable which records whether states b or e have been visited (it is set to *true* as soon as one of these states is reached). Adding such a variable would create a total of 7 additional states. Our methods just adds 5.

Note however that the resulting system in Fig. 12 is not redundancy minimal. The state d'' is not necessary since it may become unreachable even in the presence of faults after the fail-safe transformation is applied. This is the price we still have to pay for the modularity of our approach, i.e., adding history states does at present not "look ahead" which states might become unreachable even in the presence of faults.

In theory there are cases where our method of adding history states does not terminate because there are infinitely many bad fusion points. For this to happen, the state space must be infinite. If we consider the application area of embedded software, we can safely assume a bounded state space.

Given a program Σ and a general specification *SPEC*, then our combined method will find a solution to the general transformation problem iff (a) there exists one with a finite number of additional states and (b) the method of adding fail-safe fault-tolerance for fusion-closed specifications is complete. Requirement (a) ensures that our method of removing bad fusion points will terminate.

5 Conclusions

In this paper, we have presented ways on how get rid of a restriction upon which procedures that add fault tolerance [11, 12] are based, namely that specifications have to be fusion closed. Our method can be viewed as a finer grained method to add history information to a given system and hence add state space redundancy. We have shown that our method in general adds less history states than would be added using

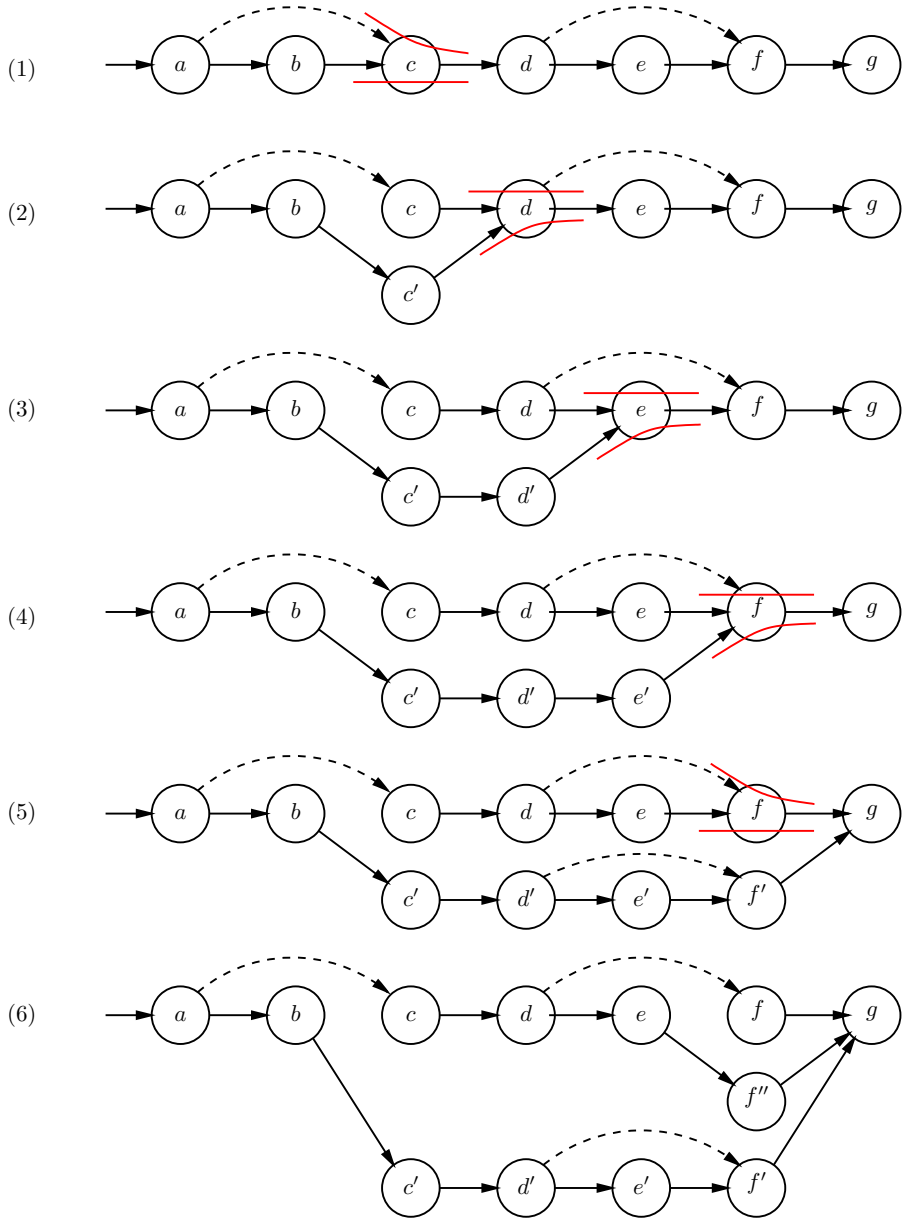

Figure 13: A more complex example. The specification is "g implies previously (b or e)".

standard history variables (which in general lead to an exponential growth of the state space). Thus, adding state redundancy using the approach presented in this paper makes addition of fault tolerance more efficient.

As future work, it would be interesting to combine our method with one of the methods to add detectors so that the resulting method is redundancy minimal. We are also investigating issues of non-masking fault-tolerance, i.e, adding tolerance with respect to liveness properties.

Acknowledgments

We wish to thank Sandeep Kulkarni for helpful discussions. Work by the first author was supported by Deutsche Forschungsgemeinschaft (DFG) as part of "Graduiertenkolleg ISIA" and Emmy Noether programme.

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 S. Kulkarni. Component based design of multitolerant systems. *IEEE Transactions on Software Engineering*, 24(1):63–78, January 1998.

[4] Anish Arora and Sandeep S. 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.

[5] Anindya Basu, Bernadette Charron-Bost, and Sam Toueg. Simulating reliable links with unreliable links in the presence of process crashes. In *Proceedings of the 10th International Workshop on Distributed Algorithms (WDAG96)*, pages 105–122, Bologna, Italy, October 1996. Springer-Verlag.

[6] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, MA, Reading, Mass., 1988.

[7] Edsger W. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[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] Felix C. Gärtner and Hagen Völzer. Redundancy in space in fault-tolerant systems. Technical Report TUD-BS-2000-06, Department of Computer Science, Darmstadt University of Technology, Darmstadt, Germany, July 2000.

[10] H. Peter Gumm. Another glance at the Alpern-Schneider characterization of safety and liveness in concurrent executions. *Information Processing Letters*, 47(6):291–294, 1993.

[11] Arshad Jhumka, Felix C. Gärtner, Christof Fetzer, and Neeraj Suri. On systematic design of fast and perfect detectors. Technical Report 200263, Swiss Federal Institute of Technology (EPFL), School of Computer and Communication Sciences, Lausanne, Switzerland, September 2002.

[12] Sandeep S. Kulkarni and Anish Arora. Automating the addition of fault-tolerance. In Mathai Joseph, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems, 6th International Symposium (FTRTFT 2000) Proceedings*, number 1926 in Lecture Notes in Computer Science, pages 82–93, Pune, India, September 2000. Springer-Verlag.

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

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

[15] Zhiming Liu and Mathai Joseph. Specification and verification of fault-tolerance, timing and scheduling. *ACM Transactions on Programming Languages and Systems*, 21(1):46–89, 1999.

[16] Heiko Mantel and Felix C. Gärtner. A case study in the mechanical verification of fault tolerance. *Journal of Experimental & Theoretical Artificial Intelligence (JETAI)*, 12(4):473–488, October 2000.

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