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

Felix C. Gärtner
RWTH Aachen
Lehr- und Forschungsgebiet Informatik IV
D-52056 Aachen, Germany

Arshad Jhumka
Technische Universität Darmstadt
Fachbereich Informatik
D-64283 Darmstadt, Germany

Abstract

The fault tolerance theories by Arora and Kulkarni [3] and by Jhumka *et al.* [10] 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 significantly reduces the number of additional states. Hence, automated methods that add fault tolerance can now be efficiently applied in environments where specifications are not necessarily fusion closed.

Contact author:

Felix Gärtner
e-mail: `fcg@acm.org`
Telephone: +49-241-80-21430
Fax: +49-241-22220

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. This technique has been applied in the area of fault-tolerant distributed 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 [16] to theoretical studies like the one by Basu *et al.* [5]. 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 [11] and more recently Jhumka *et al.* [10] 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 (weakened) 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 [8] 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 [9] (since one cannot produce specifications that relate to the history of the computations), as are specifications consisting of state transition systems (like C programs, since it is mostly data-flow like). But general temporal logic formulas which are usually used in the area of fault-tolerant program synthesis and refinement [14, 15] 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 "never ($x = 4$ and $(x = 2) \notin h$)". 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 [8], 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 avoids exponential growth of the state space, but rather causes a polynomial increase in the size of the state space in the worst case. 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 programs which we consider are non-deterministic state machines and so our application domain is that of distributed or concurrent systems.

The benefit of the proposed method is the following: Firstly, it makes the methods which automatically add detectors [10, 11] 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 sketch 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.

For lack of space, we only give proof sketches for theorems and lemmas. The detailed proofs can be found in the full version of this work [7].

2 Formal Preliminaries

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. 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]. In this paper, we are only concerned with safety properties. 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. Formally, a property S over C is a *safety property* iff 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.

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 *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. A 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.

We say that program Σ *satisfies* specification *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*.

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 $\pi(s_1, s_2, \ldots) = \pi(s_1), \pi(s_2), \ldots$

We say that a program $\Sigma_1 = (C_1, I_1, T_1)$ *extends* a program $\Sigma_2 = (C_2, I_2, T_2)$ using state projection π iff the following conditions hold:[1]

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).

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*".

Fault Models. We define a fault model F as being a program transformation, i.e., a mapping F from programs to programs. 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

[1]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].

restrict ourselves to the case where F "adds" transitions, since this is the only way to violate a safety specification. Formally, a *fault model* is a mapping F which 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. $F(T) \supset T$

The resulting program is called the *fault-affected version* or the *program in the presence of faults*. We say that a program Σ is *F-intolerant with respect to SPEC* iff 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. Formally, a fault model F must be *extension monotonic*, i.e., 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:

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

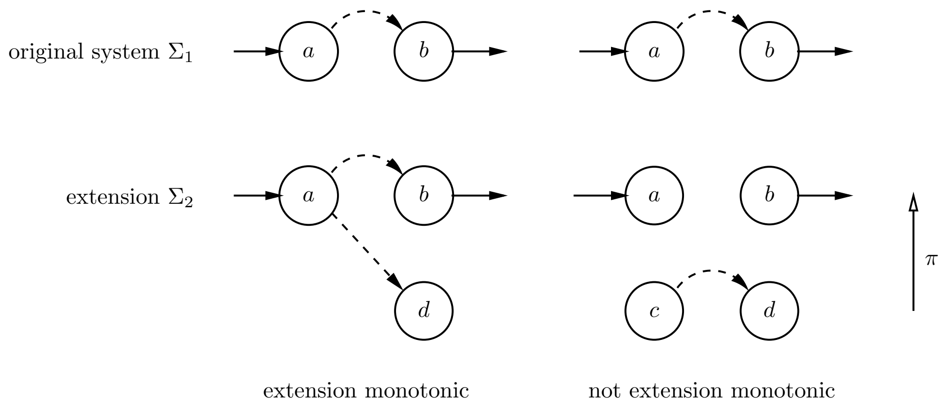

Figure 1: Examples for extension monotonic and not extension monotonic fault models.

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. Formally, an 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.

Fault-tolerant Versions. 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. Formally, 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_1 is F-intolerant with respect to F,
2. Σ_2 extends Σ_1 using π,
3. $F(\Sigma_2)$ satisfies *SPEC*.

In the remainder of this paper, F is a fault model, Σ, Σ_1 and Σ_2 are programs, *SPEC*, *SPEC*$_1$ and *SPEC*$_2$ are specifications.

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 1 (general fail-safe transformation problem) *Given 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.*

The case where *SPEC* is fusion closed has been studied by Kulkarni and Arora [11] and Jhumka *et al.* [10], i.e., they solve a restricted transformation problem where *SPEC*$_1$ is fusion-closed. We briefly recall the known solutions to this problem.

Solutions for Fusion-Closed Specifications. The basic mechanism which Kulkarni and Arora [11] and Jhumka *et al.* [10] 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, 9].

For a finite computation α 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 follows 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*.

The known automated procedures [10, 11] 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.

Adding History Variables. Consider program with one variable x which can take five different values (integers 0 to 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. Now consider the correctness specification *SPEC* = "always ($x = 4$ implies that previously $x = 2$)". 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 [10, 11].

The specification can be made fusion closed by adding a history variable h which records the entire state history. Now *SPEC* can be rephrased as *SPEC* = "always ($x = 4$ implies $\langle 2 \rangle \in h$)" or, equivalently, *SPEC* = "never ($x = 4$ and $\langle 2 \rangle \notin h$)". Now we can identify a set of bad transitions which must be prevented, e.g., from state $x = 3 \wedge h = \langle 1 \rangle$ to state $x = 4 \wedge h = \langle 1, 2, 3, 4 \rangle$. This means that all transitions to a state where $x = 4$ holds must be removed unless in the starting state $h \neq \langle 1 \rangle$ is true. In this way bad transitions are prevented and the modified system satisfies *SPEC* in the presence fault f.

Problems with 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 depicted in Figure 2. It tolerates the fault f by adding only *one* state to the state space of Σ_1 (namely, $x = 5$). 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.

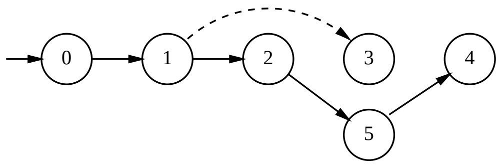

Figure 2: A redundancy-minimal version fault-tolerant program. The specification is "always ($x = 4$ implies that previously $x = 2$)".

4 Beyond Fusion Closure

Although the automated procedures of [10, 11] 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 3 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.

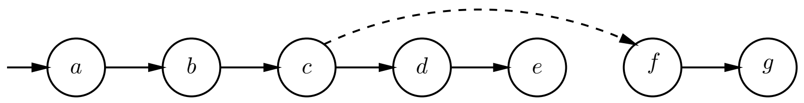

Figure 3: The fail-safe transformation can be successful even if the specification is not fusion closed. The specification in this case is "(e implies previously c) and (never g)".

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 2 (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*

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

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

Lemma 1 *For the fusion of three traces α, β, γ holds: If s occurs before s' in β then*

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

and

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

If *SPEC* is a set of traces, we recursively define an operator to generate the fusion closure of *SPEC*, denoted by *fusion-closure*$($*SPEC*$)$. It produces a set which is closed under finite applications of the *fusion* operator.

Definition 3 (fusion closure) *Given a specification SPEC, a trace* σ *is in fusion-closure*$(SPEC)$ *iff*

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

Lemma 1 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* $\sigma =$ *fusion*$($*fusion*$(\ldots$ *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 1 to construct the normal form for σ. □

Now consider the system depicted in Figure 4. The corresponding specification is: *SPEC* $=$ "f implies previously d". 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 2 is state e). In such a case we say that fault model F exploits the non-fusion closure of *SPEC*.

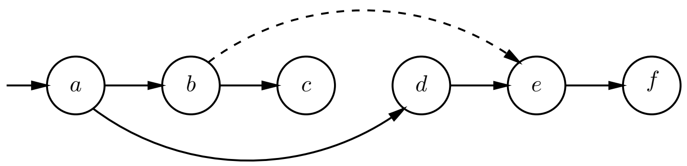

Figure 4: Example where the non-fusion closure of a specification is exploited by a fault model. The specification is "f implies previously d".

Definition 4 (exploiting non-fusion closure) *Let* Σ *satisfy SPEC. 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*$)$.

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 4 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 5. 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 that initial states are not changed by F, no F can exploit the non-fusion closure. But *SPEC* is not fusion closed.

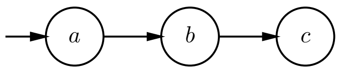

Figure 5: Example where the non-fusion closure cannot be exploited but the specification is not fusion closed. The specification is "c implies previously a".

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. 1). 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 6). 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 [10, 11]. This approach is shown in the bottom part of Figure 6. 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 [10, 11]. 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 1. 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 an algorithm (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 [10, 11] to Σ_2' with respect to SPEC_2 in fact satisfies the requirements of Definition 1, i.e., Σ_2 is in fact an F-tolerant version of Σ_1 with respect to SPEC_1.

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 5 (bad fusion point) *Let Σ be F-intolerant with respect to SPEC. State s of Σ is a* bad fusion point of Σ for *SPEC* in the presence of F *iff there exist traces $\alpha, \beta \in$ SPEC such that*

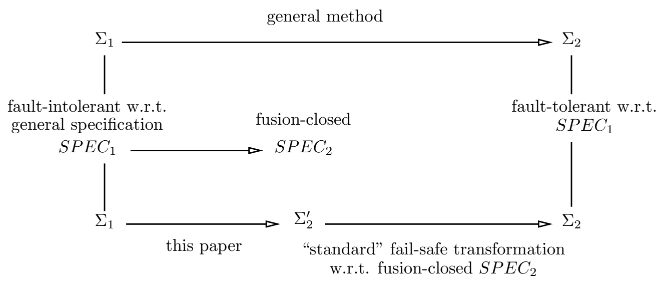
general method
Σ_1
Σ_2
fault-intolerant w.r.t. general specification $SPEC_1$
fusion-closed $SPEC_2$
fault-tolerant w.r.t. $SPEC_1$
Σ_1
Σ_2'
Σ_2
this paper
"standard" fail-safe transformation w.r.t. fusion-closed $SPEC_2$

Figure 6: Overview of transformation problem (top) and our approach (bottom). Algorithm 1 described in this paper offers a solution to the first step (i.e., $\Sigma_1 \to \Sigma_2'$).

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. 4 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) *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.*

PROOF SKETCH: The main difficulty is to prove that if *SPEC* has no bad fusion point then $F(\Sigma)$ cannot exploit the non-fusion closure. We prove this by assuming that $F(\Sigma)$ exploits the non-fusion closure and using Theorem 1 to construct a bad fusion point. □

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 7. 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.

Algorithm 1 (Removal of 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 because we assume that 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.

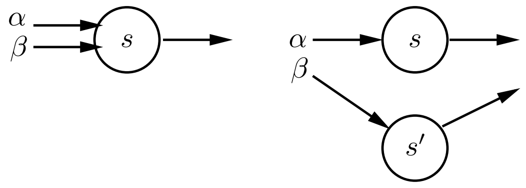

Figure 7: Splitting fusion paths.

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

Lemma 2 *Let Σ_1 be F-intolerant with respect to a general specification SPEC$_1$. The program Σ_2' which results from applying Algorithm 1 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 of Theorem 2. □

Correctness of the Combined Method. Starting from a program Σ_1, Lemma 2 shows that the program Σ_2' resulting from Algorithm 1 for removing bad fusion points enjoys certain properties (see Fig. 6). 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 [10, 11] on Σ_2', has the desired properties of the transformation problem (Definition 1).

Lemma 3 *Given F, SPEC$_1$, and Σ_1 as in Lemma 2, 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 to Σ_2' with respect to F and SPEC$_2$, where Σ_2' results from Σ_1 through the application of Algorithm 1. 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. □

Lemmas 2 and 3 together guarantee that the composition of the method described in Section 1 and the fail-safe transformation methods for fusion-closed specifications in fact solves the transformation problem for non-fusion closed specifications of Definition 1.

Theorem 3 *Let Σ_1 be F-intolerant with respect to a general specification SPEC$_1$. The composition of Algorithm 1 and the fail-safe transformation methods for fusion-closed specifications solves the general transformation problem of Definition 1.*

Examples. Finally, we present two examples of the application of our method. The top of Figure 8 (left hand side, system 1) shows the original system. The augmented system is depicted at the bottom (left hand side, 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 on the right hand side of Fig. 8. The correctness specification for the system 1 at the top is "g implies previously (b or c)". The figure 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).

Discussion. The complexity of our method directly depends on the number of bad fusion points which have to be removed and finding bad fusion points by directly applying Def. 5 is clearly infeasible even for moderately sized systems. However, 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 states x and y can be fusion points. Candidates for *bad* fusion points are all states where two execution paths merge and this can easily be checked from the transition diagram of the system.

Our method requires to check every possible fusion point whether it is a bad one. So obviously, applying our method induces a larger computational overhead during the transformation process than directly adding history variables. But as can be seen in Fig. 8, the number of states is significantly less than adding a general history variable. For example, a clever addition of history variables to the

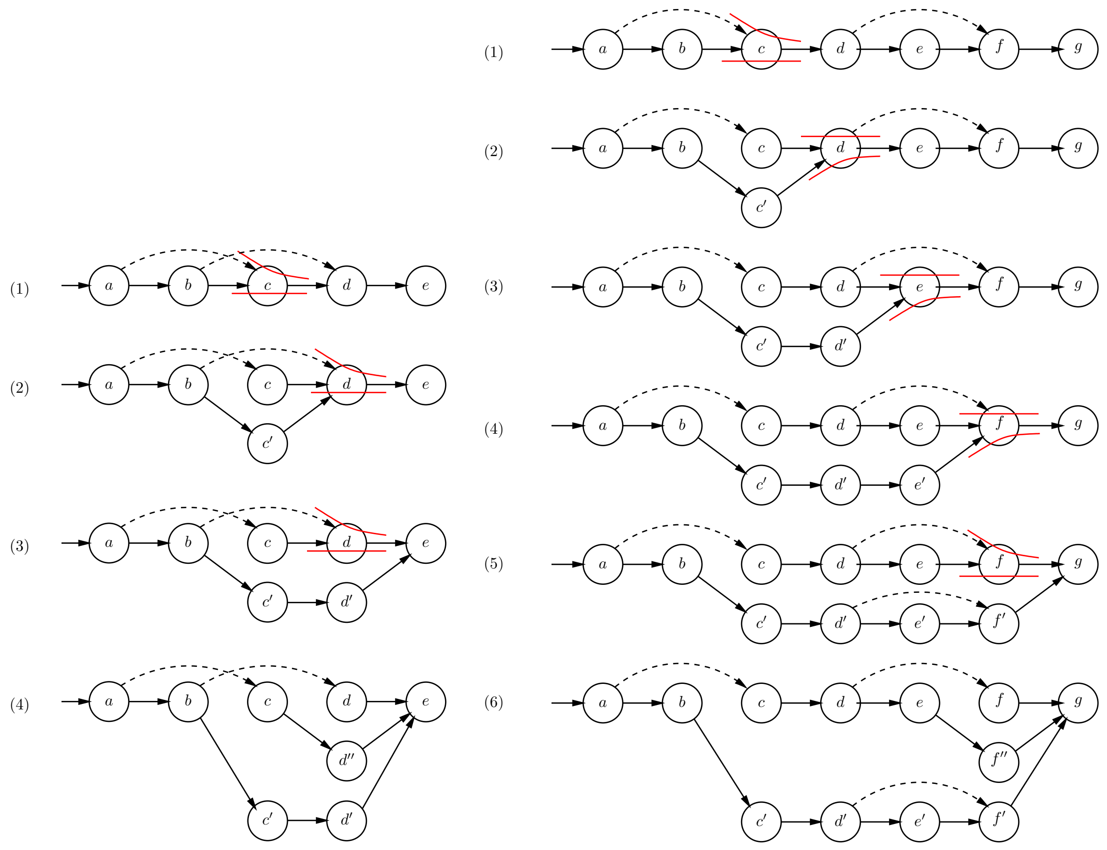
Figure 8: Removing bad fusion points. The specification for the system on the left is "(d implies previously b) and (e implies previously c)". For the system on the right it is "g implies previously (b or e)".

system in Fig. 8 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.

Although it can happen that states are refined multiple times, the number of bad fusion points (and hence the number of added states) only depends on the specification and the fault model whereas the number of added states using history variables depends on the size of the state space. For example, a program with n states will have $2 \times n$ states after adding just one bit of history information.

In general, the worst case scenario in our scheme is when every state is a bad fusion point. Assuming there are n states in the system, there are $O(n)$ bad fusion points. Assuming that faults do not affect the refined states (which, for sake of comparison is realistic since we do not assume faults to affect history variables, whenever they are used), every bad fusion point is refined, giving rise to $O(n)$ refined states. $O(n)$ bad fusion states thus give rise to $O(n^2)$ refined states. In the worst case scenario, our scheme adds an additional $O(n^2)$ states, as compared to the exponential number of additional states added by using history variables.

Note however that the resulting system in Fig. 8 is not redundancy minimal if the entire transformation problem is considered. 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.

5 Conclusions

In this paper, we have presented ways on how get rid of a restriction upon which procedures that add fault tolerance [10, 11] are based, namely that specifications have to be fusion closed. Apart from closing a gap in the detector/corrector theory [3], our method can be viewed as a finer grained method to add history information to a given system and hence add state space redundancy. This also helps to understand the principles of fault-tolerant system operation. We have shown that our method in general adds less history states than would be added using 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. At present, we are implementing our approach within the fault-tolerance synthesis framework SYNFT of Michigan State University [12].

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

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] Felix C. Gärtner and Arshad Jhumka. Automating the addition of fail-safe fault-tolerance: Beyond fusion-closed specifications. Technical Report IC/2003/23, Swiss Federal Institute of Technology (EPFL), School of Computer and Communication Sciences, Lausanne, Switzerland, April 2003.

[8] Felix C. Gärtner and Hagen Völzer. Defining redundancy in fault-tolerant computing. In *Brief Announcement at the 15th International Symposium on DIStributed Computing (DISC 2001)*, Lisbon, Portugal, October 2001.

[9] 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.

[10] 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.

[11] 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.

[12] Sandeep S. Kulkarni and Ali Ebnenasir. SYNFT: A framework for adding fault-tolerance to distributed programs. Available via email from the authors at Michigan State University, USA, 2003.

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

[14] 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.

[15] 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.

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