Automated Design of Efficient Fail-Safe Fault Tolerance

Vom Fachbereich Informatik der Technischen Universität Darmstadt genehmigte

Dissertation

zur Erlangung des akademischen Grades eines Doctor rerum naturalium (Dr.rer.nat) vorgelegt von

Arshad Jhumka

aus Rose-Hill, Mauritius

Referenten:
Prof. Dr. Neeraj Suri
Prof. Dr. Mario Dal Cin

Datum der Einreichung: 10.09.2003
Datum der mündlichen Prüfung: 10.11.2003

Darmstädter Dissertationen D17

Erklärung

Hiermit erkläre ich, die vorgelegte Arbeit zur Erlangung des akademischen Grades “Dr.rer.nat” mit dem Titel “Automated Design of Efficient Fail-safe Fault Tolerance” selbständig und ausschliesslich unter Verwendung der angegebenen Hilfsmittel erstellt zu haben. Ich habe bisher noch keinen Promotionsversuch unternommen.

Darmstadt, February 11, 2004 Arshad Jhumka

Abstract

Both the scale and the reach of computer systems and embedded devices have been constantly increasing over the last decade. As such computer systems become pervasive, our reliance on such systems increases, resulting in our expectation for such systems to continuously deliver services, even in the presence of faults, that is we expect the computer systems to be dependable. One way to ensure the continuous delivery of dependable services is replication, which however, is expensive, so we focus on the cheaper alternative, that of software-based fault tolerance.

There are different levels of fault tolerancethat can be provided, for example masking fault tolerance, fail-safe fault tolerance etc. In this thesis, we focus on providing fail-safe fault tolerance. Intuitively, a fail-safe fault-tolerant program is one where it is acceptable for such a program to "halt" when faults occur, as long as it always remains in a "safe" state. Moreover, we endeavor to synthesize efficient fail-safe fault tolerance. We used two commonly-used criteria to assess the efficiency of a fail-safe fault-tolerant program, namely (i) *error detection latency* – or latency for short –, i.e., how *fast* can a fail-safe fault-tolerant program detect an erroneous state, and (ii) *error detection coverage* – or coverage for short, i.e., the *ratio* of "harmful" errors the program can detect.

In this thesis, we present a formal framework for the design of efficient fail-safe fault-tolerant program. The framework is based on a refined theory of detectors, which introduces novel insights into their working principles. We introduce the concept of a *perfect detector*, which allows a fail-safe fault-tolerant program to have perfect detection. This means that a program, composed with perfect detectors, have optimal detection coverage. Optimal in the sense that the detectors detect all of the "harmful" errors, and make no mistakes. Then, we present the concept of fast detection, and show how a fail-safe fault-tolerant program can have both perfect, and fast error detection. In fact, the detection latency is shown to be minimal, i.e., the error is detected in 0-step. Based on these two basic notions, we present algorithms that automatically add fail-safe fault tolerance with perfect detection only, and fail-safe fault tolerance with perfect detection, and minimal detection latency.

We further develop a theory for the design of multitolerance, which is the ability of a program to tolerate multiple classes of faults. In the thesis, we explain that interference can occur between different program components when designing multitolerance, and we present a set of non-interference conditions that needs to be verified. We then present two different approaches for the design of multitolerance, and for each approach, we present two different algorithms that add fail-safe fault tolerance to several fault classes with different efficiency properties.

The algorithms presented in this thesis are particularly suitable for a class of programs termed as *bounded programs*. The property of bounded programs is that they do not have any kind of unbounded looping structure.

Keywords: Distributed Systems, Embedded Systems, Formal Methods, Fault Tolerance, Fail-Safe, Detectors, Efficiency, Multitolerance.

Kurzfassung

In den letzten Jahren durchdringen immer kleinere, eingebettete Computersysteme verstärkt unsere Lebensumwelt. Mit der Allgegenwärtigkeit solcher Systeme werden wir aber auch immer abhängiger von ihnen. Mit der Abhängigkeit steigen unsere Erwartungen an die Zuverlässigkeit der Systeme bis zu dem Punkt, an dem wir uns wünschen, dass das System auch dann noch funktioniert, wenn ein bestimmtes Maß an Fehlverhalten innerhalb der Systemkomponenten auftritt. Derartige Systeme werden als *fehlertolerant* (*fault-tolerant*) oder *verläßlich* (*dependable*) bezeichnet. Eine Möglichkeit, verlässliche Computersysteme zu bauen, besteht darin, das System mehrfach vorzuhalten, um im Fehlerfall auf ein funktionsfähiges System umschalten zu können. Eine weitere und in der Praxis häufig günstigere Alternative ist die sogenannte *software-basierte Fehlertoleranz*, um die es in dieser Arbeit geht.

Man unterscheidet verschiedene Arten von Fehlertoleranz, beispielsweise die bekannte *maskierende Fehlertoleranz* (*masking fault tolerance*). In dieser Arbeit geht es um die sogenannte *fail-safe Fehlertoleranz* (fail-safe fault tolerance). Bei fail-safe Fehlertoleranz ist es akzeptabel, wenn das System im Fehlerfall "anhält" anstatt weiterhin seinen Dienst zu erbringen. Wichtig ist lediglich, dass das System immer in einem "sicheren" Zustand verweilt. In dieser Arbeit werden Verfahren vorgeschlagen, um effiziente fail-safe-fehlertolerante Systeme aus fehler-intolerante Originalsystemen zu synthetisieren. Wir verwenden zwei bekannte Kriterien, um die Effizienz der fehlertoleranten Systeme zu messen: (1) *Fehlererkennungszeit* (error detection latency), also die "Zeit", die benötigt wird, um einen aufgetretenen Fehler zu entdecken, und (2) *Fehlererkennungsabdeckung* (*error detection coverage*), also die Rate von relevanten Fehlern, die das Programm entdecken kann.

Diese Arbeit legt die formalen Grundlagen für den Entwurf von effizienten fail-safe-fehlertoleranten Systemen. Grundlage ist eine verfeinerte Theorie sogenannter *Detektoren* (detectors). Wir definieren eine neue Klasse von Detektoren, *perfekte Detektoren* (*perfect detectors*), die es erlauben, fail-safe-fehlertolerante Systeme mit *perfekter Fehlerekennung* (*perfect detection*) und damit vollständiger Fehlererkennungsabdeckung zu synthetisieren.

Anschliessend definieren wir das Konzept der schnellen Fehlererkennung (*fast detection*), welches eine optimale Fehlererkennungszeit erlaubt. Wir stellen ein Verfahren vor, wie man ein fail-safe-fehlertolerantes System sowohl mit perfekter Fehlererkennung als auch mit schneller Fehlererkennung synthetisieren kann. Darüberhinaus ist die Fehlererkennungszeit der synthetisierten Programme ist optimal.

Die Arbeit beschäftigt sich abschließend mit dem Konzept der *Multitoleranz* (*multitolerance*), also der Fähigkeit eines Programmes, verschiedene Fehlerklassen gleichzeitig zu tolerieren. Bei Multitoleranz kann es zu einer wechselseitigen Beeinflussung (*interference*) der Detektoren für verschiedene Fehlerklassen kommen. Wir stellen eine Reihe von Nicht-Beeinflussungskriterien (*non-interference conditions*) vor, die überprüft werden müssen, um Multitoleranz zu gewährleisten. Wir stellen zwei Ansätze für den Entwurf multitoleranter Programme vor. Für jeden Ansatz geben wir zwei verschiedene Algorithmen an, die fail-safe-fehlertolerante Programme bezüglich verschiedener Fehlerklassen und mit unterschiedlicher Effizienz synthetisieren.

The Algorithmen dieser Arbeit eignen sich insbesondere für sogenannte *beschränkte Programme* (*bounded programs*). Beschränkte Programme sind Programme ohne unbeschränkte Schleifenstrukturen.

Acknowledgements

I would probably not have made this far without the help, guidance and support of many people.

I wish to thank Prof. Dr. Neeraj Suri, my adivsor, for first accepting me in his DEEDS group, and then guiding me to where I am now.

My gratitude to my friend, and colleague Martin Hiller is enormous. Ever since we started our collaboration a few years back, Martin has been a constant source of inspiration, and help.

My thanks also go to Vilgot Claesson, Örjan Askerdal, Robert Lindström, Andréas Johansson, and Adina Sarbu, who are all part of the DEEDS group. They have all contributed to create a great working atmosphere. Discussions with Andreas and Robert on various computing topics have been very interesting, as have been our numerous discussions on football.

I cannot thank Birgit Neuthe of the DEEDS group enough for all the help she afforded me with the everyday stuff. From helping me with German translation to setting up appointments, she's been of tremendous help.

Outside of the DEEDS group, I have to thank Felix Gärtner for all the interesting, and challenging discussions we've had over the times. I also have to thank Christof fetzer for all the interesting discussions we have had. My thanks also go to Sandeep Kulkarni, for taking time to clarify and explain part of the theory he developed.

I have ti thank Prof. Dr. Mario Dal Cin for kindly accepting to be a reviewer of my thesis, as well as Prof. Dr. Alex Buchmann, Prof. Dr. Mira Mezini, and Prof. Dr. Claudia Eckert for kindly taking time to read my thesis, and accepting to be on my committee.

My sincere, heartfelt thanks go to my parents, who have spared no effort in trying to get me to where I am now. Their constant moral support has been very welcoming. I also thank my brother and sister for their constant support, as well as my in-laws.

Saving the best for last: Najaat, thank you so much for everything, your support, your love, your patience, everything.

Contents

List of Figures

Chapter 1

Introduction

The design of reliable computers has been a challenge ever since computers first appeared in the middle of the 20^{th} century. In those days, computers were built out of unreliable components, such as vacuum tubes, relays, and so on. Later generations of computers were more reliable as they were built from more reliable components, such as semiconductor components, and other components from more advanced technology. Computers were expensive, and were used mainly for computation-extensive tasks, research, and defense. Nowadays, with the ever-increasing circuit density, computers are no longer expensive commodities. In fact, they are becoming more and more pervasive. They are being used in every walks of life, from safety-critical systems, such as nuclear plants control, airplanes etc, to consumer-oriented products, such as automobiles, refrigerators etc. As these computer systems pervade our lives, our expectation on their delivery of services, in spite of faults, increases. We need these computer systems to be *dependable*.

In this chapter, first, we will first briefly survey the fundamentals of dependability (Section 1.1), where we provide an overview of the main steps involved in the design of fault-tolerant systems. We then explain the motivations behind the work presented in this thesis (Section 1.2). We will then present the problem statements, and pertinent research questions that arise and explain our research contributions.

1.1 Dependability: Basic Concepts

In this section, we explain how dependable (fault-tolerant) programs are designed in general. First, we explain the fault/error/failure classification, and then we explain how dependability can be achieved. Given our focus on fault tolerance, we then briefly survey the main steps in achieving fault tolerance. Lastly, we explain how the resulting system can be validated.

The term *dependability* is defined as "the trustworthiness of a system such that reliance can justifiably be placed on the service it provides" [Lap92]. This means that the services provided by such a system are always correct, according to the system's specification, whether the environment in which it is deployed is ideal, or less than ideal (faulty).

1.1.1 Faults, Errors, and Failures:

During the construction or operation of a computer system, events may occur that can threaten the computer system's ability to deliver correct services. For example, developers of the system may have inadvertedly introduced defects (or bugs) during the construction phase. Another factor that can affect a computer system's ability to deliver correct services is the ageing of components, though its relevance may be less in software. Another example of an event that can jeopardize the computer system's operations is its deployment in noisy environments that generate unexpected events. Factors that can affect the proper functioning of a computer system, such as noise, bugs etc, are commonly referred to as *faults*.

An *error* is said to exist in a computer system when a corresponding fault is activated. Specifically, a fault in itself may not threaten the proper functioning of the system, for example, if a fault occurs in an area of memory that is not accessed, then the fault has no ability to influence any computation. However, when a fault is activated, for example a computation reaches the fault-affected area in memory, and the faulty value used during the computation, if no corrective action is taken, there is the risk of the computer system to violate its specification, i.e., do not deliver the required service. When a faulty value is used in some computation, error is said to propagate, i.e., there is *error propagation*. When the error propagates to the "output" of the computer system, a *failure* is said to happen, i.e., the behaviour of the system has deviated from what is prescribed by its specification.

Thus, to be able to develop a fault-tolerant system, one needs to understand the faults that can potentially affect the system, i.e., one needs to develop a fault model.

1.1.2 Ways of Achieving Dependability

Once a fault model has been developed, there are various ways of dealing with it, i.e., there are different ways of achieving dependability, when designing a dependable system, namely:

- **Fault Prevention:** As the name suggests, this approach tries to prevent faults from occurring in the first place. Examples of fault prevention approaches are use of sound development methodologies or use of radiation-hardened hardware.

- **Fault Tolerance:** This is the ability of a system to deliver desired level of functionality in the presence of faults, i.e., instead of preventing faults from occurring, one tries to tolerate their effects. To achieve this, the system should be able to detect and/or correct errors in the system.

- **Fault Removal:** This process deals with removal of faults, and is commonly referred to as debugging (for software).

- **Fault Forecasting:** This process helps in evaluating the consequences of faults when they occur.

1.1.3 Attributes of Dependability

Once a dependable system has been designed, one needs to measure its "dependability". There are different attributes that characterize dependability, for example:

- *Reliability* – This attribute defines the probability of a system to provide correct service over a finite period of time.

- *Availability* – This attribute defines the probability of a system to be correct at any given time.
- *Safety* – This attribute captures the extent to which a service provided by a system is safe.

Other attributes such as confidentiality and integrity are also attributes of dependability, but are more related to security issues, and we do not discuss them any further.

1.1.4 Design of Fault Tolerance

As we explained earlier (Section 1.1.2), there are various ways of achieving dependability. In this thesis, we focus mainly on *fault tolerance*. Fault tolerance is the ability of a system to provide a desired level of functionality in presence of faults. Fault tolerance is closely coupled to the fault model assumed, i.e., a fault-tolerant system may be able to tolerate one class of faults, and still not able to tolerate another types of faults.

For a system (program) to be fault-tolerant, it needs to be able to perform some important steps whenever errors (effects of faults) appear. In general, provision of fault tolerance can be divided into four stages [LA90]:

1. **Error Detection:** This step is concerned with the ability of the system to detect that some erroneous state has been reached, and that the system is in some "unsafe" state. Error detection is important, since the system is then prevented from performing unsafe actions.
2. **Damage Assessment:** After an error has been detected, one needs to determine the extent to which damage has been caused to the system. In particular, one needs to determine the extent to which error has propagated through the system.
3. **Error Processing:** Once damage assessment is done, error processing is initiated that tries to revert the system back to a non-erroneous state, i.e., a safe state. The combined actions of damage assessment, and error processing is commonly known as *error recovery*.

4. **Fault Treatment:** This step is concerned with preventing the same faults from getting activated again, and is generally performed offline.

Overall, a fault-tolerant program should be able to first detect errors, and then to recover from them. To design fault tolerance, Arora and Kulkarni observed in [AK98c, AK98a, AK95, Kul99] that two components, which they termed as *detectors*, and *correctors*, underpin the design of fault tolerance. A detector is a program component that is added to a program to detect errors in the program. Examples of detectors are executable assertions [Sai78, MAM84, Hil00], error detecting codes, snapshot procedures, comparators and so on. A corrector, on the other hand, is a program component that is added to recover from errors. Arora and Kulkarni have shown that, by using either detectors, correctors or both of them, different classes of fault-tolerant programs can be obtained, namely fail-safe fault-tolerant programs, non-masking fault-tolerant programs, and masking fault-tolerant programs. Each class of fault-tolerant programs provides a specified level of fault tolerance.

In this thesis, we focus on the design of *fail-safe fault-tolerant programs*. It was shown in [AK98c] that, to make a program fail-safe fault-tolerant, it is both necessary and sufficient to add detectors to that program. In this thesis, the approach we will present allows a program to have both perfect error detection and minimal detection latency. This in turn has the effect of constraining error propagation, hence limiting the amount of damage done in the system. Thus, by design, the damage done in presence of faults is minimal. The implication of this is that the error processing phase needs not be very complicated (sophisticated).

1.1.5 Verification and Validation of Fault Tolerance

In the design of fault-tolerant systems, one needs to verify the correctness of the system. To do this, formal methods [CW96] has often been used. The first step is to specify the properties that the system should have. The specification is usually done in some logical formalism, usually temporal logic, which can assert how the behavior of the system evolves over time. The second step is to construct a formal model for the system. In order to be suitable for verification, the model should capture those properties that must be considered to establish correctness. During the verification process, the properties that establish correctness are verified. In the dependability area, formal methods have been used to verify correctness of distributed and/or real-time protocols [KRS99, SS99b]. It has also been observed that a proper decomposition of a fault-tolerant program into its components helps in in its mechanical verification [KRS99].

Once the system has been implemented and fault tolerance mechanisms, such as detectors and correctors, have been added, the resulting "fault-tolerant" system needs to be validated. Two commonly used methods for validation are *testing*, and *fault injection*. In testing, the system is subjected to a number of test cases to ascertain that there are no bugs (faults) in the system. Bugs are suspected present when the system deviates from its specified behavior under any test case. The problem is usually to find suitable test cases which can uncover those bugs. In [SS99b], the authors adopt a formal-based approach whereby verification information is reused to drive test-case generation.

To validate the fault tolerance mechanisms, fault injection [AAA$^+$90, IT96] is often used. In fault injection experiments, faults are artificially injected in the system to create conditions that will activate those fault tolerance mechanisms. Fault injection suffers from the same problem as testing for having to find suitable test cases, as well as determining which types of faults to inject.

1.2 Motivation and Research Questions

On this background, in this section, we will discuss the motivations that underpin the work presented in this thesis. Our overall goal is *to develop a framework that allows systematic development of efficient fail-safe fault-tolerant programs.*

The motivation behind the work presented in this thesis is multifold. First, it is well-known that the design of fault-tolerant systems is inherently complex. Thus, there is a need for well-defined and sound development methodologies that can guide the software designer in the design of efficient and complex dependable systems.

Also, it is often the case that addition of fault tolerance mechanisms (i.e., detectors and correctors) interfere with the performance of the system. For example, some error detection mechanisms may be added that trigger a lot of false alarms in the system. This has the effect of affecting the performance of the system. More importantly, it has also been noted that design of efficient fault tolerance mechanisms is very often reliant on the experience of the programmers. This again points to a need for sound methodologies that can guide the programmers in the design of efficient fault tolerance mechanisms.

Further, in the start phase of the design, the software designer may not be fully aware of all the fault classes that the system will be subjected to. As the system evolves, and the system designer becomes more aware of more fault classes, additional fault tolerance mechanisms may need to be added to handle these faults. However, each time fault tolerance mechanisms are added, a complete verification of the new program is needed, which is expensive. Also, non-interference across the different fault tolerance mechanisms need to be ascertained. Thus, the ability to "add" new tolerance mechanisms without having to perform a complete verification of the program is crucial.

Overall, we endeavor to develop a framework that (i) enables the design of efficient fault tolerance mechanisms (more specifically, detectors), and (ii) enables compositional design of fault tolerance. Combined together, we provide a framework that enables systematic (compositional) design of efficient fault-tolerant programs.

Focus: In this thesis, we focus on the design of a particular class of fault tolerance, namely *fail-safe fault tolerance*. Informally, a program is fail-safe fault-tolerant if it always remains in a safe state, even in the presence of faults (We will formally define the term fail-safe fault tolerance in Chapter 3). The reason for focusing on fail-safe fault tolerance is multifold. First, fail-safe fault tolerance is often needed in critical applications, such as nuclear plants, train control systems and so on. Very often, detection is the only objective, and once an error is detected, a mechanical backup system takes over. Second, it was shown by Arora and Kulkarni in [AK98b] that to design masking fault tolerance (which is the ideal fault tolerance), one can first design a program to be fail-safe fault-tolerant and then later extended with correctors to make the program masking fault-tolerant. Thus, our approach tackles one step in the design of masking fault tolerance.

Given our focus on the design of fail-safe fault tolerance, it was shown by Arora and Kulkarni [AK98c] that it is both necessary and sufficient to compose a program with detectors to make it fail-safe fault-tolerant. Therefore, when designing such fail-safe fault-tolerant programs, we also focus on the design of detectors, i.e., program components that detect errors.

On this background, we formulate the problem statements that have driven the research presented in this thesis.

1.2.1 Problem Statements

The main goal of the work presented in this thesis has been to develop a framework that can help in the design of *efficient fail-safe fault-tolerant programs.*

While addressing the above problem, we tackled some of the following research questions:

Research Question: *How can one assess the efficiency of a fail-safe fault-tolerant program? What are the common metrics for such an assessment?*

When designing fault-tolerant programs, error detection is crucial. Very often, to validate the detectors, fault injection experiments are performed to assess the efficiency of the detectors, and common factors used for such assessment are (i) detection coverage, and (ii) detection latency. In this thesis, we thus focus on those two properties of fail-safe fault-tolerant programs, namely coverage, and detection latency.

Research Question: *What are the main properties of a detector that allow characterization of its efficiency? Can such properties be formalized?*

In Chapter 4, we develop a theory of detectors, and identify completeness and accuracy as two important properties of a detector that characterize its efficiency. We then formalize these properties and identify an important class of efficient detectors, namely perfect detectors. We explain that such a detector allows for perfect error detection, and we further explain its role in fail-safe fault tolerance. Thus, perfect detectors can be shown to have "perfect" coverage.

Research Question: *Upon the occurrence of faults, how can error propagation be limited?. How can the detection latency of a program be minimized?. Is it possible to design a fail-safe fault-tolerant program such that its detection latency is minimal?. Do the detectors included have any impact on the underlying program?.*

To tackle this question, in Chapter 5, we develop a theory of fast detectors, and explain how fail-safe fault-tolerant programs with minimal detection latency be designed. In fact, the approach we propose allows a fail-safe fault-tolerant program to have both perfect detection, and minimal detection latency.

Research Question: *Can efficient detectors be designed for several fault classes? How can their non-interference be guaranteed? Is there any methodology that can be used such that verification needs not be performed from scratch each time new detectors for new fault classes are added?.*

The motivation behind this research issue is that, during periods of perturbation, a system is subjected to faults from various sources, such as network overloads, message losses, transients, crashes and so on. It is very difficult to design a (fail-safe) fault-tolerant program to these fault classes. So, the idea is to consider one fault class at a time, and design the fault tolerance mechanisms to the fault class considered. The obvious problem is whether the fault tolerance mechanisms for different fault classes can be composed, i.e., one needs to ascertain that fault tolerance mechanisms (detectors) to a given fault class do not interfere with those of another fault class. Further, the problem is also to develop efficient fault tolerance mechanisms to several fault classes. Specifically, it would be detrimental if efficient fault tolerance mechanisms are designed to tolerate one fault class, but the tolerance mechanisms for another fault class is not efficient, resulting in an inefficient multitolerant system/program. Thus, our theory not only shows non-interference across different detectors (in terms of their behavior) for different fault classes, but our theory also shows that "composing" different perfect detectors for different fault classes preserve the efficiency of the resulting program.

Thus, in Chapter 6, we develop a theory for the design of efficient multitolerant programs. Building upon the theory of perfect detectors (Chapter 4) and the theory of fast detectors (Chapter 5, we develop a theory for the design of efficient multitolerant programs, and develop the requisite steps to show ascertain non-interference across different detectors.

Research Question: *Can the design of efficient fail-safe fault-tolerant programs be automated?*

To tackle this question, we have developed algorithms of polynomial-time complexity that automatically synthesizes a fail-safe fault-tolerant program, starting from a corresponding fault-intolerant program. We have developed examples showing how these algorithms can be used for such automatic synthesis.

Research Question: *Can we reuse the fault-intolerant program to synthesize the fault-tolerant program?*

There are two possible ways of synthesizing a fail-safe fault-tolerant program. First, one can start with a specification of the program, and then use refinement steps to first synthesize a fault-intolerant program, and then perform a fault tolerance transformation to obtain a fail-safe fault-tolerant program. The second option is to start directly with the fault-intolerant program and then transform it into a fail-safe fault-tolerant program by composing it with fault tolerance mechanisms. *In this thesis, we adopt the second methodology, and, starting from a fault-intolerant program, we transform it into a fail-safe fault-tolerant program by composing it with fault tolerance components (more specifically, with detectors).*

1.3 Research Contributions

Towards addressing all these research issues, we have developed a theory of detectors, and identified and formalized some important properties of these program components. We have identified a class of detectors called perfect detectors that allows design of efficient fail-safe fault-tolerant program.

Specifically, we will show how to design fail-safe fault-tolerant programs with perfect error detection, and minimal error detection latency. Based on the theory, we also develop polynomial-time algorithms that permit automatic synthesis of efficient fail-safe fault-tolerant programs.

Further, we develop a theory that underpins the design of multitolerant programs. We show that the class of perfect detectors allows for "non-interfering composition", i.e., perfect detectors for each fault class do not interfere with each other. Specifically, perfect detectors for different fault classes do not interfere with each other's "behavior", and they do not interfere with the efficiency of the program.

Overall, in this thesis, we make the following research contributions:

1. We first present a novel theory of detectors, formalize some important properties of detectors, and identify an important class of detectors, namely perfect detectors, that underpins design of efficient fail-safe fault-tolerant programs. We further explain their role in the design of fail-safe fault tolerance. We also provide an algorithm that automatically yields fail-safe fault-tolerant programs, with perfect detection.

2. Next, we present a novel theory of fast error detection, and building upon the theory of perfect detection, we develop an algorithm that generates fail-safe fault-tolerant programs with both perfect error detection and with minimal error detection latency.

3. We explain that, in the context of multitolerance design, some non-interference conditions need to be verified. We further explain that non-interference across detectors with respect to their behavior is not sufficient when designing efficient multitolerant programs. We therefore present a set of non-interference conditions that encompass both behavioral and performance aspects. As such, we develop a suite of algorithms that systematically adds "efficient" fail-safe multitolerance to a program. Overall, this contribution allows compositional design of efficient fail-safe fault-tolerant programs, i.e., efficient fail-safe fault-tolerant can be systematically designed.

Our contributions are in the area of fault tolerance, specifically in the field of *error detection*. We have shown how to design efficient detectors such that the fail-safe fault-tolerant programs have perfect error detection and minimal detection latency for different fault classes.

To summarize, our main contribution is *an approach that transforms a fault-intolerant program into a fail-safe fault-tolerant program with perfect error detection, and minimal detection latency, i.e., efficient fail-safe fault-tolerant program.*

1.4 Thesis Structure

The thesis is structured as follows:

Chapter 2 surveys results in the areas *design of fault tolerance*, *automated design*, *program transformation*, and *multitolerance*. We also try to put our contributions into context.

Chapter 3 introduces the formal foundations for our work and presents the terminologies used in this thesis. We also present the system model and fault model used.

Chapter 4 introduces a theory of perfect detectors, and develops a sound and complete algorithm that yields fail-safe fault-tolerant programs with perfect detection.

Chapter 5 introduces a theory of fast detectors, and develops a sound and complete algorithm that yields fail-safe fault-tolerant programs with perfect detection, and minimal detection latency.

Chapter 6 explains the concept of multitolerance. It develops a series of non-interference conditions that need to be satisfied when designing multitolerance. Several algorithms are developed that yield fail-safe multitolerant programs with varied optimal properties, as well as guaranteeing non-interference.

Chapter 7 summarizes the contributions of this thesis, and assesses their impact. We conclude by providing some pointers regarding future work.

Chapter 2

Related Work

In this chapter, we present a survey of previous work and results that are closely related to the problems addressed in this thesis. Specifically, the areas of most closely related are *design of fault-tolerant programs*, *design of effective detectors*, *automated procedures*, *error propagation analysis*, and *software implemented fault tolerance.*

2.1 Design of Fault-Tolerant Programs

One common way to implement fault-tolerant programs is to use N-Version programing [Avi85], which is however an expensive approach. Another approach has been to use Recovery Blocks [Ran75]. But the effectiveness of recovery blocks is heavily reliant on the effectiveness of the acceptance tests included. Unfortunately, little work has been done that can guide a software designer towards designing effective acceptance tests (detectors).

Leveson *et.al* presented the results of a large scale experiment to determine the effectiveness of software checks and voting in software in [LCKS90]. They explained that the effectiveness of detectors depends very much on the individual ability of the programmers to design effective detectors. Again, as in the case of Recovery Blocks, little work has been done to guide the programmers in designing effective detectors. However, to ease the use of executable assertions (which is an instance of a detector), Saib extended the FORTRAN and PASCAL languages with a software construct (called Assert) that helps in the implementation of executable assertions [Sai78]. Another approach for facilitating the use of assertions is the use of the Annotation PreProcessor tool of Rosenblum [Ros95]. A similar approach is described by Yin and Bieman [YB94]. The problem with these approaches is they do not provide guidelines pertaining to the design of effecive detectors, which is difficult, since very often, these assertions tend to be application-specific. In this work, we provide algorithms that can automatically generate perfect detectors, hence the problem of designing application-specific detectors can be effectively taken away from programmers, once the fault-intolerant program is available.

To conquer the design complexity, Arora and Kulkarni proposed a transformational approach whereby a fault-intolerant program (a program which satisfies its specification in the absence of faults), and that satisfies at least its safety specification in presence of faults) is transformed into a fault-tolerant program (either fail-safe, non-masking or masking) through the addition of detectors and/or correctors. Using this approach, they have presented fault-tolerant solutions for several problems such as distributed reset [KA98], mutual exclusion [AK98b], network management [KA97a], data transfer [AK98b], and Byzantine agreement [KA97b].

The premise is that a fault-tolerant program is a composition of a fault-intolerant program with fault tolerance components, such as detectors and correctors. The authors argue that such an approach allows for separation of concerns. Specifically, it is possible for a software designer to first focus on designing the fault-intolerant program, and then focus on adding fault tolerance to it.

In [AK98a], Arora and Kulkarni presented an stepwise approach for addition of multitolerance, i.e., the ability of being fault-tolerant to multiple classes of faults. They also argued that non-interference between different program components needs to be verified, and presented a set of non-interference conditions for that matter. In this present work, we extend the current set to include non-interference with other program properties (apart from fault tolerance) , such as perfect detection, and minimal detection latency.

Another transformational appraoch has been proposed by Joseph and Liu [Liu91, LJ92, LJ93, LJ94, LJ95]. They show how a program constructed for a fault-free system can be transformed into a fault-tolerant program for execution in faulty environments. Specifically, the addition of fault tolerance to a program is modeled by a fault-tolerant transformation that adds the necessary redundancy to the program so that the faults can be tolerated. A fault-tolerant program can be further refined using fault-tolerant refinement that preserves both the functional, and fault-tolerant properties of the program.

The fault tolerance mechanisms used are very much dependent on the fault model used. For example, in data transfer, time outs may be used to detect message losses, rather than, say, executable assertions. However, the problem of knowing in advance all the classes of faults the software can be subjected to may be difficult to solve. For example, continuing with the example on data transder, if the system designer assumes only the case where messages can be lost during data transfer, he can have an implementation such that the sending node can retry sending the loss message after a timeout. But if a fault occurs that arbitrarily corrupt the state of the program, such retry actions may not be sufficient, and safety may be compromised. Hence, weak fault models are sometimes assumed, such as Byzantine faults. In such cases, self stabilization [Dij74] has been advocated, and is getting more and more attention in the community. For example, Gouda and Multari proposed some self-stabilizing communication protocols in [GM91]. Self-stabilizing protocols have been proposed in [APSV91, DIM93, DW95, AD97, Dol97, BDDT98, Dol00] among others. However, the problem with self-stabilization is that safety may be temporarily violated. One interesting class of self-stabilization, called snap-stabilization, has been proposed by Cournier *et.al* [CDPV01] that solves this problem. A snap-stabilizing protocol is a self-stabilizing protocol meaning that starting from an arbitrary state (in response to an arbitrary perturbation modifying the memory state) it is guaranteed to behave according to its specification.

Another line of approach for design of fault tolerance, where the goal is for "scalable fault tolerance", has been investigated by Arora *et.al* [ADK01]. To achieve self stabilization, one needs to make use of system implementation. However, the authors argue that this approach does not scale very well. Hence, they propose to implement stabilization based on system specification, such that the stabilization property is guaranteed irrespective of the implementation.

The effectiveness of detectors is also affected by their placement in the software, as indicated by Hiller *et.al*in [HJS01], and the authors also demonstrate the sensitivity of the location set to the underlying fault model in [HJS02]. Once the fault-tolerant software is obtained, fault-injection experiments are conducted to evaluate the resulting dependability of the program [IT96]. However, such work do not reveal the weak spots in the software, for example, how errors propagate in the software, what are the vulnerable signals/variables. Initial work focusing on these aspects appear in [HJS01, JHS01]. To conduct these validation experiments, effective test cases are needed. Sinha and Suri investigated the applicability of formal methods in driving generation of test cases in [SS98, SS99a]. Specifically, the authors reused verification information to drive test case generation. In [JHS02b], Jhumka *et.al* proposed a formal approach for designing component-based dependable software and in [JHS02a], the authors presented a formal approach for test case generation, whereby they reuse detector design information to drive test case generation.

General surveys in the area of dependability can be found in [Cri91, Gae99a], while Gaertner presented a survey of transformational approaches in [Gae99b]

2.2 Automated Procedures

In an earlier work, Kulkarni and Arora [KA00] presented an algorithm that automates the addition of fail-safe fault tolerance to an initially fault-intolerant program. This algorithm is based on an analysis of the state transition representation of the program in the presence of faults. The algorithm is sound and complete meaning that (i) the transformed program is in fact a fail-safe fault-tolerant version of the original program, and (ii) if a fail-safe fault-tolerant version of the program exists, then the algorithm will find it. The complexity of the algorithm is polynomial in the state of the fault-intolerant program. Put in context with the work presented in this thesis, the algorithm in [KA00] always adds fail-safe fault tolerance with perfect detection. But, the algorithm can sometimes add fail-safe fault tolerance with perfect detection, and minimal detection latency to some classes of programs. By way of contrast, our work presents algorithms that always add both perfect detection, and minimal detection latency to a wider set of programs. The algorithms have polynomial complexity in the state space of the fault-intolerant program.

Chapter 3

Formal Preliminaries

In this chapter, we recall the standard formal definitions of programs, faults, fault tolerance (in particular, fail-safe fault-tolerance), and of specifications [AK98c, Kul99]. Intuitively, a program is represented as a transition system, since programs written in any imperative language can be represented as such. This chapter provides all the requisite formal basis upon which the work presented in this thesis is based.

3.1 Concurrent Systems

A *concurrent* system consists of a set of components executing together. They are usually associated with a form of communication among them. The mode of execution, and that of communication may differ from system to system. There are two main modes of execution:

1. Asynchronous or *interleaved* execution, where only one component makes a step at any time.

2. Synchronous execution, where all components make a step at the same time.

As for the communication part, we present two of the possibilities, namely

1. Shared Variables (we provide more details in Section 3.2).

2. Message Passing, where components communicate with each other by sending messages.

The work assumes an *interleaved semantics* of execution, together with the *shared variable* communication paradigm.

3.2 Programs

Definition 1 (Program) *A* program p *consists of a set of variables* V_p *and a finite set of processes. Each process contains a finite set of actions, and a finite set of variables. Each variable stores a value from a predefined nonempty, finite domain and is associated with a predefined set of initial values. An action has the form*

$$\langle name \rangle :: \langle guard \rangle \rightarrow \langle statement \rangle$$

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. The name is a unique identifier of that action.

Definition 2 (State and State Space) *We define a* state s of program p *as a possible value assignment to all variables in* p*. We also define the* state space S_p of a program p *as the set of all possible assignments of values to variables.*

Definition 3 (State Predicate) *A* state predicate of p *is a boolean expression over the state space of* p*.*

Definition 4 (Initial States) *The* set of initial states I_p *is defined by the set of all possible assignments of initial values to variables.*

Definition 5 (Enabled) *An* action ac of p is enabled in a state s *if the guard of* ac *evaluates to "true" in* s*.*

Definition 6 (Action) *An action* ac of program p *is represented by a set of state pairs* $\{(s,t) : s, t \in S_p\}$*.*

We assume that actions are deterministic, i.e., $\forall s, s', s'' : (s, s') \in ac \wedge (s, s'') \in ac \Rightarrow s' = s''$. Note that programs are permitted to be non-deterministic since multiple actions can be enabled in the same state. In particular, each non-deterministic action can be converted into a set of deterministic actions with an identical state transition relation.

Definition 7 (Program Computation) *A* computation of program p *is a weakly fair (finite or infinite) sequence of states* $s_0, s_1, \ldots$ *such that* $s_0 \in I_p$ *and for each* $j \geq 0$*,* s_{j+1} *results from* s_j *by executing the assignment of a single action which is enabled in* s_j*.*

We require that *weak fairness* implies that if a program action ac is continuously enabled, ac is eventually chosen to be executed. Weak fairness implies that a computation is *maximal* with respect to program actions, i.e., if the computation is finite, then no program action is enabled in the final state.

Definition 8 (Concatenation) *If* α *is a finite computation and* β *is a computation of* p*, we denote with* $\alpha \cdot \beta$ *the* concatenation *of both computations.*

Definition 9 (Occurs) *A* state s occurs in a computation $s_0, s_1, \ldots$ of program p *iff there exists an* i *such that* $s = s_i$*. Similarly, a* transition (s, s') occurs in a computation $s_0, s_1, \ldots$ of program p *iff there exists an* i *such that* $s = s_i$ *and* $s' = s_{i+1}$*.*

In the context of this thesis, programs are equivalently represented as state machines, i.e., a program p is a tuple $p = (S_p, I_p, \delta_p)$ where S_p is the state space of p, $I_p \subseteq S_p$ is the set of initial states. The state transition relation $\delta_p \subseteq S_p \times S_p$ is defined by the set of actions as follows : Every action ac implicitly defines a set of transitions which is added to δ_p. A transition $(s, s') \in \delta_p$ iff ac is enabled in state s and computation of the statement ac results in state s'. We say that ac *induces* these transitions. State s is called the *start state* and s' is called the *end state* of the transition.

Definition 10 (Step) *A transition from one state to another state is called* a step*.*

Definition 11 (Stuttering Step) *If (s_i, s_{i+1}) is a step, and $s_i = s_{i+1}$, then this step is a* stuttering step.

A stuttering step is an important concept in program refinement, in the sense that transitions at a lower level of abstraction appear as stuttering steps at a higher level.

Definition 12 (Computation Equivalence) *Two computations α_1 and α_2 are said to be* equivalent *if they contain identical sequence of states.*

Definition 13 (Stuttering Equivalence) *Two computations α_1 and α_2 are* equivalent under stuttering *if α_1 and α_2 are equivalent after removing stuttering steps from both computations.*

Definition 14 (Property) *A* property *is a set of computations which is closed under stuttering, i.e., if a given computation c is in property P, then all computations that are stuttering-equivalent to c are in P.*

3.3 Communication

Two processes p_r and p_w of a program p communicate as follows: for each pair of processes p_r and p_w there exists a set of "shared" variables V_s. Both processes can read the contents of any variable in V_s, but only p_w can update these variables. This defines the information flow between two processes. The set V_s represents the interface between processes p_w and p_r.

There exists a set of special variables, denoted by V_o, that are shared by some processes (that write to the variables), and the environment that reads them. These special variables are commonly referred to as the *output variables*. There exists also a special set of variables, denoted by V_i, where each of the variables is written to by the environment, and read by a process in p. Such variables are known as *input variables*. Input and output variables represent the interface of the program p with its environment.

Such program model reflects the system assumptions of distributed embedded applications (like sensors and actuators), for which part of our formal framework is targeted. Multiple initial states reflect the fact that a program

p may initially read external inputs before executing. In such cases, we additionally assume a set of special variables called the *output variables* of p in which the program should finally write the results of a computation. This model is suitable for the domain of embedded applications (like sensors and actuators). Program actions can be partitioned into two categories: (i) critical actions, and (ii) non-critical actions [AK98b]. Program actions that write output variables are critical actions. Other examples of critical actions are (i) actions that commit to a database, or (ii) actions that control progress in a nuclear control plant. Critical actions are those actions whose execution in the presence of faults can cause violation of safety.

Definition 15 (Critical and non-critical actions) *An action ac of program p with safety specification SS is said to be* critical *iff there exists a transition* (s, t) *induced by ac and* (s, t) *is a bad transition (Proposition 2) that is reachable (Definition 32) in presence of faults. An action is non-critical iff it is not critical.*

3.4 Specifications

A *specification* for a program p is a set of computations of p that is fusion-closed.

Definition 16 (Fusion Closure) *A* specification S is fusion-closed *iff the following holds for finite computations* α, γ, *a state* s *and computations* β, δ: *If* $\alpha \cdot s \cdot \beta$ *and* $\gamma \cdot s \cdot \delta$ *are in* S, *then so are* $\alpha \cdot s \cdot \delta$ *and* $\gamma \cdot s \cdot \beta$.

We will discuss the consequences of demanding fusion-closed specifications in Section 4.2.1.

Definition 17 (Satisfies) *A* computation c_p of p satisfies a specification S *iff* $c_p \in S$.

Definition 18 (Violates) *A* computation c_p of p violates a specification S *iff* c_p *does not satisfy* S.

Definition 19 (Correctness) *A* program p satisfies a specification S *iff all possible computations of p satisfy S.*

Definition 20 (Maintains) *Let p be a program, S be a specification and α be a finite computation of p. We say that α* maintains *S iff there exists a sequence of states β such that $\alpha \cdot \beta \in S$.*

Definition 21 (Safety specification) *A specification S of a program p is a* safety specification *iff the following condition holds : For every computation σ that violates S, there exists a prefix α of σ such that for all state sequences β, $\alpha \cdot \beta$ violates S.*

Using a practical system of rail crossing where trains will need to share a common track, a safety specification can be "no two trains will use the track at the same time".

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

Proof. Follows from the Definitions 20 and 21. □

Informally, the safety specification of a program states that "something bad never happens". Formally, it defines a set of "bad" finite computation prefixes that should not be found in any computation. Alpern and Schneider [AS85] have shown that every specification can be written as the intersection of a safety specification and a *liveness specification*.

Definition 22 (Liveness) *A* liveness specification *is a set of state sequences that meets the following condition : for each finite state sequence α, there exists a state sequence β such that $\alpha \cdot \beta$ is in that set.*

A example of liveness specification, following from our previous example of rail crossing, can be "eventually all trains will be able to use the track". Informally, a liveness specification determines what types of events must eventually happen, i.e., it says that "something good eventually happens".

For the work presented in this thesis, we will focus on safety specification. However, liveness issues are important since any safety specification can be satisfied by the empty program, i.e., the program that does nothing, and, thus, liveness specification helps rule out trivial implementations.

In general, if a property is finitely refutable, then it is a safety property. This means that the safety property can be refuted by inspecting only a finite prefix of a computation. On the other hand, a liveness property is not finitely refutable, i.e., it cannot be refuted by inspecting a finite prefix of a computation, rather it is refuted by inspecting infinite state sequences.

3.5 Temporal Logic

In a sequential system, the input-output semantics is adequate for analyzing the system, but is however inadequate for concurrent systems. For example, the input-output semantics cannot adequately capture specifications such as "*eventually* $(x = 2)$" or "*never* $(y = 3)$ ".

Temporal Logic is a formalism for describing sequences of transitions between states in a reactive systems. In temporal logic, a specification is a *logical formula* that describes a set of computations. In the work presented in this thesis, a semantic view is adopted, we reason about properties of a program in terms of its transitions, rather than expressing them in any specification language.

3.6 Refinement

A program can be viewed as a special type of specification. A lower level specification differs from a higher-level specification in that it contains more implementation details. Thus, we want lower-level transitions to appear as stuttering steps (Def. 11) in the higher level specification.

This can be modelled through the concept of *projection*.

Definition 23 (State Projection) *The* projection of a state s of (a lower-level specification) p on (a higher-level specification) p' *is the state obtained by considering only the variable of* p'*.*

Definition 24 (Computation Projection) *The* projection of a computation c of (a lower-level specification) p on (a higher-level specification) p' is obtained by taking the projection of each state of c (of p) on p'.

To model this, we introduce a projection function, π, from a lower-level specification p to a higher-level specification p'. Given a state s of program p, $\pi(s)$ refers to the variables of p'. We abuse the notation by defining $\pi(\alpha)$ for a projection of a computation α. Thus, π partitions the set of variables of p, V_p, into a set of *internal variables* (V_i) and a set of *external variables* (V_e). Therefore, changes to variables in V_i appears as stuttering steps in $\pi(\alpha)$.

This leads to the concept of *refinement* [AL91]. When we substitute p' for a specification S, when we say that a computation c satisfies S (Defs. 17), we really meant that the projection of that computation $\pi(c)$ satisfies S (i.e., $\pi(c) \in S$).

Refinement from a specification represents a useful way to constructing programs. Using refinement, a low-level program can be constructed from a given specification through the application of correctness-preserving refinements. With each refinement step, a lower-level program p is obtained from a higher-level program p'through the addition of more implementation details. It is those implementation details that are hidden by π.

3.7 Fault Models and Fault Tolerance

The faults that a program is subjected to can be systematically represented by actions whose execution perturbs the state of the program. Such representation is possible regardless of the type of faults (stuck-at, crash, Byzantine etc), nature of the faults (permanent, intermittent or transient), or the ability to observe the effects of the faults (detectable or not).

First, we define the term *fault class*.

Definition 25 (Simple Fault Class) *A simple fault class for a given program p over a variable v_i in p is a set of transitions (actions) over the variable v_i.*

Definition 26 (Fault Class) *A fault class F for a program p over variables $v_1 \dots v_n$ in p is a set of simple fault classes for p over $v_1 \dots v_n$.*

In this thesis, we focus on the subset of fault models that can potentially be tolerated: We disallow faults to violate the safety specification directly. For example, if a safety specification constrains the output variables of a program, the fault model prevents the *fault actions* of F to modify the output variables in such way that the fault itself results in a safety violation. However, fault actions can change the program state such that subsequent program actions violate the safety specification.

The reason for choosing such a failure model is that we target tolerable fault models. If a fault can directly violate safety, for example, by corrupting the output variables in such a way that safety can be violated, then no fail-safe fault-tolerant program exists. To see this, observe that if from state s, a fault can cause safety violation, then this program should not visit state s. If such faults can occur in every state, then all such states need to be made unreachable, i.e., the invariant of the program is an empty set. Thus, no fail-safe fault-tolerant program exists, hence our focus on tolerable fault models.

Definition 27 (Fault model) *A* fault model *F* for program *p* and safety specification *SS is a fault class F for program p over its variables that do not violate SS, i.e., if transition (s_j, s_{j+1}) is in F and $s_0, s_1, \dots, s_j$ is in SS, then $s_0, s_1, \dots, s_j, s_{j+1}$ is in SS.*

Definition 28 (Computation in the presence of faults) *A* computation of *p* in the presence of *F is a weakly p-fair sequence of states $s_0, s_1, \dots$ such that s_0 is an initial state of p and for each $j \geq 0$, s_{j+1} results from s_j by executing a program action from p or a fault action from F and there exists no program action ac such that ac is permanently enabled but never executed.*

Weakly p-fair means that only the actions of p are treated weakly fair (fault actions must not eventually occur if they are continuously enabled). We say that a *fault occurs* if a fault action is executed.

Rephrased in the transition system view, a fault model adds a set of transitions to the transition relation of p. We denote the modified transition relation by δ_p^F. Since fault actions are not treated fairly, their occurrence is not mandatory. Note that we do not rule out faults that occur infinitely often (as long as they do not directly violate the safety property).

Fault Tolerance Specifications In the absence of faults, a program p should refine its problem specification. In the presence of faulty actions, p may not refine its specifications, but can, on the other hand, refine some weaker "tolerance specification". In this thesis, we focused on fail-safe fault tolerance.

Definition 29 (Fail-safe fault-tolerance) *Let S be a specification and SS be the smallest safety specification including S, and fault class F. A program p is said to be* fail-safe F-tolerant for specification *S iff all computations of p in the presence of faults F satisfy SS.*

If F is a fault model and SS is a safety specification, we say that a *program p is F-intolerant for SS* iff p satisfies SS in the absence of faults F but violates SS in the presence of faults F. For brevity, we will write *fault-intolerant* instead of *F-intolerant for SS* if F and SS are clear from the context.

A note on critical actions introduced in 3.2: Critical actions are exactly those program actions whose execution in the presence of faults can lead to violation of safety. As such, in embedded applications such as those of plant controllers etc, the program actions that control progress while maintaining safety are critical actions.

Chapter 4

Perfect Detectors: Basis for Perfect Detection

Nowadays, there are computer systems all around us that control our everyday lives, from being present in safety-critical systems such as airplanes, to being present in consumer-oriented products, such as automobiles, washing machines etc. Especially for the consumer-oriented products, cost-effective solutions for the provision of dependability are of paramount importance, leading to the fact that software-based fault tolerance is being provided.

In this thesis, we are interested in providing efficient fail-safe fault tolerance, i.e., it is acceptable for a fail-safe fault-tolerant program to halt, as long as it remains in a safe state. The idea is to be able to detect when the program is about to violate its safety specification, and halt at that time. Thus, for the detection part, the program needs to be "upgraded" with a program component, called a *detector*. Intuitively, the detector component helps the program in detecting when "something bad" is about to happen, such that the program halts to avoid doing the "bad" thing, i.e., violate safety.

However, the design of efficient detectors is problematic. Leveson *et al.* [LCKS90] conducted a large experiment on the effectiveness of self checks, which are instances of a detector, in software. They pointed out in [LCKS90] that, among others, (i) some detectors (self-checks) detect non-existent errors, i.e., there are many false alarms (i.e., false detections), and (ii) many detectors that were designed were ineffective, i.e., they do not signal any error, when there is one in the system. In the first case, the efficiency of the system may decrease, since the system may halt prematurely, while in the second case, the safety of the system may be violated. So, we need a methodology for the design of efficient detectors.

In this chapter, we first provide a formal overview of detectors [Kul99], and explain their role in fail-safe fault-tolerant program. *Our main contribution is the development of a novel theory of detectors, that is centered around the notion of an inconsistent transition.* We further identify a special class of detectors, called perfect detectors, and explain its role in the design of fail-safe fault tolerance. Specifically, we show that composing critical actions of a program p with perfect detectors is sufficient in transforming p into a fail-safe fault-tolerant program. We then present an algorithm, based upon the theory, that, given a fault-intolerant program p with safety specification SS, and a fault class F, generates a fail-safe fault-tolerant program p', which is the fail-safe fault-tolerant version of p. The main property of perfect detectors is that they detect errors if and only if these errors may lead to violation of safety. Thus, perfect detectors can be shown to address the two problems identified by Leveson *et al.*

In [JHCS02, JHS02b, JHS02a], a set of perfect detectors was initially referred to as SS-globally consistent detectors. A SS-consistent detector is one that detects an error if and only if the error can lead to violation of safety. A set of such SS-consistent detectors is said to be SS-globally consistent.

4.1 Introduction

Safety-critical applications need to satisfy stringent dependability requirements in their provision of services. Unless sound design methods are used to synthesize such applications, the process of designing safety-critical applications is likely to be a complex one. To reduce the complexity of designing such applications, Arora and Kulkarni [AK98a] have proposed a transformational approach, whereby an initially fault-intolerant program is systematically transformed into a fault-tolerant one. The main step involved in designing a fault-tolerant program is composing the corresponding fault-intolerant program with components that (i) detect and/or (ii) correct errors that arise as a result of faults, depending on the level of fault-tolerance to be achieved. The class of programs that achieves the first goal is termed *detectors* while the class of programs that achieves the second goal is called *correctors* [AK98c].

We restrict our attention to designing *fail-safe* fault-tolerance. Intuitively this means that it is acceptable that the program "halts" if faults occur as long as it always remains in a "safe" state. This type of fault-tolerance is often used in (nuclear) power plants or train control systems where safety (avoidance of catastrophic events) is more important that continuous provision of service. In the context of the Arora/Kulkarni approach, fail-safe fault-tolerance can be achieved by merely employing detectors.

Generally, detectors can be regarded as an abstraction of many different existing fault-tolerance mechanisms. For example, a common way to achieve fault-tolerance is to replicate a critical task and schedule it on different processors. The outputs of these tasks are brought together in a voter which outputs a consistent value. The voter contains a comparator which is an instance of a detector. Another (maybe more obvious) example of a detector is the use of error detecting codes. Other error handling mechanisms like acceptance tests or executable assertions can also be formulated as detectors in the sense of Arora and Kulkarni [AK98c]. Hence, reasoning on the level of detectors makes an approach applicable to many different practical settings.

In this chapter, we present a *sound*, and *complete* algorithm for transforming an initially fault-intolerant program p into an efficient fail-safe fault-tolerant program p'. The algorithm being sound and complete, meaning that (i) the transformed program p' is in fact a fail-safe fault-tolerant version of the original program p (soundness), and (ii) if a fail-safe fault-tolerant version of the program exists, then the algorithm will find it (completeness). By efficient, we mean that the fail-safe fault-tolerant detect errors if and only if errors lead to violation of safety, thus addressing some of the problems identified by Leveson *et al* in [LCKS90], i.e., p' has perfect error detection. Overall, our approach is applicable to a class of programs, called *bounded programs*. The property of bounded programs is that there is no unbounded loop within or across processes. Embedded applications are often instances of bounded programs. Distributed algorithms such as mutual exclusion, byzantine agreement etc. are also instances of bounded programs.

Our algorithm is derived out of a refined theory of detectors. This theory develops a terminology which captures and explains the working principles of detectors better than before. The basic building block of the theory is the notion of a transition which is *inconsistent* with respect to a safety specification [Lam77]. This can be understood as follows: Executing a transition inconsistent with respect to the safety specification can lead to a violation of the safety specification if no countermeasures are taken.

Building upon this concept, we develop a theory of accurate, complete, and perfect detectors together with the necessary correctness theorems. Intuitively, a detector is *accurate* if it "preserves" correct behaviors of the system in the presence of faults. A detector is *complete* if it "rejects" incorrect behaviors in the presence of fault. A detector is *perfect* if it is accurate and complete.

In this chapter, we make the following contributions:

- We first present a *novel* theory of detectors which accurately captures the working principles of detectors.
- We identify a class of detectors, called *perfect detectors*, and explain their role, and importance in fail-safe fault-tolerance.
- Based on this theory, we provide an algorithm that systematically

transforms a fault-intolerant program into a fail-safe fault-tolerant program with perfect detection.

The chapter is structured as follows: Section 4.2 provides an overview of detectors and their role in establishing fail-safe fault tolerance. Section 4.3 defines the problem of adding fail-safe fault-tolerance using detectors. Section 4.4 develops the theory of perfect detectors. In Section 4.5, we present the algorithm that automatically generates a fail-safe fault-tolerant program from the corresponding fault-intolerant program with perfect detection capabilities. Some examples are presented in Section 4.6. We conclude the paper in Section 4.7.

4.2 An Overview of Detectors

In this section, we present a brief introduction of a detector component. For a complete formalization, we refer the reader to Kulkarni [Kul99].

A detector module d is a program component that is used to check whether its detection predicate D is "True", where D is a state predicate. Specifically, a detector d can be of the form

$$\neg Z \wedge D \rightarrow Z := True.$$

It means that if the detection predicate D is "True", then Z, the witness predicate, becomes "True". The detector component needs to satisfy three properties:

1. Safeness,
2. Progress, and
3. Stability

By safeness, we mean that the detector never allows Z to witness D incorrectly. Progress means that if D is continuously "True", Z will eventually be become "True". Stability means that once Z becomes "True", it continues to be unless D becomes "False". Examples of detectors in the literature abound, such as error detection codes, executable assertions [Hil00], comparators, and so on. However, if the detection predicate is such that it is not related to the safety specification of the program, then the error detection process will not be efficient. Hence, to design "relevant" detectors, they need to relate to the specification of the program. In the next section, we explain their role in fail-safe fault tolerance and relate it to their design.

4.2.1 Role of Detectors in Fail-Safe Fault Tolerance

We adopt the view of Arora and Kulkarni [AK98c] that a fault-tolerant program is the composition of a corresponding fault-intolerant program with fault tolerance components. Using the same system model as used in this work, Arora and Kulkarni proved that *detectors* are necessary and sufficient to establish fail-safe fault tolerance. Intuitively, a detector detects whether a given state (detection) predicate is satisfied in a given state. Instances of detectors can be executable assertions, error detection codes, self checks, and comparators.

Given our focus on fail-safe fault tolerance, we review the result of Arora and Kulkarni [AK98c] stating that detectors are *necessary* and *sufficient* to build fail-safe fault-tolerant applications. The main idea of the result is to use detectors to simply "halt" the program in a state where it is about to violate the safety specification. An important prerequisite for the Arora/Kulkarni sufficiency result is that specifications are fusion-closed. Fusion-closed specifications (Def. 16) allow to characterize a safety specification as a set of disallowed "bad" *transitions* (instead of a set of disallowed computation prefixes).

Proposition 2 *Let SS be a safety specification, p an F-intolerant program for SS for fault class F. If p violates SS then there exists a transition $t \in \delta_p$ such that for all computations σ of p holds: If t occurs in σ then $\sigma \notin SS$.*

Proof. Since p violates SS, there exists a computation σ which is not in SS. The fact that SS is a safety property implies that σ contains a minimal prefix, written $\alpha \cdot s \cdot s'$, which does not maintain SS (i.e., which prevents the computation from being in SS). This prefix has at least length 2 because all initial states of p maintain SS. We must now show that if (s, s') occurs in any other computation ρ of p, then $\rho \notin SS$:

1. For a contradiction, assume $\rho = \hat{\alpha} \cdot s \cdot s' \cdot \hat{\beta} \in SS$. We will show that $\alpha \cdot s \cdot s'$ maintains SS.
2. Since SS is a safety property and $\rho \in SS$ (step 1), all prefixes of ρ maintain SS.
3. From step 2 and because it is a prefix of ρ, computation $\hat{\alpha} \cdot s \cdot s'$ maintains SS.
4. From step 3 and definition of maintains: $\exists \hat{\delta} : \hat{\alpha} \cdot s \cdot s' \cdot \hat{\delta} \in SS$.
5. From assumption $\alpha \cdot s$ maintains SS, so from definition of maintains we have: $\exists \delta : \alpha \cdot s \cdot \delta \in SS$.
6. Because of fusion-closure of SS and the steps 4 and 5 construct: $\alpha \cdot s \cdot s' \cdot \hat{\delta} \in SS$.
7. Step 6 means that $\alpha \cdot s \cdot s'$ maintains SS, which is a contradiction to the fact that $\alpha \cdot s \cdot s'$ does not maintain SS.

$\square$

We call the transitions identified in Proposition 2 *bad transitions*. Intuitively, to maintain a safety specification now requires to keep track of the current computation and take precautions not to run into one of the bad transitions which are disallowed by the safety specification. The safety specification of a program can thus be concisely represented as a set of bad transitions. Note that, in this work, we assume that the safety specification is provided as such, i.e., the smallest specification that contains the specification. If this is not the case, and if the specification of the program is expressed as a formula in temporal logic, the set of bad transitions can be obtained in polynomial time, by considering all transitions $(s, t) : s, t \in S_p$.

From our restrictions of the fault model (Chapter 3, Section 3.7) (fault transitions cannot directly violate safety) we know that bad transitions must be program transitions (also from Proposition 2). A detector refines the guard of the corresponding action in such a way that the action is never executed whenever the computation could result in taking a bad transition.

Formally, a detector for an action implements a state predicate d which is "True" iff execution of the action starting in d maintains the specification. In the programming notation, given an action $g \rightarrow st$, a detector for this action refines the guard to $g \wedge d$. Arora and Kulkarni formulate this fact in their original work as follows [AK98a, Theorem 4.3]:

Theorem 1 (Sufficiency of detectors) *For each action ac of p there exists a predicate d such that execution of ac in a state where d holds maintains SS.*

Definition 30 (Detector for an action) *Let SS be a safety specification. An SS-*detector d monitoring program action *ac* of p *is a state predicate of p which is guaranteed to exist according to Theorem 1.*

We will simply talk about *detectors* instead of *SS-detectors* if the relevant safety specification is clear from the context. Taken together, Theorem 1 and Definition 30 state that it is sufficient to compose a given action with a relevant detector, which is guaranteed to exist, to ensure that the action executes safely.

Consider the transition system view of a program p again. We define the notions of reachable/unreachable states/transitions in the presence/absence of faults [GV00, GV01].

Definition 31 (Reachable state) *We say that a* state s is reachable by p *iff starting from an initial state of p it is possible to construct a computation which contains s using only transitions from δ_p. Otherwise s is unreachable.*

Definition 32 (Reachable transition) *A transition (s, t) of p is reachable iff state s is reachable by p. Otherwise it is unreachable.*

Definition 33 (Reachable state in the presence of faults) *We say that a* state s is reachable by p in the presence of faults F *iff starting from an initial state of p it is possible to construct a computation which contains s using only transitions from δ_p^F. Otherwise s is unreachable in the presence of faults.*

Definition 34 (Reachable transition in the presence of faults) *We say that a* transition (s, t) is reachable by p in the presence of faults *iff s is reachable by p in the presence of faults. Otherwise, (s, t) is unreachable in the presence of faults.*

Fig. 4.1 illustrates the concepts of reachable states/transitions in the absence/presence of faults.

Figure 4.1: Reachable states/transitions

Observe that, starting from an initial state, in the absence of faults, all computations of p satisfy the safety specification SS. Thus, the computations of p go through those transitions (states) of p that are reachable in the absence of faults.. However, in the presence of faults, some transitions (states) which were unreachable in the absence of faults, now become reachable. Using the above terminology, detectors *remove* some of the program transitions which were unreachable by p in the absence of faults, but become reachable in the presence of faults. In a sense, composing a program with detectors means to refine the original transition relation and eliminate certain program transitions so as to make bad transitions unreachable.

We close this section with a final remark regarding the assumption that specifications be fusion-closed. Informally spoken, fusion-closure guarantees that the entire history of a computation "is available" in the current state of the system, i.e., it is sufficient to observe the current system state to know whether the next step will result in a disallowed prefix. It has been observed [Gum93, AK98a] that specifications in the popular Unity logic [CM88] are fusion-closed, as are low-level specifications like C programs or transition systems. In general a specification that is not fusion-closed can be converted into a fusion-closed specification through the addition of history variables. How this can be done in a way that minimizes the number of additional states remains a topic for further research.

In this section, we have provided an overview of detectors and their role and importance in the design of fail-safe fault tolerance. However, little is known about whether the detectors designed are efficient or not. To address this problem, we will first define the transformation problem of fail-safe fault tolerance (Section 4.3). We will then develop a theory that underpins an algorithm that solves the transformation problem.

4.3 The Transformation Problem

In this section, we will formally state the problem of transforming a fault-intolerant program p into a fail-safe fault-tolerant version p' for a given safety specification SS and fault model F.

When deriving p' from p, only fault tolerance should be added, i.e., p' should not satisfy SS in new ways in the absence of faults. Specifically, there are two conditions to be satisfied in the transformation problem:

- If there exists a transition (s, t) in p' that is not reached by p to satisfy SS, then (s, t) cannot be used by p', since this means that there are other ways p' can satisfy SS in the absence of faults. Thus, the set of transitions of p' must be a subset of the set of transitions of p.

- Also, if there exists a state s reachable by p' in the absence of faults that is not reached by p in the absence of faults, then this means that p' can satisfy SS differently from p in the absence of faults, and such a state s should not be reached by p' in the absence of faults. Thus, the set of states reachable by p' should be a subset of the set of states reachable by p.

In general, these conditions result in the requirement that both programs should have the same set of fault-free computations. Formally, we define the transformation problem as follows:

Definition 35 (Transformation for fail-safe fault tolerance) *Let SS be a safety specification, a fault model F, and p an F-intolerant program for SS. Identify a program p' such that the following three conditions hold:*

1. *p' satisfies SS in presence of F.*
2. *In the absence of faults, every computation of p' is a computation of p.*
3. *In the absence of faults, every computation of p is a computation of p'.*

The transformation problem can also be formulated as a decision problem:

Definition 36 (Decision problem for the transformation) *Let SS be a safety specification, a fault model F, and p an F-intolerant program for SS. Does there exist a program p' such that the following three conditions hold:*

1. *p' satisfies SS in presence of F.*
2. *In the absence of faults, every computation of p' is a computation of p.*
3. *In the absence of faults, every computation of p is a computation of p'.*

Later in Section 4.5 we present a sound, and complete algorithm which solves the above transformation problem, i.e., we present an algorithm that systematically transforms a fault-intolerant program into a program that satisfies the above three conditions. Soundness of the algorithm means that the resulting program indeed solves the transformation problem. Completeness of the algorithm means that if the solution to the decision problem is true, then the algorithm will find the fail-safe fault-tolerant program.

The algorithm is based on a theory of detectors which we introduce in the following section.

4.4 A Theory of Perfect Detectors

This section presents a theory of detector components which helps in the design of efficient fail-safe fault-tolerant applications. The theory is centered around the notion of an SS-inconsistent transition which is introduced in Section 4.4.1. Using this notion, we identify correctness criteria for programs composed with so-called *perfect* detectors in Section 4.4.2. Our algorithm to add fail-safe fault tolerance presented in Section 4.5 directly follows from the theory presented now.

4.4.1 Transition Consistency in the Context of Safety Specifications

The intuition behind the definition of transition inconsistency is that if a given computation violates the safety specification, then some "erroneous" transition occurred in the computation, i.e., that transition is inconsistent with the safety specification of the program. Specifically, consider a fault-intolerant program p with safety specification SS, and a computation α that violates SS. From Propositions 1 and 2 we know that there exists a prefix σ of α that contains a bad transition.

Figure 4.2: An example to illustrate the concept of inconsistent transition

When a computation violates safety, intuitively it means that the program is on a "wrong path", and such deviation has happened earlier. This intuition is captured by the SS-inconsistency concept, as defined below.

Definition 37 (*SS*-inconsistent transitions) *Given a fault-intolerant program p with safety specification SS, and a computation* α *of p in the presence of faults. A transition* (s, s') *is* SS-inconsistent for p with respect to α in presence of faults F *iff*

- *There exists a prefix α' of α such that α' violates SS*
- *(s, s') occurs in α', i.e., $\alpha' = \sigma \cdot s \cdot s' \cdot \beta$,*
- *All transitions in $s \cdot s' \cdot \beta$ are in δ_p,*
- *$\sigma \cdot s$ maintains SS.*

We now illustrate this concept pictorially. From Fig. 4.2, transition $(7, 8)$ is SS-inconsistent for p with respect to computation $\alpha = 1 \cdot 7 \cdot 8 \cdot 9 \ldots$. α violates SS since it contains a bad transition, i.e., $(10, 11)$. Observe that transitions $(8, 9)$, and $(9, 10)$ are also SS-inconsistent for p with respect to a given computation.

```
Program P1
      var x init 1, y init 1, z init 10, c init 1 : int

      c = 1 → x := read(); c := c + 1; // value between 5 and 10
      c = 2 → y := read(); c := c + 1; // value between 5 and 15
      c = 3 → z := x + y; c := c + 1
      c = 4 → output(z); c := 1 // loop forever

F (faults):
      true → x := random [0 . . . 25]
      true → y := random [0 . . . 50]
```

Figure 4.3: Program to illustrate the concept of SS-inconsistent transitions.

We now illustrate this definition: Consider the program $P1$ in Figure 4.3 which reads two sensors, and then outputs the sum of the two readings. The safety specification SS requires the output to be always between 10 and 25. The fault transitions indicate that, from each state, the value of variable x (respectively, y) can be arbitrarily changed to a value in the range of $[0 \ldots 25]$(respectively, $[0 \ldots 50]$). Consider now computation α (states are given as triples $\langle x, y, z\rangle$, i.e., the program counter c is not explicitly given):

$$\alpha : \langle 1, 1, 10\rangle \cdot \langle 10, 1, 10\rangle \cdot \langle 10, 5, 10\rangle \cdot \langle 10, 5, 15\rangle$$

Obviously, α satisfies SS and so no program transition is SS-inconsistent. Now consider computation β which violates SS:

$$\beta : \langle 1, 1, 10\rangle \cdot \langle 10, 1, 10\rangle \cdot \langle 25, 1, 10\rangle \cdot \langle 25, 5, 10\rangle \cdot \langle 25, 5, 30\rangle$$

In β, a fault transition occurs after the second state, i.e., state $\langle 10, 1, 10\rangle$, changing the value of x to 25. The subsequent program transition from $\langle 25, 1, 10\rangle$ to $\langle 25, 5, 10\rangle$ is SS-inconsistent, since the execution of the following program transition to state $\langle 25, 5, 30\rangle$ causes a violation of the safety specification. The program transition from $\langle 25, 5, 10\rangle$ to $\langle 25, 5, 30\rangle$ is also SS-inconsistent. The first program transition and the fault transition are not SS-inconsistent.

Intuitively, an SS-inconsistent transition for a given program computation is a program transition where the subsequent execution of a sequence of program actions causes the computation to violate the safety specification. In a sense, SS-inconsistent transitions lead the program computation on the "wrong path". The requirement of a sequence of program transitions in the prefix is to capture the fact that no precaution is being taken, and the inconsistent transition captures the fact that something harmful has occurred.

Now we define SS-inconsistency independent of a particular computation. This captures the fact that, starting from such a transition, it is *possible* to violate safety, i.e., if such a transition occurs during a computation, then there is a chance that this computation will violate safety.

Definition 38 (SS-inconsistent transition for p) *Given a program p with safety specification SS. A transition (s, s') is SS*-inconsistent for p in presence of faults F *iff there exists a computation α of p in the presence of faults F such that (s, s') is SS-inconsistent for p w.r.t. α in presence of F.*

Intuitively, a transition (s, s') is SS-inconsistent for a program p if the transition starts leading the computation on the wrong path. From Fig. 4.2, transition $(7, 8)$ is SS-inconsistent for p since it has taken the computation on a possible wrong path, i.e., there can subsequently be safety violation.

In general, a transition can be SS-inconsistent w.r.t. a computation α_1, and not be SS-inconsistent w.r.t. α_2. This can be due to nondeterminism in program execution. To see this consider the program $P2$ in Figure 4.4. The safety specification SS mandates that $10 \leq d \leq 50$ at all times. Consider now the following computation α_1 of $P2$ (a state is given as $\langle w, x, y, z \rangle$):

$$\alpha_1 = \langle 1,5,1,10\rangle \cdot \langle 1,10,1,10\rangle \cdot \langle 1,45,1,10\rangle \cdot \langle 15,45,1,10\rangle \cdot \langle 15,45,15,10\rangle \cdot \langle 15,45,15,60\rangle$$

In the second state a fault occurs setting x to 45 and effectively causing α_1 to violate SS after execution of a sequence of program transitions. Notice that the transition $t = (\langle 1,45,1,10\rangle, \langle 15,45,1,10\rangle)$ is SS-inconsistent for p w.r.t. α_1.

Now consider computation α_2 of p:

$$\alpha_2 = \langle 1,5,1,10\rangle \cdot \langle 1,10,1,10\rangle \cdot \langle 1,45,1,10\rangle \cdot \langle 15,45,1,10\rangle \cdot \langle 0,45,1,10\rangle \cdot \langle 0,45,0,10\rangle \cdot \langle 0,45,0,45\rangle$$

Here again a fault happens in the second state but due to a lucky interleaving of program actions α_2 maintains SS. Hence, the same program transition t as above is not SS-inconsistent for p with respect to α_2.

Program $P2$
  var w init 1, $c1$ init 1 : int // process a
  var x init 5, y init 1, z init 10, $c2$ init 1 : int // process b

process a:
  $c1 = 1 \rightarrow w :=$ read(); $c1 := c1 + 1$; // value between 15 and 25
  $c1 = 2 \wedge x \leq 15 \rightarrow w := w + 5$; $c1 := 1$; // loop
  $c1 = 2 \wedge x > 15 \rightarrow w := w - 15$; $c1 := 1$; // loop

process b:
  $c2 = 1 \rightarrow x :=$ read(); $c2 := c2 + 1$; // value between 0 and 20
  $c2 = 2 \rightarrow y := w$; $c2 := c2 + 1$;
  $c2 = 3 \rightarrow z := y + x$; $c2 := c2 + 1$;
  $c2 = 4 \rightarrow$ output(z); $c2 := 1$; // loop

F (faults):
  true $\rightarrow x :=$ random $[10 \ldots 45]$
  true $\rightarrow w :=$ random $[1 \ldots 50]$

Figure 4.4: Program containing two concurrent processes with a transition that is both SS-inconsistent and not SS-inconsistent w.r.t. two different computations.

If we cannot find a computation in the presence of faults for which a particular transition is SS-inconsistent then we say that this transition is SS-consistent. Specifically,

Definition 39 ($\boldsymbol{SS}$-consistent transition for p) *Given a program p with safety specification SS. A* transition (s, s') is SS-consistent for p in presence of faults F *iff* (s, s') *is not SS-inconsistent for p in presence of F.*

For example, from Fig. 4.2, transition $(1, 2)$ is SS-consistent for p. Transition $(13, 14)$ is also SS-consistent for p. The notion of SS-consistent transition captures the fact that executing such a transition is inherently safe, i.e., there is no chance of safety being violated unless something harmful occurs.

The notion of an SS-inconsistent transition is a characteristic of a computation that violates SS, and is captured by the following proposition (Prop. 3).

Proposition 3 *Given an fault-intolerant program p with a safety specification SS. Every computation α of p in the presence of faults that violates SS contains an SS-inconsistent transition for p w.r.t. α in presence of F.*

Proof.

1. Because p is F-intolerant, there exists a computation α of p in the presence of faults such that $\alpha \notin SS$.
2. From step 1 and Proposition 2 there exists a bad transition (s, s') in α.
3. From step 2 and the restriction of F follows that $(s, s') \in \delta_p$.
4. From step 3 and Definition 37, (s, s') is SS-inconsistent for p w.r.t. α.

$\square$

Earlier, we have characterized inconsistent transitions by their ability of causing computations to violate safety. Since a bad transition is reachable only in the presence of faults, inconsistent transitions can also be characterized through the reachability of bad transitions.

Proposition 4 *Given a fault-intolerant program p with a safety specification SS. If (s, s') is an SS-inconsistent transition for p in the presence of faults F, then a bad transition is reachable starting from s using only program transitions from δ_p.*

Proof. The proof follows directly from the definition of *SS*-inconsistent transitions and Proposition 2. □

Reachability of bad transitions in δ_p leads to the following observation.

Proposition 5 *Given a fault-intolerant program p for safety specification SS. Every SS-inconsistent transition for p in presence of faults F is not reachable in the absence of faults F.*

Proof.

1. For a contradiction, assume the start state s of an *SS*-inconsistent transition (s, s') is reachable in the absence of faults.
2. Step 1 implies that there exists a computation $\alpha \cdot s \cdot s'$ of p in the absence of faults.
3. From the fact that (s, s') is inconsistent, and Proposition 4 there exists a computation $s \cdot s' \cdot \beta$ of p in the absence of faults in which a bad transition occurs.
4. From steps 2 and 3 follows that there exists a computation $\sigma = \alpha \cdot s \cdot s' \cdot \beta$ of p in the absence of faults containing a bad transition.
5. From step 4 and Proposition 2 there exists a computation of p in the absence of faults which violates *SS*.
6. From step 5 p violates *SS* in the absence of faults, a contradiction.

□

Note that the previous observation cannot be strengthened to an equivalence (a non-reachable transition in the absence of faults must not be *SS*-inconsistent). But it can be reformulated to characterize reachable transitions in the absence of faults as *SS*-consistent.

Corollary 1 *Given a fault-intolerant program p for a safety specification SS. Every reachable transition $(s, s') \in \delta_p$ in the absence of faults F is SS-consistent for p in the presence of faults F.*

In the next section, we introduce the notion of perfect detectors using the terminology of SS-(in)consistency.

4.4.2 Perfect Detectors

From the previous section, we observed that SS-inconsistent transitions are those transitions that can lead a program to violate its safety specification in the presence of faults, if no precautions are taken. Detectors, as we explained in Section 4.2, are a means to implement these precautions. However, as pointed out by Leveson *et al.* in [LCKS90], design of efficient detectors is inherently complex. Hence, we introduce the class of *perfect detectors*.

Perfect detectors are a means to *efficiently* implement these precautions. The definition of perfect detectors follows two design principles: A (perfect) detector d monitoring a given action ac of program p needs to (1) "reject" the starting states of all transitions induced by ac that are SS-inconsistent for p in the presence of faults, and (2) "keep" the starting states of all induced transitions that are SS-consistent for p in the presence of faults. These two properties are captured in the definition of *completeness* and *accuracy* of detectors (the notions are defined in analogy to Chandra and Toueg [CT96]).

Definition 40 (Detector accuracy) *Given a program p with safety specification SS, and a program action ac of p.* A detector d monitoring ac is SS-accurate for ac in p in the presence of faults F *iff for all transitions (s, s') induced by ac holds: if (s, s') is SS-consistent for p in the presence of F, then $s \in d$.*

The accuracy property captures the fact that efficient detectors should not make mistakes. Thus, if a detector detects that a transition is safe, then it "accepts" the state.

Definition 41 (Detector completeness) *Given a program p with safety specification SS, and a program action ac of p.* A detector d monitoring action ac is SS-complete for ac in p in the presence of faults F *iff for all transitions* (s, s') *induced by ac holds: if* (s, s') *is SS-inconsistent for p in presence of* F*, then* $s \notin d$*.*

On the other hand, the completeness property captures the notion that a detector should "reject" all harmful transitions.

Definition 42 (Perfect detector) *Given a program p with safety specification SS, and a program action ac of p. A* detector d monitoring ac is SS-perfect for ac in p in presence of faults F *iff d is both SS-complete and SS-accurate for ac in p in presence of* F*.*

Where the specification is clear from the context we will write *accuracy* instead of *SS-accuracy* (the same holds for completeness and perfection). Overall, the perfectness property of detectors captures the fact that such a detectors detect all harmful faults, and do not make mistakes.

Intuitively, the completeness property of a detector is related to the safety property of the program p in the sense that the detector should filter out all "harmful" SS-inconsistent transitions for p, whereas the accuracy property relates to the liveness specification of p in the sense that the detector should not rule out SS-consistent transitions. This intuition is captured by the following lemmas. The first one (Lemma 1) uses the accuracy property to show that the fault free behavior of a program is not affected by adding perfect detectors. Intuitively, this also means that, in the absence of faults, addition of perfect detectors to a program does not cause the original program to lose any of its behavior. The next one (Lemma 2) uses the completeness property to show that perfect detectors indeed establish fail-safe fault-tolerance. Intuitively, this also means that these detectors are efficient, in the sense that they do not make mistakes, and they also cause "rejection" of all "harmful" transitions. Jhumka *et al.* introduced the concept of SS-globally consistent detectors in [JHCS02]. As mentioned in [JHS03], a set of (SS-) perfect detectors for different actions in program p with safety specification SS is SS-globally consistent for p.

Lemma 1 (Perfect detectors and fault-free behavior) *Given a fault-intolerant program p and a set D of perfect detectors, consider program p' resulting from the composition of p and D. Then the following statements hold:*

1. *In the absence of faults, every computation of p' is a computation of p.*
2. *In the absence of faults, every computation of p is a computation of p'.*

Proof.

1. From Corollary 1, every program transition which is reachable in p is SS-consistent.

2. From construction, p' results from adding perfect detectors to p. Because they are perfect (Definition 42), they are accurate.

3. From steps 1, 2 and the definition of accuracy, all SS-consistent transitions of p are also transitions of p'.

4. Steps 1 and 3 imply that every reachable transition in p is also reachable in p'.

5. Step 4 implies that every computation of p is also a computation of p', proving the first claim of the lemma.

6. From the definition of a detector (Definition 30) follows that composition with detectors does not introduce new state transitions.

7. Step 6 implies that $\delta_{p'} \subseteq \delta_p$.

8. Step 7 implies that every computation of p' is also a computation of p, proving the second claim of the lemma.

□

Lemma 1 intuitively suggests that, in the absence of faults, program p, and its corresponding fail-safe fault-tolerant program have identical behaviors. What it also suggests is that any other detector that is designed defensively (defensive programming) interferes with the behavior of the fail-safe fault-tolerant program in the absence of faults. Specifically, it means that there exists valid (SS-consistent) transitions that are however ruled out by the detector, hence liveness is compromised.

To understand the behavior of a program in the presence of faults, we make use of the notions of critical actions, which we formalized here. Intuitively, a critical action is one which if executed in an erroneous state will cause violation of safety.

Definition 43 (Critical and non-critical actions) *Given a program p with safety specification SS, and fault model F. An action ac of p is said to be* critical *iff there exists a transition (s,t) induced by ac such that (s,t) is a bad transition (Proposition 2) that is reachable in presence of faults F. An action is* non-critical *iff it is not critical.*

Thus, the set of bad transitions reachable in the presence of faults define a set of critical actions.

Lemma 2 (Perfect detectors and behavior in the presence of faults) *Given a fault-intolerant program p for a safety specification SS. Given also a program p' by composing the critical actions of p with perfect detectors. Then, p' satisfies SS in presence of faults.*

Proof.

1. For a contradiction assume that p' violates SS. From definition of violates follows that there exists a computation σ of p' which is not in SS.

2. Step 1 and Proposition 2 imply that there a bad transition (s, s') occurs in σ.

3. Because of the restrictions on the fault model (critical variables are not affected), the transition (s, s') from step 2 must be a program transition (i.e., $(s, s') \in \delta_{p'}$)

4. From step 3, and Definition 43, there exists a critical action ac that induces the bad transition from step 3

5. From Definition 37 and step 3 the transition (s, s') is SS-inconsistent.

6. Consider the critical program action ac (from step 4) causing the bad transition. From construction of p', ac is composed with a perfect detector d.

7. From step 5 and because d is perfect, it is also complete.

8. Because d is complete (step 6), d monitors ac (step 5) and transition (s, s') induced by ac is SS-inconsistent (step 4), the definition of completeness implies that $s \notin d$.

9. Step 7 implies that $(s, s') \notin \delta_{p'}$ which contradicts step 3.

$\square$

Thus, Lemma 2 shows that perfect detectors for critical actions are sufficient for design of fail-safe fault-tolerant program. Overall, composing the critical actions of a fault-intolerant program p (resulting in p') with perfect detectors ensures that (i) in the absence of faults, p and p' have identical behavior, and (ii) in presence of faults, p' is fail-safe fault-tolerant (From Lemmas 1 and 2).

Lemma 3 (Perfect Detection and Safety Specification) *Given a fault-intolerant program p with safety specification SS, which is encoded as a set of bad transitions ss, and a fault class F. Given also a program p', such that $p' = p \setminus ss_r$, where ss_r is the set of all reachable SS-inconsistent transitions using transitions in δ_p^F. Then, the following hold:*

1. *In the absence of faults, every computation of p is a computation of p'*

2. *In the absence of faults, every computation of p' is a computation of p*

3. *In the presence of faults, p' is fail-safe fault-tolerant.*

We prove the first part of the claim:

Proof.

1. From Def. 38, ss contains only SS-inconsistent transitions for p.

2. From Propositions 3, and 4, only SS-inconsistent transitions are removed from p'.

3. From step 3, no SS-consistent transition for p is removed in p'.

4. From step 3, all SS-consistent transitions of p are also transitions of p'

5. From Corollary 1 and step 4, all reachable transitions of p in absence of faults are reachable by p' in absence of faults.

6. From step 5, every computation of p is a computation of p' in absence of faults.

$\square$

The proof of the second part of the claim follows:
Proof.

1. From Def. 38, ss contains only SS-inconsistent transitions for p.

2. From Propositions 3, and 4, only SS-inconsistent transitions are removed from p'.

3. From step 2, and by construction, no transition is added in p'

4. From step 3, no transition is added in p' that is reachable in the absence of faults

5. From step 4, $\delta'_p \subseteq \delta_p$

6. From step 5, every computation of p' is a computation of p in absence of faults.

$\square$

The proof of the third claim follows: We assume that p' is not a fail-safe fault-tolerant program, and then show a contradiction.
Proof.

1. Assume p' is not a fail-safe fault-tolerant program. There exists a computation α of p' such that α violates SS in presence of faults.

2. From Prop. 2, there exists a bad transition (s, s') in α

3. From step 3, (s, s') is not removed in p'.

4. From step 3, (s, s') is reachable using transitions in δ_p^F since (s, s') is SS-inconsistent for p in presnce of faults.

5. Contradiction, since by construction of p', all SS-inconsistent transitions for p reachable by using transitions in δ_p^F have been removed.

□

Thus, from Lemma 3, $p' = p \setminus ss_r$ solves the transformation problem. Also, removing ss_r from δ_p can be likened, following Lemma 2, to composing the critical actions of p with perfect detectors.

We now present a result on the existence of perfect detectors.

Lemma 4 (Existence of perfect detectors) *Given a program p with safety specification SS, and fault class F. For every critical action ac in p, there exists a detector D such that D is perfect for ac in p.*

Proof.

1. From assumption, action ac in p is critical

2. From 1, and Definition 43, there exists a set of SS-inconsistent transitions for p $B = \{(s,t) : (s,t) \textit{is } SS\textit{-inconsistent for } p \wedge (s,t) \textit{induced by } ac\}$.

3. Let ac_r be the set of transitions induced by ac reachable in the presence of faults.

4. From steps 3, and 3, the set $O = ac_r \setminus B$ is the set of all transitions induced by ac reachable in presence of faults that will not cause violation of SS when executed.

5. From step 4, set O does not contain any reachable transition (s, t) induced by ac that is SS-inconsistent (bad) for p.

6. From step 4, set O contain all reachable transitions (s, t) induced by ac that are SS-consistent for p.

7. From steps 5, and 6, the set $OS = \{s : (s, t) \in O\}$ defines a state predicate (thus a detector) that is perfect for ac in p.

$\square$

Thus, we have shown that for every critical action ac of a program, there exists a perfect detector for ac in p. At this point, since we for every critical action of a program, there exists a perfect detector, the question is: how do we design these?

4.4.3 Constructing Perfect Detectors

Finally, we study how to construct perfect detectors for critical actions. This will also provide a basis for automated construction of such detectors.

Theorem 2 (Constructing perfect detectors) *Given a fault-intolerant program p with safety specification SS, and fault model F. The following two statements are equivalent:*

1. *The program p' is obtained by composing each critical action ac of p with a perfect detector for ac in p in presence of F.*

2. *Each SS-inconsistent transition induced by the critical action* ac *of p reachable in the presence of F is unreachable in p' in the presence of F, and each SS-consistent transition of p reachable in the presence of F is also reachable in p' in the presence of F.*

Proof.

We assume a given critical action ac of p being composed with a detector that is perfect for ac in p in presence of F, and show the implication of the second statement from the first one.

1. ac is composed with a detector d that is perfect for ac in p in presence of F.

2. Since d is perfect, it causes all SS-inconsistent transitions induced by ac to be unreachable in p' in presence of F.

3. Since d is perfect, it causes all SS-consistent transitions induced by ac to still be reachable in p' in presence of F.

4. From steps 2 and 3, we have statement 2.

   The proof for the implication of statement 1 from statement 2 is straightforward. The first part of statement 2 (unreachability of SS-inconsistent transitions induced by a critical action ac in presence of faults F) implies completeness if a detector d monitoring ac, and the second part of statement 2 (reachability of SS-consistent transitions induced by a critical action ac in presence of faults F) implies accuracy of a detector d monitoring ac. Taken together, d is perfect for ac in p in presence of F.

$\square$

The algorithm for synthesizing perfect detectors (or fail-safe fault-tolerant programs with perfect detection) is based directly on Theorem 2.

4.5 An Algorithm for Perfect Detectors

In this section, we present a sound and complete algorithm for synthesizing fail-safe fault-tolerant programs with perfect detection. Based on the fact that composing critical actions of a fault-intolerant program p with perfect detectors results in a fail-safe fault-tolerant program p' whose behavior in the absence of faults is identical to that of p.

Theorem 3 (Correctness the of transformation algorithm) *The algorithm in Figure 4.5 solves the transformation problem of Definition 48.*

Proof. Since the algorithm constructs p' by removing the set ss_r of all SS-inconsistent transitions induced by critical actions of p reachable by using transitions in δ_p^F, we can apply Lemma 3 and Theorem 2. $\square$

add-perfect-fail-safe(δ_p, δ_F, ss: set of transitions):
{
ss_r := get-ssr(δ_p, δ_F, ss)
return ($p' = p$ where transition relation is $\delta_p \setminus ss_r$)}

get-ssr(δ_p, δ_F, ss: set of transitions):
{
ss_r := $\{(s,t)|(s,t)$ is induced by a critical action of p and (s,t) is SS-inconsistent for p in presence of F $\}$

return (ss_r)}

Figure 4.5: Algorithm to synthesize fail-safe fault-tolerant program with perfect detection.

Theorem 4 (Perfect Detection) *Given a fault-intolerant program p with safety specification SS encoded as a set ss of bad transitions, and fault class F. Program p' :=add-efficient-fail-safe(p, F, ss) has perfect detection.*

Proof.

1. All $\tau \in ss_r$ are transitions induced by critical actions (fault model)
2. From step 1, removing set ss_r is equivalent to composing critical actions of p with detectors.
3. From Lemmas 1, 2, and 3, , p' has perfect detectors.
4. From steps 3, and 3, the detectors for the critical actions of p are perfect.

$\square$

We say that *p' is fail-safe fault-tolerant to F (or fail-safe F-tolerant), and has perfect detection to F.*

Theorem 5 (Soundness and Completeness) *Algorithm* add-perfect-fail-safe *is sound and complete.*

Soundness means that the resulting program solves the transformation problem, while completeness means that if the result of the corresponding decision problem is true, i.e., the fail-safe fault-tolerant program exists, then the algorithm finds it.

Proof.

The proof of soundness (from Lemma 3), and completeness (by construction and assumption) is straight forward.

$\square$

Complexity of Algorithm *add-perfect-fail-safe*

We now provide a brief analysis of the complexity of the algorithm:

1. Assume that the number of bad transitions specified by ss be m.
2. Assume that the maximum number of transitions visited to determine reachability of a bad transition is n. Then, the number of transitions visited is $O(n)$.
3. Therefore, maximum number of transitions visited when computing set ss_r is $O(m \cdot n)$.
4. Removing set ss_r has complexity $O(m)$, since the size of set ss_r is $O(m)$.
5. Overall, the algorithm in Figure 5.1 has complexity $O(m \cdot n + m) = O(m \cdot n)$, where m is the number of bad transitions specified by ss, and n is the maximum number of transitions considered to ascertain reachability.

The complexity of our algorithm is no more than the complexity of another algorithm presented by Kulkarni, and Arora [KA00], which also has polynomial complexity in the state space of the program. An instance of algorithm *add-perfect-fail-safe* was introduced by Jhumka *et al.* in [JHCS02] that generates a set of perfect detectors. In fact, Jhumka *et al.* performed a fault-injection experiment on a medium-scale embedded system for an aircraft arrestment system to ascertain the viability of the concept of perfect detectors in [JHS03]. The main finding was that perfect detectors (i) indeed detect errors that lead to violation of safety, (ii) make no detection mistakes.

In the next section, we present several case studies showing the applicability of our approach.

4.6 Three Case Studies

In this section, we present several simple examples to show how the algorithm *add-perfect-fail-safe* works.

4.6.1 A Simple Example

Figure 4.6: Example program p in the presence of faults

In Fig. 4.6, we show a program p, together with the fault transitions that can affect it. For example, transition $(1, 7)$ is a fault transition. There are two bad transitions that are specified by the safety specification SS of the program, namely transitions $(10, 11)$, and $(20, 21)$. From Lemma 3, and algorithm *add-perfect-fail-safe*, we need to remove only the set ss of bad transitions, as specified by the safety specification SS of the program, to make it fail-safe fault-tolerant. Thus, *add-perfect-fail-safe(*$\delta_p, \delta_f, \{(10, 11), (20, 21)\}$*)* will result in the removal of all the transitions in ss from the program.

This is depicted in Fig. 4.7. Observe that there can no longer exist a computation of p' in presence of faults that will include a bad transition. Thus, the resulting program p' is fail-safe fault-tolerant, and program p' solves the transformation problem.

Figure 4.7: Fail-safe fault-tolerant program p' obtained by removing ss

4.6.2 A Majority Voter System

We recall how a triple modular redundant (TMR) majority voter system works. The system consists of three inputs $in.1$, $in.2$ and $in.3$ from three processes p_1, p_2, and p_3 respectively, and an output variable, called out. For simplicity, we consider the case where each process p_i inputs a binary value $in.i$ to the voter. In the absence of faults, all three values are identical.

For the majority voter system, the class of faults that we consider is one that can corrupt the input value of at most one of the three processes. In the absence of faults, all inputs are identical and the value of the output variable out can be set to that of any $in.i$, for any process p_i.

Thus, the fault-intolerant majority voter program can be written as follows:

$$\text{ITMR1} :: out \quad = \quad ? \quad \rightarrow \quad out \quad := \quad in.1$$

Variable $out \quad = \quad ?$ means that the output has not yet been set. In the absence of faults, the value of the output variable is set to that of $in.1$.

The faults transitions F that we can consider are those transitions that corrupt the input value $in.i$ from at most one process p_i, setting it to some arbitrary value. Thus, in our example, the fault transitions considered can be represented as such:

$$F \;::\; (in.1 = in.2) \vee (in.1 = in.3) \quad \rightarrow \quad in.1 = \quad \perp$$

In presence of faults F that can corrupt the input value from process p_1, i.e., $in.1$, the *out* variable can be wrongly set, i.e., it obtains its value from a corrupted value of *in.1*. The specification of a majority voter is that it always outputs the majority of its inputs, and its safety specification is such that it never outputs a value that is not the majority, i.e., a fail-safe majority voter will never output a corrupted value (under our assumed fault model), though it may deadlock in presence of faults. However, as we explained before, a fail-safe fault-tolerant program needs to satisfy only its safety specification in presence of faults.

Thus, every transition that sets the output variable *out* to a corrupted value should be removed. Specifically, consider set
$T = \{t : ((t(out) = t(in.1)) \wedge ((t(in.1) \neq t(in.2)) \wedge (t(in.1) \neq t(in.3)))\}$,
where $s(v)$ is the value of variable v in state s, and set T represents the set of states where variable *out* is incorrectly set, i.e., variable *out* is not set to the majority value. Hence, any transition $(s, t) : t \in T$ is a bad transition for the TMR program, ITMR1, i.e., $ss = \{(s, t) : t \in T\}$.

Thus, running algorithm *add-perfect-fail-safe* results in removing set ss from program TMR. Removing set ss from TMR means that all the remaining transitions that set the output variable *out* set *out* to a majority value, whenever *out* is not already set. In other words, all the other remaining transitions that set *out* to $in.1$, they start from a state where *out* is different from $in.1$, and where $in.1$ is equal to at least one of the other input variables, i.e., $in.1 = in.2 or in.1 = in.3$. Thus, in detector terms, we need to check that *in.1* is equal to the input value of at least one of the other processes. Hence, the fail-safe fault-tolerant program majority voter, FSTMR1, is:

FSTMR1 :: $(out \neq in.1) \wedge ((in.1 = in.2) \quad \vee \quad (in.1 = in.3)) \quad \rightarrow \quad out \quad := \quad in.1$

Theorem 6 (Fail-safe TMR) *Program FSTMR1 is fail-safe fault-tolerant with perfect detection to faults that corrupt the input of at most one process.*

Observe that if faults can arbitrarily corrupt the output *out* of the TMR system, then no fail-safe fault-tolerant TMR exists, hence our focus on tolerable fault models.

4.6.3 Token Ring

In this example, we present an example of a fail-safe fault-tolerant version of the token ring. We first recall the mutual exclusion algorithm using a token ring.

Multiple processes wait to access their critical section. They can do so provided that at any one time, at most one process is accessing its critical section. This is the safety specification for a mutual exclusion algorithm. Also, no process waits indefinitely to access its critical section, assuming that each process leaves its critical section in finite "time". This is the liveness specification of a mutual exclusion protocol.

We assume a collection of processes arranged in a ring. Mutual exclusion can be achieved in such a scenario by circulating a token among these processes, and a process accesses its critical section only upon receipt of the token. The token is circulated among the processes in a particular direction. For the token ring, the safety specification is that at most one process holds the token at any one time. In this example, we present a fail-safe fault-tolerant version of a token ring, i.e., in the presence of faults, at most one process holds the token.

Processes $0 \ldots N$ are arranged in a ring. Process $k, 0 \leq k < N$ passes the token to process $k+1$, whereas process N passes the token to process 0. Each process k has a binary variable, $t.k$, and a process $k, k \neq N$ holds the token iff $t.k \neq t.(k+1)$, and process N holds the token iff $t.N = t.0$.

The fault-intolerant program for the token ring is as follows ($+_2$ is modulo-2 addition) :

$$\text{ITR1} :: \; k \neq 0 \wedge t.k \neq t.(k-1) \rightarrow t.k := t.(k-1)$$

$$\text{ITR2} :: \; k = 0 \wedge t.k \neq t.N +_2 1 \rightarrow t.k := t.N +_2 1$$

Fault action: The fault action that we consider is

$$\text{F} :: \; \mathit{true} \rightarrow t.k := \perp$$

Note: In general, we assume faults like timing faults, message loss, or duplication etc. However, we assume that when such faults occur, a process k sets its variable $t.k =\perp$. In this sense, representing the faults as F above is representative of a large class of faults. Moreover, these faults are then detectable. Also, there can be any number of state corruptions.

The safety specification of the token ring is that at any time no more than one process holds the token. In presence of faults, especially when $t.k =\perp$, no action based on $t.k$ should be taken, just in case process $k+1$ receives a duplicate token inadvertedly. Hence, all transitions of process k that occur when the state of process $(k-1), t.(k-1) =\perp$ will lead to safety violation, and should be removed. Thus, we only allow process k to execute its action when $t.(k-1) \neq\perp$.

Thus, applying *add-perfect-fail-safe* to the fault-intolerant token ring *ITR*, the resulting fail-safe fault-tolerant token ring *FSTR* is:

$$\text{FSTR1} :: \; t.(k-1) \neq\perp \wedge k \neq 0 \wedge t.k \neq t.(k-1) \rightarrow t.k := t.(k-1)$$

$$\text{FSTR2} :: \; t.N \neq\perp \wedge k = 0 \wedge t.k \neq t.N +_2 1 \rightarrow t.k := t.N +_2 1$$

Theorem 7 (Fail-safe TR) *Program FSTR is fail-safe fault-tolerant to faults that corrupt the state of any process.*

4.7 Chapter Summary

In this chapter, we have presented a novel theory of detector components for the design of fail-safe fault-tolerant programs. This theory allows to derive a transformation algorithm which automatically adds fault-tolerance ability-with perfect detection. Specifically, we have made three important contributions in this chapter: (i) We have presented a novel theory of detectors, and (ii) identified a class of detectors called perfect detectors, and then explained their role, and importance in fail-safe fault-tolerant programs, (iii) we have provided an algorithm that adds perfect detectors to a fault-intolerant program to synthesize a fail-safe fault-tolerant program.

The motivation for perfect error detection is obvious in adaptive systems. In adaptive systems, usually (in periods of non-perturbation) a fault-intolerant program p executes. During periods of perturbation, a fault-tolerant version p' of p (with possibly lower efficiency) is switched in. If a detector is not accurate, then p' may be switched in, even when there is no perturbation, lowering the efficiency of the system. If the detector is not complete, it might fail to detect an error entirely. Hence, perfect detection is necessary if the system is to be correct and efficient.

We close this chapter with a final remark concerning an observation made by Arora and Kulkarni [AK98b, Sect. 3]. The authors observed that "based on their experience", when designing fail-safe fault-tolerant programs, the detectors for non-critical actions are trivial, i.e., "true", whereas the detectors for critical actions are non-trivial. Lemma 2 is the *first* formal justification of the validity of Arora and Kulkarni's observation since it shows that it is sufficient to compose critical actions with perfect detectors to ensure fail-safe fault-tolerance. Proving this statement was made possible by our notions of accuracy and completeness of detectors. In this sense these properties can be regarded as a concretization of "non-trivial".

Chapter 5

Fast Detectors: A Basis for Fast Error Detection

In the previous chapter 4, we introduced the concept of perfect detection (perfect detectors), i.e., the ability of a detector to detect all harmful faults, and not making any mistakes. This represents one aspect of efficiency of fail-safe fault-tolerant programs.

In this chapter, we look at a second aspect of efficiency of fail-safe fault-tolerant programs, namely fast error detection. We introduce the concept of fast detection, and is based upon the concept of SS-consistency and SS-inconsistency, as defined in Chapter 4. Further, while providing for fast error detection, we endeavor to preserve the ability for perfect error detection. In this chapter, we make the following contributions:

1. We present a theory of fast and perfect detection, and formalize a metric, called *detection latency*, that can be used to (i) estimate the detection latency of a fail-safe fault-tolerant program, and (ii) compare the detection latency efficiency of different fail-safe fault-tolerant programs .

2. We present a sound, and complete algorithm that, given a fault-intolerant program p with safety specification SS and fault model F, synthesizes a fail-safe fault-tolerant program with perfect detection, and minimal latency. As way of contrast, the algorithm of chapter 4 synthesizes fail-safe fault-tolerant programs with perfect detection only.

5.1 Introduction

Given the pervasiveness of current computer systems, their ability to tolerate effect of faults is becoming increasingly important, as shown in Chapter 4. When such (harmful) faults occur, they can corrupt the state of the program. When a variable is corrupted, and its corrupted value is used to update the value of another variable, error is said to *propagate*. If no immediate action is taken, the error may propagate beyond a given boundary. When the error propagates beyond the "system" boundary, a failure is said to occur. Thus, as we showed in Chapter 4, composing critical actions with perfect detectors will prevent the errors from propagating beyond the "system" boundary. If the system/program to be designed is only needed to be fail-safe fault-tolerant, one only needs to compose critical actions with perfect detectors.

However, as we mentioned in Chapter 1, when designing a masking fault-tolerant program, designing a corresponding fail-safe fault-tolerant program can be the first step of the process. Hence, as we explained in Chapter 1 on the design of fault tolerance, once an error is detected, it needs to be corrected too. However, the greater the error propagation, the greater is the recovery process. To see this, consider the following: assume that in a given program p, the value of a variable v_1 is used to update the value of another variable v_2. Now, a fault occurs that corrupt the value of v_1, and the resulting error is detected, before v_2 is updated. During recovery, only the value of v_1 needs to be "recovered". However, if v_2 is updated with the corrupted value of v_1, then during recovery, both the value of v_1, and v_2 needs to be recovered. Hence, the recovery process is more complicated. Thus, from the point of view of recovery, the earlier the error is detected, the simpler the error recovery process. Thus, in this chapter, we present a *theory of fast error detection*.

The theory of fast detection is also based on the notion of SS-consistency and SS-inconsistency of transitions, developed in Chapter 4. Recall that a given detector monitors a certain program action. If the state of a program is such that safety can potentially be violated by execution of program actions alone (see definition of SS-inconsistent transitions – Def 37 – Chapter 4), then these program actions need to be prevented from occurring. Specifically, the program can be halted even before safety is about to be violated, through execution of bad transitions. This has the effect of preventing errors from propagating, and corrupting the whole state space of the program, i.e., the effect of the fault is contained.

Intuitively, to achieve fast detection, not only do the critical program actions need to be monitored with perfect detectors, but other non-critical program actions too (depending on the fault model) . What this means is that we may need to refine the guards of both critical and non-critical actions, depending on the fault model. As way of contrast, the theory presented in Chapter 4 refines only the guards of critical actions.

To evaluate the "fastness" at which a fail-safe fault-tolerant program detects an error, we formalize a commonly-used metric called detection latency. For a program that is fail-safe fault-tolerant, from Chapter 4, only bad transitions induced by critical actions are removed. Thus, from the onset of a "harmful" fault to the "time" the program "halts" at a critical action (i.e., the transition induced by the critical action is removed, since it is bad), the detection latency for this fault is "maximal". Whenever we say that detectors are fast, we mean that the "time" (more specifically, the number of program transitions) it takes for the program to "halt" is less than that maximum detection latency. In this chapter, we show how the detection latency can be minimized, i.e., the fault is detected in 0-step. Further, as we composed both non-critical and critical actions with detectors, we endeavor to develop perfect detectors in such cases, thus preserving the ability of having perfect detection.

This chapter is structured as follows: In Section 5.2, we present a theory of fast error detection. We define the transformation problem for fast, and perfect error detection in Section 5.3. In Section 5.4, we present an algorithm that solves the transformation problem. We present examples to illustrate the working of the algorithm in Section 5.5. We discuss some issues concerning the approach in Section 5.6, and we summarize and conclude the chapter in Section 5.7

5.2 Fast Error Detection

Perfect detectors, introduced in Chapter 4, ensure *correctness* of the fail-safe fault tolerance transformation problem. They ensure that, in the absence of faults, liveness of the resulting program is not compromised, while also ensuring that safety is not violated in presence of faults.

We now turn to a different aspect, namely the detection latency efficiency of fail-safe fault-tolerant programs. Intuitively, we would like an error to be detected as early as possible to prevent further contamination of the program state. If a fault occurs, and no precaution is taken, then the error can propagate, and corrupt the entire state space of the program. More sophisticated recovery methods, such as a distributed reset [AG94], may then be needed to get the system into a consistent state, which are computationally expensive procedures.

In this section, we focus on explaining the relationship between *fast* detection and SS-inconsistent transitions. To see fast detection, consider a computation $\alpha = s_0 \ldots s_{i-1} \cdot s_i \cdot s_{i+1} \ldots$ of a fault-intolerant program p in the presence of faults that violates safety. Assume (s_i, s_{i+1}) is a bad transition (from Proposition 2) in α. From the algorithm *add-perfect-fail-safe* of Chapter 4, transition (s_i, s_{i+1}) would be removed from p when synthesizing the fail-safe fault-tolerant program p', so that transition (s_i, s_{i+1}) is unreachable in α. However, if transition (s_{i-1}, s_i) is a program transition, and is thus SS-inconsistent for p, then removing transition (s_{i-1}, s_i) will also make transition (s_i, s_{i+1}) unreachable in α. So, from the point of view of safety, removing transition (s_{i-1}, s_i), or (s_i, s_{i+1}) achieves the same result, that of making the bad transition (s_i, s_{i+1}) unreachable. However, not executing transition (s_{i-1}, s_i) prevents error from propagating, i.e., the variables that would have been updated and corrupted had the transition (s_{i-1}, s_i) taken place are now not corrupted, hence errors are contained.

Informally, a detector d monitoring an action ac in p is perfect for ac in p if it removes every arbitrary SS-inconsistent transition induced by ac for every violating execution, while keeping all SS-consistent transitions induced by ac. Thus, given a computation α of p that violates safety, α has a sequence of program transitions leading to a bad transition (following from the definition of SS-inconsistency of Chapter 4), where every such program transition in that sequence is SS-inconsistent for p. For fast detection, i.e., to prevent error from propagating, a given program action ac of p should be composed with a perfect detector such that the "first" SS-inconsistent transition of that sequence is removed, and that transition is induced by ac. This will allow a fail-safe fault-tolerant program to have both perfect and fast error detection. On this background, we formalize the notion of an earliest SS-inconsistent transition.

Definition 44 (Earliest SS-inconsistent transition) *Given an F-intolerant program p with safety specification SS, and a computation $\alpha = s_0 \cdot s_1 \cdots s_i \cdot s_{i+1} \cdots s_m$ of p in the presence of faults that violates SS. The* transition (s_i, s_{i+1}) is the earliest SS-inconsistent transition for p w.r.t. α *iff the following two properties hold:*

1. (s_i, s_{i+1}) *is SS-inconsistent for p w.r.t. α.*
2. (s_{i-1}, s_i) *is a transition induced by a fault action.*

Intuitively, when a computation α of a program p in the presence of faults violates the safety specification *SS* of p, there exists a suffix of the violating computation prefix of α that starts with an *SS*-inconsistent transition and ends in a bad transition. The earliest *SS*-inconsistent transition is the first *SS*-inconsistent transition in this suffix. Basically, it is the first program transition that leads the program on the wrong track. Since we have no control on fault transitions, this is the first transition which can be enabled or disabled, depending on whether it is SS-consistent, or SS-inconsistent.

Define the set $EIT_p^F(SS)$ of earliest *SS*-inconsistent transitions of a program p as the union of the earliest *SS*-inconsistent transitions over all computations of p violating SS. Define $p \setminus EIT_p^F(SS)$ as the program p' which is the same as p except that all transitions from $EIT_p^F(SS)$ have been removed from δ_p.

Definition 45 (Fast detectors) *Let p be a fault-intolerant program. A* set of perfect detectors D for program p is fast *iff p composed with D results in* $p \setminus EIT_p^F(SS)$.

At this point, we need to assess the role and impact of fast (perfect) detectors in the presence of faults. We find that, composing a fault-intolerant program with these fast detectors, and also since these fast detectors are perfect, the resulting program is indeed fail-safe fault-tolerant, and this is captured by Lemma 5.

Lemma 5 (Fast perfect detectors and behavior in the presence of faults) *Let p be a fault-intolerant program for a safety specification SS. Then p composed with a set of fast perfect detectors for p satisfies SS in the presence of faults.*

Proof. This is a generalization of the proof of Lemma 2.

1. For a contradiction, it is again assumed that p' violates SS, i.e., that there exists a computation σ of p' which is not in SS.

2. From Definition 44 it is possible to generalize Proposition 3 to state that in every violating execution there exists an earliest SS-inconsistent transition in every violating computation. Denote this transition in σ as (s, s').

3. The fact that detectors are fast and from Definintion 45 we have that all earliest inconsistent transitions are removed from p while constructing p', which is a contradiction to the occurrence of (s, s') in a computation of p'.

□

We now define, and formalize a metric to measure the "fastness" of detectors. Intuitively, the detection latency metric defines the number of program transitions executed until the program "halts" at a detector, after a "harmful" fault has occurred.

Definition 46 (Detection latency) *Let SS be a safety specification and p' be a program which has been made fail-safe fault-tolerant for SS by composing a fault-intolerant program p with a set of detectors. Consider a finite computation $\alpha = s_0 \cdots s_{i-1} \cdot s_i \cdot s_{i+1} \cdots s_m$ of p' in the presence of faults, such that:*

1. *(s_{i-1}, s_i) is a transition induced by a fault action,*
2. *all transitions in $s_i \dots s_m$ are in $\delta_{p'}$, and*
3. *starting from s_m a bad transition in SS is reachable by using a sequence of program actions of p.*

Then, the detection latency $L_p(\alpha)$ of p' w.r.t. α *is the number of transitions executed in $s_i \dots s_m$, i.e., $(m - i)$ transitions.*

Intuitively, the detection latency measures the number of (SS-inconsistent) transitions executed after the occurrence of a harmful fault, and before a detector halts the program.

Definition 47 (Maximum detection latency) *Let F be a fault model, SS be a safety specification and p' be a fail-safe F-tolerant program for SS. The* maximum detection latency LM'_p of p' *is defined as the maximum of $L'_p(\alpha)$ for all computations α of p' in the presence of faults.*

Lemma 6 (Latency of fast detectors) *Given a fault-tolerant program p' which is the result of the composition of a fault-intolerant program p with a set of fast perfect detectors. Then p' has maximum detection latency* 0.

Proof. Consider any computation $\alpha = s_0 \cdots s_i \cdots s_m$ of p' which satisfies Definition 46. We need to show that $s_i = s_m$.

1. Definition 46 implies that there exists a computation σ of p which can be written as $\sigma = \alpha \cdot \beta$ (i.e., a continuation of α) which violates SS.

2. Step 1 and Definition 44 imply that (s_i, s_{i+1}) is the earliest SS-inconsistent transition of p w.r.t. σ.

3. Step 2 and the definition of fast detectors imply that p' evolved from p by removing (among other transitions) also (s_i, s_{i+1}).

4. Step 3 implies that $s_i = s_m$, which in effect means that $L_{p'}(\alpha) = 0$.

Since we have not restricted the choice of α, the statement holds for all α. This implies that $LM_{p'} = 0$. □

Since the maximum detection latency of a fail-safe fault-tolerant program p' must be at least 0, composition of a fault-intolerant program p with fast perfect detectors results in a fail-safe fault-tolerant program p' with optimal detection latency. It remains to be shown that this composition preserves the original behavior of the fault-intolerant program in the absence of faults.

Lemma 7 (Fast perfect detectors and fault-free behavior) *Given a program p' that is the composition of a fault-intolerant program p and a set of fast perfect detectors. For p and p' holds:*

1. *In the absence of faults every computation of p' is a computation of p.*
2. *In the absence of faults every computation of p is a computation of p'.*

Proof. The proof is the same as that of Lemma 1. □

Lemmas 5 (behavior in the presence of faults), 6 (optimal detection latency) and 7 (behavior in the absence of faults) taken together show that composing a fault-intolerant program with fast, perfect detectors, the resulting program (i) preserves the original behavior in the absence of faults, (ii) is fail-safe fault-tolerant in the presence of faults, and (iii) has minimal detection latency. These lemmas will form the basis for deriving the transformation algorithm for adding fast, perfect detectors in Section 5.4.

In the next section, based on the results developed in Chapter 4 and in this section, we provide an algorithm that automates the synthesis of a fail-safe fault-tolerant algorithm with perfect and fast error detection capabilities.

5.3 The Transformation Problem for Fast and Perfect Detection

We now formally state the problem of transforming a fault-intolerant program p into a fail-safe fault-tolerant version p' for a given safety specification SS and fault model F with perfect detection, and minimal detection latency.

Again, when deriving p' from p, only fault tolerance should be added, i.e., p' should not satisfy SS in new ways in the absence of faults. For completeness, we recall the main constraints defining the transformation problem:

- If there exists a transition (s, t) in p' that is not reached by p to satisfy SS, then (s, t) cannot be used by p', since this means that there are other ways p' can satisfy SS in the absence of faults. Thus, the set of transitions of p' should be a subset of the set of transitions of p.

- Also, if there exists a state s reachable by p' in the absence of faults that is not reached by p in the absence of faults, then this means that p' can satisfy SS differently from p in the absence of faults, and such a state s should not be reached by p' in the absence of faults. Thus, the set of states reachable by p' should be a subset of the set of states reachable by p.

In general, these conditions result in the requirement that both programs should have the same set of fault-free computations. Formally, we define the transformation problem as follows:

Definition 48 (Transformation for efficient fail-safe fault tolerance) *Let SS be a safety specification, a fault model F, and p an F-intolerant program for SS. Identify a program p' such that the following four conditions hold:*

1. *p' satisfies SS in presence of F.*
2. *In the absence of faults, every computation of p' is a computation of p.*
3. *In the absence of faults, every computation of p is a computation of p'.*
4. *p' has detection latency 0.*

The transformation problem can also be formulated as a decision problem:

Definition 49 (Corresponding decision problem) *Let SS be a safety specification, a fault model F, and p an F-intolerant program for SS. Does there exist a program p' such that the following three conditions hold:*

1. *p' satisfies SS in presence of F.*
2. *In the absence of faults, every computation of p' is a computation of p.*
3. *In the absence of faults, every computation of p is a computation of p'.*
4. *p' has detection latency* 0.

Later in Section 5.4 we present a sound, and complete algorithm which solves the above transformation problem, i.e., we present an algorithm that systematically transforms a fault-intolerant program into a program that satisfies the above three conditions. Soundness of the algorithm means that the resulting program indeed solves the transformation problem. Completeness of the algorithm means that if the solution to the decision problem is true, then the algorithm will find the fail-safe fault-tolerant program.

The algorithm is based on a theory for perfect detectors which we introduce in the following section. The algorithm also synthesizes fail-safe fault-tolerant program that detects faults early, and is based on a theory for fast detectors.

5.4 Adding Efficient Fail-Safe Fault Tolerance

In this section we give an algorithm to solve the transformation problem of Definition 48 which follows from the theory presented in Section 4.4.

The basic idea of the algorithm is to remove the set of earliest inconsistent transitions from the input program p. Intuitively, the algorithm works as follows: It takes as parameters the fault-intolerant program p (in the form of its transition relation δ_p) and the fault model F (in the form of the set of fault transitions). The safety specification SS is encoded as the set of bad transitions and passed to the algorithm in variable ss.

Starting from the set of bad transitions in ss, the algorithm constructs the set it of all inconsistent transitions. From this set, it constructs the set eit of earliest inconsistent transitions. This set of transitions is removed from δ_p yielding the transition relation of the transformed program. The algorithm is presented in Figure 5.1.

add-efficient-fail-safe(δ_p, δ_F, ss: set of transitions):
    $eit := get\text{-}eit(\delta_p, \delta_F, ss)$
return ($p' = p$ where transition relation is $\delta_p \setminus eit$)

get-eit(δ_p, δ_F, ss: set of transitions):
    $it := \{(s_0, s_1) \mid \exists \alpha = s_0 \cdot s_1 \cdot s_2 \cdots$ of program transitions:
        $\exists (s, s') \in ss : (s, s')$ occurs in α and (s, s') is reachable
in $\delta_p^F\}$
    $eit := \{(s_0, s_1) \mid (s_0, s_1) \in it \wedge \exists s \in S_p : (s, s_0) \in \delta_F\}$
return (eit)

Figure 5.1: Algorithm to add efficient fail-safe fault-tolerance.

Theorem 8 (Correctness of the transformation algorithm) *The algorithm in Figure 5.1 solves the transformation problem of Definition 48. Furthermore, the resulting program has minimal detection latency.*

Proof. Since the algorithm constructs p' by removing the set of all earliest inconsistent transitions, we can apply the lemmas from Section 5.2. Lemma 5 ensures that p' satisfies the specification in the presence of faults, which proves the first requirement of Definition 48. Lemma 6 ensures that the maximum detection latency is 0, meaning that it is trivially optimal. Lemma 7 ensures that p and p' have the same fault-free behavior which proves the second and third requirements of Definition 48. □

Theorem 9 (Soundness and Completeness) *Algorithm* add-efficient-fail-safe *is sound and complete.*

Proof.

The proof of soundness (from Lemma 5), and completeness (by construction) is straight forward. □

In contrast, another algorithm for automatic synthesis of fail-safe fault-tolerance proposed by Kulkarni and Arora [KA00] generates programs with detection latency equal to the maximum length over all partial computations considered when computing set *it*, since they remove only bad transitions, i.e., the last transition in the partial execution, whereas we remove the first one.

We now provide a brief analysis of the complexity of our algorithm:

1. Assume that the number of bad transitions specified by *ss* is m.
2. Let the maximum number of computations containing any bad transition is c.
3. Thus, to compute set *it*, the number of partial computations visited is $O(m \cdot c)$.
4. Assume that the maximum number of transitions visited in any partial computation when computing set *it* is n.
5. The maximum number of transitions visited when computing set *it* is $O(m \cdot c \cdot n)$.
6. Computing set *eit* means going through set *it*, thus this step has complexity $O(m \cdot c \cdot n)$.
7. Removing set *eit* has complexity $O(m \cdot c)$, since set *eit* has size $O(m \cdot c)$.
8. Overall, the algorithm in Figure 5.1 has complexity $O(m \cdot c \cdot n + m \cdot c \cdot n + m \cdot c \cdot n + m \cdot c) = O(m \cdot c \cdot n)$, where m is the number of bad transitions specified by *ss*, c is the number of maximum number of computations containing any given bad transition, and n is the maximum number of transitions considered in any partial computation.

Also, as mentioned in the Introduction (Chapter 1), our approach targets a class of programs known as bounded programs. In bounded programs, the length of the partial executions to be considered when calculating the set *it* is finite. This means that in the program, there are no infinite or unbounbed loops, rather all loops are bounded. An instance of bounded programs can usually be found in the domain of embedded applications, more specifically applications where the output is to be written within a bounded number of steps. Another instance of bounded programs are distributed algorithms.

The algorithm *add-efficient-fail-safe* was also presented in [JHCS02] to generate SS-globally consistent detectors. The algorithm basically removed all earliest SS-inconsistent transitions such that the resulting program have both perfect detection, and minimal detection latency.

We next some examples to show the working of our algorithm.

5.5 Two Case Studies

In this section, we present two examples. For the first example, we reuse the fault-intolerant program of the first example of Chapter 4, and the second example concerns a majority voter.

5.5.1 A Simple Example

In Fig. 5.2, state 1 is an initial state, and in the absence of faults, execution goes from states $1 \ldots 6$. However, in the presence of faults, other states previously unreachable become reachable, for example, state 7. Transition $(10, 11)$ is a bad transition, specified by the safety specification. Transition $(7, 8)$ is an SS-inconsistent transition.

Figure 5.2: An example to illustrate how algorithm *add-efficient-fail-safe* works

A call to *add-efficient-fail-safe(p, F, ss)* will pass the program of Fig. 5.2 as argument, with the set of fault transitions. The variable ss holds the set of bad transitions specified by the safety specification of the program, and is equal to $\{(10, 11), (20, 21)\}$. Thus,

1. The program $p = \{(1, 2), (2, 3) \ldots (7, 8), (8, 9) \ldots\}$

2. fault $F = \{(1, 7), (1, 17), (3, 9), (3, 14) \ldots\}$

3. $ss = \{(10, 11), (20, 21)\}$

From the algorithm, we collect all earliest inconsistent transitions. We start with each bad transition specified by the safety specification. For example, transition $(10, 11) \in ss$. We "backtrack" along all possible computations in presence of faults that include transition $(10, 11)$, until a fault transition occurs. The program transition that follows the fault transition is an earliest inconsistent transition. For example, going backwards, starting from transition $(10, 11)$, we reach reach fault transition $(3, 9)$, which makes transition $(9, 10)$ an earliest inconsistent transition. Similarly, transition $(7, 8)$ is an earliest inconsistent transition. Hence, $eit = \{(9, 10), (7, 8)\}$.

Likewise, starting with transition $(20, 21) \in ss$, we have the following earliest inconsistent transitions: $\{(20, 21), (20, 19), (19, 18), (18, 17)\}$. Hence, $eit = \{(9, 10), (7, 8), (20, 21), (20, 19), (19, 18), (18, 17)\}$, and we need to remove eit from the program.

The resulting program is shown in Fig. 5.3. Observe that the transitions in ss are now unreachable in the presence of faults, which makes the program fail-safe fault-tolerant. Also, as soon as a "harmful" error occurs (i.e., those that could have brought about violation of safety), the SS-inconsistent transition is disabled/removed.

In the next example, we present the case of the majority voter, developed in the last chapter.

5.5.2 A Majority Voter System

To recall the working of the triple modular redundant majority voter, there are three processes p_1, p_2, and p_3 that inputs values $in.1$, $in.2$, and $in.3$

Figure 5.3: Fail-safe fault-tolerant program resulting from applying algorithm *add-efficient-fail-safe*

respectively, and an output variable, called *out*. For simplicity, each process p_i inputs a binary value $in.i$ to the voter. In the absence of faults, all three values are identical.

The mojority voter program is written as follows:

$$\text{ITMR1} :: out \;=\; ? \quad \rightarrow \quad out \;:=\; in.1$$

The fault transitions are represented as follows:
$(in.1 = in.2) \vee (in.1 = in.3) \quad \rightarrow \quad in.1 = \;\perp$

To compute the set *eit* of earliest SS-inconsistent transitions, we start with a transition in *ss* (recall that *ss* encodes SS by holding the set of bad transitions), and "backtrack" along every computation until a fault transition is reached. To illustrate this, consider a small example of a computation α of the majority voter in presence of faults (a state of the majority voter is represented as $\langle in.1, in.2, in.3, out\rangle$:

$$\alpha = \beta \cdot \langle 1, 1, 1, -\rangle \cdot \langle \perp, 1, 1, -\rangle \cdot \langle \perp, 1, 1, \perp\rangle \cdot \gamma$$

Recall that $\perp$ is some "bad" value. Variable $out = -$ means that *out* is not set. Transition $(\langle 1, 1, 1, -\rangle \cdot \langle \perp, 1, 1, -\rangle)$ is a fault transition, and transition

$(\langle \perp, 1, 1, -\rangle \cdot \langle \perp, 1, 1, \perp\rangle)$ is a bad transition in ss, and needs to be removed. Now, when computing set eit, the transition $(\langle \perp, 1, 1, -\rangle \cdot \langle \perp, 1, 1, \perp\rangle)$ is in eit, since it is preceded by a fault transition. However, observe that every bad transition in ss will always be preceded by a fault transition, and hence $ss \subseteq eit$. Also, since there is only one action in the majority voter, $eit = ss$. Then, we need to remove eit from the intolerant program to make it fail-safe. Thus, the set of transitions that is removed is:
$\{(s, t) : t \in T\}$, where the set T is defined as
$T = \{t : ((t(out) = t(in.1)) \wedge ((t(in.1) \neq t(in.2)) \wedge (t(in.1) \neq t(in.3)))\}$, as in Chapter 4

Running algorithm *add-efficient-fail-safe* results in removing set eit from program TMR. Removing set eit from TMR will be equivalent to removing set ss from TMR. So, the fail-safe fault-tolerant version of the majority voter program, with perfect detection, and minimal detection latency is identical to the version with only perfect detection (see below).

$$\text{FSTMR1} :: (out \neq in.1) \wedge ((in.1 = in.2) \quad \vee \quad (in.1 = in.3)) \quad \rightarrow \quad out \quad := \quad in.1$$

Theorem 10 (Fail-safe TMR) *Program FSTMR1 is fail-safe fault-tolerant with perfect detection, and minimal detection latency to faults that corrupt the input of at most one process.*

Observe that if faults can arbitrarily corrupt the output out of the TMR system, then no fail-safe fault-tolerant TMR exists, hence our focus on tolerable fault models.

In the next section, we present some discussion about the applicability of the algorithm *add-efficient-fail-safe*.

5.6 Discussion

From the examples provided, we can see that the fail-safe fault-tolerant majority voter with perfect detection, and minimal latency is identical to the fail-safe fault-tolerant majority voter with perfect detection presented in Chapter 4. This can be explained by the following: Algorithm *add-perfect-fail-safe* refines the guards of critical actions, while algorithm *add-efficient-fail-safe* refines the guards of critical, as well as non-critical actions. Since the majority voter program consists of only a critical action, the fail-safe fault-tolerant majority voter resulting from both algorithms can be expected to be identical.

However, comparing the fail-safe fault-tolerant program of the first example of Chapter 4, and that of the first example of this chapter, we find that the fail-safe fault-tolerant programs are different. This is because the program has both critical and non-critical actions. Since the algorithms refine the guards of different sets of actions, the resulting fail-safe fault-tolerant programs can be expected to be different.

Thus, given that the complexity of the algorithm *add-efficient-fail-safe* is more than that of algorithm *add-perfect-fail-safe*, but that both algorithms yield the same fail-safe fault tolerance version of the majority voter with perfect detection, and minimal detection latency, then it is better to use algorithm *add-perfect-fail-safe* to generate the fail-safe fault-tolerant majority voter version.

In general, for distributed algorithms, such as mutual exclusion, token ring, agreement problems etc, most of the actions are critical. Thus, as in the case of the majority voter, applying algorithm *add-perfect-fail-safe* or algorithm *add-efficient-fail-safe* to a fault-intolerant distributed algorithm will result in identical fail-safe fault-tolerant distributed algorithm. Intuitively, any computation of a distributed algorithm in the presence of faults consists of critical transitions and fault transitions. So, every program transition which is an earliest SS-inconsistent transition is also a bad transition, and vice-versa. Thus, removing set eit will always have result identical to when removing set ss from the fault-intolerant program of a distributed algorithm. Hence, for distributed algorithms, it is preferable to use algorithm *add-perfect-fail-safe*, since they will always have perfect detection, and minimal detection latency, and the algorithm have lower complexity.

However, algorithm *add-efficient-fail-safe* can be used to yield fail-safe fault-tolerant programs with perfect detection, and minimal detection latency for a wide class of programs which consists of both critical and non-critical actions. Embedded programs typically consist of both critical and non-critical actions, as examplified in the first example in this chapter. Then, the set *eit* needs not be equal to the set *ss* of bad transitions, as in the case of distributed algorithms.

5.7 Chapter Summary

In this chapter, we have presented a novel theory of fast detector components for the design of efficient fail-safe fault-tolerant programs. This theory builds upon the theory of perfect detectors, and allows the derivation of a transformation algorithm which automatically adds fault tolerance abilities with perfect detection and minimal detection latency to an initially fault-intolerant program. Specifically, we have made two important contributions in this chapter: (i) We have presented a theory of fast detectors, that ensures perfect detection, and minimal detection latency of fail-safe fault-tolerant program and (ii) we have developed an algorithm that adds fast, and perfect detectors to a fault-intolerant program to synthesize an efficient fail-safe fault-tolerant program. We have shown that the complexity of the algorithm is polynomial in the state space of the fault-intolerant program.

We have also shown that algorithm *add-efficient-fail-safe* is particularly suitable for a class of programs that consist of both critical and non-critical actions, while algorithm *add-perfect-fail-safe* is particularly suitable for distributed algorithms.

The motivation of minimal detection latency is for fault containment. The earlier an error is detected, the higher is the error containment. If an error is not contained, more sophisticated error recovery mechanisms may be required to correct the fault than if the error is contained. Specifically, if an error is contained, a local recovery procedure may be initiated, but if the error is not contained and the state of several processes is corrupted, local recovery mechanisms may not be adequate.

Chapter 6

Design of Efficient Multitolerance

In Chapters 4 and 5, we introduced the concept of perfect, and fast detectors respectively. We provided algorithms that, given a fault-intolerant program p, a fault model F, and a safety specification SS for p, synthesize (i) fail-safe fault-tolerant programs with perfect detection, and (ii) fail-safe fault-tolerant programs with perfect detection, and minimal detection latency. In those cases, we have considered only one given fault class, i.e., we have synthesized efficient fail-safe fault tolerance to a given fault class.

However, in a distributed setting, the nature, and type of faults occurring is varied. For example, faults can lead to message loss, corruption of program state, processor crash and so on. Thus, faults that can affect a given program can come from different sources (called fault classes), meaning that the program should be (fail-safe) fault-tolerant to these different fault classes, i.e., the program should be (fail-safe) multitolerant. This points to a methodology that can support systematic addition of such efficient multitolerance, i.e., we aim to generalize the results of Chapters 4 (addition of perfect detection) and 5 (addition of perfect and fast detection) to deal with multiple fault classes.

In this chapter, we consider two different approaches for automated synthesis of efficient fail-safe multitolerant programs. The first design approach for addition of multitolerance that we consider handles one fault class at a time, where efficient fail-safe fault tolerance to different fault classes is added in a stepwise (compositional) fashion, i.e., efficient fault tolerance is added to one given fault class, before another fault class is considered. Then, we consider a second design approach that, on the other hand, considers all fault classes at the same time.

For each design approach, we provide two algorithms for the addition of efficient multitolerance, and each algorithm (for each approach) adds some efficiency properties to the resulting multitolerant program with respect to each fault class considered. Specifically, starting from a fault-intolerant program, and the different fault classes to be tolerated, we present algorithms (for each design approach) that (i) add perfect fail-safe multitolerance to every fault class considered, and (ii) add fail-safe multitolerance with both perfect detection and minimal detection latency to every fault class considered. We show that the corresponding algorithms from each design approach yield identical fail-safe multitolerant programs. We exploit this relation to prove properties of programs generated using the second approach

The properties of the fail-safe multitolerant programs resulting from either approach are: either (i) they have minimal detection latency, and perfect detection to each fault class, or (ii) to each fault class, the fail-safe fault-tolerant programs have perfect detection. By way of contrast, Arora and Kulkarni observed in [AK98a, Kul99] that using a method that considers one fault class at a time may not yield programs that are optimal (in some sense) with respect to all fault classes, whereas the method that considers all fault models at the same time may. Here, we show that both approaches yield programs that are efficient (with respect to fault detection, and detection latency) to all fault classes. In effect, we have identified a class of multitolerant programs (i.e., fail-safe multitolerant programs) and classes of efficiency properties (i.e., perfect detection, and minimal detection latency) for which these efficiency properties can be effectively designed for each fault class considered during the design of such multitolerant programs.

The first design approach can be used when fail-safe fault tolerance to new fault classes needs to be added to a given program, whilst the second approach can be used whenever some given fault classes are re-defined.

6.1 Introduction

In this chapter, we consider the design of (efficient) multitolerance, i.e., the ability of a program to tolerate multiple classes of faults. Specifically, we restrict our attention to adding *fail-safe* fault tolerance to multiple fault classes to a given fault-intolerant program. We recall that a fault-intolerant program is one that satisfies its specification in the absence of faults, but violates it in the presence of faults. Specifically, as mentioned before, a specification is composed of two parts, namely (i) a safety specification, and (ii) a liveness specification, as indicated by Alpern and Schneider [AS85], and a fail-safe fault-tolerant program satisfies at least its safety specification in presence of faults.

In a distributed setting, the nature of faults arising is varied. For example, faults may corrupt input variables, corrupt the interfaces of processes, cause loss of messages, or processor crashes, among others. Thus, when designing a (fail-safe) fault-tolerant program, the design should be cognizant of those varied fault classes. To make a program fail-safe fault-tolerant to a given fault class, (a set of) detectors are added that handle faults from that given fault class. Thus, to transform a fault-intolerant program into a fail-safe multitolerant one, a set of detectors is added to the fault-intolerant program, where each detector handles faults from a particular fault class.

One obvious difficulty that needs to be handled, as observed by Arora and Kulkarni in [AK98a], is the fact that detectors handling different fault classes may interfere either with each other or with the program. Intuitively, this means that every computation of a given program component, in the presence of other program components, is still in its specification. For example, one condition that needs to be verified is that the new components introduced (such as detectors) should not interfere with the behavior of the original program. Thus, non-interference between different program components should be verified.

However, given our focus on efficiency properties of the fail-safe fault-tolerant programs, such as perfect detection, and minimal detection latency, the above verification conditions may not suffice. To see this, consider the following: Though addition of two detectors may not cause interference, one may cause the resulting fail-safe fault-tolerant program to lose one of its efficiency properties, either perfect detection or minimal detection latency or both. Thus, the verification conditions need to be extended to deal with those efficiency properties.

Therefore, the difficulty to be handled when adding detectors to a program p (resulting in program p') for a new fault class F_n, in addition to ascertaining non-interference across different program components, is to ensure that the new detectors included for the new fault class F_n (i) p' still has efficiency properties with respect to other fault classes, and (ii) p' has efficiency properties with respect to F_n, i.e., the resulting fail-safe multitolerant program p' has efficiency properties to all fault classes considered. In other words, when new detector components are added to a given program for a new fault class, the verification conditions are (i) there is no interference among the different program components, (ii) the new components preserve and extend the efficiency properties of the original program with respect to other fault classes. Thus, non-interference properties should include both behavioral and efficiency aspects.

There are two possible approaches for the design of multitolerance:

1. The first approach considers one fault model at a time, and
2. The second approach considers all fault models at the same time.

For the first approach, we present two algorithms, one that automatically yields fail-safe multitolerant programs, with perfect detection to all fault classes considered, and another that yields fail-safe multitolerant programs with perfect detection, and minimal detection latency to all fault classes. By way of contrast, Arora and Kulkarni argued in [AK98a] that programs designed using this appraoch can have complexity (in some sense) which is efficient for some, but not all, fault classes.

For the second approach, we present two algorithms that consider all the fault classes at the same time. The first algorithm yields a fail-safe multitolerant program with perfect detection to all fault classes considered, while the second algorithm yields fail-safe multitolerant programs with perfect detection, and minimal detection latency to all fault classes considered. We also show that the resulting fail-safe multitolerant programs obtained from the corresponding algorithms, e.g., those that add fail-safe fault tolerance with perfect detection, from each design approach are identical. For example, the fail-safe multitolerant program obtained from the algorithm that adds multitolerance with perfect detection to a program to every fault class according to the first approach is identical to the fail-safe multitolerant program obtained from the algorithm that adds multitolerance with perfect detection to every fault class according to the second approach.

Thus, the contributions in this chapter are:

1. We present non-interference conditions (behavioral and efficiency) to be verified during the design of multitolerance.

2. We present an automated approach for design of efficient fail-safe multitolerant programs by considering one fault class at a time.

3. We present an automated approach for designing efficient fail-safe multitolerant programs by considering all fault classes at the same time.

4. We also show that the programs obtained by corresponding algorithms of either approaches are identical, and that they have some efficiency properties for all fault classes.

This chapter is structured as follows: In Section 6.2, we present the non-interference conditions that have to be verified during the addition of multitolerance. In Section 6.3, we present the stepwise approach (one fault class at a time) for automatic addition of multitolerance, and provide two algorithms that yields fail-safe multitolerant programs. We present two other algorithms that handle all the fault classes at the same time in Section 6.4. We discuss and summarize the results presented in this chapter in Section 6.5.

6.2 Issues in Multitolerance Design

In this section, we present and discuss the non-interference issues involved in the design of efficient multitolerance, i.e., fail-safe fault tolerance to multiple fault classes with perfect detection, and minimal detection latency to all fault classes.

First, we define a fail-safe multitolerant program:

Definition 50 (Fail-Safe Multitolerant Program) *Given a program p with specification S, and safety specification SS, and n fault classes $F_1 \dots F_n$. A program p is said to be* fail-safe multitolerant to fault classes $F_1 \dots F_n$ *iff p is fail-safe F_i-tolerant for each $1 \leq i \leq n$.*

As mentioned in the introduction of this chapter, there are two possible approaches for the design of multitolerant program. The issues and discussions presented in this section mostly apply to a stepwise approach that considers one fault class at a time, in some fixed order $F_1 \dots F_n$ in which a fault-intolerant program p is transformed into a fail-safe multitolerant program to fault classes $F_1 \dots F_n$. In general, in the first step, the fault-intolerant program p is augmented with detectors that will make it fail-safe F_1-tolerant, i.e., the resulting program p_1 is fail-safe F_1-tolerant. Then, in the second step, the resulting program p_1 is augmented with detectors that will make it fail-safe F_2-tolerant, while preserving its fail-safe F_1-tolerance. The same is repeated, until the n^{th} step, where the program is augmented with detectors that will provide fail-safe F_n-tolerance, while preserving fail-safe fault-tolerance to $F_1 \dots F_{n-1}$. However, because of our focus on perfect detection, and minimal detection latency, these steps need to be extended to deal with those efficiency requirements.

The steps are extended as follows, below:

In the first step, when the fault-intolerant program p is augmented with detectors that will make it a fail-safe F_1-tolerant program p_1, p_1 should have perfect detection and/or minimal detection latency to F_1. The following non-interference conditions need to be verified:

1. In the absence of F_1, the detector components added to p do not interfere with p, i.e., each computation of p is in the problem specification even if it executes concurrently with the new detector components.

2. In the presence of faults F_1, each computation of the detector components is in the components' specification even if they execute concurrently with p.

3. In the presence of faults F_1, the detector components added to p provide perfect detection and/or minimal detection latency to F_1.

Note: In this section, whenever it is clear from the context, we will use the term "fail-safe fault-tolerant program (fail-safe fault tolerance)" to mean "fail-safe fault-tolerant program (fail-safe fault tolerance) with perfect detection, and/or minimal detection latency".

In the second step, when the fail-safe F_1-tolerant program p_1 is augmented with detectors that will make p_1 fail-safe fault-tolerant to F_2 (i.e., transform it into a program p_2), the following non-interference conditions need to be satisfied:

1. In the absence of F_1 and F_2, the new detectors for fail-safe fault tolerance to F_2 do not interfere with p_1, i.e., each computation of p_1 satisfies the problem specification even if p_1 executes concurrently with the new detectors.

2. In the presence of F_1, the new detectors for fail-safe fault tolerance to F_2 do not interfere with the fail-safe fault tolerance to F_1 of p_1, i.e., every computation of p_1 is in the fail-safe fault tolerance specification to F_1 even if p_1 executes concurrently with the new components.

3. In the presence of F_1, the new detectors for fail-safe fault tolerance to F_2 do not interfere with the perfect detection and/or minimal detection latency to F_1 of p_1.

4. In the presence of F_2, p_1 does not interfere with the new detectors that provide fail-safe fault-tolerance to F_2.

5. In the presence of F_2, p_1 does not interfere with the new detector components providing perfect detection, and/or minimal detection latency to F_2

In the i^{th} step, when the fail-safe F_{i-1}-tolerant program p_{i-1} is augmented with detectors that will transform it into a fail-safe F_i-tolerant program p_i (i.e, p_i is fail-safe fault-tolerant to fault classes $F_1 \ldots F_i$), the following non-interference conditions need to be satisfied:

1. In the absence of faults $F_1 \ldots F_i$, the new detectors for fail-safe fault tolerance to F_i do not interfere with p_{i-1}, i.e., each computation of p_{i-1} satisfies the problem specification even if p_{i-1} executes concurrently with the new detector components for fail-safe fault tolerance to F_i.

2. In the presence of F_1, the new detectors for fail-safe fault tolerance to F_i do not interfere with the fail-safe fault-tolerance to F_1 of p_{i-1}, i.e., every computation of p_{i-1} is in the fail-safe fault tolerance specification to F_1 even if p_{i-1} executes concurrently with the new components.

3. In the presence of F_1, the new detectors for fail-safe fault tolerance to F_2 do not interfere with the perfect detection, and/or minimal detection latency to F_1 of p_{i-1}.

4. $\vdots$

5. In the presence of faults F_i, p_{i-1} does not interfere with the new detector components that provide fail-safe fault tolerance to F_i, i.e., each computation of the detector components for fail-safe fault tolerance to F_i is in the components' specification.

6. In the presence of F_i, p_{i-1} does not interfere with the new detector components that provide perfect detection, and/or minimal detection latency to F_i.

Automated procedures that add fail-safe multitolerance to a previously fault-intolerant program need to guarantee that these conditions are met by design.

In the next section, we consider a design approach for addition of multitolerance that handles one given fault class at a time, and we then present two algorithms that automatically yield fail-safe multitolerant programs with differing efficiencies. The first algorithm, presented in the first part, yields fail-safe multitolerant programs with perfect detection to fault classes $F_1 \dots F_n$, while the second algorithm, presented in the second part, yields fail-safe multitolerant programs with perfect detection, and minimal detection latency to fault classes $F_1 \dots F_n$, by considering one fault class at a time.

Before presenting the algorithms that automatically add multitolerance, we adopt a step-by-step derivation of the algorithm, and each step of the algorithm is shown to guarantee that the non-interference conditions stated are met by design.

6.3 One-at-a-time Design of Multitolerance

In deriving both algorithms, we focus on the case of two fault classes, and the approach can be easily generalized to n fault classes.

6.3.1 Multitolerant Programs With Perfect Detection

Given a fault-intolerant program p with safety specification SS, and n fault classes $F_1 \dots F_n$ which have to be tolerated, the idea is to transform p into a program p_n that is fail-safe fault-tolerant to $F_1 \dots F_n$ with perfect detection for each fault class. To do this, we first consider fault class F_1, then F_2 until fault class F_n is handled. In this section, whenever it is clear from the context, we will use the term "fail-safe fault-tolerant program (fail-safe fault tolerance)" to mean "fail-safe fault-tolerant program (fail-safe fault tolerance) with perfect detection".

Before explaining and introducing our automated approach for addition of fail-safe multitolerance, we present a result upon which our approach is based. Intuitively, the result states that, starting with a program p_i that is fail-safe fault-tolerant to fault classes $F_1 \dots F_i$ with perfect detection to each of these fault classes, composing p_i with a perfect detector for fault class F_{i+1} such that the resulting program p_{i+1} is fail-safe fault-tolerant to F_{i+1} with perfect detection, then p_{i+1} also preserves the efficiency properties of p_i with respect to $F_1 \ldots F_i$. Said otherwise, composing a program p_i as above with perfect detectors for a new fault class F_{i+1} satisfy the verification conditions presented in Section 6.2.

Lemma 8 (Perfect detectors and multitolerance) *Given a fault-intolerant program p with safety specification SS. Given a program p_{i-1} which is fail-safe multitolerant for SS with perfect detection to fault classes $F_1 \ldots F_{i-1}$. Given also a program p_i obtained from p_{i-1} by composing critical actions of p_{i-1} with perfect detectors, such that p_i is fail-safe fault-tolerant to fault class F_i with perfect detection. Then, p_i is also fail-safe multitolerant with perfect detection to fault classes $F_1 \ldots F_{i-1}$.*

Proof. *Assume:* (i) A fault-intolerant program $p_0 = p$ with safety specification SS, (ii) Program p_{i-1} is fail-safe multitolerant for SS with perfect detection to fault classes $F_1 \ldots F_{i-1}$, (iii) a new fault class F_i which needs to be tolerated, (iv) program p_i that is fail-safe fault-tolerant for SS with perfect detection to F_i obtained by composing some critical actions of p_{i-1} with perfect detectors for F_i.

Prove: Program p_i is fail-safe multitolerant for SS with perfect detection to fault classes $F_1 \ldots F_{i-1}$.

1. From assumption, $p_{i-1} = p_0 \setminus B_1^{i-1}$, where $B_1^{i-1} \subseteq ss$ and ss is the set of bad transitions.

2. From assumption, $p_i = p_{i-1}[]_c PD_i$, where $[]_c$ means composing critical actions, and PD_i meaning perfect detectors that will tolerate fault class F_i.

3. From 3, $p_i = p_{i-1} \setminus B_i$, where $B_i = \{(s,t) : (s,t)$ is induced by a critical action $\wedge (s,t)$ is reachable in presence of $F_i \wedge (s,t) \in ss\}$.

4. From 3, since no SS-inconsistent transition is added to p_{i-1}, p_i has complete detection to fault classes $F_1 \ldots F_{i-1}$.

5. From 3, since no SS-consistent transition is removed from p_{i-1}, p_i has accurate detection to fault classes $F_1 \ldots F_{i-1}$.

6. From 4, and 5, p_i has perfect detection to fault classes $F_1 \ldots F_{i-1}$.

□

The above lemma shows that by composing critical actions with perfect detectors that makes a given program fail-safe fault-tolerant to a new fault class, fail-safe fault tolerance with perfect detection to previous fault classes is preserved, i.e., there is no interference between the new detector components with detector components for previous fault classes either at the behavioral level or at the efficiency level. This result allows us to reuse algorithm *add-perfect-fail-safe* (see Chapter 4), since *add-perfect-fail-safe* generates perfect detectors for a given fault class.

Step 1 in Multitolerance Design

To synthesize a program that is fail-safe fault-tolerant to F_1, starting from a fault-intolerant program p, with perfect detection to F_1, we first need to compute the set ss_1 of bad transitions for p reachable in the presence of faults F_1 (reachable by using transitions in $\delta_p^{F_1}$)(from Lemmas 2, 3). We then remove those transitions from program p, to obtain a program p_1, which is fail-safe F_1-tolerant, with perfect detection for F_1, as shown in Fig 6.1, where ss is the set of bad transitions that the safety specification SS of p rejects.

$p_1 :=$ add-perfect-fail-safe(p, F_1, ss)

Figure 6.1: The first step in the design of multitolerant programs with perfect detection.

At this point, we need to verify the non-interference conditions for the first step of the transformation.

First, by construction, p_1 has perfect detection to F_1.

Second, we need to prove that "*In the absence of F_1, the detector components added to p do not interfere with p, i.e., each computation of p is in the problem specification even if it executes concurrently with the new detector components*".

Proof. By construction (using algorithm *add-efficient-fail-safe*, the detector components for fail-safe fault tolerance to F_1 do not interfere with p $\square$

Third, we need to prove that "*In the presence of faults F_1, each computation of the detector components is in the components' specification even if they execute concurrently with p, i.e., p does not interfere with the new detector components.*".

Proof. By construction, p does not interfere with the detector components for fail-safe fault tolerance to F_1. $\square$

Step 2 in Multitolerance Design

In the second step of the multitolerance addition procedure, we consider fault class F_2, and we transform program p_1 (which is fail-safe fault-tolerant to F_1) into a program $p_2$that is fail-safe fault-tolerant to F_2, while preserving the existing fail-safe fault tolerance to F_1. Specifically, we compute the set ss_2 of bad transitions that are reachable in presence of faults F_2, and we remove those transitions from program p_1 to obtain program p_2, which is fail-safe fault-tolerant, with perfect detection to both F_1, and F_2, as shown in Fig 6.2.

$$p_2 := \text{ add-perfect-fail-safe}(p_1, F_2, ss)$$

Figure 6.2: The second step in the design of multitolerant programs with perfect detection.

By construction, p_2 has perfect detection to F_2, i.e., p_1 does not interfere with the new detector components that provide perfect detection to fault class F_2.

To verify the other non-interference properties, we first need to prove that "*In the absence of F_1 and F_2, the detector components added to p_1 for fail-safe fault tolerance to F_2 do not interfere with p_1, i.e., each computation of p_1 is in the problem specification even if it executes concurrently with the new detector components*".

Proof. By construction, the new detector components for fail-safe fault tolerance to F_2 do not interfere with p_1. $\square$

We now prove the second part of the non-interference conditions, which is "*In the presence of F_1, the new detectors for fail-safe fault tolerance to F_2 do not interfere with the fail-safe F_1-tolerance of p_1, i.e., every computation of p_1 is in the fail-safe F_1-tolerance specification even if p_1 executes concurrently with the new components.*"

Proof. We prove this by contradiction. We first assume that there exists a computation in presence of F_1 that violates safety, and show that such a computation cannot exist, i.e., a contradiction.

1. Given p_2 = add-efficient-fail-safe(p_1, F_2, ss)

2. Assume that there is a computation α in presence of F_1 that violates safety

3. From step 3 and Proposition 2, α contains a bad transition τ that is reachable in presence of F_1.

4. By construction of p_1, $\tau \notin \delta_{p_1}$

5. By construction of p_2, transition τ is not added to δ_{p_1}.

6. From steps 3, 4, and 5, we have a contradiction.

$\square$

We now prove that "*In the presence of F_1, the detector components for fail-safe fault tolerance to F_2 do not interfere with the perfect detection of p_1 to F_1.*"

Proof.

1. From the fact that the new detector components do not interfere with the fail-safe F_1-tolerance of p_1 in the presence of F_1, we deduce that the detector components for F_1 are complete.

2. Since only SS-inconsistent transitions are removed, the detector components for F_1 are accurate.

3. From steps 1, and 3, the detector components for F_1 are perfect.

$\square$

The proof of the last part, which is "*In the presence of F_2, p_1 does not interfere with the new detectors that provide fail-safe fault tolerance to F_2.*"

Proof. By construction, p_1 does not interfere with the new detector components for fail-safe fault tolerance to F_2. $\square$

Step K in Multitolerance Design

In the k^{th} step $(3 \leq k \leq n)$, we consider fault class F_k, and we transform a fail-safe $F_1 \dots F_{k-1}$-tolerant program p_{k-1}, i.e., p_{k-1} is fail-safe fault-tolerant to fault classes $F_1 \dots F_{k-1}$, into program p_k that is fail-safe fault-tolerant to F_k, while also preserving the fail-safe fault tolerance to $F_1 \dots F_{k-1}$ i.e., program p_k is fail-safe $F_1 \dots F_k$-tolerant. To do this, we compute the set ss_k of bad transitions that are reachable in presence of faults F_k, and we remove those transitions from program p_{k-1} to obtain program p_k which have fail-safe fault tolerance with perfect detection to fault classes $F_1 \dots F_k$. The k^{th} step is shown in Fig 6.3.

$$p_k := \text{ add-perfect-fail-safe}(p_{k-1}, F_k, ss)$$

Figure 6.3: The k^{th} step in the design of multitolerant programs with perfect detection.

With this step, we need to verify that non-interference conditions are guaranteed.

First, p_k has perfect detection to F_k by construction, i.e., program p_{k-1} does not interfere with the perfect detection of the new detector components for fault class F_2.

To verify the other non-interference properties, we prove that "*In the absence of $F_1 \dots F_k$, the detector components added to p_{k-1} for fail-safe fault tolerance to F_k do not interfere with p_{k-1}, i.e., each computation of p_{k-1} is in the problem specification even if it executes concurrently with the new detector components*".

Proof. By construction, the new detector components for fail-safe fault tolerance to F_k do not interfere with p_{k-1}. □

We now prove the i^{th} part ($2 \leq i < k$) of the non-interference conditions, which is "*In the presence of F_i, the new detectors for fail-safe fault tolerance to F_k do not interfere with the fail-safe fault tolerance to F_i of p_{k-1}, i.e., every computation of p_{k-1} is in the fail-safe fault tolerance specification to F_i even if p_{k-1} executes concurrently with the new detector components.*"

Proof. We prove this by contradiction. We first assume that there exists a computation in presence of F_i that violates safety, and show that such a computation cannot exist, i.e., a contradiction.

1. Given p_k = add-efficient-fail-safe(p_{k-1}, F_k, ss)
2. Assume that there is a computation α in presence of F_i that violates safety
3. From step 3 and Proposition. 2, α contains a bad transition τ that is reachable in presence of F_i.
4. By construction of p_{k-1}, $\tau \notin \delta_{p_{k-1}}$
5. By construction of p_k, τ is not added to $\delta_{p_{k-1}}$
6. From steps 3, 4 and 5, we have a contradiction.

□

We now prove that "*In the presence of F_i, the new detector components for fail-safe fault tolerance to F_k do not interfere with the perfect detection of p_{k-1} to F_i.*"

Proof.

1. From the fact that the new detector components do not interfere with the fail-safe F_i-tolerance of p_{i-1} in the presence of F_i, we deduce that the detector components for F_i in p_{i-1} are complete.
2. Since only SS-inconsistent transitions are removed, the detector components for F_i in p_{i-1} are accurate.

3. From steps 1, and 3, the detector components for F_i are perfect.

$\square$

The proof of the last part, which is "*In the presence of F_k, p_{k-1} does not interfere with the new detectors that provide fail-safe fault tolerance to F_k.*"

Proof. By construction, p_{k-1} does not interfere with the new detector components for fail-safe fault tolerance to F_k. $\square$

Observe that, in general, because (i) the new detector components that provide fail-safe fault tolerance to F_k do not interfere with the fail-safe fault tolerance of p_{k-1} to all fault classes F_i $(1 \leq i < k)$, and (ii) only bad transitions are removed from p_{k-1}, the perfect detection to all fault classes F_i is preserved.

In general, the algorithm for automatic synthesis of fail-safe multitolerant programs with perfect detection to all fault classes is shown in Fig. 6.4

```
add-perfect-fail-safe-multitolerance(p, [F1 ... Fn], ss: set of transitions):

        {i := 1; p0 := p
        while (i ≤ n) do {
            pi := add-perfect-fail-safe(p(i-1), Fi, ss);
            i := i + 1;} od
        return(pn)}
```

Figure 6.4: The algorithm adds fail-safe fault tolerance to n fault classes, with perfect detection to every fault class

Theorem 11 (Multitolerance with perfect detection) *Given a fault-intolerant program p with safety specification SS, and n fault classes $F_1 \ldots F_n$. Algorithm* add-perfect-fail-safe-multitolerance$(p, [F_1 \ldots F_n], ss)$ *returns a program that is fail-safe fault-tolerant to $F_1 \ldots F_n$, with perfect detection to all the fault classes.*

In this section, we have presented a stepwise approach for the automatic design of multitolerance. We have proved that every step of the algorithm guarantees that there is no interference between the new detector components and those existing fail-safe fault tolerance mechanisms for other fault classes, as well as no interference with their perfect detection to those fault classes. In the next two sections, we will present examples to show the working of the algorithm.

6.3.2 A Simple Example

One-at-a-Time Addition of Perfect Fail-Safe Fault Tolerance

In this section, we present a small example to illustrate how algorithm *add-perfect-fail-safe-multitolerance* works. Fig 6.5 shows the fault-intolerant program in presence of faults F_1. In this example, transitions $(10, 11)$ and $(20, 21)$ are bad transitions.

Figure 6.5: Fault-intolerant program in the presence of F_1 – first iteration of the algorithm

During the first iteration through the algorithm, both bad transitions are removed since both are reachable in presence of F_1 (through the call to *add-perfect-fail-safe*). The resulting fail-safe fault-tolerant program with perfect detection to F_1 is shown in Fig. 6.6. Denote it by p_1.

Then, for the second iteration through *add-perfect-fail-safe-multitolerance*, we need to add perfect fail-safe fault tolerance to F_2 to p_1, while preserving the fail-safe fault tolerance with perfect detection of

Figure 6.6: Resulting fail-safe fault-tolerant program p_1 to F_1

p_1 to F_1. First, we consider p_1 in presence of F_2, as shown in Fig. 6.7

Figure 6.7: Resulting fail-safe fault-tolerant program p_1 in presence of F_2

In the presence of F_2, no bad transition is reachable, since all of them has been removed during the previous pass. So, the program is also fail-safe fault-tolerant to F_2, while maintaining the fail-safe fault-tolerance to F_1. The resulting program (p_2) is shown in Fig. 6.8. Observe that in presence of F_1 or F_2, p_2 will never violate the safety specification.

6.3.3 Token Ring

The token ring was described in Chapter 4. Processes $0 \ldots N$ are arranged in a ring. Process $k, 0 \leq k < N$ passes the token to process $k+1$, whereas

Figure 6.8: Resulting fail-safe multitolerant program p_2 to F_1 and F_2 with perfect detection to both fault classes.

process N passes the token to process 0. Each process k has a binary variable, $t.k$, and a process $k, k \neq N$ holds the token iff $t.k \neq t.(k+1)$, and process N holds the token iff $t.N = t.0$.

The fault-intolerant program for the token ring is as follows ($+_2$ is modulo-2 addition) :

$$\text{ITR1} :: \; k \neq 0 \wedge t.k \neq t.(k-1) \rightarrow t.k := t.(k-1)$$

$$\text{ITR2} :: \; k = 0 \wedge t.k \neq t.N +_2 1 \rightarrow t.k := t.N +_2 1$$

Fail-Safe Fault Tolerance to Fault Class F_1: First, we consider a fault class where fault actions can corrupt the state of a single process k, which can be any process.

Fault action: The fault class that we consider is

$$\text{F} :: \; |\{k : t.k = \perp\}| = 0 \rightarrow t.k := \; \perp$$

Running algorithm *add-perfect-fail-safe-multitolerance* will result in the following program after the first iteration

$$\text{1-FSTR1} :: |\{k : t.k = \perp\}| = 1 \wedge t.(k-1) \neq \perp \wedge k \neq 0 \wedge t.k \neq t.(k-1) \rightarrow t.k := t.(k-1)$$

$$\text{1-FSTR2} :: |\{k : t.k = \perp\}| = 1 \wedge t.N \neq \perp \wedge k = 0 \wedge t.k \neq t.N +_2 1 \rightarrow t.k := t.N +_2 1$$

Theorem 12 (Fail-safe TR) *Program 1-FSTR is fail-safe fault-tolerant to faults that corrupt the state of a single process k, which can be any process.*

Fail-Safe Fault Tolerance to Fault Class F_2: Second, we consider a fault class where fault actions can corrupt the state of any two processes k and l.

Fault action: The fault class that we consider is

$$\text{F} :: |\{k : t.k = \perp\}| = 1 \wedge t.k \neq \perp \rightarrow t.k := \perp$$

The second iteration of algorithm *add-perfect-fail-safe-multitolerance* will result in the following program:

$$\text{2-FSTR1} :: |\{k : t.k = \perp\}| \leq 2 \wedge t.(k-1) \neq \perp \wedge k \neq 0 \wedge t.k \neq t.(k-1) \rightarrow t.k := t.(k-1)$$

$$\text{2-FSTR2} :: |\{k : t.k = \perp\}| \leq 2 \wedge t.N \neq \perp \wedge k = 0 \wedge t.k \neq t.N +_2 1 \rightarrow t.k := t.N +_2 1$$

Theorem 13 (Fail-safe TR) *Program 2-FSTR is fail-safe fault-tolerant to faults that corrupt the state of at most two processes k and l, which can be any process.*

Fail-Safe Fault Tolerance to Fault Class F_{N+1}: Finally, we consider a fault class where fault actions can corrupt the state of n $(3 \leq n \leq N+1)$ processes.

Fault action: The fault action that we consider is

$$\text{F} :: |\{k : t.k =\perp\}| = n-1 \wedge t.k \neq\perp \rightarrow t.k := \perp$$

The n^{th} iteration of algorithm *add-perfect-fail-safe-multitolerance* will result in the following program:

$$n\text{-FSTR1} :: |\{k : t.k =\perp\}| \leq n \wedge t.(k-1) \neq\perp \wedge k \neq 0 \wedge t.k \neq t.(k-1) \rightarrow t.k := t.(k-1)$$

$$n\text{-FSTR2} :: |\{k : t.k =\perp\}| \leq n \wedge t.N \neq\perp \wedge k = 0 \wedge t.k \neq t.N +_2 1 \rightarrow t.k := t.N +_2 1$$

Theorem 14 (Fail-safe TR) *Program n-FSTR is fail-safe fault-tolerant to faults that can corrupt the state of any number of processes, upto all n processes.*

From program n-FSTR, we know that when $n = N+1$, $|\{k : t.k =\perp\}| \leq n$ is always "True", so program n-FSTR simplifies to:

$$\text{MFSTR1} :: t.(k-1) \neq\perp \wedge k \neq 0 \wedge t.k \neq t.(k-1) \rightarrow t.k := t.(k-1)$$

$$\text{MFSTR2} :: t.N \neq\perp \wedge k = 0 \wedge t.k \neq t.N +_2 1 \rightarrow t.k := t.N +_2 1$$

Program MFSTR is identical to the fail-safe fault-tolerant token ring program presented by Arora and Kulkarni in [AK98a]. However, our intermediate programs are different. This is because, in [AK98a], certain bad transitions that are unreachable in the presence of a fault class F_i were already removed. Thus, though the overall multitolerant program is correct, the intermediate programs adopted a more defensive approach, by removing more transitions that are necessary. As way of contrast, our approach removes only bad transitions that are reachable in the presence of faults.

In this section, we have considered the automated design of multitolerance with perfect detection to all fault classes, by considering one fault class at a time. In the next section, we consider the automated design of multitolerance with perfect detection, and minimal detection latency to all fault classes by considering one fault class at a time.

6.3.4 Multitolerant Programs With Perfect Detection and Minimal Detection Latency

In the previous section, we presented an algorithm (together with necessary proofs of non-interference) that yields fail-safe multitolerant programs to n fault classes, with perfect detection to every fault class. The algorithm considers the fault classes in a given total order.

In this section, an algorithm is developed (along with relevant proof of non-interference) that yields fail-safe multitolerant programs to n fault classes, with *perfect detection, and minimal detection latency to every fault class*. Again, the fault classes are considered in a given total order. Intuitively, the approach builds partly upon algorithm *add-efficient-fail-safe*, whereby, for each fault class, the set of earliest SS-inconsistent transitions is computed, and these transitions are then removed from the given program.

Given a fault-intolerant program p with safety specification SS, and n fault classes $F_1 \ldots F_n$, the idea is to transform p into a program p_n that is fail-safe fault-tolerant to $F_1 \ldots F_n$ with perfect detection and minimal detection latency to each fault class, by first considering fault class F_1, then F_2 until F_n is considered. Specifically, given a fault class F_i, and a program p_{i-1} that is fail-safe fault-tolerant with perfect detection, and minimal detection latency to fault classes $F_1 \ldots F_{i-1}$, p_{i-1} is transformed into a program p_i which is fail-safe fault-tolerant with perfect detection, and minimal detection latency to F_i, while preserving the fail-safe fault tolerance with perfect detection and minimal detection latency of p_{i-1} to $F_1 \ldots F_i$.

Apart from having to verify non-interference between a program p_{i-1} and the new detector components for F_2, we also need to verify that the perfect detection, and minimal detection latency of program p_{i-1} to fault classes $F_1 \ldots F_{i-1}$ are not interfered with when adding fail-safe fault tolerance with perfect detection, and minimal latency to F_2. Thus, the set of non-interference conditions, presented in Section 6.2, is extended with those conditions that guarantee that no interference exists between the new detector components for F_i, and the minimal detection latency of p_{i-1} to fault classes $F_1 \ldots F_{i-1}$.

In this section, we show the stepwise addition of fail-safe fault tolerance with perfect detection, and minimal detection latency to two fault classes F_1 and F_2. The procedure can be easily generalized to n fault classes.

Note: In this section, whenever it is clear from the context, we will use the term "fail-safe fault-tolerant program (fail-safe fault tolerance)" to mean "fail-safe fault-tolerant program (fail-safe fault tolerance) with perfect detection, and minimal detection latency".

Step 1 in Design of Efficient Multitolerance

Given are: (i) A fault-intolerant program p with safety specification SS, and (ii) fault classes $F_1 \dots F_n$ to be tolerated. To transform p into a program p_1 that is fail-safe fault-tolerant to F_1, set eit_1 of earliest SS-inconsistent transitions for p in presence of faults F_1 is computed, and this set of transitions is then removed from p. The program p_1 obtained is fail-safe fault-tolerant, with perfect detection, and minimal detection latency to F_1. This first step is shown in Fig 6.9.

$$eit_1 := \text{get-eit}(p, F_1, ss)$$
$$p_1 := p \setminus eit_1$$

Figure 6.9: The first step in the design of multitolerant programs with perfect detection and minimal latency.

Proposition 6 *Program p_1 is fail-safe fault-tolerant, with perfect detection, and minimal detection latency to F_1.*

Proof. The proof is based on Lemmas 5, 6 and 7, which ensure that p_1 satisfies its safety specification in presence of F_1, and have perfect detection, and minimal latency to F_1. □

We now need to show that this construction of p_1 satisfies the non-interference properties defined in Section 6.2.

First, we need to prove that "*In the absence of F_1, the detector components added to p do not interfere with p, i.e., each computation of p is in the* *problem specification even if it executes concurrently with the new detector components*".

Proof.

1. From Lemma 7, p_1 and p have the same behavior in the absence of faults.

2. From step 1, each computation of p is in the problem specification even if p executes concurrently with the detector components for fail-safe fault tolerance to F_1.

3. From step 3, the detector components for fail-safe fault tolerance to F_1 do not interfere with p.

□

Secondly, we need to prove that "*In the presence of faults F_1, each computation of the detector components is in the components' specification even if they execute concurrently with p, i.e., p does not interfere with the new detector components.*".

Proof.

1. From Prop 6, p_1 is fail-safe F_1-tolerant.

2. From step 1, each computation of the detector components for fail-safe fault tolerance to F_1 is in their specification even if they execute concurrently with p

3. From step 3, p does not interfere with the detector components for fail-safe F_1-tolerance.

□

In fact, these two non-interference conditions are guaranteed by construction of p_1 since the synthesis method is identical to algorithm *add-efficient-fail-safe(p, F_1, ss).* It is also guaranteed that p_1 has perfect detection, and minimum detection latency to F_1.

Step 2 in Design of Efficient Multitolerance

Next, fault class F_2 is considerd, and program p_1 (which is fail-safe fault-tolerant to F_1) is transformed into a program p_2 that is fail-safe fault-tolerant to F_2, while preserving the fail-safe fault tolerance to F_1, i.e., program p_2 is fail-safe F_1, F_2-tolerant. To achieve this, the set eit_2 of earliest SS-inconsistent transitions for p in presence of faults F_2 is computed, and these transitions are removed from program p_1 to obtain program p_2. Program p_2, designed as such, is fail-safe fault-tolerant, with perfect detection, and minimal detection latency to both F_1, and F_2. The design of p_2 is shown in Fig 6.10.

$$\begin{aligned} eit_2 &:= \text{get-eit}(p, F_2, ss) \\ p_2 &:= p_1 \setminus eit_2 \end{aligned}$$

Figure 6.10: The second step in the design of multitolerant programs with perfect detection and minimal latency

Proposition 7 (Fail-safe fault tolerance of p_2 to F_2) *Given a fault-intolerant program p with safety specification SS, two fault classes F_1 and F_2, and a program p_1 which is fail-safe fault-tolerant to F_1i.e., fail-safe F_1-tolerant. Then, $p_2 = p_1 \backslash$get-eit(p, F_2, ss) (i) is fail-safe fault-tolerant to F_2, (ii) has perfect detection in presence of F_2, and (iii) has minimal detection latency in presence of F_2.*

Observe that the set eit_2 is the set of earliest SS-inconsistent transitions for p in presence of F_2, but to obtain program p_2, the set eit_2 needs to be removed from program p_1.

To prove the correctness of such a step, we need to prove the following:

1. Program p_2 is fail-safe fault-tolerant to F_2 with perfect detection, and minimal detection latency, i.e., we prove Proposition 7.

2. We need to fail-safe fault-tolerance to F_1 in presence of F_1 is preserved, as explained in Section 6.2.

3. We need to prove that the perfect detection, and minimal detection latency of p_1 to F_1 is preserved

First, we prove that p_2 is fail-safe fault-tolerant to F_2.
Proof.

1. Given: $p_2 = p_1 \backslash \text{get-eit}(p, F_2, ss)$

2. There are two cases to consider:
   (i) $\exists \tau \in \text{get-eit}(p, F_2, ss)$ and $\tau \in \delta_{p_1}$
   (ii) $\exists \tau \in \text{get-eit}(p, F_2, ss)$ and $\tau \notin \delta_{p_1}$

3. (i) From step 3(i), $\tau \notin \delta_{p_2}$ since $\tau \in \text{get-eit}(p, F_2, ss)$ and $p_2 = p_1 \backslash$get-eit(p, F_2, ss)
   (ii) From step 3(ii), $\tau \notin \delta_{p_2}$ since $\tau \notin \delta_{p_1}$, and $p_2 = p_1 \backslash$ get-eit(p, F_2, ss)

4. From step 3, $\forall \tau \in \text{get-eit}(p, F_2, ss)$, $\tau \notin \delta_{p_2}$.

5. From step 4 and from Defs. 37 and 44, and construction of *get-eit*, bad transitions in ss reachable in presence of F_2 can no longer be reached.

6. From step 5, p_2 is fail-safe fault-tolerant to F_2.

□

We now prove the second part of Proposition 7: p_2 has perfect detection to F_2.
Proof.

1. Given: $p_2 = p_1 \backslash \text{get-eit}(p, F_2, ss)$

2. From Prop. 7 (i), p_2 is fail-safe fault-tolerant to F_2

3. From step 3, no computation of p_2 in presence of F_2 will violate SS.

4. From step 3, the new detector components for fail-safe fault tolerance to F_2 are complete.

5. From step 3 and Def. 44, all $\tau \in \text{get-eit}(p, F_2, ss)$ are SS-inconsistent for p

6. From step 5, the new detector components for fail-safe fault tolerance to F_2 are accurate.

7. From steps 4, and 6, the new detector components for fail-safe fault tolerance to F_2 are perfect.

$\square$

We now prove the third part of 7: p_2 has minimal detection latency to F_2.

Proof.

1. Given: $p_2 = p_1 \backslash$ get-eit(p, F_2, ss)

2. From Prop. 7 (i), $\forall \tau \in$ get-eit(p, F_2, ss), $\tau \notin \delta_{p_2}$

3. From step 3, p_2 has detection latency 0, i.e., minimal detection latency, to F_2.

$\square$ We have proved that this way of designing fail-safe fault tolerance to F_2 is correct, and that p_2 has perfect detection, and minimal detection latency to F_2.

However, we have yet to show that the construction preserves the fail-safe fault-tolerance to F_1, i.e., we need to verify that there are no interference.

To achieve this, we first need to prove that "*In the absence of F_1 and F_2, the detector components added to p_1 for fail-safe fault tolerance to F_2 do not interfere with p_1, i.e., each computation of p_1 is in the problem specification even if it executes concurrently with the new detector components*".

Proof.

1. From Prop. 7 (ii), p_2 has perfect detection to F_2.

2. From step 1, the new detector components for fail-safe fault tolerance to F_2 are perfect in p_1.

3. From step 3 and Lemma. 1, the new detector components for fail-safe fault tolerance to F_2 do not interfere with p_1.

□

We now prove the second part of the non-interference conditions, which is "*In the presence of F_1, the new detectors for fail-safe fault tolerance to F_2 do not interfere with the fail-safe F_1-tolerance of p_1, i.e., every computation of p_1 is in the fail-safe F_1-tolerance specification even if p_1 executes concurrently with the new components.*"

Proof. We prove this by contradiction. We first assume that there exists a computation in presence of F_1 that violates safety, and show that such a computation cannot exist, i.e., a contradiction.

1. Given $p_2 = p_1 \backslash \text{get-eit}(p, F_2, ss)$
2. Assume that there is a computation α in presence of F_1 that violates safety
3. From step 3 and Proposition. 2, α contains a bad transition τ.
4. From step 3 and by construction of p_1, τ is not reachable in presence of F_1.
5. From step 4, and by construction of p_2, no new transition is introduced, hence τ is still unreachable.
6. From steps 3, 4 and 5, we have a contradiction.

□

The proof of the third part, which is "*In the presence of F_2, p_1 does not interfere with the new detectors that provide fail-safe fault tolerance to F_2.*"

Proof. By Proposition 7(i), p_1 does not interfere with the new detector components. □ We have proved that the new added components to p_1 adds fail-safe fault tolerance with perfect detection, and minimal detection latency to F_2, and that this addition preserves the fail-safe fault tolerance to F_1. Thus, we now need to prove that, in presence of F_1, the perfect detection, and minimal detection latency of p_1 to F_1 is preserved.

Proof. We prove that perfect detection to F_1 is preserved in presence of F_1.

1. Given: $p_2 \ = \ p_1 \backslash$ get-eit(p, F_2, ss)

2. From the first design step, p_1 has perfect detection to F_1

3. By construction, the new detector components for F_2 do not interfere with fail-safe fault tolerance of p_1 to F_1, hence completeness of detector components for F_1 is preserved.

4. From Def. 38, every transitions $\tau \in$ get-eit(p, F_2, ss) is SS-inconsistent for p.

5. From step 4, and by construction, no SS-consistent transition is removed, hence accuracy of the detector components for F_1 is preserved.

6. From steps 3, and 5, perfect detection is preserved.

$\square$

Proof. We now prove that minimal detection latency to F_1 is preserved in presence of F_1.

1. Given: $p_2 \ = \ p_1 \backslash$ get-eit(p, F_2, ss)

2. From step 1, no transition is added.

3. From step 3, set eit_1 is still "removed"

4. From step 3, minimal detection latency to F_1 is preserved.

$\square$

We have, at this point, proved the correctness of the transformation step of program p_1 into program p_2. The procedure of adding fail-safe multitolerance can be easily generalized for n fault classes.

The algorithm to design fail-safe multitolerance to n fault classes, with perfect detection, and minimal detection latency is shown in Fig. 6.11.

add-efficient-fail-safe-multitolerance($p, [F_1 \ldots F_n], ss$: set of transitions):

$\{i := 1;\ p_0 := p$
while $(i \leq n)$ do $\{$
$\quad eit_i :=$ get-eit(p, F_i, ss);
$\quad p_i := p_{i-1} \setminus eit_i$;
$\quad i := i + 1;\}$ od
return$(p_n)\}$

Figure 6.11: Algorithm *add-efficient-fail-safe-multitolerance* adds fail-safe fault tolerance to n fault classes, with perfect detection, and minimal detection latency to every fault class

Theorem 15 (Synthesis of Efficient Multitolerance) *Given a fault-intolerant program p with safety specification SS, and n fault classes $F_1 \ldots F_n$. Algorithm* add-efficient-fail-safe-multitolerance *adds fail-safe fault tolerance to $F_1 \ldots F_n$ to p, with perfect detection, and minimal detection latency to all fault classes.*

6.3.5 A Simple Example

One-at-a-Time Addition of Fail-Safe Fault Tolerance with Perfect Detection and Minimal Detection Latency

In this section, we will present a small example to illustrate the workings of algorithm *add-efficient-fail-safe-multitolerance*. For continuity, we reuse the same example as before. The fault-intolerant program is identical to the program of Fig 6.5, and is depicted in the presence of F_1. Recall that, for this program, transitions $(10, 11)$ and $(20, 21)$ are bad transitions.

During the first iteration through algorithm *add-efficient-fail-safe-multitolerance*, the call to *get-eit* causes transitions $(7, 8)$, $(18, 19)$ and $(19, 20)$

to be tagged as earliest SS-inconsistent transitions. These transitions are then removed from the fault-intolerant program. The resulting fail-safe fault-tolerant program p_1 has perfect detection, and minimal detection latency to F_1. Program p_1 in presence of F_1 is shown in Fig. 6.12.

Figure 6.12: Resulting fail-safe fault-tolerant program with perfect detection, and minimal detection latency to F_1

Then, we consider program p_1 in presence of F_2, as shown in Fig. 6.13

Figure 6.13: Program p_1 in presence of F_2

In the second iteration through *add-efficient-fail-safe-multitolerance*, we need to add fail-safe fault tolerance with perfect detection, and minimal detection latency to F_2 to p_1, while preserving the fail-safe fault tolerance with perfect detection, and minimal detection latency of p_1 to F_1. First, we consider p_1 in presence of F_2, as shown in Fig. 6.13. The call to *add-efficient-fail-safe-multitolerance* causes transitions $(9, 10), (17, 18), (20, 21)$ to be considered as earliest SS-inconsistent transitions.

Observe that the call to *add-efficient-fail-safe-multitolerance* in the second iteration refers to the fault-intolerant program p, instead of p_1. This is so because had the call referred to p_1, transition $(17, 18)$ would not have been included. Given that transition $(19, 20)$ has been removed in the first iteration, if p_1 is referred to, then no path from a fault transition of F_2 to a bad transition, using only only program transitions, would be observed, i.e. no path from transition $(17, 18)$ to bad transition using only program transitions would be observed. So, transition $(17, 18)$ would not have been included as an earliest SS-inconsistent transition.

The resulting fail-safe multitolerant program p_2 to fault classes F_1 and F_2 with perfect detection, and minimal detection latency to both is shown in Fig. 6.14.

Figure 6.14: Resulting fail-safe fault-tolerant program p_2 in presence of F_2

In this section, we have considered the approach where fault classes are considered in some fixed total order, and presented two algorithms that automates the addition of multitolerance. Another possible design approach for multitolerance considers all fault classes at the same time.

In the next section, we present two algorithms that add multitolerance, while considering all fault classes at the same time.

6.4 All-at-a-time Design of Multitolerance

In the previous section (Section 6.3), we presented two algorithms that synthesize fail-safe multitolerant programs to n fault classes, by considering one fault class at a time. The first algorithm ensures that the resulting fail-safe multitolerant program has perfect detection to all fault classes, while the second algorithm ensures that the multitolerant program has perfect detection, and minimal detection latency to all fault classes.

In this section, we consider another design approach where all the fault classes are considered at the same time, and we present two algorithms based on this design approach that achieve the same goals as the algorithms of Section 6.3. The first algorithm yields fail-safe multitolerant programs to n fault classes $F_1 \ldots Fn$ with perfect detection, by considering all the fault classes at the same time, while the second algorithm, that again handles all fault classes at the same time, yields fail-safe multitolerant programs to fault classes $F_1 \ldots F_n$ with perfect detection, and minimal latency. We also show that fail-safe multitolerant programs obtained from corresponding algorithms of either design approach are identical. We further exploit this relation to prove properties of the fail-safe multitolerant programs obtained using the algorithms presented in this section.

Design of multitolerance while considering all fault classes at the same time still requires the verification of non-interference between the different program components. Since all fault classes are considered at the same time, adding fault tolerance to one fault class entails verification that the new detector components do not interfere with the fail-safe fault tolerance to all other fault classes. This problem is tackled by showing that the fail-safe multitolerant program obtained using the all-at-a time algorithm is identical to the fail-safe multitolerant program obtained by using the corresponding one-at-a-time algorithm.

6.4.1 Multitolerance with Perfect Detection

In this section, we present an algorithm that adds fail-safe multitolerance with perfect detection to fault classes $F_1 \ldots F_n$ by considering fault classes at the same time.

The algorithm,*add-perfect-fail-safe-multitolerance-all*, is shown in Fig. 6.15.

add-perfect-fail-safe-multitolerance-all($p, [F_1 \ldots F_n], ss$: set of transitions):
$\{$ cobegin
$\|\|_{i=1}^{n}\ \ ss_{r_i}$:=get-ssr(p, F_i, ss); coend
$ss_r := \bigcup_{i=1}^{n} ss_{r_i}$
return($p_n := p \setminus ss_r$)$\}$

Figure 6.15: Algorithm *add-perfect-fail-safe-multitolerance-all* adds fail-safe fault tolerance to n fault classes, with perfect detection to every fault class by considering all fault classes at the same time.

Theorem 16 *Given a fault-intolerant program p with safety specification SS, and n fault classes $F_1 \ldots F_n$. Algorithm* add-perfect-fail-safe-multitolerance-all *adds fail-safe fault tolerance to $F_1 \ldots F_n$, with perfect detection to all fault classes, while considering all fault classes at the same time.*

To prove this, we make the following observation.

Proposition 8 *Given a fault-intolerant program p with safety specification SS, and n fault classes $F_1 \ldots F_n$. Given two programs p_n :== add-perfect-fail-safe-multitolerance $(p, [F_1 \ldots F_n]), ss$, and p'_n :=add-perfect-fail-safe-multitolerance-all$(p, [F_1 \ldots F_n], ss)$. Then, $p_n = p'_n$.*

To prove the above proposition, we need to show the following:

1. Every transition removed in p_n is also removed in p'_n.
2. Every transition removed in p'_n is also removed in p_n.

Proof. We consider any given transition τ that is removed in p_n.

1. τ is removed in p_n

2. From step 1, $\exists i : 1 \leq i \leq n : \tau \in$get-ssr$(p_{i-1}, F_i, ss)$

3. From step 3 $\exists$ a computation α of p_{i-1} in presence of F_i s.t τ occurs in α

4. From step 3 α is also a computation of p in presence of F_i and τ occurs in α

5. From step 4, and by construction of p'_n, τ is removed in p'_n

$\square$

We now prove the second part:

Proof. We prove this by contradiction, i.e., we assume there is a transition τ that is removed in p'_n but not in p_n, and show a contradiction.

1. $\exists i : 1 \leq i \leq n : \tau \in$get-ssr$(p, F_i, ss)$

2. From step 1, there exists a comptutation α of p in presence of F_i s.t τ occurs in α.

3. Since $\tau \notin \delta_{p_n}$, $\forall i : 1 \leq i \leq n$, τ is not reachable by p_{i-1} in presence of F_i

4. By construction of $p_{i-1}, i > 1$, either τ is also unreachable by p in presence of F_i, or τ is already removed from p_n

5. From assumption, steps 3, and 4, we have a contradiction.

$\square$

Thus, we have proved that algorithms *add-perfect-fail-safe-multitolerance* and *add-perfect-fail-safe-multitolerance-all* yield identical fail-safe multitolerant programs with perfect detection. We present a simple example in the next section to illustrate the working of the algorithm.

6.4.2 A Simple Example

All-at-a-Time Addition of Fail-Safe Fault Tolerance with Perfect Detection

Again, we use the same example as before to illustrate the working of algorithm *add-perfect-fail-safe-multitolerance-all*. The fault-intolerant program in presence of F_1 and F_2 is shown in Figs. 6.16 and 6.17 respectively.

Figure 6.16: Fault-intolerant program in presence of F_1

Figure 6.17: Fault-intolerant program in presence of F_2

In the presence of F_1, bad transitions $(10, 11), (20, 21)$ are reachable, while in the presence of F_2, the same set of bad transitions is reachable (call to *get-ssr*). These transitions are then removed from the program to yield a fail-safe fault-tolerant program with perfect detection to both F_1 and F_2, as shown in Fig. 6.18.

Figure 6.18: Resulting fail-safe multitolerant program p_2 to F_1 and F_2 with perfect detection to both fault classes.

Observe that the resulting program in Fig. 6.18 is identical to the program shown in Fig. 6.8 (obtained using the approach that considers fault classes one at a time).

In the next section, we present an algorithm that adds fail-safe fault tolerance to n fault classes $F_1 \ldots F_n$, with perfect detection, and minimal detection latency to all fault classes, while considering all fault classes at the same time.

6.4.3 Multitolerance with Perfect Detection and minimal detection latency

In this section, we present an algorithm that adds fail-safe multitolerance to fault classes $F_1 \ldots F_n$with perfect fault detection, and minimal detection latency, while considering all fault classes at the same time.

The algorithm,*add-efficient-fail-safe-multitolerance-all*, is shown in Fig. 6.19.

Theorem 17 *Given a fault-intolerant program p with safety specification SS, and n fault classes $F_1 \ldots F_n$. Algorithm* add-efficient-fail-safe-multitolerance-all *adds fail-safe fault tolerance to $F_1 \ldots F_n$, with perfect detection, and minimum detection latency to all fault classes, while considering all fault classes at the same time.*

add-efficient-fail-safe-multitolerance-all($p, [F_1 \dots F_n], ss$: set of transitions):

{ cobegin
$||_{i=1}^{n}$ eit_i :=get-eit(p, F_i, ss); coend
$eit := \bigcup_{i=1}^{n} eit_i$
return($p_n := p \setminus eit$)}

Figure 6.19: Algorithm *add-efficient-fail-safe-multitolerance-all* adds fail-safe fault tolerance to n fault classes, with perfect detection, and minimal detection latency to every fault class by considering all fault classes at the same time.

To prove this, we make the following observation.

Proposition 9 *Given a fault-intolerant program p with safety specification SS, and n fault classes $F_1 \dots F_n$. Given two programs p_n :== add-efficient-fail-safe-multitolerance $(p, [F_1 \dots F_n]), ss$, and p'_n :=add-efficient-fail-safe-multitolerance-all($p, [F_1 \dots F_n], ss$). Then, $p_n = p'_n$.*

To prove the above proposition, we need to show the following:

1. Every transition removed in p_n is also removed in p'_n.
2. Every transition removed in p'_n is also removed in p_n.

Proof. The proof is trivial, by construction.

1. In both constructions, the set eit_i is computed in the same way, and then removed.
2. From step 1, every transition removed in p_n is also removed in p'_n
3. From step 1, every transition removed in p'_n is also removed in p_n

□

6.4.4 A Simple Example

One-at-a-Time Addition of Fail-Safe Fault Tolerance with Perfect Detection and Minimal Detection Latency

As before, we reuse the same example to illustrate the working of algorithm *add-efficient-fail-safe-multitolerance-all*. The fault-intolerant program in presence of F_1 and F_2 is shown in Figs. 6.20 and 6.21 respectively.

Figure 6.20: Fault-intolerant program in presence of F_1

Figure 6.21: Fault-intolerant program in presence of F_2

In the presence of F_1, transitions $(7,8),(18,19),(1,20)$ considered earliest SS-inconsistent transitions, while in the presence of F_2, transitions $(9,10),(17,18),(20,21)$ are earliest SS-inconsistent transitions. These transitions are then removed from the program to yield a fail-safe fault-tolerant program with perfect detection to both F_1 and F_2, as shown in Fig. 6.22.

Figure 6.22: Resulting fail-safe multitolerant program p_2 to F_1 and F_2 with perfect detection and minimal detection latency to both fault classes when considering all fault classes at the same time.

Observe that the resulting program in Fig. 6.22 is identical to the program shown in Fig. 6.14 (obtained using the approach that considers fault classes one at a time).

6.5 Chapter Summary

In this chapter, we have presented four different algorithms that yields fail-safe multitolerant programs, with various efficiency properties, such as perfect detection, and minimal detection latency for all fault classes, using different design approaches.

We have considered two possible approaches for the design of multitolerance, namely (i) one that considers one fault class at a time, and (ii) another that considers all fault classes at the same time. We first considered the approach which adds multitolerance by considering one fault class at a time, and we presented two algorithms, namely *add-perfect-fail-safe-multitolerance*, and *add-efficient-fail-safe-multitolerance*, which add fail-safe multitolerance to a previously fault-intolerant program, with various optimal properties. We explained that, during the addition of multitolerance, some non-interference conditions between different program components need to be verified. However, we extended the proof obligations to include non-interference with the efficiency properties of the program. When fail-safe multitolerance with perfect detection is added, the non-interference conditions are similar to those proposed by Arora and Kulkarni in [AK98a], and are guaranteed by the use of algorithm *add-perfect-fail-safe-multitolerance*. For provision of fail-safe multitolerance with perfect detection, and minimal detection latency, we verified non-interference between various program components, as well as verified that those efficiency properties are not compromised.

We then considered the approach where all the fault classes are handled at the same time. We provided two algorithms, namely *add-perfect-fail-safe-multitolerance-all*, and *add-efficient-fail-safe-multitolerance-all* that add fail-safe multitolerance with perfect detection, and perfect detection and minimal detection latency respectively to an initially fault-intolerant program. We show that the corresponding fail-safe multitolerant programs are identical to those obtained using the one-at-a-time design approach. This means that all non-interference conditions are satisfied, as well as optimal properties preserved.

The algorithms based on the one-at-a-time approach can be used to add fault tolerance to new fault classes. Specifically, assume a fail-safe fault-tolerant program F_n to n fault classes $F_1 \ldots F_n$. If fail-safe fault tolerance to fault class F_{n+1} needs to be added, those algorithms can be used, without having to recompute the fail-safe fault tolerance to all the other fault classes. On the other hand, the algorithms based on the all-at-a-time approach can be used when a fault class is re-defined, or removed. Also, this also means that for fail-safe multitolerance, efficiency properties such as perfect detection can be designed for all fault classes.

Chapter 7

Conclusion and Future Work

In this thesis, we have presented a framework for the design of efficient fail-safe fault tolerance. Such an approach is bound to raise several questions. We first address some of the issues raised in Section 7.1. In Section 7.2, we summarize the contributions made in this thesis, and we discuss their impact in Section 7.3. In Section 7.4, we outline some possible future avenues.

7.1 Discussion

In this section, we address some of the issues our approach has raised.

Arora and Kulkarni [AK98c] also give a formal definition of detectors. Isn't every detector according to Arora and Kulkarni a perfect detector? No. Arora and Kulkarni [AK98c, AK98a] define a detector to be a component which relates two predicates with each other: a detection predicate X (describing the presumed "bad" state like the crash of a process), and a witness predicate Z (indicating that this state holds). The *safeness* condition of the detector [AK98a] mandates that $Z \Rightarrow X$, i.e., the witness is never wrong, while the *progress* and *stability* conditions of the detector [AK98a] mandates that if X is true for long enough, Z will eventually witness this fact and it will do this until X is falsified again. If the detection predicate can be evaluated atomically by the processes, then the detection predicate can be equivalent to the witness predicate. However, a detector does not necessarily guarantee that the detection predicate has any meaningful connection to the correctness specification. So even if the witness predicate is equivalent to the detection predicate, there is no guarantee that it detects bad or inconsistent states with respect to a safety specification.

However, Arora and Kulkarni [AK98a] prove that for any safety specification and every action there exists a detection predicate such that executing the action when the predicate holds maintains the specification. They also indicate that a weakest predicate may exist. However, they do not explain how such predicates (detectors) can be obtained. Our theory gives a guideline how to find this weakest detection predicate d. Assuming that $d \equiv X \equiv Z$, every detector in the sense of Arora and Kulkarni is perfect. Further, in the sense of Arora and Kulkarni, the weakest predicate exists for critical actions, while in our case, we make no such distinction.

How do perfect detectors in our work compare to Chandra and Toueg's perfect failure detectors? There is a close relationship between our terminology and that of the failure detector theory of Chandra and Toueg [CT96]. Kulkarni [Kul99] argues that these failure detectors can be regarded as an instance of detectors in the sense of this paper. The accuracy property of Chandra and Toueg also limits the number of mistakes a detector can make. The completeness property of Chandra and Toueg also refers to the ability of a detector to detect *all* faults.

In this paper, we have assumed bounded programs, i.e., programs with finite state space. What is the impact if the program is unbounded? If unbounded programs are considered, the method has to deal with an infinite state space and in the worst case loses completeness, i.e., it may not terminate. However, the method remains sound. This is analogous to the situation in the area of model checking where the failure to invalidate the specification on any finite subset of the state space says nothing about satisfaction of the specification on an infinite state space. Because our method is transition-based, unfortunately, it does not allow to reason directly on the level of guarded command programs, which can be regarded as a finite representation of an infinite transition system.

In this paper, we provided an algorithm for automating design of fail-safe fault tolerance. Is it efficient? Can the theory be used as a stand-alone? There are two main contributions of the present work: Firstly, the transformation algorithm automates the addition of fault-tolerance and is efficient in the sense that it has polynomial time complexity in the "size" of the specification (the number of bad transitions) and the "size" of the program (the number of reachable transitions). If the program is given as a guarded command program, it must first be translated into a state machine.

Secondly, we provided a theory which allowed the derivation of the transformation algorithm and is used to prove its correctness. The theory can be regarded as a refinement of the detector theory of Arora and Kulkarni

[AK98c] and better explains the working principles of detectors, e.g., allows a natural way to formulate and explain accuracy and completeness properties. For example, Leveson *et al.* [LCKS90] observed that the efficiency of a detector is dependent on its location, i.e., which action the detector monitors. Our theory contributes to this by proving that it is sufficient to monitor critical actions with perfect detectors for fail-safe fault tolerance. This saves the programmer of having to try different detectors at different locations to add fail-safe fault tolerance.

Kulkarni and Arora [KA00] presented an algorithm which also solves the transformation problem defined in Section 4.3. Isn't this algorithm the same as the algorithm presented in this paper? No. The algorithm by Kulkarni and Arora [KA00] also works on the state-transition representation of the pogram but does more work than absolutely necessary: by adding detectors, it also removes *non-reachable* transitions from the transition relation. So while the effect of the transformation is the same, the form of the added detectors is different. The ability to formulate this difference is one of the contributions of our theory.

In this paper, we made use of bad transitions. How are those transitions obtained? Is the generation process computationally expensive? If the safety specification is given as a state invariant, i.e., a predicate ϕ on system states (without using history variables), then it is relatively easy to compute the set of bad transitions. For this, it is just necessary to inspect all possible transitions (s, s') and check whether s satisfies ϕ and s' satisfies $\neg\phi$. This is feasible if the set of transitions is bounded. In practice, most safety specifications are state invariants.

Not every safety specification can be represented as a predicate on system states, even if it is fusion-closed. As an example, consider a system consisting of three states s_1, s_2, s_3 and the correctness specification $SS = \{s_1 \cdot s_2 \cdot s_3\}$. There is no "bad state" in this program, but there is a bad transition (s_1, s_3). We are not aware of any method to efficiently calculate these transitions from an abstract representation of the specification (e.g., a temporal logic formula).

In the definition of SS-inconsistency, there exists a sequence of program transitions after the occurrence of faults that eventually lead to violation of safety. What is the impact of such a requirement? In the definition of SS-inconsistency, we require that there exists a sequence of program transitions that eventually lead to violation of safety. The reason behind this requirement is based on the fact that fault cannot directly violate safety, which can then only be violated by a (bad) program transition. Thus, when safety is violated, at least one program transition is executed (which is the bad transition itself). However, depending on the fault model, there can then be a sequence of program transitions that ultimately leads to the bad transition being executed. Also, the reason for considering only when there exists a sequence of program transitions that ultimately lead to safety violation is that one can prevent (bad) program transitions from occurring, however this is not possible for fault transitions. The impact of such a requirement is that it allows the definition of the earliest SS-inconsistent transition that underpins fast detection. If such a requirement is "removed", then assuming that a fault transition is an "earliest inconsistent transition", one cannot prevent it from occuring.

Can the algorithm *add-perfect-fail-safe* be used to synthesize fail-safe fault-tolerant programs with perfect detection, and optimal detection latency? We have shown through examples how the use of algorithms *add-perfect-fail-safe*, and *add-efficient-fail-safe* yield the same results for distributed algorithms. However, for other classes of programs, the results will be different.

But, there is a sense in which algorithm *add-perfect-fail-safe* is equivalent to algorithm *add-efficient-fail-safe*. Since all SS-inconsistent transitions can possibly lead to safety violation, then if we treat each such SS-inconsistent transition as bad, then the set ss of bad transitions is extended to include the set of transitions that are SS-inconsistent for p. Then, since all earliest inconsistent transitions are SS-inconsistent, they are also included in set ss.

Running algorithm *add-perfect-fail-safe* thus removes the earliest inconsistent transitions, which then gives the same result as that when running algorithm *add-efficient-fail-safe*. However, the set *eit* of earliest inconsistent transitions still needs to be determined. That is, algorithm *add-efficient-fail-safe* can call *add-perfect-fail-safe* for generation of efficient fail-safe fault-tolerant programs.

Is our assumption of such a fault model as assumed in this thesis valid? What it the impact of choosing a fault model where faults can directly violate safety? In this thesis, we have assumed fault models that can be tolerated. Specifically, we have discarded fault models where faults can directly lead to violation of safety. We can analyze the impact of such an assumption for two general cases.

In the case of distributed algorithms, it is seldom the case that faults can lead directly to violation of safety. To see this, consider, for example, a mutual exclusion protocol. When one process is executing in its critical section, even if a fault happens, the fault cannot just cause another process to start accessing its critical section. The fault can however *cause* the process to enter its critical section, by "enabling" a transition which would have otherwise been disabled. Kulkarni and Ebnenasir termed such specifications as fault-safe specifications [KE02].

For the case of embedded applications, such a fault model is still valid. For example, the output register can be replicated in such a way that the probability of more than a majority of registers being corrupted is always 0. However, this ensures that safety is never violated by faults.

If we allow faults to directly violate safety, then our algorithms can be extended to deal with such a case. When designing fail-safe fault tolerance, we need to take steps to prevent the program from reaching those states from where faults can directly violate safety. If such faults can occur from any state, then we need to prevent the program from reaching any state, i.e., there is no fail-safe fault-tolerant program.

7.2 Summary of Research Contributions

In this section, we present brief summaries of the main contributions made in this thesis. The aim was to develop a framework that can allow systematic development of efficient (fail-safe) fault-tolerant programs, where efficiency was characterized by such commonly-used metrics as detection coverage and detection latency.

7.2.1 Perfect Detection

In Chapter 4, we developed a theory of detectors, and identified a class of detectors, called perfect detectors, that are crucial in the design of fail-safe fault-tolerant programs. The theory is believed to capture the working principles of detectors better than before. We showed, among others, that composing critical actions of a program with perfect detectors ensures fail-safe fault tolerance in presence of faults, i.e., composing critical actions of a program with perfect detectors is sufficient to ensure fail-safe fault tolerance. We also showed that, in the absence of faults, liveness is not compromised. In practical terms, this means that, whenever an error is flagged, there is a "harmful" error in the system, i.e., it is not a false alarm. We have presented examples to show the viability of our approach.

As indicated by Leveson *et.al* in [LCKS90], the design of "effective" detectors is problematic, and the effectiveness is heavily reliant on the experience of the software designers/programmers. Though the authors of [LCKS90] did not explicitly indicate what they meant by "effectiveness" of detectors, we have shown that "effectiveness" is captured by the completeness, and accuracy properties of detectors. To lessen the impact of such requirements as experience of programmers on the design of effective detectors, we provided an algorithm that yields a fail-safe fault-tolerant program with perfect detection, by composing the critical actions of the corresponding fault-intolerant program with perfect detectors. This is achieved by removing those bad transitions that are reachable in presence of faults.

In general, to validate the fault tolerance mechanisms incorporated in a program, fault injection experiments [IT96, AAA$^+$90] are usually conducted.

In particular, they are used to quantify the coverage of the mechanisms, such as in [Hil00], where the coverage is the ratio of the number of faults detected to the number of faults injected. However, by design, the fail-safe fault-tolerant programs obtained from the algorithm has "perfect" coverage, since the detectors are perfect.

We have also shown that the automatic synthesis of fail-safe fault-tolerant programs has polynomial time complexity in the size of the state space of the fault-intolerant program.

7.2.2 Fast Detection

In Chapter 5, we developed a theory of fast detectors, and identified a class of detectors, called fast detectors, that ensures minimal detection latency, as well as perfect detection. The idea behind fast detection is to prevent errors from propagating and corrupting the entire state of the program. However, when designing fast detectors, one problem can be that these fast detectors are not perfect. We have therefore identified the class of fast detectors that ensures both perfect detection, and minimal detection latency.

As before, design of effective detectors is problematic. Also, when designing fault-tolerant systems, when fault injection experiments are conducted to determine the effectiveness of the fault tolerance mechanisms, detection latency of these mechanisms is usually evaluated, and is taken to be the minimum time between the onset (injection) of a fault and its detection. Fault injection experiments can be a computationally expensive process to evaluate detection latency. Thus, to tackle the problem of designing perfect detectors while ensuring minimal detection latency, we provided an algorithm that achieves that. By construction, the detection latency of the fail-safe fault-tolerant program to the fault class is 0 (minimal).

7.2.3 Design of One-at-a-time Multitolerance

Building upon the design of perfect and fast detectors, we aimed at generalizing the results to deal with multiple fault classes. In Chapter 6, we addressed the problem of adding efficient fail-safe multitolerance, i.e., the ability of a program to tolerate multiple classes of faults. We argued that, in a distributed environment, the nature and types of faults affecting a system is varied, and thus the design of fault-tolerant systems needs to be cognizant of such diversity. There are two possible ways of designing multitolerance, and in Chapter 6, we presented algorithms that build upon each approach.

The first approach deals with addition of multitolerance in a stepwise fashion, that is adding fail-safe fault tolerance to a given fault class one at a time. We explained that during the addition of multitolerance to an initially fault-intolerant program, the program is extended with detector components that handle faults from each fault class. Consequently, there can be interference between different program components, for example between detector components for different fault classes, or between the program and the detector components for some fault class. This interference needs to be handled, and "removed", since it may prevent the different program components from satisfying their problem specification. Some of the non-interference conditions were first presented by Arora and Kulkarni in [AK98a]. However, given our focus on efficient fail-safe fault tolerance, we explained that during addition of multitolerance, the detector components for different fault classes should not interfere with the optimal properties (perfect detection, and minimal detection latency) of the fail-safe multitolerant program to different fault classes. Therefore, we have extended the set of non-interference conditions to include those relating to program properties. Therefore, any algorithm that automatically adds multitolerance to a program needs to reflect those non-interference requirements.

We developed two algorithms that automatically add multitolerance to a fault-intolerant program. The first algorithm transforms a fault-intolerant program into a fail-safe multitolerant program, with perfect detection to all fault classes considered. The second algorithm transforms a fault-intolerant program into a fail-safe fault-tolerant program with perfect detection, and minimal detection latency to every fault class considered. Each algorithm was incrementally developed, and each design stage was proved to handle the non-interference conditions (between program components, and program properties).

7.2.4 Design of All-at-a-time Multitolerance

In Chapter 6, we also focused on the second approach for designing multitolerance. In this approach, all the fault classes are considered at the same time. We developed two algorithms that add fail-safe multitolerance to a fault-intolerant program, by considering all fault classes at the same time.

The first algorithm yields a fail-safe multitolerant program with perfect detection to all fault classes, while the second algorithm yields a fail-safe multitolerant program with perfect detection, and minimal latency to all fault classes. We have shown that the programs yielded using these algorithms are identical to those yielded by the corresponding algorithm that considers one fault class at a time.

7.3 Impact

We now discuss briefly the impact our contributions have on design of fail-safe fault tolerance. Our theory provides a general yet powerful basis for understanding the working principles of detectors. Specifically, we have been able, through our defined notion of perfect detectors, to explain design decisions in the design of fault-tolerant programs. For example, only critical actions of programs were composed with detectors. How these detectors were designed or what properties should they possess were mostly intuitive, or based on experience. Our contribution has shown that, for fail-safe fault tolerance, the detectors need to be perfect, and that it is sufficient to compose critical actions with such detectors.

We have also shown that, in order to have minimal detection latency, the detectors of non-critical actions are non-trivial, i.e., they are also perfect. By way of contrast, Arora and Kulkarni observed in [AK98b] that, "according to their experience", detectors of non-critical actions are trivial, i.e., true. Thus, we conclude that there are times when their approach can yield minimal detection latency, but may be not always. Arora and Kulkarni also observed in [AK98b] that the detectors of critical actions are non-trivial, while Leveson *et.al* observed in [LCKS90] that design of effective detectors is difficult. Though the authors never explicitly clarified the meaning of "non-trivial", or "effective", our theory has enabled us to determine the properties that underpin the notions of "non-triviality" and "effectiveness", i.e., the properties of completeness, and accuracy.

Further, we have provided a generalization of our approach by looking at the design of multitolerant programs. We have provided algorithms that can automatically add efficient fail-safe multitolerance, and that for certain classes of fault tolerance, and efficiency properties, the impact of the order in which fault classes are handled can be minimized.

7.4 Future Work

Our work on automated synthesis of fail-safe fault tolerance has opened up several new avenues for future research. Some of them are outlined below.

One of the main assumptions underpinning our work has been that specification are fusion closed. Fusion closure guarantees that the history of the computation is "available" in the current state of the system, i.e., by just looking at the current state, one can determine whether the next step is a bad one or not. A specification that is not fusion closed can be made fusion closed by adding history variables. However, adding another variable leads to an exponential increase in size of the state space of the program. This suggests two possible avenues: (i) Is there a way of converting non fusion closed specifications into a fusion closed specification that minimizes the number of states added?. (ii) Does there exist a class of non fusion closed specifications which have sufficiently "nice" features such that the absence fusion closure is irrelevant?.

In this thesis, we have looked at two properties of fail-safe fault-tolerant programs, namely perfect detection, and detection latency. However, there are other properties that can be investigated. One such property is *availability*. In fact, in the course of this work, we have observed that one needs to adopt a pessimistic look of a computation in order to have fast detection. However, for availability, one needs to adopt a more optimistic outlook. How can availability be modeled, and its impact on design decisions will be investigated.

Bibliography

[AAA$^+$90] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J. C. Fabre, J. C. Laprie, E. Martins, and D. Powell. *"Fault Injection for Dependability Validation: A Methodology and Some Applications"*. *IEEE Transactions on Software Engineering*, 16(2):166–182, 1990.

[AD97] Yehuda Afek and Shlomi Dolev. Local stabilizer. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing (PODC97)*, 1997.

[ADK01] A. Arora, M. Demirbas, and S. Kulkarni. *"Graybox Stabilization"*. In *Proceedings of the International Conference on Dependable Systems and Networks*, 2001.

[AG94] Anish Arora and Mohamed G. Gouda. Distributed reset. *IEEE Transactions on Computers*, 43(9):1026–1038, September 1994.

[AK95] Anish Arora and Sandeep S. Kulkarni. Designing masking fault-tolerance via nonmasking fault-tolerance. In *Proceedings of the 14th IEEE Symposium on Reliable Distributed Systems (SRDS95)*, pages 174–185, 1995.

[AK98a] Anish Arora and Sandeep S. Kulkarni. Component based design of multitolerant systems. *IEEE Transactions on Software Engineering*, 24(1):63–78, January 1998.

[AK98b] Anish Arora and Sandeep S. Kulkarni. Designing masking fault tolerance via nonmasking fault tolerance. *IEEE Transactions on Software Engineering*, 24(6), June 1998.

[AK98c] Anish Arora and Sandeep S. Kulkarni. Detectors and correctors: A theory of fault-tolerance components. In *Proceedings of the 18th IEEE In-*

ternational Conference on Distributed Computing Systems (ICDCS98), May 1998.

[AL91] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, May 1991.

[APSV91] Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Self-stabilization by local checking and correction. In *FOCS91 Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, pages 268–277, 1991.

[AS85] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.

[Avi85] A. Avizienis. *"The N-Version Approach to Fault-Tolerant Software"*. *IEEE Transactions on Software Engineering*, 39(4):1491 – 1501, 1985.

[BDDT98] Joffroy Beauquier, Sylvie Delaët, Shlomi Dolev, and Sébastien Tixeuil. Transient fault detectors. In *Proceedings of the 12th International Symposium on DIStributed Computing (DISC'98)*, number 1499 in Lecture Notes in Computer Science, pages 62–74, Andros, Greece, September 1998. Springer-Verlag.

[CDPV01] A. Cournier, A. K. Datta, F. Petit, and V. Villain. *"Snap-stabilizing PIF Algorithm in Arbitrary Rooted Networks"*. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 91 – 98, 2001.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, MA, Reading, Mass., 1988.

[Cri91] Flaviu Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, February 1991.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[CW96] E. M. Clarke and J. M. Wing. *"Formal Methods: State of the Art and Future Directions"*. *ACM Computing Surveys*, 28(4):626 – 643, 1996.

[Dij74] Edsger W. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

[DIM93] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7:3–16, 1993.

[Dol97] Shlomi Dolev. Self-stabilizing routing and related protocols. *Journal of Parallel and Distributed Computing*, 42(2):122–127, 1997.

[Dol00] Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000.

[DW95] Shlomi Dolev and Jennifer L. Welch. Self-stabilizing clock synchronization in the presence of Byzantine faults. In *Proceedings of the Second Workshop on Self-Stabilizing Systems*, pages 9.1–9.12, 1995.

[Gae99a] Felix C. Gaertner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys*, 31(1):1–26, March 1999.

[Gae99b] Felix C. Gaertner. 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.

[GM91] Mohamed G. Gouda and Nicholas J. Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40(4):448–458, April 1991.

[Gum93] 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.

[GV00] 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.

[GV01] 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.

[Hil00] M. Hiller. *"Executable Assertions for Detecting Data Errors in Embedded Control Systems"*. In *Proceedings International Conference Dependable Systems and Networks*, pages 24 – 33, 2000.

[HJS01] M. Hiller, A. Jhumka, and N. Suri. *"An Approach for Analyzing the Propagation of Data Errors in Software"*. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 161 – 170, 2001.

[HJS02] M. Hiller, A. Jhumka, and N. Suri. On the placement of software mechanisms for detection of data errors. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 135 – 144, 2002.

[IT96] R.K. Iyer and D. Tang. *Experimental Analysis of Computer System Design Dependability*, chapter 5 in *Fault-Tolerant Computer System Design*. Prentice Hall, 1996.

[JHCS02] A. Jhumka, M. Hiller, V. Claesson, and N. Suri. *On Systematic Design of Globally Consistent Executable Assertions in Embedded Software*. In *Proceedings LCTES/SCOPES*, pages 74–83, 2002.

[JHS01] A. Jhumka, M. Hiller, and N. Suri. *"Assessing Inter-Modular Error Propagation in Distributed Software"*. In *Proceedings of the 20th Symposium on Reliable Distributed Systems*, pages 152 – 161, 2001.

[JHS02a] A. Jhumka, M. Hiller, and N. Suri. *"An Approach to Specify and Test Component-Based Dependable Software"*. In *Proceedings of the 7th International Symposium on High Assurance Systems Engineering*, 2002.

[JHS02b] A. Jhumka, M. Hiller, and N. Suri. *"Component-Based Synthesis of Dependable Embedded Software"*. In *Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 111 – 128. Lecture Notes in Computer Science (LNCS), 2002.

[JHS03] A. Jhumka, M. Hiller, and N. Suri. *"A Framework for the Design and Validation of Efficient Fail-Safe Fault-Tolerant Programs."*. In *To Appear, Proceedings Software and Compilers for Embedded Systems (SCOPES)*, 2003.

[KA97a] S. Kulkarni and A. Arora. *"Once-and-forall Management Protocol (OFMP)"*. In *Proceedings of the 5th International Conference on Network Protocols*, 1997.

[KA97b] Sandeep S. Kulkarni and Anish Arora. *Compositional design of multitolerant repetitive Byzantine agreement.* In *Proceedings of the 18th International Conference on the Foundations of Software Technology and Theoretical Computer Science, Kharagpur, India*, pages 169 – 183, 1997.

[KA98] S. Kulkarni and A. Arora. *"Multitolerance in Distributed Reset"*. *Chicago Journal of Theoretical Computer Science*, 1998(4), 1998.

[KA00] 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.

[KE02] S. Kulkarni and A. Ebnenasir. *"Complexity of Adding Fail-Safe Fault Tolerance"*. In *Proceedings International Conference on Distributed Computing Systems*, 2002.

[KRS99] Sandeep S. Kulkarni, John Rushby, and Natarajan Shankar. A case-study in component-based mechanical verification of fault-tolerant programs. In Anish Arora, editor, *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Workshop on Self-Stabilizing Systems*, pages 33–40, Austin, TX, USA, June 1999. IEEE Computer Society Press.

[Kul99] Sandeep S. Kulkarni. *Component Based Design of Fault-Tolerance.* PhD thesis, Department of Computer and Information Science, The Ohio State University, 1999.

[LA90] P. A. Lee and T. Anderson. *"Fault Tolerance - Principles and Practice"*. volume 3 of *Dependable Computing and Fault-Tolerant Systems*. Springer Verlag, 1990.

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, March 1977.

[Lap92] J. C. Laprie. *"Dependability: Basic Concepts and Terminology"*. In *Dependable Computing and Fault-Tolerant Systems series*, volume 5. Springer-Verlag, 1992.

[LCKS90] N. Leveson, S. S. Cha, J. C. Knight, and T. J. Shimeall. *The Use of Self-Checks and Voting in Software Error Detection: An Empirical Study. IEEE Transactions on Software Engineering*, 16(4):432–443, 1990.

[Liu91] Zhiming Liu. *Fault-tolerant programming by transformations*. PhD thesis, University of Warwick, Department of Computer Science, 1991.

[LJ92] Zhiming Liu and Mathai Joseph. Transformation of programs for fault-tolerance. *Formal Aspects of Computing*, 4(5):442–469, 1992.

[LJ93] Zhiming Liu and Mathai Joseph. Specification and verification of recovery in asynchronous communicating systems. In Jan Vytopil, editor, *Formal Techniques in Real-time and Fault-tolerant Systems*, chapter 6, pages 137–165. Kluwer, 1993.

[LJ94] Zhiming Liu and Mathai Joseph. Stepwise development of fault-tolerant reactive systems. In *Formal techniques in real-time and fault-tolerant systems*, number 863 in Lecture Notes in Computer Science, pages 529–546. Springer-Verlag, 1994.

[LJ95] Zhiming Liu and Mathai Joseph. A formal framework for fault-tolerant programs. In C. M. Mitchell and V. Stavridou, editors, *Mathematics of Dependable Computing*, pages 131–148. Oxford University Press, 1995.

[MAM84] A. Mahmood, D. M. Andrews, and E. J. McCluskey. *"Executable Assertions and Flight Software"*. In *Proceedings of the 6th AIAA/IEEE Digital Avionics Systems Conference (DASC-6)*, pages 346 – 351, 1984.

[Ran75] B. Randell. *"System Structure for Software Fault Tolerance". IEEE Transactions on Software Engineering*, 1(2):220 – 232, 1975.

[Ros95] D. S. Rosenblum. *"A Practical Approach to Programming with Assertions". IEEE Transactions on Software Engineering*, 21(1):19 – 33, 1995.

[Sai78] S. H. Saib. *"Executable Assertions: An Aid to Reliable Software"*. In *Proceedings of 11th Asilomar Conference on Circuits, Systems and Computers*, pages 277 – 281, 1978.

[SS98] N. Suri and P. Sinha. *"On the Use of Formal Methods for Validation"*. In *Proceedings of the 28th International Symposium on Fault-Tolerant Computing*, pages 390 – 401, 1998.

[SS99a] P. Sinha and N. Suri. *"Identification of Test Cases Using a Formal FI Approach"*. In *Proceedings of the 29th International Symposium on Fault Tolerant Computing*, 1999.

[SS99b] P. Sinha and N. Suri. *"On the Use of Formal Techniques for Analyzing Dependable Real-Time Protocols"*. In *Proceedings Real Time Systems Symposium*, pages 126 – 135, 1999.

[YB94] H. Yin and J. M. Bieman. *"Improving Software Reliability With Assertion Insertion"*. In *Proceedings of the International Test Conference*, pages 831 – 839, 1994.

CV

Personal Data:

- Name: Arshad Jhumka
- Date of Birth: 14.04.1974
- Nationality: Mauritian

School Education

- Jan. 1985 – Dec. 1989: Royal College, Port-Louis, Mauritius (SC)
- Jan. 1990 – Dec. 1991: Royal College, Port-Louis, Mauritius (HSC)

University Education

- Oct. 1992 – Jun. 1995: University of Cambridge, England (BA Computer Science)
- 1999: MA Computer Science, University of Cambridge
- Feb. 2000 – Nov. 2002: Chalmers University of Technology, Gothenburg, Sweden
- Nov. 2002 – Nov. 2003: TU - Darmstadt, Germany (PhD Computer Science)