Fault-tolerant Programming By Transformations

Zhiming Liu[1]
University of Warwick

[1]Address for correspondence: Department of Computer Science, University of Warwick, Coventry CV4 7AL. This work was supported by research grant GR/D 11521 from the Science and Engineering Research Council.

Abstract

It has been usual to consider that the steps of program refinement start with a program specification and end with the production of the text of an executable program. But for fault-tolerance, the program must be capable of taking account of the failure modes of the particular architecture on which it is to be executed. In this thesis we shall develop a formal framework which shows how a program constructed for a *fault-free* system can be transformed into a *fault-tolerant* program for execution on a system which susceptible to failures.

Physical faults are modelled by a set of atomic actions whose semantics are defined in the same way as the semantics of usual program actions. The interference of fault actions on the execution of a program is then defined by the *failure semantics*. The behaviours of the program on a system with the specified set of fault actions are simulated by a *fault transformation* of the program into its *fault-affected version*. The properties of such a behaviour (called the *fault properties*) are studied by reasoning about the fault-affected version.

The addition of fault-tolerance to a program is modelled by a *fault-tolerant* transformation which introduces the necessary redundancy in the program so that the specified faults can be tolerated. A fault-tolerant program can be further refined by using *fault-tolerant refinement* which preserves both the functional and the fault-tolerant properties of the program.

——— To my grandparents and my parents

for their love, encouragement and support

Acknowledgements

I would like to record my most grateful thanks to my supervisor, Professor Mathai Joseph, for his guidance, encouragement through the past more than three years, for his constant willingness to find time for our discussions, and for his careful revisions of every earlier version of each chapter of this thesis. I fell deeply indebted for the care and love from Mathai, his wife Anita and their children, without these my life in a foreign land would have been much harder.

I am grateful to Professor Zhou Chaochen who was my supervisor when I was doing my MSc in 1985-1988 in the Software Institute of China. I felt that he was a father-like supervisor giving me love and hope. I started my research career in computer science under his guidance. The experience and general background that I have achieved under his guidance and encouragement are invaluable.

I would like to thank my colleague, Asis Goswami, for helpful discussions, for his friendly advice and good suggestions. My sincere thanks also go to Jozef Hooman for carefully reading some of the results presented as a research report, and for many detailed suggestions for improvement. I acknowledge the helpful discussion with Professor R.J.R. Back about the ideas of modelling faults and using the refinement calculus for fault-tolerant programming, when we met during the Fourth Refinement Workshop in Cambridge in January 1991.

This work was supported by research grant GR/D 11521 from the Science and Engineering Research Council.

Special thanks go to Richard Kent and Vivien Kent for their help and friendship during my time in Coventry.

Finally, I would like to thank my wife, Hong. Her eternal love, infinite patience and constant encouragement have sustained me throughout the past three years.

Declaration

In this thesis, the UNITY notation by Chandy and Misra (1988) has been used to provide high level specification of fault-tolerant programs. The fault-tolerant refinement framework in this thesis has been developed by adapting the action system formalism and its refinement calculus by Back (1978).

Contents

Chapter 1

Introduction

The problem of *making computing reliable* has been with us since the days of the first computers, which were built using relays, vacuum tubes, delay line and cathode-ray storage tubes, and other relatively unreliable components. The reliability of the second and later computer generations has significantly improved due to the introduction of semiconductor components, magnetic core storage elements and advanced technology. However, we can never expect to obtain a system which is totally reliable.

At the same time, by using new components and techniques, hardware developments are producing systems with ever decreasing cost and size, and with increasing power and complexity. These advances have made computers an inherent part of many aspects of present day life and will further add to the use of computer systems in many different areas. The more computer systems are involved in our daily life, the more it is essential to ensure their reliability. Errors in a computer used in a banking system, or a company data system, may cause the loss of vast sums of money. And with errors in a system for air traffic control, or nuclear reactor control, human lives may be in great danger.

1.1 Aim of The Thesis

There are several different ways in which a program can be developed using formal rules that guarantee that it will *satisfy* a specification when executed on a fault-free system [Lam83, AL88, Bac87, Bac88, BW89, Bac89]. The aim of this thesis, however, is to develop a formal framework within which we can describe the behaviour of *fault-tolerant* programs when executed on systems which may exhibit any of a set of specified *failure* properties. Such a framework provides the necessary starting point for the development of methods for constructing fault-tolerant programs.

Fault-tolerant programs are required for applications where it is essential that failures do not cause a program to have unpredictable execution behaviours. We assume that the failures do not arise from errors in the program, since methods such as those mentioned above can be used to construct error-free programs. So, the only failures we shall consider are those caused by the underlying hardware and software faults referred to *physical faults* in this thesis. Many such failures can be *masked* from the program using automatic error detection and correction methods, but there is a limit to the extent to which this can be achieved at reasonable cost in terms of the resources and the time needed for correction.

When the nature or frequency of the errors makes automatic detection *and* correction infeasible, it may still be possible that error *detection* can be performed. It is desirable that fault-tolerant programs are able to perform predictably under these conditions: for example when using memory with single bit error correction and double bit error detection which operates even when the error correction is not effective. In fact, the provision of good program level fault-tolerance can make it possible to reduce the amount of expensive system error correction needed, as program level error recovery can often be focussed more precisely on the damage caused by an error than a general-purpose error correction mechanism.

The task is then to develop programs which perform predictably in the presence of *detected* system failures, and this requires the representation of such failures in the execution of a program. Earlier attempts to use formal proof methods for verifying the properties of fault-tolerant programs (e.g. [JMS87], [HW88]) were based on an *informal* description of the effects of faults, and this limits their applicability.

In this framework, we shall instead model the physical faults as a set of *atomic action* which performs state transformations in the same way as other program actions, making it possible to extend a semantic model to include fault actions and to use a single consistent method of reasoning for both program and fault actions. These fault actions interfere with the execution of a program in a way that is defined by the *failure semantics* of the program, derived from the semantics of the program and the fault actions. The behaviour of a program on a system with a specified set of fault actions is then simulated by a transformation of the program into a *fault-affected version*. The semantics of such a fault-affected version of a program is then proved to be the same as the failure semantics of the program.

Based on the failure semantics, transformations are applied to a high level and non-fault-tolerant program specification. The final step of the transformation is to produce a program which is executable, efficient and fault-tolerant on a system with a specified set of faults. The transformations used in such a development procedure include *refinement transformations*, a *fault-tolerant transformation* and *fault-tolerant refinement transformations*. Refinement transformations are required to preserve the functional correctness of non-fault-tolerant programs. Fault-tolerant transformation is needed to provide fault-tolerance to non-fault-tolerant programs. And fault-tolerant refinement transformations are required to preserve both the functional and fault-tolerant properties of fault-tolerant programs. Such a transformational approach provides a way of converting fault properties and fault-tolerant properties into functional properties. Fault properties and fault-tolerant properties can thus be reasoned about using similar formal methods as used for reasoning about the functional properties of programs.

The framework and the transformations provide a basis for the development of methods for the systematic construction of fault-tolerant programs. In this thesis, we illustrate how existing techniques for program fault-tololerance can be represented and applied in our framework. An obvious extension to our work is to incorperate such techniques into a program development method: it may even possible to use the transformational approach to develop more techniques for fault-tolerance.

1.2 Basic Concepts

There is a widely accepted terminology that is used for describing fault-tolerance. Many of these terms are helpful for understanding the formal definitions used in this thesis.

Systems

A *computing system* consists of a program and a hardware machine. The program defines a set of actions that transform the *state* of program variables. The hardware machine has enough memory to store the state (values) of the variables. It executes the program by carrying out the state transformations following the flow of control defined by the program. A behaviour of the system is a sequence of state transitions. The desired behaviours of a system can be defined or characterized by a *program specification*. During the transition from one state to another, the system may pass through a number of intermediate states which cannot be accessed from outside the action carrying out the state transition.

Reliability and Failures

The *reliability* of a system is taken to be a measure of the success with which it meets its specification [RLT78, AL81]. Nothing can be said about the reliability of a system without knowing its specification, because without this there is no way to determine whether a behaviour succeeds or fails.

A *failure* of a system occurs when the behaviour of the system deviates from what is required by its specification. The reliability of a system is inversely related to the frequency of the occurrence of failures.

Errors and Faults

Computing systems are unreliable because of software and hardware *faults*. The execution of a program enters an *error state* when such a fault occurs. In an error state, further processing without using any corrective actions by the system will usually lead to a failure, even in the absence of any subsequent occurrence of faults.

Software design faults are due to mistakes in the design or construction of the software. Software with design faults will not satisfy its specification, and hence it will not behave correctly even if it runs on a fault-free hardware. Hardware with faults may not be capable of carrying out the state transformations defined by the program. A hardware component may malfunction either because of physical degradation or because it is badly designed.

Therefore, an error state is entered when a fault action transforms the state, instead of the desired program action defined by the specification. Such a way of viewing the effect of a fault suggests the idea that a system fault can be modelled as just another kind of action that performs state transformations. And a class of faults can then be modelled as an environment (a program) that interferes with the behaviour of the system.

Fault-tolerance

Taking reliability as a requirement for a computing system, two distinct approaches, *fault-intolerance* and *fault-tolerance*, can be used in order to reach the goal of reliable computing. These approaches are applicable to both the hardware and the software of the system. In the fault-intolerance approach, reliability is assured by *a priori fault avoidance* and *fault removal* [Avi76].

Fault avoidance is concerned with design methodologies and the selection of techniques and technologies which aim to avoid the introduction of faults during the design and construction of a system.

In principle, software design faults can be avoided by using formal specification, verification and implementation methods, By using such methods, a program can be developed to guarantee that the execution of the program on a fault-free hardware machine is always correct with respect to its specification. For formal specification and verification techniques, the reader can refer to [Hoa69, OG76, Gri81], and to [Bac78, Lam83, AL88, Bac88, Bac89, Mor90] for techniques of implementing programs from specifications.

Fault removal is concerned with checking the implementation of a system and eliminating any faults which are thereby exposed. This elimination takes place before regular use of the system begins.

In both principle and practice, fault avoidance and removal are insufficient for reliable operation in the hardware. Physical components, even though without design faults, can be faulty because of their age and deterioration. Thus fault-tolerance is needed. This is based on the use of protective redundancy: a system can be designed to be *fault-tolerant* by incorporating additional components and algorithms which attempt to ensure that the occurrence of an error state does not result in later system failures [RLT78].

Fault-tolerance is by no means a new concept. In the 1950s, there were proposals for adding redundant hardware components to systems to achieve higher reliability [Neu56]. Since then, redundancy techniques have been generalized from hardware redundancy to software redundancy. Much work has been done in developing methods for structuring these additional components [FF73, HLMR74] in sequential systems. These methods have then been extended to concurrent systems [Ran75, RLT78]. The implementations of the redundant structures need algorithms to restore the system state after faults occur, and recovery protocols to restart the system from the restored state. Work also has been done in writing these algorithms [Rus80, KT87] and implementing the structures [RT79, Kim82, CAR83, JC86]. Yet these methods, algorithms, implementations are generally described in rather informal terms, with no formal proofs of correctness.

For some applications or at some development stages, informal descriptions and discussions are acceptable. However, for safety-critical applications or from the point of view of generality, more than just confidence in the working of fault-tolerant techniques is required [Min91a, Min91b, Min91c]. There is a need for a formal framework within which design faults of the redundant components can be avoided, and fault properties and fault-tolerant properties can be formally and rigorously discussed at every stage of the system development. This forms the motivation of the objective of this thesis.

1.3 Principles and Techniques

To make a program tolerate physical faults, redundancy techniques have to be used in the program. Using such techniques, the program is changed into a fault-tolerant version by adding programs, or program segments, or instructions which serve to provide either error detection or recovery. These redundant components would not be needed if the program was to be executed on a fault-free hardware. No more new design faults should be introduced into a program because of the introduction of these redundant components. This implies that the redundant components have to be free of design faults, and implemented with very high reliability.

Techniques for fault-tolerance can be classified into backward and forward recovery techniques. *Backward error recovery* aims to restore the system to a valid error-free state which occurred prior to the manifestation of the fault [Ran75]. Forward recovery aims to identify the error and, based on this knowledge, corrects the system state containing the error [BC81]. Recovery techniques can also be classified into *static* [FF73, Ran75] and *dynamic* [Woo81, Had82, KT87] error recovery depending on the way of coordinating the recovery operations for different parts of the program.

Clearly, it is out of the question and not necessary to add redundancy to each basic computational operation for error checking and recovery. This should be done for a computational unit consisting of a group of basic operations. A scheme for doing this is outlined in [FF73]. In such a scheme, multiple implementations and executions of one algorithm are employed to produce a set of resulting states. A majority voting scheme is then used to determine which of the states is to be accepted as the outcome of that execution. For this scheme, assumptions are needed that faults are so rare and the voting mechanism is free of faults to ensure that only correct states can be voted to be the outcome. Information contained in intermediate states before the voted outcome cannot be accessed outside this algorithm. Under these assumptions, we take the voting mechanism as the redundant part for providing error detection and error recovery: if faults occur in the execution, the correct resulting state is determined by the voting mechanism.

A recovery block scheme was proposed in [HLMR74] for backward recovery. A recovery block consists of a primary alternate which corresponds to the program performing the intended computation, an acceptance test, and one or more other alternates. When a recovery block is entered, the initial state is saved and then the primary alternate is executed. After the execution of an alternate completes, the acceptance test is executed to determine whether or not faults occur in the execution. If the test is passed, the recovery block is exited and any further alternates are ignored. A further alternate, if one exists, is entered if the preceding alternate fails to pass the test. Before an alternate is so entered, the system state is restored with the saved initial state. This technique is based on the assumption that the acceptance test cannot fail, and the storage used for saving the initial state will never be corrupted by any faults in the system. Recovery blocks must be atomic in the sense that intermediate information cannot be accessed outside the block.

The recovery block technique was extended to the conversation technique for dealing with error recovery in interacting processes [Ran75]. Work has been focused on backward error recovery in the conversation scheme [RT79, FGL82, Kim82].

Recovery blocks and conversations can be designed as atomic actions termed as *fault-tolerant atomic actions* [LRT78, CR86, JC86, SMR87]. Within the atomic action scheme, it has been informally shown that backward and forward recovery techniques can be used in one recovery block or one conversation [CR86, JC86].

As mentioned before, these existing techniques are described in non-mathematical terms and the assumption that the redundant components must be free of design faults cannot be rigorously reasoned about. Also, the application of each of the schemes is limited by the restrictions made for the computing system model and the program structures. In many cases, we cannot achieve fault-tolerance by using only backward or only forward recovery techniques in a program. For some applications, we cannot use only static or only dynamic recovery techniques because of the requirements of the program and the architecture of the hardware. We should not be forced to use only a multiple implementation scheme or only a recovery block-conversation scheme because the expense in storage or time. We need a framework which is formal so that the properties of fault-tolerant programs can be rigorously reasoned about and developed, and which makes it more feasible to decide which fault-tolerance techniques should be used, and where.

The choice of fault-tolerance strategy must be based upon the properties of the faults that we know and the properties of the program defined by its specification. Properties of faults include the way that a fault can affect the execution of the program and if necessary, the probability of the occurrence of a fault. For instance, in principle, backward error recovery can provide fault-tolerance for any program, but forward error recovery can be used only when the error state can be identified and remaining information in the erroneous state can be used. In practice, however, backward error recovery cannot be used in a program which interacts with a changing environment that cannot be turned back under any circumstances. Application of backward error recovery is also restricted by the amount of resources of the system, such as storage, and the time constraints of the software. Therefore, the choice between backward and forward error recovery in a program must be based upon the properties of the faults and the program.

1.4 Failure Semantics and Fault-tolerant Refinement

The failure semantics of a program defines the behaviour of a program when it runs on a fault-prone hardware system. In a semantic model, such a failure semantics of a program is obtained from the semantics of the program and the semantics of the actions representing the faults of the system. Specification and verification techniques for reasoning about the failure behaviours are also required.

It is traditionally taken to be an extremely hard task to define the precise effect of a fault on the program. In this thesis, we will carry out this task following the rule: never under estimate the damage made by a fault. When an error occurs, the worst case is when the state could be anything but a "good" state. Any information that we do not know exactly in an error state could be anything but "good" information. Based on the failure semantics of the program which is so defined, a fault transformation on the program transforms the program into a fault-affected version, the fault-free semantics of which equals to the failure semantics of the original program.

A program can be developed from its specification stage by stage, by refinement transformations [Bac78, Bac88, Bac89, Mor90, Lam83, AL88], until it can be executed on the hardware. While the development of the program proceeds from a higher level to a lower level, more is known about the faults at the lower level than at the higher level, because more information can be introduced about the use of the resources of the system by the program. Such changes of the fault properties are described by the refinement of the fault-affected version derived by the refinement step corresponding to the development step of the original program. Hence, the choice of the fault-tolerance strategy for a part of the program may not be made at any one stage, and if it has been made, it may need to be remade at the next stage, e.g. backward error recovery could be changed to forward error recovery or *vice versa*. This leads to the transformational approach developed in this thesis for dealing with faults and fault-tolerance.

Consider a program with a defined failure semantics. The failure semantics defines the class of faults of the system and it is responsibility of the designer of the system to guarantee that no other faults occur. For the given failure semantics of a program, the designer of the program has to transform the program into a program which tolerates the specified faults by adding redundant components. The failure semantics of transformed program has to be proved to satisfy the specification of the original program. Intuitively, a fault-tolerant program obtained by such a transformation must be a refinement of the original program. This is because (1) if no fault occurs in an execution of the fault-tolerant program, the execution must satisfy the specification of the original program and, (2) if faults have occurred in an execution, it also satisfies the specification. The first condition implies that the transformation described above is a refinement, and the second condition says that not every refinement can make a program fault-tolerant. A transformation satisfying these two conditions is termed *fault-tolerant refinement*.

The failure semantic model provides a formal way for classifying the faults of a system and defining assumptions about the faults. Intuitively, the more restrictive the class of faults [Cri90], or the stronger the assumptions about the faults [Lam84], the better is the failure semantics of the program required, and consequently the simpler and cheaper it is to achieve the fault-tolerance. The worst case is when a program is of a failure semantics that the execution of the program may fail at any time with arbitrary malicious behaviour. A better case is when the program may fail by stopping but without damaging any storage of the system. The best case is that the failure semantics of the program is the same as its semantics and hence no fault-tolerance is needed.

Given a program with a specified failure semantics, it sometimes can be refined so that the refined version has a better failure semantics. Such a refinement must be done by making a part of the program fault-tolerant. For example, consider a program P consisting of a set of processes with a single shared resource. Assume that any time the resource can be used by only one process to execute code in its critical section. Assume that a process may fail and destroy the resource. By duplicating the code in the the critical section of each process, we may be able to refine P to a program P' to ensure that no process will fail during the execution of its critical section, or at least that the shared resource will not be destroyed.

The failure semantic model is also useful for stepwise refinement and modular design of reactive systems. Consider a given part P_1 of a reactive program. We want to design another part P_2 so that the program composed of P_1 and P_2 satisfies the given specification of the reactive system. If we are also given the failure semantics of P_1, we can try to design the other part to be P_2' such that the failure semantics of the composite program is better than that of the composite program of P_1 and P_2. Again consider a program P_1 consisting a set of processes with a single shared resource. We want to design a program P_2 such that the composed program of P_1 and P_2 guarantees: (1) the resource can be used by only one process at any time and, (2) if any process using the resource will eventually release it, then any process requesting the resource will eventually get it. If P_2 is designed based on the semantics but not on the failure semantics of P_1, the composite program of P_1 and P_2 may fail and a failed process in P_1 may be allowed to get the resource even though P_2 is free from faults. This may cause starvation in the processes which have not failed. Based on the failure semantics of P_1, we can design a program P_2' such that no failed process can be allowed get the resource. More detailed consideration of this problem will be presented in Chapter 5. Therefore, this framework is feasible to use and helpful to the designers at different level of a system to communicate with each other and make decisions together.

1.5 Fault-tolerance and Atomic Actions

Fault-tolerant properties are often discussed in terms of atomic actions. The ideas behind atomic actions can be traced back at least to Floyd's seminal paper [Flo67] in which he characterized programs by their input/output relations. As a programming construct, an atomic action is a piece of program that behaves as a simple "primitive" with regard to its environment, while it may possess a "complicated" internal structure [Lom77, Kim82, JC86, Bac88]. During the execution of an atomic action, intermediate states will never be observed by the computation outside the atomic action. In other words, during the execution of an atomic action, variables whose values are read by steps of the atomic action can be modified only by other steps inside the atomic action.

Even when faults are not taken into account, consideration of concurrency is crucial for the concept of atomicity of actions. Two concurrent processes with a shared variable can interact in such a way that intermediate state of variables within one process are observed by the other. If one process modifies the shared variable while the other is being executed, the behaviour of the first can be affected. Similarly, a process that halts because of a failure will leave intermediate states of variables exposed outside the atomic action.

If both concurrency and fault-tolerance are to be addressed, the atomicity of an action means that intermediate states, which may be correct or erroneous, cannot be observed by computations outside the action. Error states should be corrected within the action to prevent error propagation from the computation of this action to that outside the action. These requirements for atomic actions were studied in [Gra78, Ree83, Tay86, HW87]. In this thesis, we will formalize these requirements to be the properties to be preserved by the transformations in our framework.

1.6 Reasoning About Fault-tolerant Programs

As mentioned earlier, the concept of an atomic action is related to the so-called inductive assertion method for proving properties of programs [Flo67]. In this technique, an initial state and a final state can be associated with the flow of control of the system as it enters and leaves each component of an atomic action. An assertion called a *precondition* and an assertion called a *postcondition* are associated respectively with the initial and final states and can be used to specify the computation carried out by the component. The pre- and postconditions of the components of an atomic action constitute a decomposition of the specification (i.e. the pre- and postconditions) of the atomic action. And the specifications of the atomic actions of a system constitute the specification of the system.

The technique for reasoning about programs using pre- and postconditions, which was firstly well-established for sequential programs, has been generalized to concurrent programs using atomic actions [OG76, AFR80, LG81, CM88, Bac89]. The safety properties of a concurrent program are specified by assertions which are preserved by every action of the program. Liveness (progress) properties are specified as the pre- and postconditions of some actions and proved by using certain fairness conditions that guarantee that these actions will be eventually executed. Therefore, the correctness and reliability of a system depends upon the correctness and reliability of each of its actions, and the correctness and reliability of an action depends upon the correctness and reliability of each of its components. The pre- and postcondition technique and the weakest precondition calculus developed from this technique were used for the development of a refinement calculus of programs [BW89, Bac89, Mor90].

Refinement of programs provides a useful way for constructing large and complex programs. Using this method, an original high level program specification is transformed by a sequence of correctness preserving transformations into an executable and efficient program.

In this thesis, we will use the technique for reasoning about fault-tolerant programs by using pre- and postconditions of atomic actions. As mentioned before, refinement framework developed based on this technique will be adapted and extended for developing a fault-tolerant program. The refinement of a program changes the way in which the execution of the program uses the system resources. Along with each refinement step of the program, more information about the use of the system resources can be known. When system faults are concerned, such information is helpful for analyzing and reasoning about the effect of system faults on the execution of the program. We will take advantage of the refinement framework and apply it to fault-tolerant programs by taking fault-tolerant properties as a kind of correctness to be preserved by the refinement transformations. We believe that, using this approach, we can ease the difficulties in developing programs which may be already rather complex and which become even more complex because of the requirements of fault-tolerance.

1.7 Organization of the Thesis

Chapter 2 uses UNITY framework [CM88] to describe the general problems involved in the design of fault-tolerant programs. It is shown that, in this chapter, system faults with various assumptions can be modelled as fault actions which simulate the activity of the fault environment. The interference of the fault environment with the execution of a program P designed for a fault-free system is then described as a *fault transformation* $\mathcal{F}$ which transforms P into a program $\mathcal{F}(P)$. Each execution step of $\mathcal{F}(P)$ is either an execution of an action of P or a fault action of the fault environment. Based on this way of modelling the faults, the fault properties of the program P can be formally reasoned by defining the properties of $\mathcal{F}(P)$. Fault-tolerant properties are introduced to a program P by applying a *recovery* *transformation* $\mathcal{R}$ to P and obtain a program $\mathcal{R}(P)$. The general requirements and properties of fault-tolerant programs are then defined and discussed by reasoning about the program $\mathcal{F}(\mathcal{R}(P))$.

In Chapter 3, based on the extended action system formalism [Bac87] and its semantic model, the *failure semantics* of a program with respect to a specified set of fault actions is defined from the semantics of the program and the semantics of the fault actions. The fault transformation and recovery transformation introduced in Chapter 2 are extended to programs written as action systems. This allows us to refine the fault properties and fault-tolerant properties of a program along with the refinement of the functional properties. The introduction of general ideas for presenting the effect of faults along with refinement steps and the definition of *fault-tolerant refinement* is one of our main objectives in this chapter. The another one is to show how we can refine a program to improve its failure semantics. General refinement rules and sequential refinement rules are also presented.

Chapter 4 extends the ideas and methods of Chapter 3 to parallel fault-tolerant programs and uses these methods to deal with various aspects of fault-tolerant programs. These include using atomicity or conversion techniques for fault-tolerance, dynamic checkpointing and recovery techniques [Woo81, Had82, KT87].

Chapter 5 demonstrates, by considering an example of a resource allocation problem, the application of the transformational approach developed in the research of this thesis to stepwise refinement and modular design of fault-tolerant reactive systems. A special case of the problem of solving conflicts among processes in distributed systems in the presence of faults of the processes is considered. Fault-tolerant broadcasts [SG84] and the *Byzantine General problem* [LSP82] are involved in this example. Rather than providing new solutions for the *Byzantine General problem*, it is shown that the existing solutions can be understood and used in a new way.

Chapter 6 summarizes the results of the research presented in this thesis. It also considers the related formal approaches to fault-tolerance, and topics for further research.

Chapter 2

A Theory for Fault-tolerance

2.1 Introduction

In this chapter we use the UNITY framework [CM88] to describe the problems involved in the design of a fault-tolerant program. Our objective is to solve these problems by using transformations to construct a fault-tolerant program from a program constructed for a fault-free system.

Let a program P be described as a set of atomic actions: assume that if an action is chosen for execution, it will be executed without interference from other actions in the program. Suppose that a program P *satisfies* a specification Sp when it is executed on a *fault-free* system; informally, a program is said to satisfy a specification if every property defined by the specification is a property of any execution of the program or, equivalently, no execution of the program violates any property defined by the specification. Now consider the execution of program P on a system which may exhibit faults. Let the effect of each physical fault on P be modelled as a *fault action* which transforms a *good* state into an *error* state leading to a condition which violates the specification Sp. Physical faults appear therefore as actions of a *fault program* F which models the *fault environment* in which P is executed. F is assumed not to change an error state into a good state. Let a fault at any point during the execution of P take it to an error state in which a boolean variable f is $true$.

The interference by the fault environment F on the execution of program P can then be defined as a *fault transformation* $\mathcal{F}$ which transforms program P into the program

$$\mathcal{F}(P) = P[]F$$

where the *union composition* $P[]F$ is defined as the union of the sets of actions of P and F [CM88]. The execution of a program P in the fault environment F is then equivalent to the execution of $P[]F$ in a *fault-free* environment.

The behaviour of $\mathcal{F}(P)$ will usually not satisfy Sp. But by applying a *fault-tolerant transformation* $\mathcal{T}$, it may be possible that $\mathcal{F}(\mathcal{T}(P))$ satisfies Sp. Unfortunately, this cannot always be achieved, though $\mathcal{F}(\mathcal{T}(P))$ may be shown to satisfy a weaker but still acceptable specification. One kind of fault-tolerant transformation is a *recovery transformation*.

Let P have an initial execution sequence $[S_0]A_1[S_1]A_2\ldots A_m[S_m]\ldots A_n[S_n]$ in which each action A_i transforms state S_{i-1} into state S_i. Assume that this sequence ends in state S_n because of failure, i.e. because a fault has occurred and transformed the good state S_n into an error state. Let S be a state which is the union of the disjoint substates each of which is a substate of one of the states $S_m,\ldots,S_n$. In general, such substates representing partial information (e.g. local states of processes) have been saved at some of the execution points $S_m,\ldots,S_n$, and will be used to calculate a global state S for recovery after the occurrence of faults. If the execution of P can be restarted from state S and still satisfies Sp, then the state S is said to be *backward consistent* with the interrupted execution sequence. Alternatively, the execution of P can be continued from a *possible future* state S_{n+k}. If there exists an execution sequence $[S_0]A_1[S_1]A_2\ldots A_m[S_m]\ldots A_n[S_n]A_{n+1}[S_{n+1}]\ldots A_{n+k}[S_{n+k}]\ldots$ of P which satisfies Sp, then S_{n+k} is said to be *forward consistent* with the interrupted execution sequence. A *consistent* state of an execution sequence is a state which is either backward or forward consistent with the execution sequence (see Section 2.4).

If an error state in the execution of $\mathcal{F}(P)$ can be transformed into a good state which is consistent with the interrupted execution of P, then the execution of P can resume. This transformation can be performed by adding a set P_R of *recovery actions*, called a *recovery program*. The recovery actions of P_R are enabled only when a fault occurs and are assumed not to be affected by the fault environment F, i.e. no fault occurs during the execution of a recovery action. The desired effect of the recovery program P_R can be therefore viewed as arising from the application of a transformation $\mathcal{R}$ on P such that the program

$$\mathcal{F}(\mathcal{R}(P)) = P[]P_R[]F$$

satisfies an acceptable specification though it may not behave exactly as P.

The programs P, F and P_R can be described as UNITY programs. On the one hand, they are programs which are nondeterministic, with no restrictions on either *when*, *where* or *how* program F interferes P or P recovers from errors. On the another hand, they specify clearly *what* they achieve in their executions. They may contain unbounded nondeterminism and may therefore not be executable. This allows us to specify at a high level how programs are to behave and to separate the logical behaviour from the implementation on a specific architecture [CM88, Bac88, BS89].

After a summary of the basic features of UNITY in Section 2.2, Section 2.3 defines the effect of faults as a fault transformation. Following a formal definition of *consistency* in Section 2.4, the *recovery transformation* is defined in Section 2.5, and it is shown to apply to both *backward* and *forward* recovery. Section 2.6 discusses the fault-tolerant properties of the fault-tolerant program $\mathcal{R}(P)$. Finally Section 2.7 discusses these results.

2.2 Summary of UNITY

The UNITY system (Chandy and Misra [CM88]) provides the foundation for an approach to designing parallel programs systematically for a variety of architectures and applications. The approach concentrates on the stepwise refinement of specifications, with the production of code left until the final stages of design. Hence the UNITY system involves a model of computation, a notation for specification and a theory for proving the correctness of specifications.

This section presents an outline of the basic features and notation of UNITY, sufficient for our use in this chapter; some of the notation and the assumptions of UNITY in [CM88] have been slightly changed so that they can be more easily used for our description. We also present a formal semantics for UNITY which differs in some details from the informal account in [CM88].

2.2.1 The UNITY Notation

The following subset of the UNITY notation will be used in the later part of this chapter.

Primitive commands

A primitive command is a conditional multiple assignment of the following form:

$$\begin{aligned}
x_1, \ldots, x_m := {} & e_{11}, \ldots, e_{1m} \ \textbf{if}\ b_1 \\
\sim {} & e_{21}, \ldots, e_{2m} \ \textbf{if}\ b_2 \\
\sim {} & \vdots \\
\sim {} & e_{n1}, \ldots, e_{nm} \ \textbf{if}\ b_n,
\end{aligned}$$

where each e_{ij} is a simple expression and b_k is boolean expression.

Thus, a multiple assignment consists of a variable list on the left side and simple expression lists with corresponding boolean expressions on the right side. If there is more than one component simple expression list whose associated boolean expressions is $true$, any one of them can be chosen for assignment; if no boolean expression is $true$, the variable values are left unchanged.

Statements

Primitive commands and parallel composition of primitive commands, $c_1 \parallel \ldots \parallel c_n$, where $n > 1$, are UNITY statements. In the parallel composition $c_1 \parallel \ldots \parallel c_n$, the variables on the left side of the assignments are required to be different. We may use the term $action$ wherever the term $statement$ is used.

Programs

A UNITY program has a *program-name*, a *declare-section* which declares the types of the program variables, an *initially-section* which defines the initial values of some variables in this program and an *assign-section* which is a statement list, $S_1 [] \ldots [] S_n$.

Since the types of variables will not be considered in the semantic model that will be presented in Section 2.2.3, the *declare-section* is not discussed here any further. The initially-section is a set of equations which is *proper* in the sense that it can be used to determine the values of variables named in the program. Here, the initially-section will generally be the conjunction of a set of predicates about variables in the program. Therefore, a UNITY program can be abstractly denoted as

$$P = (Init_P, Act_P),$$

where $Init_P$ is a set of predicates over the initial values of the variables named in P, and Act_P is a set of actions (statements) which define the assign-section of the program P. We always assume that the conjunction of the predicates in $Init_P$ is not $false$.

Example 2.2.1 An example of the UNITY syntax

The declare-section declares the type of the variables x and y to be integer. The initially-section, $x = 0$, $y = 0$, defines the initial values of the variables x and y as 0. The assign-section consists two actions, the first of which increases x by 1 if $x \leq 3 \times (y + 1)$ holds and the second of which increases y by 1 if $x \geq 3 \times y$.

Program $Exp1$
  declare
    x, y: $integer$
  initially
    $x = 0, y = 0$
  actions
    $x := x + 1$ **if** $x \leq 3 \times (y + 1)$
    $[]$
    $y := y + 1$ **if** $x \geq 3 \times y$
End $\{Exp1\}$

There are two different kinds of composition of programs. The *union* $P_1 [] P_2$ of two programs P_1 and P_2 is obtained by combining their statements,

$$P_1 [] P_2 \triangleq (Init_{P_1} \cup Init_{P_2}, Act_{P_1} \cup Act_{P_2})$$

For the union composition $P_1 [] P_2$, it is required that the conjunction of the predicates in $Init_{P_1} \cup Init_{P_2}$ is not $false$.

Unlike union, *superposition* describes modifications of a given program by adding new variables and assignments, but not altering existing variables. Thus, superposition preserves all the properties of the original program.

2.2.2 UNITY Logic

We use Q, Q', R , R' to denote arbitrary predicates, P to denote a program and A to denote a statement of a program. We adopt the convention that a program P with $Act_P = \emptyset$ is equivalent to a program P_1 with only one statement which does not change the program state, such as $x := x$. We denote a program P with $Act_P = \emptyset$ as $\emptyset$.

As in Hoare logic [Hoa69, Dij76], the specification[1] $\{Q\}A\{R\}$ defines a statement A which when executed in a state satisfying predicate Q terminates in a state satisfying predicate R. Execution of any statement of a program is assumed to terminate. A is universally or existentially quantified over the statements Act_P; a property which holds for all points of the execution of a program P is defined using universal quantification while a property which holds eventually is defined using existential quantification.

[1]This is Hoare logic for total correctness —the original proposal in [Hoa69] used the notation $Q\{A\}R$ and dealt with partial correctness.

The logical operators **unless**, **stable**, **invariant**, **ensures**, **leads-to** ($\longmapsto$) are used to describe the safety and progress properties of programs. Following UNITY, these operators are defined for a program P in terms of the $\underline{Sat}$ relation for the formulas listed below.

- $P\ \underline{Sat}\ (Q\ \textbf{unless}\ R) \;\Leftrightarrow\; \forall A : A \in Act_P :: \{Q \wedge \neg R\}A\{Q \vee R\}$
- $P\ \underline{Sat}\ (\textbf{stable}\ Q) \;\Leftrightarrow\; P\ \underline{Sat}\ (Q\ \textbf{unless}\ false)$
- $P\ \underline{Sat}\ (\textbf{invariant}\ Q) \;\Leftrightarrow\; (Init_P \Rightarrow Q) \wedge P\ \underline{Sat}\ (\textbf{stable}\ Q)$
- $P\ \underline{Sat}\ (Q\ \textbf{ensures}\ R) \;\Leftrightarrow\; P\ \underline{Sat}\ (Q\ \textbf{unless}\ R)\ \wedge (\exists A : A \in Act_P :: \{Q \wedge \neg R\}A\{R\})$
- Program P satisfies Q *leads-to* R, $P\ \underline{Sat}\ (Q \longmapsto R)$, iff it can be derived by a finite number of applications of the following three inference rules:

$$\frac{P\ \underline{Sat}\ (Q\ \boldsymbol{ensures}\ R)}{P\ \underline{Sat}\ (Q \longmapsto R)}, \qquad \frac{P\ \underline{Sat}\ (Q \longmapsto R),\ \ P\ \underline{Sat}\ (R \longmapsto R')}{P\ \underline{Sat}\ (Q \longmapsto R')},$$

  and for any set W:

$$\frac{(\forall m : m \in W :: P\ \underline{Sat}\ (Q(m) \longmapsto R))}{P\ \underline{Sat}\ ((\exists m : m \in W :: Q(m)) \longmapsto R)}.$$

A fixed point of a program is a program state such that execution of any statement of the program in that state leaves the state unchanged. The execution of the program *terminates* if it reaches it fixed point. A predicate ($Fixedpoint(P)$) defining such a state can be deduced from the text of program P. $Fixedpoint(P)$ is defined as follows: for each assignment in P, the left and right sides of the assignment are equal in the value, i.e. if $Assign_P$ is the set of assignments with simple expressions occurring in P,

$$Fixedpoint(P) \triangleq \forall A \in Assign_P : A = (X := E) :: gA \Rightarrow X = E$$

where gA is the boolean condition of A.

Example 2.2.2 The program $Exp1$ in Example 2.2.1 has the following properties which constitute a specification of $Exp1$:

$Sp[Exp1]$:

$$Exp1 \ \underline{Sat} \ \textbf{invariant} \ (3 \times (y-1) \leq x \leq 3 \times y + 4)$$

$$Exp1 \ \underline{Sat} \ \textbf{stable} \ (x \geq k)$$

$$Exp1 \ \underline{Sat} \ \textbf{stable} \ (y \geq k)$$

$$Exp1 \ \underline{Sat} \ (x = k \longmapsto x = k + 1)$$

$$Exp1 \ \underline{Sat} \ (y = k \longmapsto y = k + 1)$$

A set of properties of the union composition [] will be used in this chapter. We list below these properties, without presenting the proofs which can be found in [CM88].

Theorem 2.2.1 (The Union Theorem)

1. $P_1 [] P_2 \ \underline{Sat} \ (Q \ \textbf{unless} \ R) \Leftrightarrow$
$(P_1 \ \underline{Sat} \ (Q \ \textbf{unless} \ R)) \wedge (P_2 \ \underline{Sat} \ (Q \ \textbf{unless} \ R))$

2. $P_1 [] P_2 \ \underline{Sat} \ (Q \ \textbf{ensures} \ R) \Leftrightarrow$
$(P_1 \ \underline{Sat} \ (Q \ \textbf{ensures} \ R)) \wedge (P_2 \ \underline{Sat} \ (Q \ \textbf{unless} \ R)) \vee$
$(P_2 \ \underline{Sat} \ (Q \ \textbf{ensures} \ R)) \wedge (P_1 \ \underline{Sat} \ (Q \ \textbf{unless} \ R))$

Corollary 2.2.1

1. $(P_1 [] P_2 \ \underline{Sat} \ \textbf{stable} \ Q) \Leftrightarrow ((P_1 \ \underline{Sat} \ \textbf{stable} \ Q) \wedge (P_2 \ \underline{Sat} \ \textbf{stable} \ Q))$

2. $$\frac{P_1 \ \underline{Sat} \ (Q \ \textbf{unless} \ R) \ , \ P_2 \ \underline{Sat} \ \textbf{stable} \ Q}{P_1 [] P_2 \ \underline{Sat} \ (Q \ \textbf{unless} \ R)}$$

3. $$\frac{P_1 \ \underline{Sat} \ \textbf{invariant} \ Q, \ P_2 \ \underline{Sat} \ \textbf{stable} \ Q}{P_1 [] P_2 \ \underline{Sat} \ \textbf{invariant} \ Q}$$

4. $$\frac{P_1 \ \underline{Sat} \ (Q \ \textbf{ensures} \ R) \ , \ P_2 \ \underline{Sat} \ \textbf{stable} \ Q}{P_1 [] P_2 \ \underline{Sat} \ (Q \ \textbf{ensures} \ R)}$$

5. $$\frac{P_1 \ \underline{Sat} \ (Q \ \textbf{ensures} \ \neg Q)}{P_1 [] P_2 \ \underline{Sat} \ (Q \ \textbf{ensures} \ \neg Q)}$$

6. $$\frac{P_1 \ \underline{Sat} \ (Q \longmapsto R) \ , \ P_2 \ \underline{Sat} \ \textbf{stable} \ Q}{P_1 [] P_2 \ \underline{Sat} \ (Q \ \longmapsto \ R)}$$

7. $$\frac{P_1 \ \underline{Sat}\ (Q \longmapsto R)\ ,\ P_2 \ \underline{Sat}\ (Q \longmapsto R)}{P_1 [] P_2 \ \underline{Sat}\ (Q \ \longmapsto \ R)}$$

2.2.3 A Semantics of UNITY

We use a simple semantics for UNITY programs in which each statement in a program is executed atomically: if a statement is chosen for execution, it is assumed to be executed without any interference from the other statements in the program. Because of this atomicity, a program can be viewed as being sequential and nondeterministic.

A *state* of a program P is a function from the program variables $Var(P)$ to their value space. For a set of variables $Y \subseteq Var(P)$, a *sub-state* $S|_Y$ of state S is the restriction of S to Y. Ψ_P is used to denote the set of all the states of program P.

The semantics of a statement $A \in Act_P$ is a function:

$$\Gamma(A): \ \Psi_P \rightarrow \mathbb{P}(\Psi_P)$$

This function can be extended to be a function over the powerset $\mathbb{P}(\Psi_P)$, i.e. for a set Ψ of states

$$\Gamma(A)(\Psi) = \bigcup_{S \in \Psi} \Gamma(A)(S)$$

Semantics of primitive commands

Let A be a primitive command of the following form:

$$\begin{array}{ll} x_1,\ldots,x_m := & e_{11},\ldots,e_{1m} \text{ if } b_1 \\ & \sim e_{21},\ldots,e_{2m} \text{ if } b_2 \\ & \sim \vdots \\ & \sim e_{n1},\ldots,e_{nm} \ \textbf{if } b_n \end{array}$$

Let $bb \triangleq b_1 \vee b_2 \vee \ldots \vee b_n$,

$$\Gamma(A)(S) \triangleq$$
$$\begin{cases} \{S[x_1/e_{i1}(S),\ldots,x_m/e_{im}(S)] \mid 1 \leq i \leq n \wedge b_i(S) = true\} & \text{if } bb(S) = true \\ \{S\} & \text{if } bb(S) = false \end{cases}$$

where $S[x_1/e_{i1}(S), \ldots, x_m/e_{im}(S)]$ is the state in which the value of each x_j is $e_{ij}(S)$ and the value of any variable other than the x_j ($1 \leq j \leq n$) remains the same as the value in S.

We will not formally define the semantics $e_{ij}(S)$ of expressions e_{ij} in a state S, leaving the reader to intuitively treat this as the evaluation of the expressions.

Semantics of statements

Let $A = A_1 \parallel A_2$, where A_1 and A_2 are primitive commands.

$$\Gamma(A)(S) \triangleq \Gamma(A_1)(S) \cap \Gamma(A_2)(S)$$

Sequence notation

For the definition of the semantics of programs, we shall use the following notation for sequences. Let V be a set of elements and $V^\dagger$ be the set of all the finite and infinite sequences over V and σ, σ' be sequences in $V^\dagger$:

1. $<>$ is the empty sequence;
2. $< a, \ldots, b >$ is the sequence of elements $a, \ldots, b$;
3. $\hat{}$ concatenates two sequences:

$$< a > \hat{}< b >=< a, b >, \quad < a > \hat{}<>=< a >$$

4. $\sigma' \underline{\propto} \sigma$ is $true$ if σ' is a prefix of σ;
5. $\sigma' \propto \sigma$ is $true$ if $\sigma' \underline{\propto} \sigma$ and $\sigma' \neq \sigma$;
6. $\#\sigma$ is the length of the sequence σ and $\#\sigma = \infty$ if σ is infinite;
7. $\sigma(i-1)$ is the ith element of σ, $1 \leq i \leq \#\sigma$;
8. $head(\sigma)$ and $last(\sigma)$ denote respectively the first and last elements of a non-empty sequence σ (for a finite sequence σ);
9. $rest(\sigma)$ denotes the sequence obtained from σ by removing the first element of a non-empty sequence σ:

$$\sigma =< head(\sigma) > \hat{} rest(\sigma)$$

10. $front(\sigma)$ denotes the sequence obtained from σ by removing the last element of a non-empty finite sequence σ:

$$\sigma = front(\sigma)\,\hat{}\,< last(\sigma) >$$

11. $e \in \sigma$ denotes that e is an element of σ;

12. $\sigma' \subseteq \sigma$ denotes that each element of σ' is an element of σ;

13. $head(k, \sigma)$ is the sequence of the first k elements of σ;

14. if σ' is a prefix of σ, then $\sigma - \sigma'$ is the sequence obtained from σ by removing the prefix σ'.

Semantics of programs

An *execution* of a program P is a sequence of states each of which, except the first (which is determined by the initial condition $Init_P$), is produced from the previous state by executing one of the statements in Act_P.

If V is the value space of program P, an *observation* θ of P is a sequence of states which is defined as the function $\theta : \ Var(P) \longrightarrow V^{\dagger}$ satisfying the following conditions:

OB1. $\theta(x) \neq <>$, for any $x \in \ Var(P)$,

OB2. $\#\theta(x) = \#\theta(y)$ (denoted by $\#\theta$) for any $x, y \in \ Var(P)$,

OB3. for each $i < \#\theta$, $\theta[i]$ is a state of P defined as $\theta[i](x) = \theta(x)(i)$ for any $x \in \ Var(P)$,

OB4. $\theta[0]$ is an initial state of P,

OB5. $\forall\, i : 0 < i < \#\theta :: (\theta[i] = \theta[i-1]) \vee (\exists\, A \in Act_P : \theta[i] \in \Gamma(A)(\theta[i-1]))$

We use the sequence notation to describe a property of a function from $Var(P)$ to the sequences $V^{\dagger}$, e.g. for the functions θ and θ', $\theta\,\hat{}\,\theta'$ is the function such that $\theta\,\hat{}\,\theta'(x) = \theta(x)\,\hat{}\,\theta'(x)$ for each $x \in \ Var_P$.

A function $\beta : \ Var(P) \longrightarrow V^{\dagger}$ satisfying Condition **OB2** is said to be a sub-observation of θ, denoted $\beta \ll \theta$, if there is an observation θ' and a function $\gamma : \ Var(P) \longrightarrow V^{\dagger}$ satisfying Condition **OB2** such that

$$\theta = \theta'\,\hat{}\,\beta\,\hat{}\,\gamma$$

Obviously, given a statement A, $\Gamma(A)$ can be extended to be a function over the set OB_P of the finite observations of P, i.e.

$$\Gamma(A)(\theta) = \{\theta \,\hat{}\, S \mid S \in \Gamma(A)(last(\theta))\}$$

An *execution* E of P is an infinite observation of P. It is said to exhibit *fairness* (or *justice* [MP83, GP89]) if for any $i \geq 0$ and $A \in Act_P$, there is some $k \geq i$ such that $E[k+1] \in \Gamma(A)(E[k])$. The semantics of a program P is the set $\Gamma(P)$ of its fair executions. Two programs P and P' are said to be *equivalent*, denoted $P \equiv P'$, if they have the same semantics,

$$P \equiv P' \triangleq \Gamma(P) = \Gamma(P')$$

Within this semantic model, the following equivalences hold for a program P:

$$(Init_P \Rightarrow Fixedpoint(P)) \Rightarrow (P \equiv \textbf{skip})$$

and,

$$P[]\textbf{skip} \equiv P$$

where **skip** denotes the program which never changes the values of the program variables. We also use **skip** to denote a command or an action which does not change the values of the program variables.

This semantic model permits *finite stuttering*, i.e. in any execution, a state can be repeated consecutively at most a finite number of times before the execution reaches the fixed point of P. The stuttering property is required when dealing with the refinement of reactive programs [Lam83, AL88, Bac89].

2.3 Modelling Faults as Transformations on Programs

Let P be a program satisfying the specification Sp when executed on a fault-free system. If P is executed on a fault-prone system then, in general, the execution will not satisfy Sp. The physical faults of the system interfere with the execution at state S in such a way that either the execution of a statement A in Act_P correctly terminates with a $good$ state or a *fault action* transforms S into an *error state* which leeds to a state which violates the specification Sp.

2.3.1 The Fault Transformation

Assume that the presence or absence of faults is represented by the value of a boolean variable f which cannot be changed by P, i.e. $f = true$ when a fault occurs. The transformation by a fault action can therefore be modelled as a statement A which is atomic and satisfies

$$\{true\}A\{f\}$$

The interference by all the physical faults of the system can be composed as a program F which defines the set of fault actions representing the fault environment. The effect on the original program P of the fault environment F is described as a transformation $\mathcal{F}$:

$$\mathcal{F}(P, F) = P[]F$$

We shall write this as

$$\mathcal{F}(P) \triangleq P[]F$$

when there is no confusion about the fault environment F. And $\mathcal{F}$ is compositional, in the sense that if $P = P_1[]P_2$,

$$\mathcal{F}(P) \equiv \mathcal{F}(P_1)[]\mathcal{F}(P_2)$$

$\mathcal{F}(P, F)$ is called the *fault affected version* of P $w.r.t.$ F. Initially, f is $false$ and its value is never changed by program P. Execution of the fault environment program F can make f *true*, but not subsequently *false*. Therefore, each execution step of the fault affected program $\mathcal{F}(P, F)$ is either a correct execution of a statement of the original program P or a fault transformation of states made by the fault environment F. Whenever such a fault transformation occurs, execution stays in the error state in which f is $true$ but statements in both P and F may continue to be executed in error states and change the value of such a state.

More formally, a state S of $\mathcal{F}(P)$ is a *good* state if $f(S) = false$ and an *error* state otherwise. The effect of the fault transformation $\mathcal{F}$ is described in terms of two functions $good$ and $error$. Given any observation θ of $\mathcal{F}(P)$, $\theta = good(\theta)^\frown error(\theta)$ where $good(\theta)$ is an observation of P containing only good states, i.e. $good(\theta)[i]$ is a good state for $i < \#good(\theta)$, and $error(\theta)$ is either empty or contains only error states. Obviously, if no fault occurs in an observation θ of $\mathcal{F}(P)$, it is then an observation of P and $\mathcal{F}(P)$ satisfies the condition:

$$\forall \theta \in OB_{\mathcal{F}(P)} : (error(\theta) =<>\Rightarrow \theta \in OB_P)$$

If there are no faults, $\mathcal{F}(P)$ is equivalent to P:

$$F = \emptyset \Rightarrow (\mathcal{F}(P, F) \equiv P)$$

The following example shows a fault affected version of program $Exp1$.

Example 2.3.1 Assume that the faults may cause the value of x to be any natural number. Such a fault environment can be therefore simulated by a program F given as:

Program F
  initially
    $x = 0, f =$ *false*
  actions
    $[]_n : n \in \mathbb{N} ::\ x\ ,\ f := n\ ,$ *true*
End $\{F\}$

where $[]_n$ is used to combine all the actions within the scope of n, i.e. $n \in \mathbb{N}$

The fault affected version of $Exp1$ is then

$$\mathcal{F}(Exp1, F) = Exp1[]F$$

which is equivalent to the program:

Program $\mathcal{F}(Exp1)$
  initially
    $x = 0, y = 0, f =$ *false*
  actions
    $x := x + 1$ **if** $x \leq 3 \times (y + 1)$
   $[]$
    $y := y + 1$ **if** $x \geq 3 \times y$
   $[]$
    $[]_n : n \in \mathbb{N} ::\ x\ ,\ f := n\ ,$ *true*
End $\{\mathcal{F}(Exp1)\}$

2.3.2 Fail-stop Error Behaviour

In practice, it is usually not possible to design a fault-tolerant program without any constraints on the behaviour of both the original program P and the faults F in error states. For the implementation of fault-tolerant programs, a *fail-stop* assumption is often required [SS83] to guarantee that no further statements of P will be executed after a fault occurs.

To make P fail-stop, it must first be transformed. To allow this to be done, we introduce a syntactic change to the UNITY notation. Given a boolean expression b and a primitive command A of the following form:

$$\begin{aligned} x_1,\ldots,x_m := \; & e_{11},\ldots,e_{1m} \;\; \textbf{if } b_1 \\ & \sim e_{21},\ldots,e_{2m} \;\; \textbf{if } b_2 \\ & \sim \vdots \\ & \sim e_{n1},\ldots,e_{nm} \;\; \textbf{if } b_n, \end{aligned}$$

$b \to A$ is defined to be

$$\begin{aligned} x_1,\ldots,x_m := \; & e_{11},\ldots,e_{1m} \;\; \textbf{if } b_1 \wedge b \\ & \sim e_{21},\ldots,e_{2m} \;\; \textbf{if } b_2 \wedge b \\ & \sim \vdots \\ & \sim e_{n1},\ldots,e_{nm} \;\; \textbf{if } b_n \wedge b, \end{aligned}$$

and for a statement $A = A_1 \parallel A_2$,

$$b \to A \triangleq b \to A_1 \parallel b \to A_2$$

The f-*stop* action of an action A is the action $\neg f \to A$. The f-*stop transformation* $\mathcal{T}_f$ transforms program P to a program $\mathcal{T}_f(P)$ by transforming each $A \in Act_P$ into its f-stop action $\neg f \to A$. The f-*stop fault transformation* $\mathcal{F}_f$ is then defined as

$$\mathcal{F}_f(P, F) \triangleq \mathcal{F}(\mathcal{T}_f(P), F) \equiv \mathcal{T}_f(P)[]F$$

Because f cannot be changed by any action of P, we have

$$\mathcal{T}_f(P) \equiv P$$

and

$$(F = \emptyset) \Rightarrow \mathcal{F}_f(P, F) \equiv P$$

2.3.3 Finite Error Behaviour

The proof of progress properties for fault-tolerant programs must be based on some *finite error behaviour* assumption, i.e. faults can occur only finitely often in each execution of a program. In general, fault-tolerance cannot be achieved without the assumption that faults do not occur infinitely often. Such an assumption can be dealt with in the following two ways.

First, define the assumption as a fairness condition. Let $P = P_1[]P_2$. An execution E of P is said to be *fair* $w.r.t.$ P_1, if for each $i \geq 0$ and each statement $A \in Act_{P_1}$, there is some $k \geq i$ such that $E[k+1] \in \Gamma(A)(E[k])$. E is *bounded* $w.r.t.$ P_2 if there exists $i \geq 0$ such that

$$\forall\, k \geq i\, \exists\, A \in Act_{P_1} : E[k+1] \in \Gamma(A)(E[k])$$

The finite error behaviour assumption is then equivalent to saying that for programs P and P' and the fault environment F, $\mathcal{F}(P, F)$ (which is either $P[]F$ or $\mathcal{T}_f(P)[]F$) is bounded $w.r.t.$ F.

However, to use this fairness condition to prove progress properties for fault-tolerant programs, the UNITY logic would have to be extended to accommodate a bounded fairness condition, and this is far from trivial.

We thus choose an alternative way. The fault environment F is modelled as a program with a statement $bound := 1$ in Act_F. The value of $bound$ is assumed not to be changed by either the statements of the original program P or any statement of F other than the assignment $bound := 1$. The initial condition of F satisfies $Init_F \Rightarrow bound = 0$, and each statement in Act_F other than $bound := 1$ is of the form;

$$(bound = 0) \rightarrow A$$

where A satisfies $\{true\}A\{f\}$. Therefore, the fault actions in the fault affected program

$$\mathcal{F}(P, F) = P[]F$$

or its f-stop version

$$\mathcal{F}_f(P, F) = \mathcal{T}_f(P)[]F$$

can be *effectively* executed a finite but unbounded number of times in each execution of $\mathcal{F}(P, F)$.

The bounded error behaviour assumption means that faults can only occur less than k times in each execution, $k \in \mathbb{N}$. To meet this assumption, the fault environment F can be simulated in the following way. Initially, let $bound = 0$, i.e. $Init_F \Rightarrow (bound = 0)$. The variable cannot be changed by any statement of the original program P. Each statement in Act_F is of the form:

$$(bound < k) \rightarrow (A \parallel (bound := bound + 1))$$

where A satisfies $\{true\}A\{f\}$ and A does not modify $bound$.

Dealing with the finite error and bounded error behaviour assumptions in this way allows the fairness assumption in UNITY and the UNITY logic to be sufficient to specify progress properties of fault-tolerant programs.

2.3.4 Fault Transformations for Concurrent Processes

The fault transformation defined in the two previous subsections is based on the assumption that each fault in F may affect the execution of any statement in P. Often only some parts of a program can be affected by certain kinds of faults: e.g. the execution of a process of a program may only be affected by faults in the processor on which it is being executed.

In UNITY, no special notation is used for processes and communication channels: a process can be represented by a program and a channel by a variable which is shared by two processes and whose values satisfy the specification of some communication mechanism, e.g. synchronous or asynchronous communication. For implementation as a communicating system, a program can be partitioned into processes by refinement mapping, and a program P with n processes can be described as

$$P = p_1 [] \ldots [] p_n$$

This is one way to parallelize the execution of a UNITY program. Partitioning a program for parallel execution will be further discussed in Chapter 4 of this thesis.

For $i : 1 \leq i \leq n$, let the fault environment for process p_i, i.e. the physical faults that may occur while executing p_i, be specified as F_i with the boolean variable f_i as the fault indicator. If $F \triangleq F_1 [] \ldots [] F_n$, the fault transformation

$$\mathcal{F}(P, F) \triangleq \mathcal{F}(p_1, F_1)[] \ldots []\mathcal{F}(p_n, F_n)$$

and in its fail-stop version,

$$\mathcal{F}_f(P, F) \triangleq \mathcal{F}(\mathcal{T}_{f_1}(p_1), F_1)[] \ldots []\mathcal{F}(\mathcal{T}_{f_n}(p_n), F_n)$$

Example 2.3.2 The program $Exp1$ can be implemented by two processes p_1 and p_2, where *Exp1* $\equiv p_1[]p_2$ and

Program p_1
  initially
    $x = 0, u = 0$
  actions
    $x\ ,\ u := x + 1\ ,\ u + 1$ **if** $x \leq 3 \times (v + 1)$
End $\{p_1\}$

and

Program p_2
  initially
    $y = 0, v = 0$
  action
    $y\ ,\ v := y + 1\ ,\ v + 1$ **if** $u \geq 3 \times y$
End $\{p_2\}$

Let F be the faults on the processor on which p_1 is being executed, and let there be no faults on the processor on which p_2 is being executed. Apply the fault transformation to $p_1[]p_2$,

$$\mathcal{F}(p_1[]p_2, F) \equiv \mathcal{F}(p_1, F)[]p_2$$

where $\mathcal{F}(p_1, F)$ is given as

Program $\mathcal{F}(p_1, F)$
  initially
    $x = 0, u = 0, f =$ *false*
  actions
    $x\ ,\ u := x + 1\ ,\ u + 1$ **if** $x \leq 3 \times (v + 1)$
   $[]$
    $[]_n : n \in \mathbb{N} ::\ x\ ,\ f := n\ ,$ *true*
End $\{\mathcal{F}(p_1, F)\}$

The way in which that a fault action affects a part of a program will be discussed in more detail in Chapter 3.

2.4 Consistency and Reachability

To make the execution of P *recoverable* from an error state which violates the specification Sp due to the interference of F, the system has to be restored to a good state from which the interrupted execution of P can resume. Such a good state is described in terms of *consistency* with the interrupted execution of P.

Given a program P, for any state S of P let $Reach(S)$ be the set of states which are reachable from S by executing P,

$$Reach(S) \triangleq \{S' \mid \exists\, \theta, \theta' \in OB_P : S = last(\theta) \wedge S' = last(\theta') \wedge \theta \underline{\propto} \theta'\}$$

Let,

$$Reachable(S, S') \triangleq S' \in Reach(S)$$

and, as an abbreviation,

$$Reachable(S) \triangleq Reachable(init_P, S)$$

where $init_P$ is an initial state of P.

Let θ be a finite observation of P. S is a *possible future* state of P for θ if there is an observation of P which extends θ to S. Such a state S is said to be *forward consistent* (*ForwCon*) with θ:

$$ForwCon(S, \theta) \triangleq Reachable(last(\theta), S)$$

For a forward consistent state S of θ, a *forward recovered* observation of θ from S is an observation θ' such that

$$(\theta \underline{\propto} \theta') \wedge (last(\theta') = S)$$

Sometimes, only a subset of the variables $Var(P)$ need to be restored for recovery. For this case, and to provide more generality, we define forward consistency for substates. Let $\mathcal{X}$ be a set of disjoint subsets of $Var(P)$ and $\Psi = \{S_X \mid X \in \mathcal{X}\}$, where S_X is a substate over X; let $\overline{\mathcal{X}} = Var(P) - \cup\mathcal{X}$, where $\cup\mathcal{X}$ is the union of all the sets in $\mathcal{X}$. We say that Ψ is *forward-consistent* with θ, if there exists a state S such that:

$$(\forall X \in \mathcal{X} : (S|_X = S_X)) \wedge (S|_{\overline{\mathcal{X}}} = last(\theta)|_{\overline{\mathcal{X}}}) \wedge ForwCon(S, \theta)$$

We may also wish to consider a 'state' which is the union of sub-states previously reached at different points in this execution, provided that this 'state' could have been reached in some execution of P. The consistency of such a state S relies on whether the execution can continue from the state S.

Let $\mathcal{X} = \{X_0, \ldots, X_{n-1}\}$ be a partition of $Var(P)$. Consider a set of sub-states $\Psi = \{S_{X_i} \mid X_i \in \mathcal{X}\}$ which occur in a sub-observation β of θ during the execution of program P. More precisely stated, let $\theta = \theta_1 \hat{}\beta\hat{}\gamma$. For each $X_i \in \mathcal{X}$ there exists k_i such that $i < j \Rightarrow k_i < k_j$ and Ψ satisfies condition **C**:

$$\forall X_i \in \mathcal{X} : \beta(k_i)|_{X_i} = S_{X_i} \qquad (\mathbf{C})$$

Ψ is *backward consistent* with θ, denoted $BackwCon(\Psi, \theta)$, if there exists a function $\beta' : \ Var(P) \longrightarrow V^{\dagger}$ such that:

BC1 $\wedge$ **BC2** $\wedge$ **BC3**

where **BC1**, **BC2** and **BC3** are defined as follows: for each $X_i \in \mathcal{X}$,

BC1. $\forall j : 0 \leq j \leq k_i :: \beta'(j)|_{X_i} = \beta(j)|_{X_i}$

BC2. $\forall j > k_i : \beta'(j)|_{X_i} = \beta(k_i)|_{X_i}$

BC3. $\theta_1 \hat{}\beta' \in OB_P$

where $\theta_1 \hat{}\beta' \in OB_P$ is called a *backward recovered* observation of θ from Ψ.

If $BackwCon(\Psi, \theta)$, for β and β' satisfying these conditions, let $\theta' = \theta_1 \hat{}\beta'$. Then θ is said to be a *backward consistent prefix* of θ' and denoted as $\theta \underline{\propto}_{bc} \theta'$.

Similar to the case of forward consistency, for a subset $\mathcal{X} \subseteq \mathcal{P}(Var(P))$, let $\Psi_\beta = \{S_{X_i} \mid X_i \in \mathcal{X}\}$ satisfy Condition **C**. Ψ_β is *backward-consistent* with θ, if there exists a state S such that:

$$\forall X \in \mathcal{X} : S|_X = S_X \wedge BackwCon(S, \theta)$$

Obviously, $BackwCon(\Psi_\beta, \theta) \Rightarrow ForwCon(\Psi_\beta, \theta_1)$

A state S is *consistent* with θ if it is either forward or backward consistent with θ,

$$Consistent(S, \theta) \triangleq ForwCon(S, \theta) \vee BackwCon(S, \theta)$$

When there is no confusion, we will simplify the definitions and notation by omitting θ and writing $Consistent(S)$.

Example 2.4.1 For each finite observation θ of program $Exp1$, the initial state S_0 in which $x = 0$ and $y = 0$ is backward consistent with θ. And any state S in which

$$(last(\theta)(x) < S(x)) \wedge (last(\theta)(y) < S(y)) \wedge (3 \times S(y) \leq S(x) \leq 3 \times (S(y) + 1))$$

is forward consistent with θ.

2.5 The Recovery Transformation

We describe here *what* should be done to make the execution of program P recoverable from interruption by faults in F; in other words, what should be done to recover from an observation θ of $\mathcal{F}(P, F)$ where $error(\theta) \neq <>$. Intuitively, when the execution θ reaches an error state, some actions called *recovery actions* are needed to transform the error state to a good state S which is consistent to the observation $good(\theta)$. The execution of P then restarts from state S and continues with the recovered observation from S. The faults defined by F may occur once again in the restarted execution. Therefore, to make the execution of P recoverable from faults in F, P has to be transformed into a program $\mathcal{R}(P)$ by adding a set of recovery actions which are described as a *recovery program*.

For specifying such a recovery program, we introduce an auxiliary variable ob. The value space of ob is the set of observations of $\mathcal{F}(P, F)$

$$\{\theta : \ Var(P) \longrightarrow V^\dagger\}$$

where $\theta[i]$ is a good state for each $i < \#\theta$, ob is the sequence of states of the execution of P and the last state of ob is the last good state in the execution of $\mathcal{F}(P, F)$.

A state predicate Q over $Var(P)$ is extended to be a state predicate Q_{ob} over $Var(P) \cup \{ob\}$ such that for a state S over $Var(P) \cup \{ob\}$, $Q_{ob}(S) = true$ if

$$Q(S|_{Var(P)}) \wedge \forall x \in Var(P) : last(ob)(x) = S(x)$$

For θ and θ' in the value space of ob, let $Ext(\theta, \theta') = true$ if

$$\exists S \in \Psi_{Var(P)} : \forall x \in Var(P) : \theta(x) = \theta'(x) \hat{} < S(x) >$$

The predicate $ForwExt(\theta, \theta') = true$ if

$$ForwCon(last(\theta), \theta') \wedge \theta' \underline{\propto} \theta$$

With a forward recovery action, instead of restarting the execution of P from $last(\theta')$ and moving forward to the state $last(\theta)$, the current observation θ' is altered directly to θ.

Similarly, the predicate $BackwExt(\theta, \theta') = true$ if

$$BackwCon(last(\theta), \theta') \wedge \theta' \underline{\propto}_{bc} \theta$$

We define

$$ConsExt(\theta, \theta') \triangleq ForwExt(\theta, \theta') \vee BackwExt(\theta, \theta')$$

Now let the specifications of P and F over $Var(P)$ be transformed into specifications over $Var(P) \cup \{ob\}$ following the rules given below.

1. Each $\{Q \wedge \neg f\}A\{R\}$ in the specification of P is transformed into

   $$\{Q_{ob} \wedge \neg f \wedge ob = \theta\}A\{R_{ob} \wedge Ext(ob, \theta)\}$$

   i.e. correct execution of a statement extends ob by one state following the definition of this statement.

2. Each $\{Q\}A_f\{R\}$ in the specification of F is transformed into

   $$\{Q \wedge ob = \theta\}A_f\{R \wedge ob = \theta\}$$

i.e. the history of faults is not recorded in ob.

3. Each $\{f\}A\{f\}$ in P or F is transformed into

$$\{f \wedge ob = \theta\}A\{f \wedge ob = \theta\}$$

i.e. if A is not f-stop, transformation from the error state is not recorded, and if A is f-stop, execution of A in a state S in which f is $true$ will not change the values of the program variables or the value of ob.

Now we can see that for any constant observation θ_0, the fault affected program $\mathcal{F}(P, F)$ satisfies

$$\mathcal{F}(P, F)\ \underline{Sat}\ \textbf{stable}(f \wedge ob = \theta_0)$$

And if we have the finite error behaviour assumption, $\mathcal{F}(P, F)$ also satisfies

$$\mathcal{F}(P, F)\ \underline{Sat}\ \textbf{stable}((\neg f)_{ob} \wedge (bound = 1))$$

and

$$\mathcal{F}(P, F)\ \underline{Sat}\ (\neg f \wedge (bound = 1) \wedge (ob = \theta_0) \longmapsto (\theta_0 \propto ob))$$

From these two properties, if no fault occurs during an execution of P on a system with possible faults F, the execution is an execution of the program P.

The specification of the recovery program P_R of P $w.r.t.$ the fault environment F has the following features:

R1. P_R has the same variables as P (including the fault indicator f and the auxiliary variable ob).

R2. Its initial conditions are that of P together with the conditions $f = false$ and $ob =< init_P >$, for f and ob, where $init_P$ is an initial state of P.

R3. P_R has no effect if it is executed in a good state: for each set of values $\{v_x \mid x \in Var(P)\}$ of the variables $Var(P)$ and an observation θ_0 of P,

$$P_R\ \underline{Sat}\ \textbf{stable}(\neg f \wedge (ob = \theta_0) \wedge \forall\, x \in Var(P) : (x = v_x))$$

R4. When the execution of $\mathcal{F}(P, F)$ reaches an error state, P_R will be eventually invoked to make the execution recovered, i.e. P_R satisfies the following *conditional property* [CM88]

Hypothesis: $\mathcal{F}(P, F)$ $\underline{Sat}$ **stable**$(f \wedge (ob = \theta_0))$
Conclusion: $\mathcal{F}(P, F)[]P_R$ $\underline{Sat}$ $f \wedge (ob = \theta_0) \longmapsto (\neg f)_{ob} \wedge ConsExt(ob, \theta_0)$

The addition of the actions P_R satisfying the above conditions is performed by the *recovery transformation* $\mathcal{R}$:

$$\mathcal{R}(P) = P[]P_R$$

and $\mathcal{R}(P)$ is called the *fault-tolerant* program of P $w.r.t.$ the faults F.

From the compositionality of $\mathcal{F}$, we have

$$\mathcal{F}(\mathcal{R}(P), F) \equiv \mathcal{F}(P, F)[]P_R$$

Therefore, for the fault transformation without the f-stop assumption,

$$\mathcal{F}(\mathcal{R}(P), F) \equiv P[]F[]P_R$$

and for the transformation with the f-stop assumption,

$$\mathcal{F}(\mathcal{R}(P), F) \equiv \mathcal{T}_f(P)[]F[]P_R$$

Let $ConFun \subseteq (OB_P \rightarrow \Psi_P)$ be the set of functions from OB_P to the states Ψ_P of program P such that

$$\forall\, cons \in ConFun\, \forall\, \theta \in OB_P : Consistent(cons(\theta), \theta)$$

And similarly let

$$ForwConFun \triangleq \{cons \in ConFun \mid \forall\, \theta \in OB_P : ForwCon(cons(\theta), \theta)\}$$

and

$$BackwConFun(\theta) \triangleq \{cons \in ConFun \mid \forall\, \theta \in OB_P : BackwCon(con(\theta), \theta)\}$$

The recovery program P_R is then described as:

Program P_R
$[]_{cons}$: $cons \in ConFun$ ::
$f := false$ **if** f $\|$ $(\|_x \ x \in Var(P) :: x := cons(ob)(x)$ **if** f
End $\{P_R\}$

Note that:

1. The variable $cons$ used in P_R is a logical variable that is not changed by the program.

2. $[]_{cons}$: $(\|_x)$ in P_R is used to combine all the statements within the scope of S (the scope of x).

3. When the execution of $\mathcal{F}(P[]P_R, F)$ reaches a state in which f is $true$, the recovery program will eventually be executed to restore the system to a consistent state from which program P can continue its execution.

4. Program P_R here may have unbounded nondeterminism because of the use of $[]_{cons}$ and so $ConFun$ may be infinite.

5. The program P_R provides information about *what* should be done by the execution of the recovery program, but no information about the implementation of P_R: refinement of P_R will be discussed in the next chapter.

6. The program P_R provides an uniform schema for both *forward* and *backward* recovery.

7. The semantics of P_R and the auxiliary variable ob is given by **R3** and for a statement A in P_R such that

$$\{f\}A\{\neg f \wedge \forall x \in Var(P) : x = cons(ob)(x)\}$$

A satisfies the specification

$$\{f \wedge ob = \theta_0\}A\{(\neg f)_{ob} \wedge ConsExt(ob, \theta_0)\}$$

In the backward error recovery technique [AL81], when a fault occurs the system recovers by starting from a consistent state composed of substates previously reached. A program which can perform backward recovery is a special case of the recovery program P_R and can be described as:

Program $P_{\overleftarrow{R}}$
$[]_{cons}: \ cons \in BackwConFun ::$
$f := \mathit{false}$ **if** $f \parallel (\|_x \ x \in Var(P) :: x := cons(ob)(x)$ **if** f
End $\{P_{\overleftarrow{R}}\}$

Example 2.5.1 For the fault affected version of $Exp1$, an implementation of $Exp1_{\overleftarrow{R}}$ is given as follows: when a fault occurs $Exp1_{\overleftarrow{R}}$ will eventually assign the variables x and y with their initial values 0,

Program $Exp1_{\overleftarrow{R}}$
initially
$x = 0, y = 0, f = \mathit{false}$
actions
$x \ , \ y \ , \ f \ := 0 \ , \ 0 \ , \mathit{false}$ **if** f
End $\{Exp1_{\overleftarrow{R}}\}$

In some cases, forward recovery may allow a program to recover from an error whose effects cannot be overcome with backward recovery. As in the case of backward recovery, variables $Var(P)$ have to be assigned to a state S when f is $true$ in the current state. However, for forward recovery, such a state must be good and forward consistent with the current observation ob. Thus, the forward recovery program $P_{\overrightarrow{R}}$ can be described as:

Program $P_{\overrightarrow{R}}$
$[]_{cons}: \ cons \in ForwConFun ::$
$f := \mathit{false}$ **if** $f \parallel (\|_x \ x \in Var(P) :: x := cons(ob)(x)$ **if** f
End $\{P_{\overrightarrow{R}}\}$

Example 2.5.2 For the fault affected version of $Exp1$, an implementation of $Exp1_{\overrightarrow{R}}$ is given as: when a fault occurs $Exp1_{\overrightarrow{R}}$ will eventually assign the variables x and y with the smallest natural numbers v_x and v_y such that

$$(v_y > y) \wedge (3 \times v_y \leq v_x \leq 3 \times (v_y + 1))$$

Obviously such smallest numbers are $3 \times (y + 1)$ and $y + 1$.

Program $Exp1_{\overrightarrow{R}}$
initially

$x = 0, y = 0, f = \mathit{false}$
actions
$x\ ,\ y\ ,\ f\ := 3 \times (y+1)\ ,\ y+1\ ,\ \mathit{false}$ **if** f
End $\{Exp1_{\overline{R}}\}$

2.6 Fault-tolerant Properties of $\mathcal{R}(P)$

This section discusses the properties of the fault-tolerant program $\mathcal{R}(P)$ by describing what properties of the original program P remain true for the program $\mathcal{F}(\mathcal{R}(P), F)$.

Theorem 2.6.1

1. $\mathcal{R}(P) \equiv P$

2. *If there are no faults, the execution of* $\mathcal{F}(\mathcal{R}(P), F)$ *is the same as that of* P*,*

$$(F = \emptyset) \Rightarrow ((\mathcal{F}(\mathcal{R}(P), F) \equiv P)$$

Proof: To prove $\mathcal{R}(P) \equiv P$, from **R1** and **R2** we know that $\mathcal{R}(P)$ and P has the same set of variables. **R3** implies that if

$$P[]P_R\ \underline{Sat}\ \textbf{invariant}\neg f$$

then

$$P[]P_R \equiv (P[]\textbf{skip}) \equiv P$$

$\neg f$ is really invariant in $P[]P_R$ because it is invariant in both P and P_R.

The second result is shown by the first and the fact that

$$(F = \emptyset) \Rightarrow (\mathcal{F}(P, F) \equiv P)$$

□

Theorem 2.6.2 (Fault-tolerance)

$\mathcal{R}(P)$ can tolerate the faults specified by F, i.e. for any constant observation θ_0,

$$\mathcal{F}(P, F)[]P_R \ \underline{Sat} \ (f \wedge (ob = \theta_0) \longmapsto ((\neg f)_{ob} \wedge ConsExt(ob, \theta_0)))$$

Proof: It is easy to see from the text that P_R satisfies the conditional property **R4**,

Hypothesis: $\mathcal{F}(P, F)$ $\underline{Sat}$ **stable**$(f \wedge (ob = \theta_0))$
Conclusion: $\mathcal{F}(P, F)[]P_R$ $\underline{Sat}$ $(f \wedge (ob = \theta_0) \longmapsto ((\neg f)_{ob} \wedge ConsExt(ob, \theta_0)))$

The definition of $\mathcal{F}$ guarantees the hypothesis. Therefore, the conclusion holds, which is the result of the theorem.

□

Theorem 2.6.3 (Invariant Properties)

*If P $\underline{Sat}$ **invariant** Q, then*

$$\mathcal{F}(\mathcal{R}(P), F) \ \underline{Sat} \ \textbf{invariant}(\neg f \Rightarrow Q)$$

Proof: From the definition of $\mathcal{F}$ and $\mathcal{R}$, we have

$$Init_P \Leftrightarrow Init_{\mathcal{F}(\mathcal{R}(P),F)}$$

Therefore,

$$(Init_P \Rightarrow Q) \Leftrightarrow (Init_{\mathcal{F}(\mathcal{R}(P),F)} \Rightarrow Q)$$

Hence,

$$Init_{\mathcal{F}(\mathcal{R}(P),F)} \Rightarrow Q$$

Also we have

$$(P \ \underline{Sat} \ \textbf{stable} Q) \Rightarrow (P \ \underline{Sat} \ \textbf{stable}(\neg f \Rightarrow Q))$$

and

$$(F \ \underline{Sat} \ \textbf{stable} f) \Rightarrow (F \ \underline{Sat} \ \textbf{stable}(\neg f \Rightarrow Q))$$

and

$$(P_R \ \underline{Sat}\ \textbf{stable}\neg f) \Rightarrow (P_R \ \underline{Sat}\ \textbf{stable}(\neg f \Rightarrow Q))$$

because P_R has no effect on good states.

We thus have

$$P \ \underline{Sat}\ \textbf{invariant}(\neg f \Rightarrow Q)$$

$$F \ \underline{Sat}\ \textbf{invariant}(\neg f \Rightarrow Q)$$

$$P_R \ \underline{Sat}\ \textbf{invariant}(\neg f \Rightarrow Q)$$

From the union theorem in [CM88], this implies that

$$\mathcal{F}(\mathcal{R}(P), F) \ \underline{Sat}\ \textbf{invariant}(\neg f \Rightarrow Q)$$

□

Without the finite error behaviour assumption, stable properties and progress properties of $\mathcal{F}(\mathcal{R}(P), F)$ cannot be derived for the general recovery program P_R from that of the original program P. The following theorems are given under the finite error behaviour assumption.

Theorem 2.6.4 (Stable Properties)

In the state after which no fault ever occurs and no further recovery is ever required, $\mathcal{F}(\mathcal{R}(P), F)$ will satisfy all the stable properties of P,

$P \ \underline{Sat}\ \textbf{stable}\ Q \Rightarrow$
$\mathcal{F}(\mathcal{R}(P), F) \ \underline{Sat}\ \textbf{stable}((last(front(ob(bound))) = 1) \wedge Q)$

Proof: First, note that

$$P \ \underline{Sat}\ \textbf{stable}((last(front(ob(bound))) = 1) \wedge Q)$$

because Q is stable in P and $bound$ is not affected by P. And from the finite error behaviour assumption, we have

$$F \ \underline{Sat}\ \textbf{stable}((last(front(ob(bound))) = 1) \wedge Q)$$

If the current state is error, $bound$ could not have been 1 before, i.e.

$$\mathcal{F}(\mathcal{R}(P), F)\ \underline{Sat}\ \textbf{invariant}\ ((last(front(ob(bound))) = 1) \Rightarrow \neg f)$$

P_R always leave good states unchanged. Therefore,

$$P_R\ \underline{Sat}\ \textbf{stable}((last(front(ob(bound))) = 1) \wedge Q)$$

Thus finally we have proved the theorem from the *Union Theorem* □

Similarly we have the theorem for progress properties.

Theorem 2.6.5 (Progress properties)

1. *If* P $\underline{Sat}$ $(Q$ **ensures** $R)$, *then*

$$\mathcal{F}(\mathcal{R}(P), F)\ \underline{Sat}\ ((last(front(ob(bound))) = 1) \wedge Q\ \textbf{ensures}\ R)$$

2. *If* P $\underline{Sat}$ $(Q \longmapsto R)$, *then*

$$\mathcal{F}(\mathcal{R}(P), F)\ \underline{Sat}\ ((last(front(ob(bound))) = 1) \wedge Q \longmapsto R)$$

Proof: The second result is derived from the first one and the inductive rules of $\longmapsto$. It is therefore sufficient to show the first result only.

As we have obtained in the proof of the last theorem,

$$\mathcal{F}(\mathcal{R}(P), F)\ \underline{Sat}\ \textbf{invariant}\ ((last(front(ob(bound))) = 1) \Rightarrow \neg f)$$

we have

$$F\ \underline{Sat}\ \textbf{stable}\ (last(front(ob(bound))) = 1) \wedge Q$$

and

$$P_R\ \underline{Sat}\ \textbf{stable}\ (last(front(ob(bound))) = 1) \wedge Q$$

Since P $\underline{Sat}$ (Q **ensures** R), and $last(front(ob(bound))) = 1$ is stable in P, we have

$$P \ \underline{Sat} \ ((last(front(ob(bound))) = 1) \wedge Q \ \textbf{ensures} \ R)$$

Hence from the Corollaries of the *Union Theorem*, if P $\underline{Sat}$ (Q **ensures** R), then

$$\mathcal{F}(\mathcal{R}(P), F) \ \underline{Sat} \ ((last(front(ob(bound))) = 1) \wedge Q \ \textbf{ensures} \ R)$$

is proved to be true. $\square$

Theorem 2.6.6 (Termination)

1. *If* P $\underline{Sat}$ **stable** R *and* P $\underline{Sat}$ ($Init_P$ **ensures** R), *then*

$$\mathcal{F}(\mathcal{R}(P), F) \ \underline{Sat} \ (Init_P \longmapsto (bound = 1) \wedge \neg f \wedge R)$$

2. *If* P $\underline{Sat}$ **stable** R *and* P $\underline{Sat}$ ($Init_P \longmapsto R$), *then*

$$\mathcal{F}(\mathcal{R}(P), F) \ \underline{Sat} \ (Init_P \longmapsto (bound = 1) \wedge \neg f \wedge R)$$

3. *Specially, if* P $\underline{Sat}$ ($Init_P \longmapsto Fixedpoint(P)$),

$$\mathcal{F}(\mathcal{R}(P), F) \ \underline{Sat} \ (Init_P \longmapsto (bound = 1) \wedge \neg f \wedge Fixedpoint(P))$$

Proof: The second result is derived from the first by using the inductive rules of $\longmapsto$, and the third result is a special case of the second. Therefore, it is sufficient to only prove the first result.

If P $\underline{Sat}$ ($Init_P$ **ensures** R), then

$$P \ \underline{Sat} \ (Init_P \ \textbf{unless} \ R)$$

Thus,

$$P \ \underline{Sat} \ \textbf{stable}(Init_P \vee R)$$

and so

$$P \underline{Sat} \textbf{ invariant}(Init_P \vee R)$$

From Theorem 2.6.3, we have,

$$\mathcal{F}(\mathcal{R}(P), F) \underline{Sat} \textbf{ invariant}(f \vee Init_P \vee R)$$

Since we have

$$\mathcal{F}(\mathcal{R}(P), F) \underline{Sat} (true \longmapsto (bound = 1) \wedge \neg f)$$

Hence,

$$\mathcal{F}(\mathcal{R}(P), F) \underline{Sat} (true \longmapsto ((bound = 1) \wedge \neg f \wedge (Init_P \vee R)))$$

P, F and P_R satisfies

$$\textbf{stable } ((bound = 1) \wedge \neg f \wedge (Init_P \vee R))$$

and P $\underline{Sat}$ $(Init_P$ **ensures** $R)$ implies that

$$P \underline{Sat} ((bound = 1) \wedge \neg f \wedge (Init_P \vee R) \textbf{ ensures } ((bound = 1) \wedge \neg f \wedge R))$$

Therefore, from $Init_P \longmapsto true$ and the *Union Theorem*, we obtain

$$\mathcal{F}(\mathcal{R}(P), F) \underline{Sat} (Init_P \longmapsto (bound = 1) \wedge \neg f \wedge R)$$

□

Under the *bounded* error behaviour assumption, similar properties of $\mathcal{R}(P)$ can be proved in the same way.

Since the termination of a program P is defined by its fixedpoint, we assume that no faults will occur after the execution of P reaches its fixedpoint. The following theorem describes a fault-tolerant property of $\mathcal{R}(P)$ in terms of probability.

2.7 Discussion

This chapter proposes a unifying theory for the design of fault-tolerant programs from programs specified for fault-free systems. The interference of faults is modelled as an interleaving of the program actions and the fault-actions [Cri85] using the fault transformation $\mathcal{F}$. A fault-action is modelled as a state transformation which violates the specification of the original program. Based on this model of faults, recovery actions are introduced by the recovery transformation $\mathcal{R}$ which are invoked whenever faults have occurred and transform a error state into a state which is consistent with the interrupted execution. The fault affected version and fault-tolerant version of a program P are union compositions of P with the fault program and recovery program respectively. This makes it possible for the UNITY logic to be sufficient for reasoning about the logical behaviour of both the fault-affected version and fault-tolerant version of a program designed for a fault-free system. By separating the logical behaviour from the implementation, the theory is simple and appropriate for a wide variety of applications and architectures. Of course, the simplicity and generality are achieved by making the refinement of programs more important. The next chapter will show how to refine the fault affected-version and the fault-tolerant version of a program along with the refinement of the functional properties of the program.

Chapter 3

Refinement of Fault-tolerant Programs

3.1 Introduction

Refinement from specification provides a useful way of systematically constructing programs. Using this method, an original high level program specification is transformed by a sequence of correctness preserving refinements into an executable and efficient program [Lam83, AL88, Bac87, Bac88, BW89, Bac89]. It has been usual to consider that the steps of refinement end with the production of the text of an executable program.

Chapter 2 has described how to reason about, at a high level, properties of fault-affected programs and fault-tolerant programs. Given a problem specification, we began by proposing a general solution strategy (usually such a strategy is broad, admitting many solutions). Next, without taking physical faults into account, we gave a specification (program) of the solution strategy and proved that this strategy (as specified) solved the problem (as specified). Then, by using the recovery transformation, we introduced the fault-tolerant strategy based on the analysis of the fault properties characterized by the fault transformation. The analysis of the fault properties and the fault-tolerant strategy at this stage were both broad (once again, admitting may solutions).

When we consider a specific set of target architectures, we may choose to narrow the solution strategy for both the original program and its fault-tolerant mechanisms, e.g. sometimes we may choose to use only forward recovery and sometimes only backward recovery. This means that we have to refine the original program and its fault-tolerant version.

Along with each refinement step, $P \sqsubseteq P'$, of the original program, more information is introduced about the use of the resources of the system and the nature of the possible faults in the system (e.g. which processors and channels may fail) as all of these factors can affect the execution of the program. This implies that the fault-affected version of P' is to be obtained by adding more details to the fault environment of P.

Also, the choice of recovery strategy depends on the structure of the original program, e.g. if the original program has the appropriate structure, recovery constructs such as *recovery blocks* or *conversations* [Ran75] can be introduced so that consistent states can be calculated more efficiently.

Our objective in this chapter is to refine the fault-tolerant properties along with the refinement of the functional properties of the original program. The high level design of fault-tolerant programs in the UNITY framework provides a good starting point. However, UNITY does not provide formal methods for implementing programs although there is an informal suggestion that a program specification can be implemented on various architectures by mapping the specification to an architecture. To develop a framework which allows fault-tolerant refinement and preserves the abstract theory of the UNITY framework, we will extend UNITY by combining it with Back's *action system formalism* and its refinement calculus [Bac87].

3.2 The Extended Action System Formalism

UNITY provides methods for constructing programs from specifications and allows the proof that a program satisfies its specification to be determined from the program text. On the other hand, the action system formalism provide a refinement framework allowing a high level program specification to be transformed by a sequence of correctness preserving transformations into an executable and efficient program. The refinement calculus was first described in [Bac87, Bac88] as a formalization of the stepwise refinement approach for systematic construction of sequential programs and has been further elaborated in [Bac87, BS88, AL88, BW89] to deal with parallel programs. This formalization corresponds with the informal method given in [CM88] for implementing program specifications on various architectures by using mappings. Furthermore, by taking each assignment as a guarded command with its boolean expression as the guard, a UNITY program can be treated as an action system. Also the UNITY logic can be applied for the high level specification of action systems using the definition given in Section 2.2 of Chapter 2.

3.2.1 Commands and Actions

A *program* P can be described as an *action system* which is a pair $(Init_P, Act_P)$: $Init_P$ is the set of initial conditions of the program variables and Act_P is a set of actions. Each *action* $A \in Act_P$ is of the form $g \to c$, where g is a boolean condition and c is a command [Bac88, BS89]. We use gA and cA respectively to denote the *guard* g and the *body* c of action A, and we abbreviate the action $true \to c$ to c if there is no confusion.

A *command* c is defined as follows,

$$
\begin{array}{lll}
c & ::= \overline{x}_i := \overline{x'}_i.Q & (nondeterministic\ \ assignment) \\
& \mid\ c_1 \parallel \ldots \parallel c_n, \quad n > 1 & (c_i\ is\ an\ \ assignment) \\
& \mid\ c_1; \ldots ; c_n, \quad n > 1 & (sequential\ \ composition) \\
& \mid\ \textbf{if}\ A_1 [] \ldots [] A_n\ \textbf{fi}, \quad n \geq 1 & (conditional\ \ composition) \\
& \mid\ \textbf{do}\ A_1 [] \ldots [] A_n\ \textbf{od}, \quad n \geq 1 & (iterative\ \ composition)
\end{array}
$$

Here $\overline{x}_i$ and $\overline{x'}_i$ are lists of variables and Q is a condition over the values of program variables [BK83]. In the sequential composition, each c_i is a command, and each A_i in the conditional composition and the iterative composition commands is an action. We write $IF_{i=1}^n A_i$ for **if** $A_1 [] \ldots [] A_n$ **fi** and similarly $DO_{i=1}^n A_i$ for the iterative composition command.

The effect of the nondeterministic assignment command is to assign to the list of variables $\overline{x}_i$ some value(s) $\overline{x'}_i$ satisfying condition Q. (Note that this may introduce unbounded nondeterminism: we shall discuss this later).

As in UNITY, the set $Act_P = \{A_1, \ldots, A_n\}$ of actions can also be represented as a list of actions $A_1 [] \ldots [] A_n$. If P_1 and P_2 are programs, then the *union composition* program $P_1 [] P_2$ is $(Init_{P_1} \cup Init_{P_2}, Act_{P_1} \cup Act_{P_2})$. No special notation is used for processes and communication channels. Parallel execution of a program will be studied in Chapter 4.

3.2.2 A Semantic Function

An action of a program in the action system formalism is to be executed as a simple "primitive" command in a UNITY program: if an action is chosen for execution, it is assumed to be executed without any interference from the other actions in the program. Because of this atomicity, a program can be viewed as being sequential and nondeterministic.

States : Let V be the value space of program P and let $\{\perp, \textbf{abort}\} \subseteq V$. A *state* S of a program P is a function from the program variables $Var(P)$ to the value space V such that

$$\begin{aligned}&\forall x \in Var(P) : (S(x) = \perp) \vee\\ &\forall x \in Var(P) : (S(x) = \textbf{abort}) \vee\\ &\forall x \in Var(P) : (S(x) \neq \perp) \wedge (S(x) \neq \textbf{abort})\end{aligned}$$

The state in which the value of each variable is $\perp$ is simply denoted as $\perp$ and this stands for nontermination. And **abort** denotes the state in which the value of each variable is **abort**. This state satisfies the assertion

$$(ABORT(\textbf{abort}) = true) \wedge \{false\}c\{ABORT\}$$

where $ABORT$ is a state predicate which is only $true$ in state **abort**.

A *sub-state* $S|_Y$ is the restriction of S to Y where $Y \subseteq Var(P)$. Ψ_P is used to denote the set of all the states of program P.

Semantics of commands : The semantics of a command c is a function:

$$\Gamma(c) : \Psi_P \rightarrow \mathbb{P}(\Psi_P)$$

This function can be extended to be a function over the powerset $\mathbb{P}(\Psi_P)$ in the following way. For a set Ψ of states

$$\Gamma(c)(\Psi) = \bigcup_{S \in \Psi} \Gamma(c)(S)$$

We define $\Gamma(c)(\perp) = \{\perp\}$ and $\Gamma(c)(\textbf{abort}) = \{\textbf{abort}\}$

The semantics Γ of the execution of a command c in a state S is defined as follows.

1. Assignment:

$$\Gamma(\overline{x}_i := \overline{x'}_i.Q)(S) =$$
$$\begin{cases} \{S' \mid Q(S') \wedge \forall y.\, y \notin \overline{x}_i : S'(y) = S(y)\} & \text{if } Q \neq false \\ \{\textbf{abort}\} & \text{if } Q = false \end{cases}$$

2. Parallel composition:

$$\Gamma(\overline{x}_i := \overline{x'}_i.Q_1 \parallel \overline{y}_i := \overline{y'}_i.Q_2)(S) = \Gamma(\overline{x}_i \,\hat{}\, \overline{y}_i := \overline{xy}'.Q_1 \wedge Q_2)(S)$$

3. Sequential composition: $\Gamma(c_1; c_2)(S) = \Gamma(c_2)(\Gamma(c_1(S))$

4. Conditional composition:

$$\Gamma(IF_{i=1}^{n} A_i)(S) = \begin{cases} \bigcup\limits_{g_i(S)=true} \Gamma(cA_i)(S) & \text{if } gg(S) = true \\ \{\textbf{abort}\} & \text{if } gg(S) = \mathit{false} \end{cases}$$

where $gg = gA_1 \vee \ldots \vee gA_n$.

5. Iterative composition: Let $\mathcal{A} = A_1[] \ldots []A_n$. We first define

$$\Gamma(\mathcal{A})(S) = \bigcup_{gA_i(S)=true} \Gamma(cA_i)(S)$$

within $\mathbb{P}(\Psi_P)$ which is the complete partial order domain (CPO) under the inclusion between sets of states of P. For $k \geq 0$, we define Γ^k inductively as:

- $\Gamma^0(\mathcal{A})(S) \triangleq \{S\}$,
- $\Gamma^k(\mathcal{A})(S) \triangleq \Gamma(\Gamma^{k-1}(\mathcal{A})(S))$, if $k \geq 1$.

Let Y be a set of states and $\mathcal{L}(S)$ be the least fixed point[1] of the equation

$$Y = \{S\} \cup \Gamma(\mathcal{A})(Y)$$

and

$$M = \{S \mid \forall\, i : 1 \leq i \leq n : gA_i(S) = \mathit{false}\} \cup \{\bot\}$$

Then

$\Gamma(DO_{i=1}^{n} A_i)(S) =$

$$\begin{cases} \Gamma F(S) \cap M & \text{if } \exists\, k \geq 0 : \forall\, S' \in \Gamma^k(\mathcal{A})(S) : g\mathcal{A}(S') = \mathit{false} \\ \mathcal{L}(S) \cup \{\bot\} & \text{otherwise} \end{cases}$$

where $g\mathcal{A} = gA_1 \vee \ldots \vee gA_n$.

Semantics of actions : The semantics of an action $A = g \rightarrow c$ is a function:

[1]The existence of such a fixed point can be proved in set theory.

$$\Gamma(A): \Psi_P \rightarrow \mathbb{P}(\Psi_P)$$

such that

$$\Gamma(A)(S) = \begin{cases} \Gamma(c)(S) & \text{if } g(S) = \mathit{true} \\ \emptyset & \text{if } g(S) = \mathit{false} \end{cases}$$

As in the case of commands, $\Gamma(A)(\perp) = \{\perp\}$ and $\Gamma(A)(\textbf{abort}) = \{\textbf{abort}\}$.

Semantics of programs : Observations, executions and the semantics $\Gamma(P)$ of a program P are defined in the same way as for UNITY programs in Section 2.2.3. We choose the *strong fairness condition* for the execution of action systems. Formally speaking, an execution E of a program P is *fair* if for each $i \geq 0$ and any action A of P,

$$gA(E(i)) \Rightarrow (\exists\, j \geq i : E(j+1) \in \Gamma(A)(E(j))) \vee (\exists\, j > i : \neg g(A)(E(j)))$$

Therefore the semantic model for UNITY is consistent with the semantic model for action systems.

3.2.3 Reasoning About Action Systems

As in the definition of UNITY, the Hoare triple $\{Q\}A\{R\}$ defines an action A which when executed in a state satisfying the predicate Q terminates in a state satisfying the predicate R. A is universally or existentially quantified over the actions; a property which holds for all points of the execution of a program is defined using universal quantification while a property which holds eventually is defined using existential quantification. The Hoare triple $\{Q\}c\{R\}$ for a command c refers similarly to the execution of c in a state satisfying the predicate Q and terminating in a state satisfying the predicate R. The Hoare triples of the components of an action A are obtained from a decomposition of the Hoare triple of A.

Under the fairness condition we have chosen, the logical operators **unless**, **stable**, **invariant**, **ensures**, **leads-to** ($\longmapsto$) used to describe the safety and progress properties of programs can be defined in a similar way to that in Section 2.2 of Chapter 2. However, since the semantics of an action A is defined to be $\Gamma(A)(S) = \emptyset$ if $g(A)(S) = \mathit{false}$, the definition of **unless** needs to be slightly changed into the following form:

$$P\ \underline{Sat}\ (Q\ \textbf{unless}\ R) \Leftrightarrow \forall\, A : A \in Act_P :: \{gA \wedge Q \wedge \neg R\}cA\{Q \vee R\}$$

The definitions of the other operators remains the same as given in Section 2.2 of Chapter 2.

3.3 Refinement of Action System

In the refinement frameworks in [Bac78, Bac80, Bac87, Mor87, Mor90], the notation of correctness that must be preserved by refinement transformations is input-output correctness. This means that an action system $P = (Init_P, A_1[] \ldots []A_n)$ can be treated as one command

$$c_0;\ \textbf{do}\ A_1[] \ldots []A_n\ \textbf{od}$$

where if $\overline{x}$ is the list of the variables of P, the command c_0 is an assignment

$$\overline{x} := \overline{x'}.Init_P$$

Refinement is then applied to the command

$$c_0; \textbf{do}\ A_1[] \ldots []A_n\ \textbf{od}$$

Temporal properties (e.g. safety and liveness properties) of parallel programs are not in general preserved by such refinements. However, in this thesis, we allow the high level specification of a program be written in UNITY logic. A program expressed by an action system constructed from such a specification must satisfy the specification. The refinement transformations of such a program are restricted to those that preserve the UNITY specification. We also take the convention that if the high level specification of a program written in UNITY logic or as action system only asserts an input-output property of the program, all the transformations in the refinement calculus are allowed on this program; that is, the action system constructed from the specification can be treated as one command, as mentioned above. In this section, we give a summary of the refinement calculus for sequential programs and discuss about the kind of refinements that will preserve a UNITY specification.

3.3.1 Refinement of Commands

Let c and c' be commands. Command c is said to be *refined* by the command c', denoted $c \sqsubseteq c'$, if

$$\{Q\}c\{R\} \Rightarrow \{Q\}c'\{R\}$$

for any predicates Q and R. Refinement captures the notion of command c' preserving the correctness of command c. Hence if c is totally correct with respect to a given precondition Q and a postcondition R and $c \sqsubseteq c'$, then c' will also be totally correct with respect to the precondition Q and postcondition R.

The commands c and c' are said to be (*refinement*) *equivalent*, denoted $c \equiv c'$, if

$$(c \sqsubseteq c') \wedge (c' \sqsubseteq c)$$

The refinement relation between commands is reflexive and transitive, and hence it is a preorder. For any commands c, c' and c'',

1. $c \sqsubseteq c$,
2. $(c \sqsubseteq c') \wedge (c' \sqsubseteq c'') \Rightarrow (c \sqsubseteq c'')$

The sequential program constructors are also monotonic with respect to the refinement relation. That is, if $c_i \sqsubseteq c_i'$, for $i = 1, \ldots, n$, then

1. $c_1; \ldots ; c_n \ \sqsubseteq \ c_1'; \ldots ; c_n'$
2. **if** $g_1 \rightarrow c_1 [] \ldots [] g_n \rightarrow c_n$ **fi** $\sqsubseteq$ **if** $g_1 \rightarrow c_1' [] \ldots [] g_n \rightarrow c_n'$ **fi**
3. **do** $g_1 \rightarrow c_1 [] \ldots [] g_n \rightarrow c_n$ **od** $\sqsubseteq$ **do** $g_1 \rightarrow c_1' [] \ldots [] g_n \rightarrow c_n'$ **od**

This is summarized in the following theorem (the proof can be found in [Bac80, Bac87]).

Theorem 3.3.1 (Refinement of Commands)

(i) The refinement relation is reflexive and transitive.

(ii) If $t \sqsubseteq t'$, then $c(t) \sqsubseteq c(t')$, for any command c that contains t as a subcommand.

3.3.2 Context Dependent Replacement

The refinement theorem allows the replacement of a subcommand t in $c(t)$ with some other command t', if $t \sqsubseteq t'$. However, if a command (an action, or a program)

$c(t)$ contains a command t, it is possible that the replacement of t by t' does preserve correctness of $c(t)$, although $t \sqsubseteq t'$ does not hold. This kind of *context dependent replacements* can be handled by the following technique.

Let us add *assert commands* of the form $\{Q\}$ to our language, where Q is a predicate. Such a command $\{Q\}$ is defined to be the nondeterministic assignment $\varepsilon := \varepsilon.Q$, where ε denotes the empty list of variables. The effect of this command is the same as **skip** if Q holds in the initial state. Execution of this command in a state in which Q does not hold will abort. Assume now that we can prove

1. $c(t) \sqsubseteq c(\{Q\}; t)$ and
2. $\{Q\}; t \sqsubseteq t'$

By monotonicity in the refinement theorem, we then have $c(t) \sqsubseteq c(t')$.

The first step is called *context introduction*, while the second step is called *refinement in context* [Bac87].

3.3.3 Refinement of Actions

Given actions $A = g \to c$ and $A' = g' \to c'$, A is said to be *refined* by A', denoted $A \sqsubseteq A'$, if

$$\{Q\}A\{R\} \Rightarrow \{Q\}A'\{R\}$$

for every precondition Q and every postcondition R. The refinement of actions is characterized by the following theorem, the proof of which was given in [Bac88].

Theorem 3.3.2 (Refinement of Actions) *Let $A = g \to c$ and $A' = g' \to c'$ be two actions. Then $A \sqsubseteq A'$ if and only if*

(i) $g' \Rightarrow g$ *and*

(ii) $\{g'\}; c \sqsubseteq c'$

The first condition says that A is always enabled when A' is enabled, and the second says that the body of A is refined by the body of A' whenever A' is enabled.

However, command constructors are *not* monotonic with respect to refinement relation between actions: $A_i \sqsubseteq A'_i$ implies neither of the following refinements, $i = 1, \ldots, n$,

1. **if** $A_1[] \ldots []A_n$ **fi** $\sqsubseteq$ **if** $A'_1[] \ldots []A'_n$ **fi**
2. **do** $A_1[] \ldots []A_n$ **od** $\sqsubseteq$ **od** $A'_1[] \ldots []A'_n$ **od**

Hence, we are not permitted to freely replace an action in an action system with its refinement unless the guards of the actions are equivalent.

3.3.4 Preserving UNITY Properties

The following theorem shows that the application of the refinement relation between commands to individual actions of an action system preserves the properties specified by UNITY logic.

Theorem 3.3.3 *Let* $P = (Init_P, A_1[] \ldots []A_n)$ *and* $P' = (Init_{P'}, A'_1[] \ldots []A'_n)$ *be programs such that*

(i) for $i = 1, \ldots, n$, $gA_i \Leftrightarrow gA'_i$;

(ii) for $i = 1, \ldots, n$, $cA_i \sqsubseteq cA'_i$;

(iii) $Init_{P'} \Rightarrow Init_P$.

Then, for any predicates Q *and* R,

1. $(P\ \underline{Sat}\ (Q\ \textbf{unless}\ R)) \Rightarrow (P'\ \underline{Sat}\ (Q\ \textbf{unless}\ R))$
2. $(P\ \underline{Sat}\ (\textbf{stable}\ Q)) \Rightarrow (P'\ \underline{Sat}\ (\textbf{stable}\ Q))$
3. $(P\ \underline{Sat}\ (\textbf{invariant}\ Q)) \Rightarrow (P'\ \underline{Sat}\ (\textbf{invariant}\ Q))$
4. $(P\ \underline{Sat}\ (Q\ \textbf{ensures}\ R)) \Rightarrow (P'\ \underline{Sat}\ (Q\ \textbf{ensures}\ R))$
5. $(P\ \underline{Sat}\ (Q \longmapsto R)) \Rightarrow (P'\ \underline{Sat}\ (Q \longmapsto R))$

Proof: From the definition of the refinement relation between commands, we have, for any commands c and c' and guard g,

$$c \sqsubseteq c' \Rightarrow (g \rightarrow c) \sqsubseteq (g \rightarrow c')$$

Therefore,

1. $(P\ \underline{Sat}\ (Q\ \textbf{unless}\ R)) \Leftrightarrow \forall\, A : A \in Act_P :: \{gA \wedge Q \wedge \neg R\}cA\{Q \vee R\} \Rightarrow \forall\, A' : A' \in Act_{P'} :: \{gA' \wedge Q \wedge \neg R\}cA'\{Q \vee R\} \Leftrightarrow (P'\ \underline{Sat}\ (Q\ \textbf{unless}\ R))$
2. $(P\ \underline{Sat}\ (\textbf{stable}\ Q)) \Rightarrow (P'\ \underline{Sat}\ (\textbf{stable}\ Q))$ can be derived from the **unless** case.
3. $(P\ \underline{Sat}\ (\textbf{invariant}\ Q)) \Rightarrow (P'\ \underline{Sat}\ (\textbf{invariant}\ Q))$ can be derived from the **stable** case.
4. $(P\ \underline{Sat}\ (Q\ \textbf{ensures}\ R)) \Rightarrow (P'\ \underline{Sat}\ (Q\ \textbf{ensures}\ R))$ can be derived from the **unless** case and the fact that

   $$\exists\, A : A \in Act_P :: \{gA \wedge Q \wedge \neg R\}A\{R\} \Rightarrow$$
   $$\exists\, A' : A' \in Act_{P'} :: \{gA \wedge Q \wedge \neg R\}A'\{R\}$$

5. $(P\ \underline{Sat}\ (Q \longmapsto R)) \Rightarrow (P'\ \underline{Sat}\ (Q \longmapsto R))$ can be derived from the **ensures** case and deductive rules for $\longmapsto$.

The theorem is thus proved. □

As mentioned before, we are not permitted to freely replace an action A_i in Act_P of an action system P with its refinement A'_1, but we are permitted to replace A_i by A'_i if they have equivalent guards and the body of A_i is refined by the body of A'_i. The following theorem, the proof of which is similar to that of Theorem 3.3.3, shows that we are permitted another kind of replacement.

Theorem 3.3.4 *Let* $P = (Init_P, A_1[] \ldots []A_n)$ *and* $P' = (Init_{P'}, A'_1[] \ldots []A'_n)$ *be programs such that*

(i) for $i = 1, \ldots, n$, $P\ \underline{Sat}\ \textbf{invariant}\ (gA_i \Leftrightarrow gA'_i)$*;*

(ii) for $i = 1, \ldots, n$, $cA_i \sqsubseteq cA'_i$*;*

(iii) $Init_{P'} \Rightarrow Init_P$.

Then, for any predicates Q *and* R*,*

1. $(P\ \underline{Sat}\ (Q\ \textbf{unless}\ R)) \Rightarrow (P'\ \underline{Sat}\ (Q\ \textbf{unless}\ R))$
2. $(P\ \underline{Sat}\ (\textbf{stable}\ Q)) \Rightarrow (P'\ \underline{Sat}\ (\textbf{stable}\ Q))$
3. $(P\ \underline{Sat}\ (\textbf{invariant}\ Q)) \Rightarrow (P'\ \underline{Sat}\ (\textbf{invariant}\ Q))$

4. $(P \underline{Sat} (Q \textbf{ ensures } R)) \Rightarrow (P' \underline{Sat} (Q \textbf{ ensures } R))$

5. $(P \underline{Sat} (Q \longmapsto R)) \Rightarrow (P' \underline{Sat} (Q \longmapsto R))$

Refinement for parallel programs will be discussed in Chapter 4.

3.4 Fault Transformation for Action Systems

A program represented as an action system contains more program constructs, such as sequential composition, than a UNITY program, i.e. it contains more execution information. Therefore, more details about the effect of the faults on the execution of the program can be described. The modelling of the interference of the execution by faults is also more complicated since the fault transformation cannot always be defined as the simple union of the program with the fault program. If we had started by characterizing the interference by fault with so much execution information, it would have been difficult to achieve the simple fault-tolerant properties that we have achieved in Section 2.6. We overcome this difficult by treating the fault transformation for an action system as a refinement of the fault transformation given in Chapter 2. This means that we are allowed to refine the fault-affected version of a program P along with the refinement of P.

Faults are still modelled as a program F consisting of a set of atomic actions. In the fault environment F, the execution of an action A of program P may be interrupted by the execution of a fault action at any point: such a *failure execution* of P is defined by the *failure semantics*.

3.4.1 Failure Semantics

For a program P and a fault environment F, assume that P has a boolean variable f to indicate the presence of a fault, and that the value of f is never changed in P. We define, in this subsection, the failure semantics without the f-stop assumption. The f-stop failure semantics will be given in the next subsection.

Failure Semantics of Commands

Given a program P and the fault environment F, we use the function

$$\Gamma_F(c) : \Psi_P \to \mathbb{P}(\Psi_P)$$

to denote the *failure semantics* of a command c $w.r.t.$ F.

Let a primitive command be an assignment or a parallel composition of assignments. The failure semantics $\Gamma_F(c)$ of a primitive command c in a state S has the intuitive meaning that either c is correctly executed or a fault occurs:

$$\Gamma_F(c)(S) = \Gamma(c)(S) \cup \bigcup_{a \in F} \Gamma(a)(S)$$

The failure semantics of a sequential composition command $c_1; c_2$ in S is the failure semantic execution of c_1 in S followed by the failure semantic execution of c_2. In other words, faults may occur in the execution of either or both c_1 and c_2:

$$\Gamma_F(c_1; c_2)(S) = \Gamma_F(c_2)(\Gamma_F(c_1)(S))$$

The failure semantics of a conditional composition command $IF_{i=1}^n A_i$ in S is the failure semantic execution of the body of an action when its guard holds in the state S. For S, if there is no such an action in the command, the execution aborts. That is

$$\Gamma_F(IF_{i=1}^n A_i)(S) = \begin{cases} \bigcup\limits_{gA_i(S)=true} \Gamma_F(cA_i)(S) & \text{if } gg(S) = true \\ \{\textbf{abort}\} & \text{if } \neg gg(S) = true \end{cases}$$

where $gg = gA_1 \vee \ldots \vee gA_n$.

The failure semantics of an iterative composition command $DO_{i=1}^n A_i$ in S is the iteration of the failure semantic execution of the body of an action with a true guard. This iteration terminates when the guards of all the actions are $false$. Let the notation $\mathcal{A}$ and the set M of states be as defined in the defintion of the semantics of $DO_{i=1}^n A_i$. We define the failure semantics of each iteration:

$$\Gamma_F(\mathcal{A})(S) = \bigcup_{gA_i(S)=true} \Gamma_F(cA_i)(S)$$

Let $\mathcal{L}_F(S)$ be the least fixed point of the equation

$$Y = \{S\} \cup \Gamma_F(\mathcal{A})(Y)$$

Then $\Gamma_F(DO_{i=1}^n A_i)(S)$ is defined as

$\Gamma_F(DO_{i=1}^n)(S) =$

$$\begin{cases} \mathcal{L}_F(S) \cap M & \text{if } \exists\, k \geq 0 : \forall\, S' \in \Gamma_F^k(\mathcal{A})(S) :: g\mathcal{A}(S') = \mathit{false} \\ \mathcal{L}_F(S) \cap M \cup \{\perp\} & \text{otherwise} \end{cases}$$

Assume that $\Gamma_F(c)(\perp) = \{\perp\}$ and $\Gamma_F(c)(\textbf{abort}) = \{\textbf{abort}\}$ for any command c. From the failure semantics of conditional commands and iterative commands, we see that faults can cause the execution of an action (or command) to be **abort** or $\perp$ even though this cannot occur if there are no faults. If the execution is **abort** or $\perp$, we cannot distinguish whether it has occurred normally or as the effect of a fault.

Failure Semantics of Actions

The failure semantics of the execution of an action A in a state S is the failure execution of its body if its guard is $true$ in S:

$$\Gamma_F(A)(S) = \begin{cases} \Gamma_F(cA)(S) & \text{if } gA(S) = true \\ \emptyset & \text{otherwise} \end{cases}$$

We also define $\Gamma_F(A)(\perp) = \{\perp\}$ and $\Gamma_F(A)(\textbf{abort}) = \{\textbf{abort}\}$ for any action A.

Failure Semantics of Programs

From the failure semantics of a command and an action, the *failure observation* of the program P $w.r.t.$ F can be derived as a function $\theta : \; Var(P) \longrightarrow V^\dagger$ satisfying the following conditions:

FOB1. $\theta(x) \neq <>$, for any $x \in Var(P)$,

FOB2. $\#\theta(x) = \#\theta(y)$ (denoted by $\#\theta$) for any $x, y \in Var(P)$,

FOB3. for each $i < \#\theta$, $\theta[i]$ is the state of P defined as $\theta[i](x) = \theta(x)(i)$ for any $x \in Var(P)$,

FOB4. $\theta[0]$ is an initial state of P,

FOB5. $\forall\, i : 0 < i < \#\theta :: (\theta[i] = \theta[i-1]) \vee (\exists\, A \in Act_P : \theta[i] \in \Gamma_F(A)(\theta[i-1]))$

A *failure execution* E of a program P $w.r.t.$ F is an infinite failure observation of P. It said to exhibit *fairness* if any action A of P which is continuously enabled from a execution point in E will be eventually executed following the failure semantics of A. The *failure semantics* of the program P $w.r.t.$ to F is the set $\Gamma_F(P)$ of the fair failure executions of P $w.r.t.$ F.

3.4.2 Fail-stop Failure Semantics

Under the fail-stop assumption, the failure semantics of a command c $w.r.t.$ the fault environment F is a function:

$$\Gamma_F^f(c):\ \Psi_P \to \mathbb{P}(\Psi_P)$$

which is called f-*stop failure semantics* of c.

The f-stop failure semantics of a primitive command c in a state S has the following meaning: if S is a good state, then the execution of c is of the failure semantics, and if S is an error state, the execution of c will not be carried out.

$$\Gamma_F^f(c)(S) = \begin{cases} \{S\} & \text{if } f(S) = true \\ \bigcup\limits_{a\in F} \Gamma(a)(S) \cup \Gamma(c)(S) & \text{if } f(S) = false \end{cases}$$

In a state S, the f-stop failure semantics of the sequential composition $c_1; c_2$ has the meaning that the f-stop failure semantic execution of c_1 is followed by the f-stop failure semantic execution of c_2:

$$\Gamma_F^f(c_1; c_2)(S) = \Gamma_F^f(c_2)(\Gamma_F^f(c_1)(S))$$

If S is a good state, the f-stop failure semantic execution of a conditional composition command $IF_{i=1}^n A_i$ is the f-stop failure semantic execution of the body of an action with a true guard. The execution aborts, if the guard of each action A_i in $IF_{i=1}^n A_i$ is $false$ in S. If S is an error state, the f-stop execution of $IF_{i=1}^n A_i$ leaves it unchanged. That is

$$\Gamma_F^f(IF_{i=1}^n A_i)(S) = \begin{cases} \bigcup\limits_{gA_i(S)=true} \Gamma_F^f(cA_i)(S) & \text{if } (\neg f \wedge gg)(S) = true \\ \{\textbf{abort}\} & \text{if } (\neg f \wedge \neg gg)(S) = true \\ \{S\} & \text{if } f(S) = true \end{cases}$$

where $gg = gA_1 \vee \ldots \vee gA_n$.

Similar to the case of failure semantics, we define the f-stop failure semantics for the iterative composition $DO_{i=1}^n A_i$ in the following way.

Use the semantics

$$\Gamma_F^f(A_1[]\ldots[]A_n)(S) = \begin{cases} \bigcup\limits_{g_i(S)=true} \Gamma_F^f(cA_i)(S) & \text{if } f(S) = \mathit{false} \\ \emptyset & \text{if } f(S) = \mathit{true} \end{cases}$$

and the least fixed point $\mathcal{L}_F^f(S)$ of the equation

$$Y = \{S\} \cup \Gamma_F^f(A_1[]\ldots[]A_n)(Y)$$

$\Gamma_F(DO_{i=1}^n A_i)(S)$ is defined as

$\Gamma_F(DO_{i=1}^n A_i)(S) =$

$$\begin{cases} \mathcal{L}_F^f(S) \cap M & \text{if } \exists\, k \geq 0 : \forall\, S' \in \Gamma_F^{fk}(\mathcal{A}(S) :: \neg f \vee g\mathcal{A}(S') = \mathit{false} \\ \mathcal{L}_F^f(S) \cap M & \text{otherwise} \end{cases}$$

where $M = \{S \mid \neg f \vee g\mathcal{A}(S) = \mathit{false}\}$.

An action $A = g \rightarrow c$ is enabled in a state S in the fault environment F iff S is a good state and g holds in S. If A is enabled in state S, the f-stop failure semantic execution of A is the f-stop failure semantic execution of its body:

$$\Gamma_F^f(A)(S) = \begin{cases} \Gamma_F^f(c)(S) & \text{if } \neg f \wedge g(S) = \mathit{true} \\ \emptyset & \text{otherwise} \end{cases}$$

We also define $\Gamma_F^f(a)(\perp) = \{\perp\}$ and $\Gamma_F^f(a)(\textbf{abort}) = \{\textbf{abort}\}$ for each command or action a.

An f-*stop failure observation* of the program P $w.r.t.$ F is defined in the same way as a failure observation except that condition **FOB5** is changed to:

$$\forall\, i < \#\theta : (\theta[i] = \theta[i-1]) \vee (\exists\, A \in Act_P : \theta[i] \in \Gamma_F^f(A)(\theta[i-1]))$$

Similarly, an f-*stop failure execution* E of a program P $w.r.t.$ F is an infinite f-stop failure observation of P. The f-*stop failure semantics* of the program P $w.r.t.$ to F is the set $\Gamma_F^f(P)$ of the fair f-stop failure executions of P $w.r.t.$ F.

In the f-stop failure semantic model, faults will never cause **abort** or $\perp$ to occur more frequently than in the original program. Obviously, if there are no faults, i.e. F is empty, each failure execution is the same as some execution of P:

$$F = \emptyset \Rightarrow \Gamma_F(P) = \Gamma(P)$$

and

$$F = \emptyset \Rightarrow \Gamma_F^f(P) = \Gamma(P)$$

Programs P and P' are said to be *fault-prone equivalent* $w.r.t.$ F if they are equivalent and have the same failure semantics:

$$P \equiv_F P' \triangleq P \equiv P' \wedge \Gamma_F(P) = \Gamma_F(P')$$

It may be noticed that equivalent programs may not be fault-prone equivalent. Also from the definition of Γ_F^f, we have

$$P \equiv_F P' \Rightarrow \Gamma_F^f(P) = \Gamma_F^f(P')$$

3.4.3 Fault Transformations

When a program is refined and language constructs such as sequential composition, conditional composition and iterative composition are introduced, we must assume that faults can interfere with any component command of the program. This subsection provides the definition of the fault transformation which can be applied to an action system at each refinement step.

Given a program $P = (Init_P, A_1[] \ldots []A_m)$ and a fault environment F, we define the fault transformation $\mathcal{F}$ as

$$\mathcal{F}(P, F) \triangleq (Init_P, \mathcal{F}(A_1, F)[] \ldots []\mathcal{F}(A_n, F))$$

where each $\mathcal{F}(A_i, F)$ is obtained from A_i by transforming it into $gA_i \rightarrow \mathcal{F}(cA_i, F)$. In this definition, the fault transformation $\mathcal{F}(c, F)$ for a command c is defined compositionally below.

1. For a primitive command c,

   $\mathcal{F}(c, F) \triangleq$ **if** $true \rightarrow c[]F$ **fi**

2. $\mathcal{F}(c_1; c_2, F) \triangleq \mathcal{F}(c_1, F);\ \mathcal{F}(c_2, F)$

3. $\mathcal{F}(IF_{i=1}^{n} A_i, F) \triangleq IF_{i=1}^{n} \mathcal{F}(A_i, F)$

4. $\mathcal{F}(DO_{i=1}^{n}, F) \triangleq DO_{i=1}^{n} \mathcal{F}(A_i, F)$

The following theorem establishes the soundness of the fault transformation in terms of the failure semantics.

Theorem 3.4.1 *Given a program P and its fault environment F, the fault transformation $\mathcal{F}$ satisfies*

$$\Gamma(\mathcal{F}(P, F)) = \Gamma_F(P)$$

Proof: From the definitions of observations and failure observations, it is only required to prove that for each state S and action $A = g \rightarrow c$ of P,

$$\Gamma_F(A)(S) = \Gamma(\mathcal{F}(A, F))(S)$$

Case 1 : if c is a primitive command, then

$$\begin{aligned}
\Gamma_F(A)(S) &= \begin{cases} \Gamma_F(c)(S) & \text{if } g(S) = true \\ \emptyset & \text{otherwise} \end{cases} \\
&= \begin{cases} \Gamma(c)(S) \cup \bigcup\limits_{a \in F} \Gamma(a)(S) & \text{if } g(S) = true \\ \emptyset & \text{otherwise} \end{cases} \\
&= \Gamma(g \rightarrow \textbf{ if } true \rightarrow c[]F \textbf{ fi})(S) \\
&= \Gamma(\mathcal{F}(A, F))(S)
\end{aligned}$$

Case 2 : if $c = c_1; c_2$ and $\Gamma_F(c_i) = \Gamma(\mathcal{F}(c_i, F))$ for $i = 1, 2$, then

$$\begin{aligned}
\Gamma_F(A)(S) &= \begin{cases} \Gamma_F(c_1)(\Gamma_F(c_2)(S)) & \text{if } g(S) = true \\ \emptyset & \text{otherwise} \end{cases} \\
&= \begin{cases} \Gamma(\mathcal{F}(c_1, F))(\Gamma(\mathcal{F}(c_2, F))(S)) & \text{if } g(S) = true \\ \emptyset & \text{otherwise} \end{cases} \\
&= \begin{cases} \Gamma(\mathcal{F}(c_1, F); \mathcal{F}(c_2, F))(S) & \text{if } g(S) = true \\ \emptyset & \text{otherwise} \end{cases} \\
&= \begin{cases} \Gamma(\mathcal{F}((c_1; c_2), F))(S) & \text{if } g(S) = true \\ \emptyset & \text{otherwise} \end{cases} \\
&= \Gamma(g \rightarrow \mathcal{F}((c_1; c_2), F))(S) \\
&= \Gamma(\mathcal{F}(A, F))(S)
\end{aligned}$$

Case 3 : if $c =$ **if** $A_1[] \ldots []A_n$ **fi**, $A_i = g_i \rightarrow c_i$ for $i = 1, \ldots, n$, then

$$\Gamma_F(A)(S) = \begin{cases} \bigcup_{g_i(S)=true} \Gamma_F(c_i)(S) & \text{if } (g \wedge gg)(S) = true \\ \{\textbf{abort}\} & \text{if } (\neg gg \wedge g)(S) = true \\ \emptyset & \text{if } g(S) = false \end{cases}$$
$$= \Gamma(g \rightarrow \textbf{if } \mathcal{F}(A_1, F)[] \ldots []\mathcal{F}(A_n, F) \textbf{ fi})(S)$$
$$= \Gamma(\mathcal{F}(A, F))(S)$$

Case 4 : if $c =$ **do** $A_1[] \ldots []A_n$ **od**, $A_i = g_i \rightarrow c_i$ for $i = 1, \ldots, n$, then

$$\Gamma_F(A)(S) =$$

$$\begin{cases} \Gamma_F(c)(S) & \text{if } g(S) = true \\ \emptyset & \text{otherwise} \end{cases}$$
$$= \begin{cases} \mathcal{L}_F(S) \cap M & \text{if } g(S) \wedge \exists\, k \geq 0 : \forall\, S' \in \Gamma_F^k(\mathcal{A}(S) :: g\mathcal{A}(S') \\ \mathcal{L}_F(S) \cap M \cup \{\perp\} & \text{if } \neg(\exists\, k \geq 0 : \forall\, S' \in \Gamma_F^k(\mathcal{A}(S) :: g\mathcal{A}(S')) \wedge g(S) \\ \emptyset & \text{if } g(S) = false \end{cases}$$

Here $\mathcal{L}_F(S)$ is the least fixed point of the equation

$$Y = \{S\} \cup \Gamma_F(A_1[] \ldots []A_n)(Y)$$

within the CPO $\mathbb{P}(\Psi_P)$. Since

$$\Gamma_F(A_1[] \ldots []A_n)(Y) = \Gamma(\mathcal{F}(A_1, F)[] \ldots []\mathcal{F}(A_n, F))(Y)$$

$\mathcal{L}_F(S)$ is also the least fixed point of the equation

$$Y = \{S\} \cup \Gamma(\mathcal{F}(A_1, F)[] \ldots []\mathcal{F}(A_n, F))(Y)$$

and

$$\Gamma_F^k(\mathcal{A}) = \Gamma^k(\mathcal{F}(\mathcal{A}, F))$$

We thus have

$$\Gamma_F(A) = \Gamma(\mathcal{F}(A, F))$$

□

From this theorem and the failure semantics, we have the following corollary.

Corollary 3.4.1 *Given a program P and its faults F,*

$$F = \emptyset \Rightarrow \mathcal{F}(P, F) \equiv P$$

Therefore, as in the case of UNITY programs, a fault-free execution of the program $\mathcal{F}(P, F)$ is a fault-prone execution of program P and vice-versa, and $\mathcal{F}(P, F)$ is equivalent to P if no fault occurs. Furthermore, as shown in the following theorem, $\mathcal{F}(P, F)$ is equivalent to the union composition of P with a program F_P which can be derived from the text of P and F.

Theorem 3.4.2 *Given the program P and a fault environment F, there is a program F_P such that*

$$\mathcal{F}(P, F) \equiv P[]F_P$$

Proof: Let $P = (Init_P, A_1[] \ldots []A_n)$, $A_i = g_i \rightarrow c_i$, we prove that for each A_i, there is a F_i such that

$$\mathcal{F}(A_i, F) \equiv \textbf{if } A_i[]F_i \textbf{ fi}$$

Case 1 : if c_i is a primitive command,

$$\begin{aligned}\mathcal{F}(A_i, F)) &= g_i \rightarrow \textbf{if } true \rightarrow c_i[]F \textbf{ fi} \\ &\equiv \textbf{if } g_i \rightarrow c_i[]g_i \rightarrow F[]\neg g_i \rightarrow \textbf{skip fi}\end{aligned}$$

Case 2 : if $c_i = c_{i1}; c_{i2}$ and $\mathcal{F}(c_{ij}, F) \equiv \textbf{if } c_{ij}[]F_{c_{ij}} \textbf{ fi}$ for $j = 1, 2$, then

$$\begin{aligned}\mathcal{F}(A_i, F)) &= g_i \rightarrow \mathcal{F}((c_{i1}; c_{i2}), F) \\ &= g_i \rightarrow \mathcal{F}(c_{i1}, F); \mathcal{F}(c_{i2}, F) \\ &\equiv g_i \rightarrow \textbf{if } c_{i1}[]F_{c_{i1}} \textbf{ fi}; \textbf{ if } c_{i2}[]F_{c_{i2}} \textbf{ fi} \\ &\equiv \textbf{if } g_i \rightarrow c_{i1}; c_{i2}[]g_i \rightarrow c_{i1}; F_{c_{i2}}[]g_i \rightarrow F_{c_{i1}}; c_{i2} \\ &\quad []g_i \rightarrow F_{c_{i1}}; F_{c_{i2}}[]\neg g_i \rightarrow \textbf{skip fi}\end{aligned}$$

Case 3 : if $c_i = IF_{j=1}^{n_i} A_{ij}$, and $\mathcal{F}(cA_{ij}, F) \equiv \textbf{if } cA_{ij}[]F_{ij} \textbf{ fi}$,

$$\begin{aligned}\mathcal{F}(A_i, F) &= g_i \rightarrow IF_{j=1}^{n_i} \mathcal{F}(A_{ij}, F) \\ &\equiv g_i \rightarrow IF_{j=1}^{n_i}(gA_{ij} \rightarrow \textbf{if} cA_{ij}[]F_{ij}\textbf{fi}) \\ &\equiv g_i \rightarrow IF_{j=1}^{n_i}(gA_{ij} \rightarrow cA_{i1}[]gA_{ij} \rightarrow F_{ij}) \\ &\equiv \textbf{if } []_{j=1}^{n_i}(g_i \wedge gA_{ij} \rightarrow cA_{ij}[]g_i \wedge gA_{ij} \rightarrow F_{ij})[]\neg g_i \rightarrow \textbf{skip fi} \\ &\equiv \textbf{if } A_i[]([]_{j=1}^{n_i} g_i \wedge gA_{ij} \rightarrow F_{ij})[]\neg g_i \rightarrow \textbf{skip fi}\end{aligned}$$

Case 4 : the proof for the case of iterative composition is similar to **Case 3**.

Therefore,

$$\begin{aligned}\mathcal{F}(P,F) &= (Init_P, \mathcal{F}(A_1,F)[] \ldots [] \mathcal{F}(A_n,F))\\ &\equiv (Init_P, \textbf{if } A_1 [] F_{A_1} \textbf{ fi } [] \ldots [] \textbf{ if } A_n [] F_{A_n} \textbf{ fi})\\ &\equiv P[](Init_P, F_{A_1}[] \ldots [] F_{A_n})\end{aligned}$$

□

We now define the $\oplus$ composition as

$$P \oplus F \triangleq P[]F_P$$

where F_P is obtained from the proof of Theorem 3.4.2. And we have

$$(P_1[]P_2) \oplus F \equiv (P_1 \oplus F)[](P_2 \oplus F)$$

Corollary 3.4.2 *If P is a UNITY program taken as an action system, then*

$$P \oplus F \equiv P[]F$$

Proof: Directly from the proof of Theorem 3.4.2 and the fact that each action A of P is a primitive command.

□

Fail-stop Fault Transformation

As for the definition of the f-stop fault transformation of a UNITY program, we define the f-stop fault transformation $\mathcal{F}_f$ of a program P to be the composition of the fault transformation $\mathcal{F}$ and a *f-stop transformation* $\mathcal{T}_f$,

$$\mathcal{F}_f(P,F) \triangleq \mathcal{F}(\mathcal{T}_f(P),F)$$

The f-stop transformation $\mathcal{T}_f$ transforms an action A into it *f-stop version* $\mathcal{T}_f(A)$:

$$\mathcal{T}_f(A) \triangleq \neg f \wedge gA \rightarrow \mathcal{T}_f(cA)$$

The f-stop transformation $\mathcal{T}_f$ transforms each command c into its *f-stop version* $\mathcal{T}_f(c)$:

1. for a primitive command c,
   $\mathcal{T}_f(c) \triangleq$ **if** $\neg f \rightarrow c [] f \rightarrow$ **skip fi**
2. $\mathcal{T}_f(c_1; c_2) \triangleq \mathcal{T}_f(c_1); \mathcal{T}_f(c_2)$,
3. $\mathcal{T}_f(IF_{i=1}^n A_i) \triangleq$
   if $\neg f \wedge gA_1 \rightarrow \mathcal{T}_f(cA_1) [] \ldots [] \neg f \wedge gA_n \rightarrow \mathcal{T}_f(cA_n) [] f \rightarrow$ **skip fi**,
4. $\mathcal{T}_f(DO_{i=1}^n A_i) \triangleq DO_{i=1}^n \mathcal{T}_f(A_i)$

Given a program $P = (Init_P, A_1 [] \ldots [] A_n)$, its f-stop version $\mathcal{T}_f(P)$ is defined as:

$$\mathcal{T}_f(P) \triangleq (Init_P, \mathcal{T}_f(A_1) [] \ldots [] \mathcal{T}_f(A_n))$$

Similar to the case of fault transformation, the f-stop fault transformation can be justified by the following theorem.

Theorem 3.4.3 *Given a program P and its fault environment F, the f-stop fault transformation $\mathcal{F}_f$ satisfies*

$$\Gamma(\mathcal{F}_f(P, F)) = \Gamma_F^f(P)$$

Proof: As for Theorem 3.4.1. □

From Theorem 3.4.2, the following corollary is obvious.

Corollary 3.4.3 *Given a program P and its fault environment F,*

$$\mathcal{F}_f(P, F) \equiv \mathcal{T}_f(P) \oplus F$$

Thus there always exists a program $F_{\mathcal{T}_f(P)}$ which can be derived from the texts of P and F such that

$$\mathcal{F}_f(P, F) \equiv \mathcal{T}_f(P) [] F_{\mathcal{T}_f(P)}$$

3.4.4 Generalization of the Fault Transformation

So far the fault program F has been treated as a presentation of the fault environment in which a program P is to be executed. The failure semantics and the fault-tolerant transformations have been defined based on the assumption that each fault action in F may interrupt the execution of any action of P, and that each occurrence of a fault in F will stop the execution of any action of P if the f-stop semantics or transformation is used. Such an assumption simplifies the failure semantics, the fault transformations and the discussion of their properties. But in general, the execution of an action or a command of P can be affected by only those faults that affect the resources that it uses. The greater the set of faulty resources used, the more fault actions will be involved.

Therefore, for a program P and its fault environment F, each command c of P has its own fault environment $ENV_F(c)$ which is a subset of F. $ENV_F(c)$ represents the faults which may occur during the execution of c. The fault environment of a command (or an action A) c can be calculated from that of the primitive commands in c (or the action A) using the rules described below.

$$ENV_F(c_1; c_2) = ENV_F(c_1) \cup ENV_F(c_2)$$

$$ENV_F(IF_{i=1}^{n} A_i) = \bigcup_{i=1}^{n} ENV_F(cA_i)$$

$$ENV_F(DO_{i=1}^{n} A_i) = \bigcup_{i=1}^{n} ENV_F(cA_i)$$

$$ENV_F(A) = ENV_F(cA)$$

Based on the above discussion, the definition for the failure semantics $\Gamma_F(c)$ of a primitive command c is then generalized to:

$$\Gamma_F(c)(S) = \Gamma(c)(S) \cup \bigcup_{a \in ENV_F(c)} \Gamma(a)(S)$$

The generalized definitions for failure semantics for the composition commands actions, and thus programs can be then obtained in the compositional way given in Section 3.4.1.

Note that fault occurrence is different from *error propagation*. This is illustrated in the example below.

Example 3.4.1 Consider a program P consisting of two actions, $x := e; y := x$ and $z := y$. Assume that that a fault action A can only occur during the execution of the action $x := e; y := x$ and that it changes the value of x. Although the fault A does not occur in the execution of the action $z := y$, the error caused by A in the execution of $x := e; y := x$ will be propagated to the execution of $z := y$ and z will get an incorrect value.

Depending on such a failure semantics, we can generalize the fault transformation $\mathcal{F}$.

The fault-affected version $\mathcal{F}(c, F)$ of a primitive command c under F is now defined as:

$$\mathcal{F}(c, F) \triangleq \textbf{if } true \rightarrow c[]ENV_F(c) \textbf{ fi}$$

The fault-affected versions of composition commands and action can then be defined compositionally as in Section 3.4.3.

For a program $P = (Init_P, A_1[] \ldots []A_n)$, let

$$F_{rest} = F - \bigcup_{i=1}^{n} ENV_F(A_i)$$

The fault actions in F_{rest} represent the faults which may occur during the time after the termination and before the starting of the execution of an action in the program.

The fault-affected version $\mathcal{F}(P, F)$ is now generally defined as:

$$\mathcal{F}(P, F) = (Init_P, \mathcal{F}(A_1, F)[] \ldots []\mathcal{F}(A_n, F)[]F_{rest})$$

From now on, we can make it clear that the claim " *no fault will occur during the execution of a command c or an action A in the fault environment F*" has the formal meaning which can be described in any one of the following ways:

(a) The fault environment of the command c or the action A is empty, e.g.

$$ENV_F(c) = \emptyset$$

(b) The failure semantics of the command c or the action A equals that of the corresponding semantics, e.g.

$$\Gamma_F(c) = \Gamma(c)$$

(c) The fault-affected version of c or A is equivalent to c or A respectively, e.g.

$$\mathcal{F}(c, F) \equiv c$$

The old version of the fault transformation of a program is thus a special case of the generalized one, if it is assumed that the fault environment of each command c in P is F. Another special case is that the fault environment of each action of P is empty and hence

$$\mathcal{F}(P, F) = P[]F$$

In particular, let $P = P_1[]\ldots[]P_k$ be the union of k programs (e.g. processes) and F_{P_i} be the fault environment of each command of program P_i. Then

$$\mathcal{F}(P, F) = P_1 \oplus F_{P_1}[]\ldots[]P_k \oplus F_{P_k}$$

where $F = \bigcup_{i=1}^{k} F_{P_i}$.

Similarly, the fail-stop assumption can be applied to only a part of a program P. In communicating systems, for example, it is difficult or even impossible to restore the state of the channels after faults occur. Error messages which have been transferred to the channel cannot be discarded. To prevent a process from transferring an error message, the sending command in the process has to be fail-stop. This means a sending command cannot be executed after faults occur in the process.

To make a command (or an action) c in a program P f-stop, P is transformed to a program P' by replacing c in P with its f-stop version. And then the fault transformation is applied to P' to obtain the general version of the f-stop fault-affected program of P.

All the results proved in Section 3.4.3 for the fault transformation still hold for the generalized version described above.

From the discussions about the failure semantics and fault transformation that have been presented in this section, we can reason about both the functional and fault properties of a command c or an action A by reasoning about the functional properties of $\mathcal{F}(c)$ or $\mathcal{F}(A)$. Rules for doing this reasoning are described below and will be used, in the following two sections to support the discussion about refinement of fault and fault-tolerant properties of a program.

For a primitive command c,

$$\frac{\{Q\}c\{R\},\ \ \{Q\}ENV_F(c)\{R_f\}}{\{Q\}\mathcal{F}(c)\{R \vee R_f\}}$$

For a sequential composition command $c_1; c_2$,

$$\frac{\{Q\}\mathcal{F}(c_1)\{R_1\},\ \ \{R_1\}\mathcal{F}(c_2)\{R\}}{\{Q\}\mathcal{F}(c_1; c_2)\{R\}}$$

For a conditional composition command $IF_{i=1}^n$,

$$\frac{\{Q\}\mathcal{F}(A_i)\{R\},\ \ i = 1, \ldots, n}{\{Q\}\mathcal{F}(IF_{i=1}^n A_i)\{R\}}$$

For an iterative command $DO_{i=1}^n A_i$,

$$\frac{\{Q_1 \wedge gA_i\}\mathcal{F}(A_i)\{Q_1\},\ \ Q \Rightarrow Q_1,\ \ Q_1 \wedge \neg gg \Rightarrow R,\ \ i = 1, \ldots, n}{\{Q\}\mathcal{F}(DO_{i=1}^n A_i)\{R\}}$$

where $\neg gg \triangleq \bigwedge_{i=1}^n \neg gA_i$.

For an action A, if $\{Q\}cA\{R\}$, then

$$\{Q \wedge gA\}A\{R\}$$

$$\{\neg gA\}A\{false\}$$

Using induction and the definition of the fault transformation, we can prove that for a command c (an action A),

$$\{Q\}\mathcal{F}(c)\{R\}) \Rightarrow \{Q\}c\{R\}$$

3.5 Refining Fault Properties

Refinement of a program P changes the way in which it uses the resources of the system. The fault environment F, on the other hand, is a representation of the effects of faults in the resources on the execution of P. When a higher level program P is refined to a lower level program P', the corresponding fault environment F of P must therefore be refined to a lower level representation F' which is more concrete than F. This is illustrated in the example below, in which we just give the actions of the programs.

Example 3.5.1 Consider the design of a program which computes the sum of the squares of the elements X_i in a sequence X of n real numbers.

We can start with the following specification.

$$P_0 : \quad s := \sum_{i=1}^{n} X_i^2$$

At this stage, all we know about the fault environment F_0 is that a fault may occur in the execution of the primitive command in the program. When a fault occurs, s may contain any value except the desired one. F_0 is thus written as one action:

$$F_0 : \quad s, \; f := s_0.(s_0 \neq \sum_{i=1}^{n} X_i^2), \; \mathit{true}.$$

We then refine P_0 into P_1 by splitting the computation into two parts, squaring and summing.

$$P_1 : \quad M := < X_1^2, \ldots, X_n^2 >; \; s := \sum M[i]$$

Now we know that faults may occur during the computations of both commands. However, since the operations $< X_1^2, \ldots, X_n^2 >$ and $\sum$ are not executable, we do not know how the values of the variables are changed by the fault action. The fault environment F_1 remains the same as F.

Let us refine P_1 further into program P_2 and implement squaring and summing by a loop.

$$\begin{array}{ll} P_2 : & M := <>; \; i := 1; \; s := 0 \\ & \textbf{do } i \leq n \rightarrow M := M \,\hat{}\, < X_i^2 >; \; i := i + 1 \\ & [] \;\; M \neq <> \rightarrow y := \mathit{head}(M); \; M := \mathit{rest}(M); \; s := s + y \\ & \textbf{od} \end{array}$$

At this stage, let us assume that faults can only occur in the execution of the concatenation $M \,\hat{}\, < X_i^2 >$ and the operation $\mathit{rest}(M)$. Furthermore, we know the failure semantics of both $M := M \,\hat{}\, < X_i^2 >$ and $M := \mathit{rest}(M)$ is to keep M unchanged if a fault occurs.

The fault environment F is then represented by the program.

$$F_2 : \quad M, f := M, \mathit{true}$$

which is in the fault environments of both

$$M := M \,\hat{}\, < X_i^2 > \; \text{ and } \; M := \mathit{rest}(M)$$

The fault-affected versions of P and P_1 are then refined to the program:

$$
\begin{array}{ll}
\mathcal{F}(P_2, F_2): & M := <>;\ i := 1;\ s := 0 \\
& \textbf{do}\ i \leq n \rightarrow \textbf{if}\ M := M \,\hat{}\, < X_i^2 > [] F_2\ \textbf{fi};\ i := i + 1 \\
& []\ \ M \neq <> \rightarrow y := \mathit{head}(M); \\
& \quad \textbf{if}\ M := \mathit{rest}(M) [] F_2\ \textbf{fi}; \\
& \quad s := s + y \\
& \textbf{od}
\end{array}
$$

For $k = 1, 2$ in this example, F_k is called a *presentation* of F_{k-1} (and F_{k-1} is called an *abstraction* of F_k) corresponding to the refinement $P_{k-1} \sqsubseteq P_k$.

Let c be a command (or an action) in program P and $c \sqsubseteq c'$. In general, by analyzing the operations used in c and c', the fault environment $ENV_F(c)$ can be refined to a presentation $ENV(c')$ such that

$$\mathcal{F}(c, F) \sqsubseteq \mathcal{F}(c', F')$$

where the change of F into F' is defined by removing from F the fault actions F_c which can only occur in the execution of c and then adding the fault actions in $ENV(c')$:

$$F' = (F - F_c) \cup ENV(c')$$

F' is called a *presentation* of F (and F is called an *abstraction* of F') corresponding to the refinement step from P to the program P' which is obtained by replacing c in P with c'.

We take the convention that when the refinement $P \sqsubseteq P'$ is done for the original program P, $\mathcal{F}(P) \sqsubseteq \mathcal{F}(P')$ implies that a presentation F' of the fault environment F in $\mathcal{F}(P)$ is also obtained in $\mathcal{F}(P')$

The choice of the fault-tolerant strategies is based on the analysis fault properties of a program. On the other hand, required details about the fault properties depend on the strategy which is chosen for fault-tolerance. For backward recovery and replication techniques, the important information is about the parts of the program in which faults may occur. It is not necessary to care about the way in which a fault may change the variables. However, forward recovery techniques depend on use of partial information remaining in the error state and on the identification of the cause of fault.

3.6 Refining Fault-tolerant Properties

In Section 2.5, we introduced a set of recovery actions P_R to satisfy a high level specification and defined the recovery transformation $\mathcal{R}$ for a program P and its fault environment.

Consider a program P and its fault environment F written as action systems. We apply the same recovery transformation $\mathcal{R}$ to P:

$$\mathcal{R}(P) = P[]P_R$$

Here the recovery program P_R is written as an action system:

Program P_R:
$f \rightarrow (\|_x : x \in Var(P) :: x := v_x).REC_P \; ; \; f := false$
End $\{P_R\}$

where

$$REC_P \triangleq \exists\, cons \in ConFun\, \forall\, x \in Var(P) : (v_x = cons(ob)(x))$$

Since, as mentioned before, the failure semantic execution of an action A of P may abort or fail to terminate unlike its execution in a fault-free environment, the fault-affected program $\mathcal{F}(P, F)$ does not have the property:

$$\mathcal{F}(P, F)\ \underline{Sat}\ \textbf{stable}\ (f \wedge (ob = \theta_0))$$

Hence, the fault-tolerant program $\mathcal{R}(P)$ does not satisfy the property:

$$\mathcal{F}(P, F)[]P_R\ \underline{Sat}\ (f \wedge ob = \theta_0) \longmapsto ((\neg f)_{ob} \wedge ConsExt(ob, \theta_0))$$

As an example, consider a program P with two actions $x := 2$ and $y := 4/x$, where $4/x$ is defined as the division of 4 by x and is undefined if $x = 0$. Let the initial value of x be 1. Although both of these two action will terminate in a fault-free environment, $y := 4/x$ will not terminate if a fault occurs and changes the value of x to 0.

Therefore, none of the theorems in Section 2.6, except Theorem 2.6.1, generally hold for the program

$$\mathcal{F}(\mathcal{R}(P), F) \triangleq \mathcal{F}(P, F)[]P_R$$

However, if we transform P into a program P_f by transforming each action A (not necessarily the commands in A) of P into its f-stop version A', all the properties in the specification of P_R given in Section 2.5 will hold for $\mathcal{F}(P_f, F)[]P_R$. This implies that all the theorems in Section 2.6 will hold for $\mathcal{F}(P_f, F)[]P_R$.

This section presents refinements for fault-tolerant versions of a program. Such refinements are described as *fault-tolerant* refinements. It will be shown that problems such as ruling out abortion and nontermination caused by the fault environment, can be solved by fault-tolerant refinement, using intermediate states to implement P_R. We use the same symbol P_R to represent the action in P_R.

A program P is called a *fault-tolerant* program $w.r.t.$ the fault environment F, if every execution of P can eventually recover from any error states by restarting from a consistent state. This is formalized as: there is a program P_f which is obtained from P by transforming a set (which is possibly empty) of actions of P into their f-stop versions and

$$\mathcal{F}(P_f, F)\ \underline{Sat}\ (f \wedge (ob = \theta_0) \longmapsto (\neg f)_{ob} \wedge ConsExt(ob, \theta_0))$$

Therefore, for each program P, $\mathcal{R}(P)$ is a program which tolerates F.

Let us restrict the refinement relation between programs to the class of programs which tolerate the fault environment. Such a restricted relation is called *fault-tolerant refinement*.

Let P be a fault-tolerant program with fault environment F'. P is said to be *fault-tolerant refined* by a program P' with fault environment, denoted $P \sqsubseteq_{T(F,F')} P'$, if

$$\mathcal{F}(P, F) \sqsubseteq \mathcal{F}(P', F')$$

When there is no confusion, we write $P \sqsubseteq_{T(F)} P'$ for $P \sqsubseteq_{T(F,F')} P'$.

We below present some rules for fault-tolerant refinement.

FT-Rule 1 *Fault-tolerance can be introduced either before or after the refinement of the original program.*

$$\frac{P \sqsubseteq P'}{\mathcal{R}(P) \sqsubseteq_{T(F)} \mathcal{R}(P')}$$

FT-Rule 2 *Fault-tolerant refinement is transitive.*

$$\frac{P \sqsubseteq_{T(F)} P_1, P_1 \sqsubseteq_{T(F)} P_2}{P \sqsubseteq_{T(F)} P_2}$$

Since the bodies of the backward and forward recovery actions are refined versions of the body of the recovery action P_R, we have the following rules.

FT-Rule 3 *Backward and forward recovery are implementations of the general recovery transformation, for a program P*

1. $\mathcal{R}(P) \sqsubseteq_{T(F)} P[]P_{\overleftarrow{R}}$

2. $\mathcal{R}(P) \sqsubseteq_{T(F)} P[]P_{\overrightarrow{R}}$

The example below shows a fault-tolerant program obtained by using the fault-tolerant transformation and refinement rules, and in which the execution recovers by re-starting from the initial state whenever a fault occurs [JM83, HH87].

Example 3.6.1 Let $\forall\, x \in \mathit{Var}(P) : x = v_x$ be the initial state of program P and

Program $P_{R'}$:
$f \rightarrow \|_x: x \in \mathit{Var}(P) :: x := v_x \; ; \; f := \mathit{false}$
End $\{P_{R'}\}$

then

$$\mathcal{R}(P) = P[]P_R \sqsubseteq_{T(F)} P[]P_{\overleftarrow{R}} \sqsubseteq_{T(F)} P[]P_{R'}$$

Let $P = (\mathit{Init}_P, A_1[] \ldots []A_n)$ be a program and I be an non-empty subset of $\{1, \ldots, n\}$. For $i = 1, \ldots, n$, we define a transformation $\mathcal{R}_I$ by

$$A'_i = \begin{cases} A_i & \text{if } i \notin I \\ \textbf{if } A_i[]\neg gA_i \rightarrow \textbf{skip fi}; \textbf{if } \neg f \rightarrow \textbf{skip}[]P_R \textbf{ fi} & \text{if } i \in I \end{cases}$$

and

$$\mathcal{R}_I(P) = (\mathit{Init}_P, A'_1[] \ldots []A'_n)$$

FT-Rule 4 *Given* $P = (Init_P, A_1[] \dots [] A_n)$ *and* I *as an non-empty subset of* $\{1, \dots, n\}$,

1. $\mathcal{R}(P) \sqsubseteq_{T(F)} \mathcal{R}_I(P)$

2. $I \subseteq J \Rightarrow \mathcal{R}_I(P) \sqsubseteq_{T(F)} \mathcal{R}_J(P)$

FT-Rule 5 *If an action* A'_i *of* $\mathcal{R}_I(P)$ *can be written as* $a_{i1}; a_{i2}$, *let*

$$A''_i = a_{i1}; \textbf{if}\ \neg f \rightarrow a_{i2} [] f \rightarrow P_R\ \textbf{fi}$$

Then

$$\mathcal{R}_I(P) \sqsubseteq_{T(F)} \mathcal{R}_I(P)[A'_i / A''_i]$$

It is noted that for any command c

$$c \equiv \textbf{skip};\ c \equiv c;\ \textbf{skip}$$

And we assume here that the fault environment of **skip** is always empty:

$$\mathcal{F}(c) \equiv \mathcal{F}(\textbf{skip};\ c) \equiv \mathcal{F}(c;\ \textbf{skip})$$

We can therefore introduce the recovery action P_R (and its refined versions), by replacing such a **skip** with P_R, at any suitable place. This implies that we can put the recovery action before a command which when executed in an error state may abort or not terminate. In this way, we can rule out the possible **abort** and $\perp$ caused by faults.

3.7 Fault-tolerant Refinement of Commands

Given a command c in a program P, we say that c is *fault-tolerant refined* by a command c', denoted $c \sqsubseteq_{T(F)} c'$ if

$$c \sqsubseteq \mathcal{F}(c')$$

From the definition of the refinement relation $\sqsubseteq$, $c \sqsubseteq_{T(F)} c'$ iff for any assertions Q and R,

$$\{Q\}c\{R\} \Rightarrow \{Q\}\mathcal{F}(c')\{R\}$$

This implies that c' is either a command whose fault environment is empty:

$$c' = \mathcal{F}(c')$$

or c' contains component commands that overcome the faults occur in the execution of c'. In both cases, c' will always carry out the computation that c is required to carry out in a fault-free environment. But c' does not need to tolerate faults outside it. In other words, starting from a good state, the execution of c' always produces the correct result with respect to the specification of c, but starting from an error state, it can only be guaranteed that no further damage caused by faults in the execution of c' will be exposed.

The fault-tolerant refinement relation $\sqsubseteq_{T(F)}$ is stronger than the refinement relation $\sqsubseteq$ between commands:

$$c \sqsubseteq_{T(F)} c' \Rightarrow (c \sqsubseteq \mathcal{F}(c') \sqsubseteq c')$$

The relation $\sqsubseteq_{T(F)}$ is thus transitive:

$$c \sqsubseteq_{T(F)} c' \sqsubseteq_{T(F)} c'' \Rightarrow c \sqsubseteq_{T(F)} c''$$

Furthermore,

$$c \sqsubseteq c' \sqsubseteq_{T(F)} c'' \Rightarrow c \sqsubseteq_{T(F)} c''$$

But it is not, in general, reflexive,

$$c \sqsubseteq_{T(F)} c$$

does not generally hold.

A command c is *fault-tolerant* if it is fault-tolerant refined by itself:

$$c \sqsubseteq \mathcal{F}(c)$$

Since there is no reflexivity for $\sqsubseteq_{T(F)}$, replacement of a component command t with its fault-tolerant refinement t' in a command c cannot guarantee $c \sqsubseteq_{T(F)} c[t/t']$.

However $\sqsubseteq_{T(F)}$ has the properties described in the following theorem.

Theorem 3.7.1 (Fault-tolerant Refinement of Commands)

For $i = 1, \ldots, n$, let $c_i \sqsubseteq_{T(F)} c_i'$ and c be a command containing t as a sub-command, then

1. $c_1; \ldots; c_n \sqsubseteq_{T(F)} c_1'; \ldots; c_n'$
2. **if** $g_1 \rightarrow c_1 [] \ldots [] g_n \rightarrow c_n$ **fi** $\sqsubseteq_{T(F)}$ **if** $g_1 \rightarrow c_1' [] \ldots [] g_n \rightarrow c_n'$ **fi**
3. **do** $g_1 \rightarrow c_1 [] \ldots [] g_n \rightarrow c_n$ **od** $\sqsubseteq_{T(F)}$ **do** $g_1 \rightarrow c_1' [] \ldots [] g_n \rightarrow c_n'$ **od**
4. $(c \sqsubseteq_{T(F)} c) \wedge (t \sqsubseteq_{T(F)} t') \Rightarrow c(t) \sqsubseteq_{T(F)} c[t/t']$

Proof: From the definition of $\sqsubseteq_{T(F)}$ and Theorem 3.3.1. □

We say that a command c' is of better failure semantics than command c, if

$$\mathcal{F}(c) \sqsubseteq \mathcal{F}(c')$$

Fault-tolerant refinement of commands provides a formal way for describing how to improve the failure semantics of a program or a part of a program.

Theorem 3.7.2 (Improving Failure Semantics)

For a program P and its fault environment F, if c is a command of P and is fault-tolerant refined by c', then

$$\mathcal{F}(P) \sqsubseteq \mathcal{F}(P[c/c'])$$

where $P[c/c']$ is the program obtained from P by replace c with c'.

Proof: Note that $\mathcal{F}(c) \sqsubseteq c$ always holds. The theorem is then proved from the transitivity and monotonicity of the refinement relation of commands. □

It is noted that with programs of finite size, and a finite amount of hardware, it is impossible to tolerate infinite occurrences of faults during the execution of a part of a program. We next present a fault-tolerant refinement rule which can be used when the number of occurrences of faults is assumed to be bounded (Section 2.3.3).

FT-Rule 6 *For $i = 1, \ldots, k$, assume that*

i) $c_1 \sqsubseteq c_i$,

ii) $c_1 \sqsubseteq c_1; \ldots; c_k$,

iii) $c_1 \sqsubseteq \mathcal{F}(c_1; \ldots; c_k)[\mathcal{F}(c_i)/c_i]$

iv) the number bound *of occurrences of faults in the fault environment of* $c_1; \ldots; c_k$ *is less than* k.

Then

$$c_1 \sqsubseteq_{T(F)} c_1; \ldots; c_k$$

How large k should be depends upon the assumption about the bound of the occurrences of faults. We focus on the construction of the commands described in the above theorem and leave the quantitative argument about such a bound to the designer. We will use it in a fault-tolerant context to keep it independent of the argument about the bounds.

For a program P, we define a command $Save_P$. The execution of $Save_P$ in a good state S saves the value of each variable x in S in a newly introduced variable rec_x:

$$Save_P = \textbf{if}\ \neg f \rightarrow \|_x\!: x \in Var(P) :: rec_x := x [] f \rightarrow \textbf{skip fi}$$

Corresponding to the *checkpointing* command $Save_P$, we define a *restoring* command $Restore_P$:

$$Restore_P = \|_x\!: x \in Var(P) :: x := rec_x$$

It is assumed that faults do not occur in the executions of $Save_P$ and $Restore_P$, and that any *checkpointing* variable rec_x cannot be corrupted by faults.

For $j = 1, \ldots, k$, let $c \sqsubseteq c_j$ and

$$Recr_j \triangleq \textbf{if } f \rightarrow Restore_P; c_j [] \neg f \rightarrow \textbf{skip fi}$$

Let

$$Recr[c :: \ j : 1..k :: \ c_j]_P \triangleq Recr_1; \ldots; Recr_k$$

We then define a *fault-tolerant block* of c:

$$[\mid c :: \ j : 1..k :: \ c_j \ \mid]_P \triangleq Save_P; \textbf{if } f \rightarrow c [] \neg f \rightarrow c; Recr_1; \ldots; Recr_k \textbf{ fi}$$

Corollary 3.7.1 *In the fault-tolerant block* $[\mid c :: \ j : 1..k :: \ c_j \ \mid]_P$*, the commands* $c, Recr_1, \ldots, Recr_k$ *satisfy the first three conditions described in* **TF-Rule** *6.*

Given a program $P = (Init_P, A_1 [] \ldots [] A_n)$, let

$$Recr_P \triangleq f \rightarrow Recr[IF_{i=1}^n A_i :: \ j : 1..k :: \ c_{ij}]_P$$

and

$$Redo_P \triangleq f \rightarrow \textbf{do } f \rightarrow Restore_P; c \textbf{ od}$$

where $IF_{i=1}^n A_i \sqsubseteq c$.

We assume that the initial value of each checkpointing variable rec_x is the same as that of x:

$$Init \triangleq Init_P \cup \{rec_x = x\}$$

We can have a program

$$P' \triangleq (Init, Save_P [] A_1 [] \ldots [] A_n [] Recr_P)$$

The values of the checkpointing variables constitute a state which is always backward consistent with the current observation restricted to $Var(P)$,

$$\mathcal{F}(P') \ \underline{Sat} \ (\textbf{invariant } BackwCon(Recv_P, ob))$$

where $Recv_P = \{rec_x \mid x \in Var(P)\}$ and ob is the observation variable of P. This discussion leads to the refinement rule presented below.

FT-Rule 7 *Let* $P = (\mathit{Init}_P, A_1[] \ldots [] A_n)$. *Then*

1. $\mathcal{R}(P) \sqsubseteq_{T(F)} P[]P_{\overline{R}} \sqsubseteq_{T(F)} (\mathit{Init}, \mathit{Save}_P[]A_1[] \ldots []A_n[]\mathit{Recr}_P)$
2. $\mathcal{R}(P) \sqsubseteq_{T(F)} P[]P_{\overline{R}} \sqsubseteq_{T(F)} (\mathit{Init}, \mathit{Save}_P[]A_1[] \ldots []A_n[]\mathit{Redo}_P)$
3. *Specifically,*

   $P[]P_{\overline{R}} \sqsubseteq_{T(F)} (\mathit{Init}, \mathit{Save}_P[]A_1[] \ldots []A_n[]f \rightarrow \mathit{Restore}_P)$

The rule described below shows how to use intermediate states for recovery.

FT-Rule 8 *Let* $P \sqsubseteq_{T(F)} P$ *and* c *be a command of* P. *Then for any fault-tolerant block* $[\mid c :: \ j : 1..k :: \ c_j \ \mid]_P$ *of* c,

$$P \sqsubseteq_{T(F)} P[c/[\mid c :: \ j : 1..k :: \ c_j \ \mid]_P]$$

Proof: From Corollary 3.7.1. □

Instead of refining c into a block with alternative c_j, c itself can be repeated under certain conditions in the fault environment. A little bit more generally, let $c \sqsubseteq c'$ and

$$[\mid c :: \ \textbf{redo}\ c' \ \mid]_P \triangleq Save_P;\ \textbf{if}\ f \rightarrow c[]\neg f \rightarrow c;\ Redo_{c'}\ \textbf{fi}$$

where

$$Redo_{c'} \triangleq \textbf{if}\ \neg f \rightarrow \textbf{skip}[]f \rightarrow \textbf{do}\ f \rightarrow Restore_P; c'\ \textbf{od fi}$$

This redo block provides fault-tolerance in a fault-environment with finite error assumption.

FT-Rule 9 *If* $c \sqsubseteq c'$, *and the number of occurences of fault in the fault environment of* $[\mid c :: \ \textbf{redo}\ c' \ \mid]_P$ *is finite,*

$$c \sqsubseteq_{T(F)} [\![c :: \textbf{redo } c']\!]_P$$

In a fault-tolerant program P, recovery is carried out by execution of an action $Recr$ such that

$$gRecr \Rightarrow f$$

The action $Recr$ is either in Act_P or contained in an action $A \in Act_P$ as:

$$A = A[c; \textbf{if } B[\,]Recr \textbf{ fi}]$$

We say a recovery action $Recr$ is *redundant* if it is never enabled under the fault environment. Formally, $Recr$ is redundant if

(a) $Recr \in Act_P$ and $\neg gRecr$ is invariant in $\mathcal{F}(P, F)$, or

(b) $Recr$ is contained in an action $A \in Act_P$ as:

$$A = A[c; \textbf{if } B[\,]Recr \textbf{ fi}]$$

and the command c has the property

$$\{true\}\mathcal{F}(c, F)\{\neg gRecr\}$$

FT-Rule 10 *Any redundant recovery action in a fault-tolerant program P can be replaced by* **skip**

3.8 Example: A Protocol for Communication Over Faulty Channels

In this section, we consider an example of fault-tolerant programming using the methods presented in this chapter. The problem is to design a protocol that guarantees reliable communication from a sender to a receiver in spite of faults in the communication channel between them. In the following programs, we will omit the declaration and initial values of the variables if they are clear from the context.

The *Sender* process produces an infinite sequence ms_0 of data. The *Receiver* process reads in a sequence mr satisfying the following specification:

invariant $mr \underline{\propto} ms_o$ (i.e. mr is a prefix of ms_0)
$\#mr = n \longmapsto \#mr = n+1$ (i.e. the length of mr increases eventually)

If the sender and the receiver communicate over an unbounded reliable FIFO channel c, the communication between them can be implemented using the following program $Sender\text{-}Receiver$:

Program $Sender\text{-}Receiver$
initially
$ms, mr, c = ms_0, <>, <>$
actions
$c, ms := c\,\hat{}\,head(ms), rest(ms)$ (*Sender*)
[]
$c \neq <> \rightarrow c, mr := rest(c), mr\,\hat{}\,head(c)$ (*Receiver*)
End $\{Sender\text{-}Receiver\}$

The FIFO channel c can be implemented as the following program:

Program C
initially
$cs, cr = <>, <>$
actions
$cs \neq <> \rightarrow cs, cr := rest(cs), cr\,\hat{}\,head(cs)$
End $\{C\}$

Let program P be

Program P
initially
$ms, mr, cs, cr = ms_0, <>, <>, <>$
actions
$cs, ms := cs\,\hat{}\,head(ms), rest(ms)$
[]
$cr \neq <> \rightarrow mr, cr := mr\,\hat{}\,head(cr), rest(cr)$
End $\{P\}$

Then,

$$Sender\text{-}Receiver \sqsubseteq C[]P$$

Now assume that the channel C is faulty and has the following behaviour:

1. any message sent along the channel may be lost; however, only a finite number of messages can be lost consecutively;
2. any message sent along the channel may be replicated, but no message can be replicated forever;
3. messages are not permuted – i.e. messages are delivered in the order in which they are sent; and
4. messages are not corrupted – i.e. their contents are not altered.

A program F simulating such fault environment can be given as:

Program F
  declare
    $b, f\ :\ boolean$
  initially
    $cs\ , cr, b, f = <>, <>, false, false$
  actions
    $\neg b \wedge cs \neq <> \rightarrow cs, f := rest(cs), true$ (loss)
   []
    $\neg b \wedge cs \neq <> \rightarrow cr, f := cr\,\hat{}\,head(cs), true$ (duplication)
   []
    $b := true$ (guarantee the finiteness of consecutive faults)
End $\{F\}$

Consider the refinement $C \sqsubseteq C_1$, where

Prgram C_1
  initially
    $cs, cr, b = <>, <>, false$
  actions
    $cs \neq <> \rightarrow cs, cr, b := rest(cs), cr\,\hat{}\,head(cs), false$
End $\{C_1\}$

The behaviour of a fault channel can be simulated by a program $FC \triangleq \mathcal{F}(C_1)$ with fail-stop assumption:

Program $\mathcal{F}(C_1)$

declare
$b, f \ : \ boolean$
initially
$b, f = false, false$
actions
$\neg b \wedge cs \neq <> \rightarrow loss$
[]
$\neg b \wedge cs \neq <> \rightarrow duplication$
[]
$\neg f \wedge cs \neq <> \rightarrow correct\text{-}transfer$
[]
$b := true$ (guarantee the finiteness of consecutive faults)
End $\{\mathcal{F}(C_1)\}$

where

1. $loss \triangleq cs, f := rest(cs), true$
2. $duplication \triangleq cr, f := cr\,\hat{}\,head(cs), true$
3. $correct\text{-}transfer \triangleq b, cs, cr := false, rest(cs), cr\,\hat{}\,head(cs)$

And $\mathcal{F}(Sender\text{-}Receiver)$ (or $\mathcal{F}(C[]P)$) is refined to the program $\mathcal{F}(C_1[]P)$ given below.

Program $\mathcal{F}(C_1[]P)$
declare
$b, f \ : \ boolean$
initially
$ms, mr, cs, cr\ ,\ b, f = ms_0, <>, <>, <>, false, false$
actions
$\neg f \rightarrow cs, ms := cs\,\hat{}\,head(ms), rest(ms)$
[]
$\neg f \wedge cr \neq <> \rightarrow mr, cr := mr\,\hat{}\,head(cr), rest(cr)$
[]
$\neg b \wedge cs \neq <> \rightarrow loss$
[]
$\neg b \wedge cs \neq <> \rightarrow duplication$
[]
$\neg f \wedge cs \neq <> \rightarrow correct\text{-}transfer$
[]

$b := true$ (guarantee the finiteness of consecutive faults)
End $\{\mathcal{F}(C_1[]P)\}$

To design a recovery program for $Sender\text{-}Receiver$, let the type of cs and cr be sequence variables whose elements are pairs $(integer,\ data\ item)$. Let $C_1[]P$ be refined to P_1:

Program P_1
declare
$ks, kr\ :\ integer,$
initially
$cs, cr, ks, kr, b\ = <>, <>, 1, 0, false$
actions
$cs\ ,\ ks := cs\,\hat{}\,(ks, ms(ks)), ks + 1$
[]
$cr \neq <> \rightarrow mr, cr, kr := mr\,\hat{}\,head(cr).val, rest(cr)\ ,\ kr + 1$
[]
$cs \neq <> \rightarrow cs, cr, b := rest(cs), cr\,\hat{}\,head(cs), false$
End $\{P_1\}$

Here, some message has been lost if $head(cr).dex > kr + 1$ and some message has been duplicated if $head(cr).dex \leq kr$. Therefore, the fault indicating variable f can be implemented as:

$$f \Leftrightarrow (lost \vee duplic)$$

where

$$lost \triangleq head(cr).dex > kr + 1$$

and

$$duplic \triangleq head(cr).dex \leq kr$$

It is sufficient to assume that only the receiving command is f-stop. Such a fail-stop version of P_1 is then given as $\mathcal{T}_f(P_1)$:

Program $\mathcal{T}_f(P_1)$
initially

$cs, cr, ks, kr, b = <>, <>, 1, 0, false$
actions
$cs, ks := cs \hat{} (ks, ms(ks)), ks + 1$
[]
$\neg f \wedge cr \neq <> \rightarrow mr, cr, kr := mr \hat{} head(cr).val, rest(cr), kr + 1$
[]
$cs \neq <> \rightarrow b, cs, cr := false, rest(cs), cr \hat{} head(cs)$
End $\{\mathcal{T}_f(P_1)\}$

The fault affected version $\mathcal{F}(C_1[]P)$ can be refined to $\mathcal{F}(\mathcal{T}_f(P_1))$:

Program $\mathcal{F}(\mathcal{T}_f(P_1))$
initially
$cs, cr, ks, kr, b = <>, <>, 1, 0, false$
actions
$cs, ks := cs \hat{} (ks, ms(ks)), ks + 1$
[]
$\neg f \wedge cr \neq <> \rightarrow mr, cr, kr := mr \hat{} head(cr).val, rest(cr), kr + 1$
[]
$\neg b \wedge cs \neq <> \rightarrow loss$
[]
$\neg b \wedge cs \neq <> \rightarrow duplication$
[]
$cs \neq <> \rightarrow correct\text{-}transfer$
[]
$b := true$ (guarantee the finiteness of consecutive faults)
End $\{\mathcal{F}(\mathcal{T}_f(P_1))\}$

The recovery program for P_1 in $\mathcal{R}(P_1)$ is refined to P_{1R} given below.

Program P_{1R}
initially
$ks, kr = 1, 0$
actions
$lost \rightarrow$ **do** $cr \neq <> \rightarrow cr := rest(cr)$ **od** ;
do $cs \neq <> \rightarrow cs := rest(cs)$ **od** ;
$ks := kr + 1; cs, ks := cs \hat{} (ks, ms(ks)), ks + 1;$
$cr, cs := cr \hat{} head(cs), rest(cs)$
[]
$duplic \rightarrow$ **do** $duplic \wedge (cr \neq <>) \rightarrow cr := rest(cr)$ **od**
End $\{P_{1R}\}$

Hence, the fault-tolerant program $\mathcal{R}(P_1)$ is fault-tolerant refined to $\mathcal{T}_f(P_1)[]P_{1R}$.

This transformational approach can be also used to establish proof obligations if one does not want to check the refinement at each step. We can proof that $\mathcal{T}_f(P_1)[]P_{1R}$ tolerates the specified faults, i.e. $\mathcal{F}(\mathcal{T}_f(P_1)[]P_{1R})$ satisfies the specification of $Sender\text{-}Receiver$. We give this proof in the following.

Proof: To prove the invariant $mr \underline{\propto} ms_0$:

a) $mr\underline{\propto}ms_0$ is $true$ initially, since $mr = \langle\rangle$ initially.

b) Each action in $\mathcal{F}(\mathcal{T}_f(P_1))$ and P_{1R} leaves $mr\underline{\propto}ms_0$ stable. This can be shown by proving that

$$\forall\, i \leq \#mr : mr(i) = ms_0(i)$$

All the actions in $\mathcal{F}(\mathcal{T}_f(P_1))$ and P_{1R}, except the second one in $\mathcal{F}(\mathcal{T}_f(P_1))$, i.e. the receiving action, do not modify mr while the fail-stop receiving action can transfer $head(cr).val$ to mr only when

$$head(cr).dex = kr + 1 = \#mr + 1$$

Since

$$cr \neq \langle\rangle \Rightarrow (head(cr).val = ms(head(cr).dex) = ms_0(head(cr).dex))$$

the fail-stop receiving action also leaves

$$\forall\, i \leq \#mr : mr(i) = ms_0(i)$$

stable.

Hence $mr\underline{\propto}ms$ is invariant.

To prove the progress property $\#mr = n \longmapsto \#mr = n + 1$:

a) It is easily seen that $\#mr = kr$ is invariant, therefore

$$(\#mr = n \longmapsto \#mr = n + 1) \Leftrightarrow (kr = n \longmapsto kr = n + 1)$$

b) From both $\mathcal{F}(\mathcal{T}_f(P_1))$ and P_{1R}, we have

$$\textbf{stable}\ (kr \geq n)$$

and

$$(kr = n \wedge head(cr).dex = n + 1) \longmapsto (kr = n + 1)$$

c) From the recovery program P_{1R},

$kr = n \wedge head(cr).dex \leq n$ **ensures**
$(cr =<> \vee head(cr).dex > n) \wedge kr = n$

d) From the $correct\text{-}transfer$ action and the action $b := true$,

$$kr = n \wedge cr =<> \longmapsto kr = n \wedge cr \neq <>$$

e) From the recovery program P_{1R},

$$kr = n \wedge head(cr).dex > n + 1 \longmapsto (ks = n + 2 \wedge head(cr).dex = n + 1)$$

Hence we proved the progress property.

□

The fault-tolerant program $\mathcal{T}_f(P_1)[]P_{1R}$ can be further refined by introducing a channel ack which is implemented by two sequence variables $(ackr, acks)$. After it receives a message from $Sender$, $Receiver$ sends an acknowledgment message to $Sender$ to inform $Sender$. $Sender$ sends the next message only after it receives from $Receiver$ the acknowledgment of receiving the last message. Such an implementation was presented in [CM88]

3.9 Discussion

This chapter presented the refinement of fault properties and fault-tolerant properties along with the refinement of the functional properties of the original program. In general, refinement provides the method which allows large programs to be developed step by step, part after part. Refinement of fault-affected programs and fault-tolerant refinement of fault-tolerant programs defined in this chapter can also support such a purpose. That is, the fault properties can be analyzed step by step, for a part of the original program at a time; fault-tolerant properties can be implemented step by step, making part after part of the original program fault-tolerant. This is useful for the design and development of large fault-tolerant program.

From the discussion of the failure semantics of a program, we can see that faults can cause the execution of an action not to terminate, even though the action can be proved to terminate in a fault-free environment. It is also possible that a non-terminating action terminates because of the occurrence of a fault in its execution. Similarly, faults may introduce more or less **abort** in programs. Therefore, the semantic objects $\perp$ and **abort** in the semantic model are necessary for the treatment of fault-tolerance.

In this chapter, we defined fault-tolerant blocks and *redo* blocks which are based on the assumptions that the *Save* and *Restore* commands cannot be affected by any faults and values of the *recording variables* can never be corrupted by any faults. To fulfill such an assumption, it is required that the hardware has enough redundant storage to make the recording variables safe, and the *Save* and *Restore* commands can have multiple implementations to ensure that they will always give the correct output.

Chapter 4

Fault-tolerant Refinement for Parallel Programs

4.1 Introduction

In this chapter, we deal with the fault-tolerance issues in parallel programs. Both static and dynamic structuring techniques for fault-tolerant parallel programs will be considered.

The action system formalism provides two different ways for making parallel the execution of a program described as an action system. Each way follows the crucial requirement that all actions the program are atomic: in any parallel execution of a program, two actions can be executed in parallel only if they do not share variables in common.

In the first case, we can model the parallel execution of a program $P = (Init_P, Act_P)$ by introducing a set $Proc = \{p_1, \ldots, p_k\}$ of processes and partitioning the actions Act_P among these processes. This is termed as a *concurrent action system* in [Bac88]. A *shared variable* is a variable that is referred by two or more actions in different processes, while a *private variable* is a variable that is referred to only by actions in one process. Two actions of P can be executed in parallel if and only if they are not in the same process and do not share variables in common. Therefore a concurrent system P with a set $Proc$ of k processes can be written as

$$P = (Init_P, p_1[] \ldots []p_k)$$

where p_i denotes the actions that are assigned to process p_i, $1 \leq i \leq k$.

In a concurrent system, communications between processes are carried out by executing the action body referring to shared variables.

In the other case, we model the parallel execution of P by partitioning the variables $Var(P)$ among the processes $Proc$. This is referred to as a *distributed action system* in [Bac88]. Two actions of a distributed system can be executed in parallel if and only if they do not refer to variables in one common process. A *shared* (or *joint*) action is an action that refers to variables in two or more processes, while a *private* action only refers to variables in one process. A distributed system P with a set $Proc$ of k processes can be then written as:

$$P = (Init_P, Proc, A_1[p_{11}, \ldots, p_{1k_1}][] \ldots []A_n[p_{n1}, \ldots, p_{nk_n}])$$

where $Proc = \{p_1, \ldots, p_k\}$ is a partition of the state variables of P and each $A_j[p_{j1}, \ldots, p_{jk_j}]$ is an action shared by the processes $p_{j1}, \ldots, p_{jk_j}$ in $Proc$, $1 \leq j \leq n$.

The action $A_j[p_{j1}, \ldots, p_{jk_j}]$ is assumed to be executed jointly by the processes $\{p_{j1}, \ldots, p_{jk_j}\}$ that share the action A_j. And these processes are synchronized for the execution of A_j. The action body cA_j provides communication between the processes: variables of one process may be updated by A_j in a way that depends on the value of the variables of other processes sharing this action.

A sequential action system is a special case of a parallel action system where all actions (or all the variables) are assigned to the same process. Another special case is that each action (or each variable) is assigned to its own process. The system is then executed with maximal parallelism.

We can also view a concurrent action system

$$P = (Init_P, p_1[] \ldots []p_k)$$

as a distributed system, with the private variables of each process p_i being local and the shared variables being partitioned among new processes.

Because actions are executed atomically, a parallel computation can be modelled by a sequential computation with interleaved execution of actions. Hence, the set of possible executions of an action system will be the same for a concurrent or distributed action system as for a sequential action system. This allows us to separate the logical behaviour from implementation issues.

In both of the concurrent and distributed cases, the degree of parallelism that can be achieved during the execution of a program P depends up on the atomicity of the actions in the program. We can sometimes decompose an action into a number of smaller actions to increase the parallelism. Refinement techniques for sequential programs, that we summarized in Section 3.3, have been extended to refine the atomicity of actions [Bac88, Bac89].

When both parallelism and fault-tolerance are considered, we realize that a fault-tolerant version of a program P obtained by using the refinement rules presented in Chapter 3 may decrease the degree of parallelism of P. In the fault-tolerant version of P, an action which contains a recovery command (or a checkpointing command or a restoring command) cannot be executed in parallel with any other action. This is because that these additional commands refer to all the variables of program P. In this chapter, we will solve this problem by introducing more fault-tolerant rules which allow the actions containing the additional commands to be executed in a more independent manner. Moreover, under certain conditions, such actions should be split up into a number of smaller actions to increase the parallelism.

After this introduction, we present in Section 4.2 a summary of Back's refinement of parallel programs. As in the case of the refinement of sequential programs, sometimes, we may restrict the refinement to preserve some temporal properties. In Section 4.3, we present more fault-tolerant refinement rules and show that these rules preserve the atomicity of the original program. Refinement of such atomicity is described in Section 4.4, being concerned with fault-tolerance. An extensive discussion in Section 4.5 shows that our development leads to a framework which can formally support the use and the interpretation of existing techniques, such as the multiple implementation scheme, recovery blocks and the conversation scheme, and atomic action techniques, that we mentioned in Chapter 1. Furthermore, these techniques can be used and interpreted at every stage during the development of a fault-tolerant program. Finally in Section 4.6, we consider how, by using our framework, to model recovery propagation in asynchronous communicating systems. Based on this, we show that dynamic checkpointing and recovery can be achieved by using the transformational approach.

4.2 Refinement of Parallel Programs

This section summarizes the refinement of parallel programs presented in [Bac88, Bac89].

4.2.1 Refining Atomicity

Given an action A of a program P, under certain conditions, A need not to be executed atomically and it can be replaced by a set of smaller actions that are to be executed atomically. Refinement transformations performing such replacements is referred to *refining atomicity* in [Bac88]. The conditions under which we can replace an action A by a number of new actions are determined by the commutativity between the new actions and the old actions other than A.

Commutativity between actions

Let A_1 and A_2 be two actions. The sequential composition (which is not a command or an action), $a = A_1; A_2$ determines, for any state S, three sets Ψ_0, Ψ_1, Ψ_2, where $\Psi_k = \Gamma(A_k)(\Psi_{k-1})$, $k = 1, 2$ and $\Psi_0 = \{S\}$. The semantics of a, denoted $\Gamma(a)$, is defined as follows:

$$\Gamma(a)(S) = \Gamma(A_2)(\Gamma(A_1)(S))$$

for each state S. This is equivalent to

$$\Gamma(a)(S) = \begin{cases} \emptyset & \text{if } \Psi_1 = \emptyset \\ \Psi_2 & \text{otherwise} \end{cases}$$

An action B is said to *commute left* with an action A, denoted $B; A \sqsubseteq A; B$, if and only if

(i) for any state S,

$$\Gamma(A; B)(S) \neq \emptyset \Rightarrow \Gamma(B; A)(S) \neq \emptyset$$

and

(ii) for any state S,

$$(\Gamma(B; A)(S) \neq \emptyset \wedge \{\textbf{abort}, \perp\} \cap \Gamma(B; A)(S) = \emptyset)$$

implies that

$$(\Gamma(A; B)(S) \subseteq \Gamma(B; A)(S))$$

The first condition means that whenever $A; B$ is a possible computation sequence for a state S, $B; A$ is a possible computation sequence for S. The second condition means that if $B; A$ is a possible computation that does not lead to abortion or nontermination for a state S, then any final state of $A; B$ is a final state of $B; A$.

An action B is said to *commute right* with an action A if action A is commute left with action B, i.e. $A; B \sqsubseteq B; A$. An action B commutes with an action A, denoted $A; B \equiv B; A$, if it commutes both left and right with A.

Conditions for commutativity

Let $A = g \rightarrow c$ and $A' = g' \rightarrow c'$ be actions. The way in which actions can enable and disable each other is captured by the following definitions.

- A *cannot enable* A', if $\{\neg g'\}A\{\neg g'\}$,
- A *must enable* A', if $\{\neg g'\}A\{g'\}$,
- A *cannot disable* A', if $\{g'\}A\{g'\}$,
- A *must disable* A', if $\{g'\}A\{\neg g'\}$,
- A *excludes* A', if $g \Rightarrow \neg g'$,
- A *cannot precede* A', if $\{true\}A\{\neg g'\}$.

We also say that A cannot be followed by A' if A cannot precede A'.

The following theorem, the proof of which was presented in [Bac88], gives sufficient conditions for two actions to commute.

Theorem 4.2.1 (Commutativity of Actions) *Let $A = g \rightarrow c$ and $A' = g' \rightarrow c'$ be actions. Then $A; A' \sqsubseteq A'; A$, if*

(i) A' cannot be disabled by A, and

(ii) A cannot be disabled by A', and

(iii) $\{g \wedge g'\}; c; c' \sqsubseteq c'; c$

Refining atomicity in action systems

Now we show how atomicity can be refined in action systems. Consider an action system

$$P = (Init_P, A[]B_1[] \ldots []B_n)$$

We want to replace the action $A = g \rightarrow c$ in P by a number of smaller actions, such that the resulting action system P' is a refinement of the original action system P.

We do this refinement in two steps. First we do a refinement

$$A = g \rightarrow c \sqsubseteq A' = g \rightarrow c'$$

where c' is of the form:

$$c' = c_0; \ \mathbf{do}\ A_1 [] \ldots [] A_m \ \mathbf{od}$$

By monotonicity of refinement, we have

$$P \sqsubseteq P[c/c']$$

The structure of c' determines the way to split up the action A. However, the new action $A' = g \rightarrow c'$ is still atomic in $P[c/c']$.

Next let P' be the program defined below:

$$P' = (Init_P, A_0 [] A_1 [] \ldots [] A_m [] B_1 [] \ldots B_n)$$

where the additional action $A_0 = g \rightarrow c_0$, which has the guard of action A and the initialization of the command c' as its body. This action is needed to initialize the computation of c' within the computation of P'.

Let the actions $B_1, \ldots, B_n$ above be termed *old actions*, and the actions $A_0, \ldots A_m$ above the *new actions*. Define gg by

$$gg = \bigvee_{i=1}^{m} gA_i$$

The following theorem, the proof of which was presented in [Bac88], presents the conditions under which the refinement

$$P[c/c'] \sqsubseteq P'$$

holds with respect to input-output total correctness.

Theorem 4.2.2 (Atomicity Refinement) *Let P, c, c' and P' be as above. Then $P[c/c'] \sqsubseteq P'$ is correct with respect to the input-output total correctness, if the following conditions are satisfied.*

1. *An old action (or the initialization $init_P$) cannot enable or disable any new action except A_0.*

2. *Action A_0 is excluded by the actions $A_1, \ldots, A_m$.*

3. *The old actions that are not excluded by the new actions are partitioned into two disjoint sets, the* left movers *$\{L_1, \ldots, L_k\}$ and the* right movers *$\{R_1, \ldots, R_j\}$, such that the following conditions hold.*

   3.1. *For each left mover L and each new action A_i, either A_i cannot be followed by L or L commutes left with A_i.*
   3.2. *For each right mover R and each new action (except A_0) or left mover C, either R cannot be followed by C or R commutes right with C.*
   3.3. *The command* **do** *$R_1[] \ldots []R_j$* **od** *is guaranteed to terminate for any initial state satisfying gg.*

From Theorem 3.3.3, the refinement $P \sqsubseteq P[c/c']$ preserves the temporal properties defined by the UNITY logic. However the conditions given in the above theorem cannot, in general, guarantee that the refinement $P[c/c'] \sqsubseteq P'$ or the refinement $P \sqsubseteq P'$ preserves the temporal properties defined by the UNITY logic. We have to strengthen the conditions to guarantee that the refinement $P \sqsubseteq P'$ preserves certain temporal properties. This is done in the theorem below, the proof of which is similar to that of Theorem 3.3.3.

Theorem 4.2.3 *Let the action system P, and actions $A_0, \ldots, A_m$ be as in Theorem 4.2.2 and let P' be the action system*

$$P' = (Init_P, A_0[]A_1[] \ldots []A_m[]B_1[] \ldots []B_n)$$

If all the conditions listed in Theorem 4.2.2 hold, we have for any predicates Q and R,

1. *if $\{Q \wedge \neg R\}cA_i\{Q \vee R\}$, for $i = 0, 1, \ldots, m$, then*

$$(P\ \underline{Sat}\ (Q\ \mathbf{unless}\ R)) \Rightarrow (P'\ \underline{Sat}\ (Q\ \mathbf{unless}\ R))$$

2. *if* $\{Q\}cA_i\{Q\}$*, for* $i = 0, 1, \ldots, m$*, then*

$$(P\ \underline{Sat}\ (\textbf{stable}\ Q)) \Rightarrow (P'\ \underline{Sat}\ (\textbf{stable}\ Q))$$

3. *if* $\{Q\}cA_i\{Q\}$*, for* $i = 0, 1, \ldots, m$*, then*

$$(P\ \underline{Sat}\ (\textbf{invariant}\ Q)) \Rightarrow (P'\ \underline{Sat}\ (\textbf{invariant}\ Q))$$

4. $(P\ \underline{Sat}\ (Q\ \textbf{ensures}\ R)) \Rightarrow (P'\ \underline{Sat}\ (\neg gg \wedge Q\ \textbf{ensures}\ gg \vee R))$

5. $(P\ \underline{Sat}\ (Q \longmapsto R)) \Rightarrow (P'\ \underline{Sat}\ (\neg gg \wedge Q \longmapsto R))$

Proof: The first three properties are easy to prove and the last one is derived from the fourth by using the induction rule of $\longmapsto$. Here we just give an informal proof of the fourth property.

Since $P\ \underline{Sat}(Q\ \textbf{ensures}\ R)$, we have

$$P\ \underline{Sat}(\neg gg \wedge Q\ \textbf{ensures}\ R)$$

and

$$P\ \underline{Sat}(\neg gg \wedge Q\ \textbf{ensures}\ gg \vee R)$$

Note that $P \sqsubseteq P[c/c']$, where c' is

$$c_0;\ \textbf{do}\ A_1[] \ldots []A_m\ \textbf{od}$$

Therefore, starting from a state in which $\neg gg \wedge Q$ holds, the execution of P' before gg is established is equivalent to the execution of $P[c/c']$, also to the execution of

$$(Init_P, A_0[]B_1[] \ldots []B_n)$$

Thus, $\neg gg \wedge Q$ will be continously hold until A_0 establishes gg (any B_i is required not to disable or enable new actions except A_0), or Q becomes true by executing A_0 and $B_1, \ldots, B_n$. The **ensures** property is then hold for P'. $\square$

Hence, when we refine the atomicity of an action system, we may have to take some temporal properties, such as invariant properties or progress properties, into account to achieve a refined version that preserves these properties.

We will adapt the example below (from [Bac88]) to illustrate atomicity refinement. However, some refined versions of the program will be slightly changed so that they can preserve some UNITY properties. The discussion will show that the requirement for atomicity refinement to preserve some UNITY properties is not very difficult to achieve.

Example 4.2.1 Compute the minimum of a function *func* in the interval $[0, 1]$. The basic idea is to generate a large number of randomly distributed point in this interval, compute the value of each point and choose the least function value found as the minimum.

Initial version We start from the following algorithm.

Program P_0
  declare
    q, y, h, b : queue of real, real, real, boolean
  initially
    $q, h, b = <>, \infty, true$
  actions
    GENERATE: $b \rightarrow b := false$;
      $q := q_0.(q_0$ consists of 10000 random values in $[0, 1])$
   []
    do CHECK: $q \neq <> \rightarrow y, q :=$ *head*(q), *rest*(q);
      $h :=$ *min*(*func*$(y), h)$ **od**
End $\{P_0\}$

Second version We refine the atomicity of the iterative command to achieve the following version.

Program P_1
  declare
    q, y, h, b : queue of real, real, real, boolean
  initially
    $q, h, b = <>, \infty, true$
  actions
    GENERATE: $b \rightarrow b := false$;
      $q := q_0.(q_0$ consists of 10000 random values in $[0, 1])$

[]
CHECK: $q \neq <> \rightarrow y, q := \textit{head}(q), \textit{rest}(q);$
$h := \textit{min}(\textit{func}(y), h)$
End $\{P_1\}$

This refinement preserves the invariant property

invariant $h = \textit{min}(f(y), h)$

Third version We now refine the GENERATE action to the action

$n := 0; x := seed; b := false;$
do $\neg b \wedge n < 10000 \rightarrow x := random(0, 1, x); q := q \hat{\ } x; n := n + 1$ **od**

We next replace the body of the GENERATE action with the above action.

Program P_2
declare
$q, y, h, b:$ queue of real, real, real, boolean;
n, x: integer, real
initially
$q, h, b = <>, \infty, true$
actions
GENERATE: $b \rightarrow$
$n := 0; x := seed; b := false$
do $\neg b \wedge n < 10000 \rightarrow$
$x := random(0, 1, x); q := q \hat{\ } x; n := n + 1$
od
[]
CHECK: $q \neq <> \rightarrow y, q := \textit{head}(q), \textit{rest}(q);$
$h := \textit{min}(\textit{func}(y), h)$
End $\{P_2\}$

Fourth version We then want to refine the atomicity of the GENERATE action.

Program P_3
declare
$q, y, h, b:$ queue of real, real, real, boolean;
n, x: integer, real
initially

$q, h, b = <>, \infty, true$

actions

GENERATE$_0$: $b \rightarrow n := 0; x := seed; b := false$

[]

GENERATE$_1$: $\neg b \wedge n < 10000 \rightarrow x := random(0, 1, x);$
$q := q\hat{\ }x; n := n + 1$

[]

CHECK: $q \neq <> \rightarrow y, q := \mathit{head}(q), \mathit{rest}(q);$
$h := \mathit{min}(\mathit{func}(y), h)$

End $\{P_3\}$

The refinement step also preserves the invariant $h = min(func(y), h)$.

Final version The GENERATE$_1$ action and the CHECK action cannot be executed in parallel, because they refer to the common variable q. We now split the CHECK action into two parts. One part only removes the next element from the buffer and the another does the actual computation of the function value and the minimum.

Program P_4

declare

q, y, h, b : queue of real, real, real, boolean;

n, x: integer, real;

c, z: boolean,real

initially

$q, h, b, c = <>, \infty, true, true$

actions

GENERATE$_0$: $b \rightarrow n := 0; x := seed; b := false$

[]

GENERATE$_1$: $\neg b \wedge n < 10000 \rightarrow x := random(0, 1, x);$
$q := q\hat{\ }x; n := n + 1$

[]

CHECK$_0$: $\neg b \wedge c \wedge q \neq <> \rightarrow z, q := \mathit{head}(q), \mathit{rest}(q);$
$c := false$

CHECK$_1$: $\neg c \rightarrow y := z; h := min(\mathit{func}(y), h); c := true$

End $\{P_4\}$

This refinement step also preserves the invariant property

invariant $h = min(\mathit{func}(y), h)$

And every refinement step preserves the progress property

$$y \in q \longmapsto h = min(\mathit{func}(y), h)$$

4.2.2 Simulations Between Programs

There is another kind of refinement which preserves the temporal properties of parallel programs. For this, we have to keep track of which variables are local to a part of the program and which are the global variables of the program. We introduce a *local variable declaration block*

$$\textbf{var}\ \overline{y} :\ c$$

as a command into our language, where $\overline{y}$ is a list of variables. The semantics $\Gamma(\textbf{var}\ \overline{y} :\ c)$ of this command is defined, for a global state S that does not contain any variable in $\overline{y}$, as

$$\Gamma(\textbf{var}\ \overline{y} :\ c)(S) \triangleq (\Gamma(c)(S \uparrow \overline{y})) \downarrow \overline{y}$$

where

$$S \uparrow \overline{y} \triangleq \{S' \mid S' \downarrow \overline{y} = S\}$$

and for a set of Ψ states that contain both the global variables and local variables $\overline{y}$,

$$\Psi \downarrow \overline{y} \triangleq \{S \downarrow \overline{y} \mid S \in \Psi\}$$

and for a state S that contains both local $\overline{y}$ and global variables $\overline{x}$,

$$S \downarrow \overline{y} \triangleq S|\overline{x}$$

is the restriction of S to the variables $\overline{x}$.

It was also proved in [Bac89] that the command constructor $\textbf{var}\ \overline{y} : c$ is monotonic with respect to the refinement relation between commands. We assume that, in this section, the initial condition $Init_P$ of a program P is implemented by a multiple assignment. Now a program P is of the form:

$$P = (\textbf{var}\ \overline{x} :\ \overline{x} := \overline{x}_0; A_1[] \ldots []A_n :\ \overline{z})$$

where $\overline{x}$ are the local variables that cannot be observed outside the system P, while $\overline{z}$ are the global variables. The program P is sometimes denoted as $P : \overline{z}$ when we are not interested in the local variables.

Union composition and hiding

Given two action systems P_1 and P_2,

$$\begin{aligned} P_1 &= (\mathbf{var}\,\overline{x} : \overline{x} := \overline{x}_0; A_1[] \ldots []A_m : \overline{z}) \\ P_2 &= (\mathbf{var}\,\overline{y} : \overline{y} := \overline{y}_0; B_1[] \ldots []B_n : \overline{u}) \end{aligned}$$

we define their *union composition* $P_1[]P_2$ to be

$$P_1[]P_2 \triangleq (\mathbf{var}\,\overline{x}, \overline{y} : \overline{x}, \overline{y} := \overline{x}_0, \overline{y}_0; A_1[] \ldots []A_m[]B_1 \ldots []B_n : \overline{z} \cup \overline{u})$$

This is the same as union operator defined before, except that here it separates the local and global variables while the definition given before just considers the global variables. This is exactly the same as the parallel composition in [Bac89]. We also assume that in this definition $\overline{x}$ and $\overline{y}$ have no variables in common. If this is not the case, then rename the local variables in the action systems so that the assumption becomes fulfilled, before the union operation is applied. We can also use substitution to rename the global variables of a program. Thus if P is a program with the global variables $\overline{z}$, then the program $A[\overline{z}/\overline{z}']$ is a program with the global variables obtained by replacing each occurrence of a variable in $\overline{z}$ by the corresponding variable in $\overline{z}'$.

Given a program,

$$P \;= (\mathbf{var}\,\overline{x} : \overline{x} := \overline{x}_0; A_1[] \ldots []A_n \;: \overline{z}_1 \,\hat{}\, \overline{z}_2)$$

hiding the global variables $\overline{z}_1$, to make them inaccessible from other actions outside the program, can be done by making them local:

$$P' = (\mathbf{var}\,\overline{z}_1 : \overline{z}_1 := \overline{z}_0; P : \overline{z}_2)$$

where

$$(\mathbf{var}\,\overline{z}_1 : \overline{z}_1 := \overline{z}_0; P : \overline{z}_2) \triangleq (\mathbf{var}\,\overline{z}_1, \overline{x} : \overline{z}_1, \overline{x} := \overline{z}_0, \overline{x}_0; A_1[] \ldots []A_n : \overline{z}_2)$$

A given program

$$P = (\textbf{var}\ \overline{v} :\ \overline{v} := \overline{v}_0; C_1 [] \ldots [] C_n :\ \overline{z})$$

can be *decomposed* into smaller programs by union and hiding. Let $A = \{A_1, \ldots, A_m\}$ and $B = \{B_1, \ldots, B_k\}$ be a partition of the actions in P and $var(A)$ and $var(B)$ be the sets of variables that are named in A and B respectively. Let

$$\begin{aligned}
\overline{x} &= var(A) - var(B) - \overline{z} \\
\overline{y} &= var(B) - var(A) - \overline{z} \\
\overline{w} &= var(A) \cap var(B) - \overline{z}
\end{aligned}$$

We can write P as

$$P = (\textbf{var}\ \overline{w} :\ \overline{w} := \overline{w}_0; P_1 [] P_2 :\ \overline{z})$$

where

$$\begin{aligned}
P_1 &= (\textbf{var}\ \overline{x} :\ \overline{x} := \overline{x}_0; A_1 [] \ldots [] A_m :\ \overline{z} \mathbin{\hat{}} \overline{w}) \\
P_2 &= (\textbf{var}\ \overline{y} :\ \overline{y} := \overline{y}_0; B_1 [] \ldots [] B_k :\ \overline{z} \mathbin{\hat{}} \overline{w})
\end{aligned}$$

Simulations between programs

Consider the program

$$P = (\textbf{var}\ \overline{v} :\ \overline{v} := \overline{v}_0; C_1 [] \ldots [] C_n :\ \overline{z})$$

Assume that we want to refine P to another program P' by replacing some actions $A_1, \ldots, A_m$ in P by some other actions $A'_1, \ldots A'_r$, possibly also changing some local variables.

We first decompose P into

$$P = (\textbf{var}\ \overline{w} :\ \overline{w} := \overline{w}_0; P_1 [] P_2 :\ \overline{z})$$

as described above. Here P_1 consists of the actions $A_1, \ldots, A_m$ to be replaced and hides the variables that are only accessed by these actions. P_1 is again a program.

We consider below the conditions under which the replacement of the old actions $A_1, \ldots, A_m$ by new actions $A'_1, \ldots, A_r$ in program P preserves any temporal properties as well as the input-output total correctness of P. In other words, we want to find the conditions under which

$$P = (\textbf{var}\ \overline{w} :\ \overline{w} := \overline{w}_0; P_1[]P_2 :\ \overline{z}) \sqsubseteq (\textbf{var}\ \overline{w} :\ \overline{w} := \overline{w}_0; P_1'[]P_2 :\ \overline{z}) = P'$$

for a program P_1' and the temporal properties of P are well preserved.

It was proved in [Bac89] that the data refinement presented in [BW89] is sufficient for this purpose.

Let c be a command with variables $\overline{u}\ \hat{}\ \overline{v}$ and c' be a command with variables $\overline{u}'\ \hat{}\ \overline{v}$. The fact that c' is constructed from c by changing the way in which the state is represented is described as c is *data refined* by c'. Such a refinement can be done by using an *encoding command* E which computes a mapping from the states over $\overline{u}\ \hat{}\ \overline{v}$ to the states over $\overline{u}'\ \hat{}\ \overline{v}$ such that there exists an inverse command E^{-1} of E satisfying

$$E; E^{-1} \equiv \textbf{skip}$$

We say that the command c is *refined by* the command c' *through the encoding* E, denoted $c \sqsubseteq_E c'$, if

$$c \sqsubseteq E; c'; E^{-1}$$

Now consider the programs P, P_1, P_2, P_1' and P' described above. Let c_0 and c_0' be the initial assignments of P_1 and P_1' respectively. For the actions $A_1, \ldots, A_m$ of program P_1 and $A_1', \ldots, A_r'$ of program P_1', let

$$A = \bigvee_{i=1}^{m} gA_i \rightarrow \textbf{if}\ A_1[] \ldots []A_m\ \textbf{fi}$$
$$A' = \bigvee_{i=1}^{r} gA_i' \rightarrow \textbf{if}\ A_1'[] \ldots []A_r'\ \textbf{fi}$$

For a guard g over $\overline{u}'\ \hat{}\ \overline{v}$, let $E(g)$ be the guard over $\overline{u}\ \hat{}\ \overline{v}$ which is the *weakest precondition* of E for the post condition g [Dij76, BW89].

We will say that program $P_1 : \overline{z}$ is *strongly simulated* by program $P_1' : \overline{z}$, denoted $P_1 \preceq_s P_1'$, if there exists an encoding command E from the local variables $\overline{x}$ of P_1 to the local variables $\overline{x}'$ of P_1' such that the following three conditions are satisfied:

(i) $\overline{x} := \overline{x}_0.true; c_0 \sqsubseteq \overline{x}' := \overline{x}_0'.true; c_0'; E^{-1}$

(ii) $A \sqsubseteq_E A'$

(iii) $gA \Rightarrow E(gA')$

This simulation is *strong* in the sense that it requires a one to one correspondence between the actions executed by P_1 and by P_1'. Even though an action execution in P_1' needs not always correspond to an execution of the same action in P_1, it must always correspond to some action execution of P_1. In practice, however, executing a simple action in P_1 will often correspond to executing a sequence of two or more actions in P_1', so that the one to one correspondence is not maintained.

Our semantic model permits *finite stuttering*, i.e. we take the program P_1 to be equivalent to the program

$$P_1^+ = (\mathbf{var}\ \overline{x}, h :\ \overline{x} := \overline{x}_0; h := h'.true; A_1 [] \ldots [] A_m [] Stutter :\ \overline{z})$$

where the action $Stutter$ is

$$Stutter = h := h'.true; \mathbf{do}\ h > 0 \rightarrow h := h'.(h' < h)\ \mathbf{od})$$

and h is assumed to range over a well-founded set with the least element 0. For any program P_2, we also have

$$P_1 [] P_2 \equiv P_1^+ [] P_2$$

We say that a program P_1 is *simulated by* program P_1', denoted $P_1 \preceq P_1'$, if $P_1^+ \preceq_s P_1'$.

We summarize the results about the simulation refinement in the following theorems the proofs of which were given in [Bac89].

Theorem 4.2.4 *Let P_1 and P_1' be two programs. Then $P_1 \preceq P_1'$ implies that the refinement*

$$(\mathbf{var}\ \overline{w} :\ \overline{w} := \overline{w}_0; P_1 [] P_2) \sqsubseteq (\mathbf{var}\ \overline{w} :\ \overline{w} := \overline{w}_0; P_1' [] P_2)$$

preserves the input-output total correctness for any choice of $\overline{w}$, $\overline{w}_0$ and P_2.

Theorem 4.2.5 *Let P_1, P_1', P_1'', P_2, P_2' be programs, $\overline{w}$, $\overline{z}$ and $\overline{z}'$ be lists of distinct variables and $\overline{w}_0$ be a list of initial values. Then the following properties hold:*

(i) $P_1 \preceq P_1$

(ii) $P_1 \preceq P_1' \preceq P_1'' \Rightarrow P_1 \preceq P_1''$

(iii) *If* $P_1 \preceq P_1'$ *and* $P_2 \preceq P_2'$ *then*

(a) $P_1 [] P_2 \preceq P_1' [] P_2'$
(b) $P_1[\overline{z}/\overline{z}] \preceq P_1'[\overline{z}/\overline{z}]$
(c) $(\textbf{var}\ \overline{w} :\ \overline{w} := \overline{w}_0; P_1) \preceq (\textbf{var}\ \overline{w} :\ \overline{w} := \overline{w}_0; P_1')$

Theorem 4.2.6 *Let* P_1 *and* P_2 *be programs. If* $P_1 \preceq P_2$*, then for any temporal formula* Φ *defined in* UNITY *logic*

$$P_1\ \underline{Sat}\ \Phi \Rightarrow P_2\ \underline{Sat}\ \Phi$$

In [Bac89], a stronger result was proven that the simulation refinement preserves any temporal formula which is insensitive to stuttering.

4.3 Fault-tolerant Atomicity

Let P be a parallel program

$$P = (Init_P, Act_P)$$

A fault-tolerant version P' of it can be obtained by using the transformation $\mathcal{R}$ and the refinement rules described in Chapter 3. However, when partitioning the program for parallelizing the execution of P', any action involved with recovery commands, which refer to all the variables of the program, cannot be executed in parallel with any other actions. The worst case is when each action of P' contains recovery commands and P' can only be executed in a sequential manner.

Consider the fault-tolerant blocks $[\mid c :: \ j : 1..k :: \ c_j \ \mid]_P$ and $[\mid c :: \ \textbf{redo}\ c' \ \mid]_P$ defined in Chapter 3. It is because the commands $Save_P$ and $Restore_P$ refer to too many variables that actions containing fault-tolerant blocks cannot be executed in parallel with other actions. On the other hand, it is sufficient if we only save and restore the state of the variables that are used for the execution of the command c.

In this section when we present a fault-tolerant refinement rule of the form $P \sqsubseteq_{T(F)} P'$, we always assume that the execution of any command c in the program P cannot stay in an error state forever without terminating. In other words, the occurrence of a fault will eventually be detected. There are two cases for this assumption, i.e. either the execution of c is being carried out with the provision of fault-tolerance or the execution of c will eventually terminate with an error state if faults occur during this execution.

Given a program $P = (Init_P, A_1 [] \ldots [] A_n)$, and a command c contained in an action of P, we define the fault-tolerant structures below.

1. For a set X of variables of P, let $Rec_X = \{rec_x \mid x \in X\}$ be a set of distinct variables, called *recording* (or *checkpointing*) variables of X, which are not named in P. The execution of the *saving* command $Save[X, Rec_X]$ of X takes a checkpoint of the current state of X, i.e.

$$Save[X, Rec_X] \triangleq \textbf{if } \neg f \rightarrow \|_x : x \in X :: rec_x := x [] f \rightarrow \textbf{skip fi}$$

2. Let $var(c)$ be the set of variables named in c and $Rec_{var(c)}$ be a set of recording variables of $var(c)$. The *saving* command $Save_c$ of c is defined as

$$Save_c \triangleq Save[var(c), Rec_{var(c)}]$$

3. Let X be a set of variables and Rec_X be the recording variables of X used in the saving command of X. The *restoring* command $Restore[X, Rec_X]$ is defined as

$$Restore[X, Rec_X] \triangleq \|_x : x \in X :: x := rec_x$$

4. For a command c, the restoring command $Restore_c$ is defined as

$$Restore_c \triangleq Restore[var(c), Rec_{var(c)}]$$

5. For a command c_j, let

$$Recr(c, c_j) \triangleq \textbf{if } \neg f \rightarrow \textbf{skip} [] f \rightarrow Restore_c; c_j \textbf{ fi}$$

6. For $j = 1, \ldots, k$, let

$$Recr[c :: j : 1..k :: c_j] \triangleq Recr(c, c_1); \ldots ; Recr(c, c_k)$$

7. Let

$$[\mid c :: j : 1..k :: c_j \mid] \triangleq Save_c; \textbf{if } f \rightarrow c [] \neg f \rightarrow c; Recr[c :: j : 1..k :: c_j] \textbf{ fi}$$

We assume that, wherever a saving command $Save_c$ or a restoring command $Restore_c$ occurs, the recording variables of c will never be corrupted and the saving command or restoring command cannot be interrupted by any fault in the fault environment.

If $c \sqsubseteq c_j$ for $j = 1, \ldots, k$, then the command $[\!\mid c :: j : 1..k :: c_j \mid\!]$ is called the *fault-tolerant version* of c with the *alternatives* $c_1, \ldots, c_k$; and the command $Recr[c :: j : 1..k :: c_j]$ is called a *recovery command* of c.

When the execution of the command $[\!\mid c, j : 1..k :: c_j \mid\!]$ starts with a good state, this state is then saved by the saving command $Save_c$. Then the command of c starts to be executed from this good state. If it terminates with a good state (i.e. no fault occurs during the execution), no alternative will be required to be executed. If it cannot terminate, no alternative can be executed (in this case no fault occurs by the assumption made at the beginning of this section). If faults occur during the execution of c, the first alternative is entered. A further alternative, if one exists, is entered if the preceding alternative fails (i.e. faults occur in its execution). But before an alternative is so entered, the state of each variable x used for this execution is restored by executing the restoring command $Restore_c$ using the values recorded in the recording variables $Rec_{var(c)}$. If an alternative terminates with a good state, any further alternatives are ignored, and the command following the fault-tolerant command is the next to be executed.

However, the fault-tolerant command $[\!\mid c :: j : 1..k :: c_j \mid\!]$ can only correct the faults occurring during the execution of c and its alternative. The faults which occur before the execution of this fault-tolerant command and the faults which occur during the execution of the last alternative will be left to be corrected by the recovery commands outside this fault-tolerant command.

It is noted that

$$[\!\mid c :: j : 1..k :: c_j \mid\!]_P \sqsubseteq [\!\mid c :: j : 1..k :: c_j \mid\!]$$

and

$$\mathcal{F}([\!\mid c :: j : 1..k :: c_j \mid\!]_P) \sqsubseteq \mathcal{F}([\!\mid c :: j : 1..k :: c_j \mid\!])$$

when only the variables $var(c)$ are concerned.

From this fact and the properties of fault-tolerant refinement discussed in Chapter 3, we have the following refinement rule.

FT-Rule 11 *Let $P \sqsubseteq_{T(F)} P$ and c be a command of P. If $[\!\mid c :: j : 1..k :: c_j \mid\!]$ is a fault-tolerant version of c, then*

$$P \sqsubseteq_{T(F)} P[c/[\!\mid c :: j : 1..k :: c_j \mid\!]]$$

Similarly, refine the fault-tolerant block $[\!\mid c :: \textbf{ redo } c' \mid\!]_P$ to the *retry version* of c $[\!\mid c :: \textbf{ redo } c' \mid\!]$ which is defined as:

$$Save_c; \textbf{if } f \rightarrow c [] \neg f \rightarrow c; \textbf{if } f \rightarrow \textbf{do } f \rightarrow Restore_c; c' \textbf{ od } [] \neg f \rightarrow \textbf{skip fi fi}$$

It is noted that if $c = c'$, we have the special case of multiple retry versions of c, that is $[\!\mid c :: \textbf{ redo } c \mid\!]$.

FT-Rule 12 *If $c \sqsubseteq c'$,*

$$c \sqsubseteq_{T(F)} [\!\mid c :: \textbf{ redo } c' \mid\!]_P \sqsubseteq_{T(F)} [\!\mid c :: \textbf{ redo } c' \mid\!]$$

From these two rules, we derive the rules below for introducing recovery into the body of an iterative command.

FT-Rule 13 *Let $P \sqsubseteq_{T(F)} P$ and $DO^n_{i=1} A_i$ be a command of P. If for $i = 1, \ldots, n$, $[\!\mid cA_i :: j : 1..m_i :: c_{ij} \mid\!]$ is a fault-tolerant version of cA_i with alternatives, then*

$$P \sqsubseteq_{T(F)} P[DO^n_{i=1} A_i / DO^n_{i=1} gA_i \rightarrow [\!\mid cA_i :: j : 1..m_i :: c_{ij} \mid\!]]$$

This rule is proved from the fact that $P[A_i/A'_i]$ can be obtained by repeatedly applying **FT-Rule 11**. to the program P.

Similarly, repeating the application of **FT-Rule 12** derives the following rule.

FT-Rule 14 *If for $i = 1, \ldots, n$, $[\!\mid cA_i :: \textbf{ redo } c_i \mid\!]$ is a retry version of cA_i, then*

$$DO^n_{i=1} A_i \sqsubseteq_{T(F)} DO^n_{i=1} gA_i \rightarrow [\!\mid cA_i :: \textbf{ redo } c_i \mid\!]$$

Given a program $P = (Init_P, A_1 [] \ldots [] A_n)$, using the rules described above we can transform each action A_i into a *fault-tolerant action* $FT[A_i :: j : 1..k_i :: c_{ij}]$:

$$FT[A_i :: \ j : 1..k_i :: \ c_{ij}] \triangleq gA_i \to [\,|\, cA_i :: \ j : 1..k_i :: \ c_{ij} \ |\,]$$

A_i can be also transformed into another fault-tolerant version $FT[A_i :: \ \textbf{redo}\ c_i]$:

$$FT[A_i :: \ \textbf{redo}\ c_i] \triangleq gA_i \to [\,|\, cA_i :: \ \textbf{redo}\ c_{ij} \ |\,]$$

From the **FT-Rule 7**, we have

$$\mathcal{R}(P) \sqsubseteq_{T(F)} (Init, Save_P[]A_1[] \ldots []A_n[]Recr_P)$$

Denote $(Init, Save_P[]A_1[] \ldots []A_n[]Recr_P)$ by $P[Save_P, Recr_P]$. Now we have the following refinement rule.

FT-Rule 15 *For a program $P = (Init_P, A_1[] \ldots []A_n)$, we can do the following fault-tolerant refinement:*

$P[Save_P, Recr_P] \sqsubseteq_{T(F)}$
$(Init, Save_P[]FT[A_1 :: j : 1..k_1 :: \ c_{1j}][] \ldots []FT[A_n :: \ j : 1..k_n :: \ c_{nj}][]Recr_P)$

Apply the fault transformation $\mathcal{F}$ to the program

$$(Init, Save_P[]FT[A_1 :: j : 1..k_1 :: \ c_{1j}][] \ldots []FT[A_n :: \ j : 1..k_n :: \ c_{nj}][]Recr_P)$$

If we can prove that

(a) for $i = 1, \ldots, n$, $FT[A_i :: \ j : 1..k_j :: \ c_{ij}]$ terminates with good states if A_i is guaranteed to terminate in a fault-free environment, i.e. one of the alternatives c_{ij} must succeed after a fault occurs in the execution of cA_i:

$$cA_i \sqsubseteq_{T(F)} cFT[A_i :: \ j : 1..k_j :: \ c_{ij}]$$

(b) faults can only occur during (not after or before) the execution of the body, $[\,|\, cA_i : j1..k_i :: c_{ij} \ |\,]$, of an action $FT[A_i :: \ j : 1..k_j :: \ c_{ij}]$,

then

$P[Save_P, Recr_P] \sqsubseteq_{T(F)}$
$(Init, FT[A_1 :: j : 1..k_1 :: \ c_{1j}][] \ldots []FT[A_n :: \ j : 1..k_n :: \ c_{nj}])$

The retry version of **FT-Rule 15** is that when a fault occurs in the execution of the body cA_i of an action A_i, the initial state of this action is restored and then the execution of refined version of cA_i is repeated. This repetition will carry on until it achieves the correct result of this action.

FT-Rule 16 *Given the programs P, $\mathcal{R}_I(P)$ as in* **FT-Rule 4**, *we have*

1. $P[Save_P, Recr_P] \sqsubseteq_{T(F)}$
   $(Init, Save_P[]FT[A_1 ::$ **redo** $c_1][] \ldots []FT[A_n ::$ **redo** $c_n][]Redo_P)$

2. *If faults cannot occur during the time between the termination of an action and the start of execution of another action, then*

   $P[Save_P, Recr_P] \sqsubseteq_{T(F)} (Init, FT[A_1 ::$ **redo** $c_1][] \ldots []FT[A_n ::$ **redo** $c_n])$

We combine these two rules to get a general one.

FT-Rule 17 *Let the programs P be as in* **FT-Rule 15**. *For $i = 1, \ldots, n$, let $FT[A_i]$ be the fault-tolerant action obtained from A_i by using* **FT-Rule 15** *or* **FT-Rule 16**. *If $Recr$ is global recovery command constructed as $Recr_P$ or $Redo_P$, then we have*

1. $P[Save_P, Recr_P] \sqsubseteq_{T(F)} (Init, Save_P[]FT[A_1][] \ldots []FT[A_n][]Recr)$

2. *If faults cannot occur during the time between the termination of an action and the start of execution of another action, then*

$$P[Save_P, Recr_P] \sqsubseteq_{T(F)} (Init, FT[A_1][] \ldots []FT[A_n])$$

After a fault-tolerant version of a command is obtained by using the rules described above, it can be further refined following the the rules described below.

FT-Rule 18 *Let $P = (Init_P, A_1[] \ldots []A_n)$ be a program and $P \sqsubseteq_{T(F)} P$. Assume that an action A_i contains a fault-tolerant version $[\mid c :: j : 1..k :: c_j \mid]$. Then if $c \sqsubseteq c'$, and for $j = 1, \ldots, k$, $c_j \sqsubseteq c'_j$,*

$$P \sqsubseteq_{T(F)} (Init_P, A_1[] \ldots []A'_i[] \ldots []A_n)$$

where $A'_i = A_i[[\mid c :: j : 1..k :: c_j \mid]/[\mid c' :: j : 1..k :: c'_j \mid]]$.

FT-Rule 19 *Let $P = (Init_P, A_1[] \ldots []A_n)$ be a program and $P \sqsubseteq_{T(F)} P$. Assume that an action A_i contains a retry version $[\mid c ::$ **redo** $c_1 \mid]$. Then if $c \sqsubseteq c'$ and $c_1 \sqsubseteq c'_1$,*

$$P \sqsubseteq_{T(F)} (Init_P, A_1[] \ldots []A'_i[] \ldots []A_n)$$

*where $A'_i = A_i[[\mid c ::$ **redo** $c_1 \mid]/[\mid c' ::$ **redo** $c'_1 \mid]]$.*

Consider the parallel execution of a program P. If P' is a fault-tolerant refined version of P obtained by using the fault-tolerant rules described above, then we claim by the theorem below that P' can have the same degree of parallelism.

Theorem 4.3.1 *Let program P' be a program obtained from program P by using the fault-tolerant rules described above. Given a partition Proc of the variables Var(P), there is a partition Proc$'$ of the variables Var(P') such that*

(i) for each $p \in$ Proc, there is a $p' \in$ Proc$'$ such that

$$\forall\, x \in \mathit{Var}(P) : (x \in p' \Leftrightarrow x \in p)$$

(ii) for $x, y \in$ Var(P), rec$_x$ and rec$_y$ are in the same process of the new partition Proc$'$ only if x and y are in the same process of the old partition Proc, i.e.

$$(\exists\, q \in \mathit{Proc}' : \mathit{rec}_x \in q \wedge \mathit{rec}_y \in q) \Rightarrow (\exists\, p \in \mathit{Proc} : x \in p \wedge y \in p)$$

(iii) if **FT-Rule 15** *or* **FT-Rule 16** *was used in obtaining P' from P, rec$'_x$ and rec$'_y$ are in the same process of the new partition Proc$'$ only if x and y are in the same process of the old partition Proc, i.e.*

$$(\exists\, q \in \mathit{Proc}' : \mathit{rec}'_x \in q \wedge \mathit{rec}'_y \in q) \Rightarrow (\exists\, p \in \mathit{Proc} : x \in p \wedge y \in p)$$

Then any two actions A'_i and A'_j of P' can be executed in parallel under the partition Proc$'$ if and only if the corresponding actions A_i and A_j of P can be executed in parallel under the partition Proc.

4.4 Refining Fault-tolerant Atomicity

Given a program P, a fault-tolerant version of P can be constructed by using the fault-tolerant transformation and fault-tolerant refinement rules described so far. Fault-tolerant programs so constructed employ global recovery actions and/or fault-tolerant atomic actions described in last section. It is noted that a fault-tolerant action $FT[A]$ can be quite large in the sense that it may be a module, a procedure or a subroutine, etc.. It is sometimes necessary to refine such a large fault-tolerant action into a group of smaller actions which are to be executed atomically but so that no error state entered by executing theses smaller actions can be propagated to actions outside this group. This means that such a refinement transformation preserves the *error atomicity* of $FT[A]$ although it may refine its *concurrent atomicity*.

Consider a fault-tolerant program

$$P = (Init_P, A[]B_1[]\ldots[]B_n)$$

where $A = g \to c$. Find a command

$$c' = c_0; \textbf{do } A_1[]\ldots[]A_m \textbf{ od}$$

that is a fault-tolerant refinement of c in the context g:

$$\{g\}; c \sqsubseteq_{T(F)} c'$$

Since P is a fault-tolerant program, we have

$$P \sqsubseteq_{T(F)} P[c/c']$$

where

$$P[c/c'] = (Init_P, g \to c'[]B_1[]\ldots[]B_n)$$

Now define a program P' by

$$P' = (Init_P, A_0[]A_1[]\ldots[]A_m[]B_1[]\ldots[]B_n)$$

where the additional action $A_0 = g \rightarrow c_0$ has the guard of action A and the initialization of the command c'as its body. This action is needed to initialize the computation of c' within the computation of P'.

Then if we can prove that

$$\mathcal{F}(P) \sqsubseteq \mathcal{F}(P')$$

we have

$$P \sqsubseteq_{T(F)} P'$$

Corollary 4.4.1 (Fault-tolerant Atomicity Refinement)

$P \sqsubseteq_{T(F)} P'$ is correct with respect to input-output total correctness, if the following conditions are satisfied.

1. *For each old action B_i, $\mathcal{F}(B_i)$ cannot enable or disable $\mathcal{F}(A_j)$ for any new action A_j except A_0.*
2. *Action $\mathcal{F}(A_0)$ is excluded by the actions $\mathcal{F}(A_1), \ldots, \mathcal{F}(A_m)$.*
3. *The fault-affected versions of the old actions that are not excluded by the fault-affected versions of the new actions are partitioned into two disjoint sets, the* left movers $\{L_1, \ldots, L_k\}$ *and the* right movers $\{R_1, \ldots, R_j\}$*, such that the following holds:*
   - *3.1. For each left mover L and each new action A_i, either $\mathcal{F}(A_i)$ cannot be followed by L or L commutes left with $\mathcal{F}(A_i)$.*
   - *3.2. For each right mover R and the fault-affected version of each new action (except A_0) or left mover C, either R cannot be followed by C or R commutes right with C.*
   - *3.3. The command* **do** $R_1[] \ldots []R_j$ **od** *is guaranteed to terminate for any initial state satisfying gg, where $gg = \bigvee_{i=1}^{m} gA_i$.*

Proof: From Theorem 4.2.2, the conditions described above imply

$$\mathcal{F}(P) \sqsubseteq \mathcal{F}(P')$$

□

This shows that, by using the fault transformation on fault-tolerant programs, refinement of parallel program can be extended to deal with refinement of fault-tolerant parallel programs. The conditions given in the corollary require that errors and error recoveries must not propagate to actions outside the group of the new actions; also errors and error recoveries outside this group must not propagate into this group.

Similarly, fault-tolerant simulations between fault-tolerant program can be defined by applying the fault transformation to the programs.

For a fault-tolerant program

$$P = (\mathbf{var}\ \overline{x} :\ \overline{x} := \overline{x}_0; A_1[] \dots [] A_n)$$

we say that P is *strongly fault-tolerant simulated* by program P', denoted $P \preceq_{T(F)} P'$, if

$$\mathcal{F}(P) \preceq_s \mathcal{F}(P')$$

where

$$P' = (\mathbf{var}\ \overline{x}' :\ \overline{x}' := \overline{x}'_0; A'_1[] \dots [] A'_m)$$

This requires that there exists an encoding command E from the local variables $\overline{x}$ of P to the local variables $\overline{x}'$ of P' such that the following conditions are satisfied:

(i) $\overline{x} := \overline{x}.true; \mathcal{F}(init_P) \sqsubseteq \overline{x}' := \overline{x}'.true; \mathcal{F}(init_{P'}); E^{-1}$

(ii) $\mathcal{F}(A) \sqsubseteq E; \mathcal{F}(A'); E^{-1}$

(iii) $gA \Rightarrow E(gA')$

where $init_P$ and $init_{P'}$ are the initial assignments of P and P' respectively, and

$$A \triangleq \bigvee_{i=1}^{n} gA_i \rightarrow \textbf{if}\ A_1[] \dots [] A_n\ \textbf{fi}$$

$$A' \triangleq \bigvee_{i=1}^{m} gA'_i \rightarrow \textbf{if}\ A'_1[] \dots [] A'_m\ \textbf{fi}$$

Note that the commands E, E^{-1}, $\overline{x} := \overline{x}.true$ and $\overline{x}' := \overline{x}'.true$ are not concerned with faults because they are used for verification but not for execution.

P is said to be *fault-tolerant simulated* by P', denoted $P \prec_{T(F)} P'$, if

$$\mathcal{F}(P)^+ \preceq_s \mathcal{F}(P')$$

Again we can also write this as

$$P^+ \preceq_{T(F)} P'$$

by assigning empty fault environment to the commands, $h := h'.true$ and

$$h := h'.true;\ \textbf{do}\ h > 0 \rightarrow h := h'.(h' < h)\ \textbf{od}$$

because they are introduced only for verifying the simulation relation.

Based on the above discussion, we describe some methods below for doing fault-tolerant refinement of parallel programs.

Consider a program $P = (\mathit{Init}_P, A[]B_1[] \ldots []B_n)$. Assume the action A is of the form:

$$A = g \rightarrow c_1; \ldots; c_m$$

Then

$$P \sqsubseteq (\mathit{Init}_P, g \rightarrow c_1; DO_{i=2}^m(b_i \rightarrow c_i)[]B_1[] \ldots []B_n)$$

provided that

(a) $g \rightarrow c_1$ enables $b_2 \rightarrow c_2$,

(b) $b_i \rightarrow c_i$ enables $b_{i+1} \rightarrow c_{i+1}$, for $i = 2, \ldots m-1$,

(c) $b_m \rightarrow c_m$ establish $\neg \bigvee_{i=2}^{m} b_i$,

(d) $b_i \rightarrow c_i$ excludes all other actions $b_j \rightarrow c_j$, $j \neq i$ and $b_1 = g$.

For $i = 1, \ldots, m$, let $A_i = b_i \rightarrow c_i$. If by using Theorem 4.2.2, we can prove that

$$P \sqsubseteq (\mathit{Init}_P, A_1[] \ldots []A_m[]B_1[] \ldots []B_n)$$

then we have the following rule.

FT-Rule 20 *If for $i = 1, \ldots, m$, $c_i \sqsubseteq_{T(F)} c'_i$, let*

$$A'_i = gA_i \rightarrow c'_i$$

Then

$$P \sqsubseteq (\mathit{Init}_P, \mathcal{F}(A'_1)[] \ldots []\mathcal{F}(A'_m)[]B_1[] \ldots []B_n)$$

Specifically, if cA is a fault-tolerant command with c_1 and c_m being respectively a saving command and a recovery command for the command $c_2; \ldots; c_{m-1}$, then

$$P \sqsubseteq (\mathit{Init}_P, \mathcal{F}(A_1)[] \ldots []\mathcal{F}(A_m)[]B_1[] \ldots []B_n)$$

provided additionally that, for any action B_j, for $j = 1, \ldots, n$, if B_j refers to a variable that a command c_i refers to, $i = 2, \ldots, m-1$, B_j excludes all the actions $b_i \rightarrow c_i$.

This rule gives the conditions under which instead of transforming a big sequential composition command into a fault-tolerant action, we can decompose a sequential composition command into a group of actions and make each of them fault-tolerant.

Consider a program $P = (\mathit{Init}_P, A[]B_1[] \ldots []B_n)$, where the action A is of the form:

$$A = g \rightarrow \textbf{if } A_1[] \ldots []A_m \textbf{ fi}$$

Let $cA \sqsubseteq_{T(F)} [\mid cA :: \ j : 1..k :: \ c_j \ \mid]$, and

$$Save_A = g \rightarrow Save_{cA}$$

$$Recr_A = b \rightarrow Recr[cA :: \ j : 1..k :: \ c_j]$$

Then we have the following rule.

FT-Rule 21 *Given a program P described above,*

$$P \sqsubseteq (\mathit{Init}_P, \mathit{Save}_A[]\mathcal{F}(A_1)[] \ldots []\mathcal{F}(A_m)[]\mathcal{F}(\mathit{Recr}_A)[]B_1[] \ldots []B_n)$$

provided that

(a) each B_i which refers to variables in var(A) excludes all the actions A_j,

(b) the action Save$_A$ establishes $\bigvee_{j=1}^{m} gA_j$,

(c) each $\mathcal{F}(A_j)$ establishes $\neg \bigvee_{j=1}^{m} gA_j$,

(d) each $\mathcal{F}(A_j)$ establishes b,

(e) Recr$_A$ excludes each B_i which refers to variables in var(A).

Under the conditions of this rule, the actions A_1, ..., A_m may be executed in parallel. However, during such an execution, interactions and error propagation and error recovery propagation are restricted within the group $\{A_1, \ldots, A_m\}$ of actions.

Assume that each A_j can be further decomposed into a sequential composition:

$$A_j = cA_j \rightarrow c_{j1}; \ldots ; c_{jk_j}$$

and the conditions (a)-(d) in **FT-Rule 20** hold for the actions $A_{jl} = b_{jl} \rightarrow c_{jl}$. In addition to these conditions, assume that the following conditions hold:

(i) each B_i which refers to variables in $var(A)$ excludes all the actions A_{jl},

(ii) for each pair of the actions A_{jl} and $A_{j'l'}$, either they exclude each other or commute with each other,

(iii) the action $Save_A$ establishes $\bigvee_{j=1}^{m} gA_j$,

(iv) the action $Save_A$, excludes all the actions A_{jl},

(v) each A_{jk_j} establishes b,

(vi) $Recr_A$ excludes each B_i which refers to variables in $var(A)$.

Let

$$Act = \{Save_A, \mathcal{F}(Recr_A)\} \cup \{\mathcal{F}(A_{jl}) \mid 1 \leq j \leq m, 1 \leq l \leq k_j\} \cup \{B_1, \ldots, B_n\}$$

Then the following refinement holds.

FT-Rule 22 *Let Act be the actions described above, and programs P and*

$$P' = (Init_P, Save_A[]\mathcal{F}(A_1)[]\dots[]\mathcal{F}(A_m)[]\mathcal{F}(Recr_A)[]B_1[]\dots[]B_n) \sqsubseteq (Init_P, Act)$$

be those described in **FT-Rule 21**. *Then,*

$$P \sqsubseteq P' \sqsubseteq (Init_P, Act)$$

4.5 Unifying and Generalizing Existing Methods

This section shows that the framework which we have developed provides a formal basis for unifying and generalizing the existing informal methods, such as multiple implementation scheme, recovery block and conversation scheme and atomic action scheme. Such an unification makes it possible for these methods to be used rigorously in the development of one fault-tolerant program. The generalization shows, within one program, how backward and forward recovery techniques can be used, and how a *fail-soft* program[1] can be developed by using our approach.

4.5.1 Recovery Blocks

The concept of *recovery block* was proposed in [HLMR74] for providing backward error recovery for a sequential program. When the execution of the program enters a recovery block, it checkpoints its state. When an error occurs during the execution of the block, the program rolls back to the checkpoint and retries its computation from the checkpoint. The retry command can be designed to be either exactly the same as the previous try or a different one. The execution of program exits the block if one of the retry copies succeeds or all the retry copies have been tried.

Let a program $P = (Init_P, A_1[]\dots[]A_n)$ be executed as one single sequential process. Applications of the **FT-Rules 8, 11, 13, 15** and **17** will transform P into fault-tolerant versions. In the fault environment, the execution of such a version provides fault-tolerance by executing the fault-tolerant structure $[\,|\, c :: \; j..k :: \; c_j \;|\,]$. From the definition and explanation given for $[\,|c :: \; j..k :: \; c_j \;|\,]$ in Section 4.3, we can see that structure $[\,|\,c :: \; j..k :: \; c_j \;|\,]$ is a formalization of the recovery block proposed in [HLMR74, Ran75]. Furthermore, such recovery blocks are generalized to include the retry blocks like $[\,|\; c :: \;$ **redo** $c' \;|\,]$ which is defined in **FT-Rules 9, 12, 14** and

[1] The term *fail-soft* appeared in [Avi76]. A program is *fail-soft* if, in the fault environment, it implements a specification which is weaker than the original specification of the program.

16. Recovery blocks can be nested: the use of **FT-Rules 14-17** provides recovery blocks while the successive use of **FT-Rules 11-13** provides nested recovery blocks. The **FT-Rules 18** and **19** allow a recovery block in a program to be further refined.

4.5.2 Conversations

Recovery block was extended to the concept of *conversation* in [Ran75] to deal with backward error recovery for interacting processes. When two or more processes enter a conversation, each of them must checkpoint its state. If a error occurs in one of the processes during the conversation, all the processes of the conversation roll back to their checkpoints and retry their computations from the checkpoints. All the processes leave the conversation together if one of the retry computations succeeds or all the retry copies have been tried. During the conversation, processes inside the conversation cannot interact with processes outside the conversation.

Conversation provides a way of coordinating error recoveries among concurrent processes to deal with a sort of uncontrolled *domino effect* [Ran75] described below.

Consider the case that a program P is partitioned into two or more processes which are executed in parallel. Let each process consist of a set of recovery blocks constructed from sequential sets of actions in this process. If a process p fails during its execution of a recovery block $\mathcal{A}$ and recovers from an error state starting from the beginning of the recovery block, another process q which has had interaction during the execution of the recovery block of p has to restart from a state which is consistent with the state recorded by $\mathcal{A}$. This recovery of process q may also causes another process r to recover, and so on.

Given a program P, a fault-tolerant version of P can be achieved by using **FT-Rules 11-19**. Partitioning the fault-tolerant version in the way described in Theorem 4.3.1, it has been shown that the conditions given in **FT-Rules 11-19** guarantee the avoidance of the domino effect in the parallel execution of the program. These conditions formalize the requirements for constructing *conversations* proposed in [Ran75]. But these rules can only allow conversations to be constructed within one action. **FT-Rules 20-22** can then be used for constructing general conversations from a group of actions. Conversations can also be nested, and may be further refined.

4.5.3 Atomic Action Scheme

Work on fault-tolerance for distributed systems [Lom77, HW88] considers the conditions for constructing a fault-tolerant atomic action: processes executing such an action do not communicate with other processes while the action is being executed, errors which occur in the execution of this action cannot be exposed to other processes and errors which occur in the execution outside this action cannot be propagated to the execution of this action.

A distributed system is modelled in the action system formalism by partitioning the variables of a program among a set of processes. The conditions described above for fault-tolerant atomic actions are guaranteed by **FT-Rules 14**-**19** and the conditions given in Theorem 4.3.1 for partitioning a fault-tolerant program obtained by using these rules. And they are preserved by **FT-Rules 20**-**22**. Fault-tolerant atomic actions can be nested and further refined by using these rules.

The discussion presented so far in this section shows that in this transformational framework, we can uniformly describe the concepts of recovery blocks, conversations and fault-tolerant atomic actions. Such an unification reflects the *duality* of message-process and object-action approaches to the provision of fault-tolerance that was studied in [SMR87].

4.5.4 Various Uses of the Rules

It is noted that the fault-tolerant refinement rules described so far in this chapter share the following common features. A fault-tolerant version P' in $P \sqsubseteq_{T(F)} P'$ is obtained by replacing a command c in P with its fault-tolerant version. The common structure $c; Recr[c :: j : 1..k :: c_j]$ (or a retry version of c) in the fault-tolerant version of c provides the tolerance of faults during the execution of c by using the information saved by the saving command $Save_c$ after the execution enters this fault-tolerant version. The structure $Recr[c :: j : 1..k :: c_j]$ in the rules consists of a sequence of alternative implementations which can accomplish the same task as that the command c was originally expected to do. This provides backward recovery approaches to the provision of fault-tolerance. When faults occur during the execution of c and the recovery succeeds, the fault-tolerant version always provides the same results as the command c was originally expected to do.

This subsection discusses how to generalize the applications of the fault-tolerant rules by changing the condition that requires each alternative of c to be a refined version of c. By so doing, we will achieve several variants for each of the above fault-tolerant rules. Each of the variants provides an approach which is often used to the provision of fault-tolerance for certain applications.

Fail-soft refinement

In many cases for the fault-tolerant version

$$[\mid c :: j : 1..k :: c_j \mid]$$

there may be an alternative which performs a less desirable operation, but one which is still *acceptable*. That is, instead of requiring that $c \sqsubseteq c_j$ for each c_j, a command c_j can be accepted if it satisfies a specification which, although it is weaker than the expected specification of c, is *acceptable*. Suppose that we are given a *desired* specification Sp and k stand-by acceptable specifications $Sp_1, \ldots, Sp_k$. We are asked to design a command such that if no fault occurs during the execution of this command, it has to meet the desired specification; if faults occur, it has to satisfy one of the acceptable specification. Assume that we can prove that for $j = 1, \ldots, k$, c_j satisfies Sp_j. Then the fault-tolerant version

$$[\mid c :: j : 1..k :: c_j \mid]$$

is one of the commands we want, if in no case that faults can occur in the execution of all the commands c and c_j for $j = 1, \ldots, k$.

In the same way, the retry version $[\mid c ::$ **redo** $c' \mid]$ can be generalized for achieving a weaker but acceptable result compared with the desired result of the command c.

The transformation of a command into a command, in the way descibed above, which performs a less desirable operation but still acceptable is termed *fail-soft refinement*.

Example 4.5.1 Assume that we have a command (or a program)

$$c = q := \textit{append}(x, q)$$

During the execution of this command, both of the values of q and x may be lost. Let

(a) $c_1 = q := q\circ < x >$

(b) $c_2 = q := q; \textit{warning} := (error : \text{“\textit{lost item}”})$

(c) $c_3 = q :=< x >; \textit{warning} := (error : \text{“\textit{lostsequence}”})$

(d) $c_4 = q :=<>; \textit{warning} := (error : \text{“\textit{lost sequence and item}”})$

Assume that the semantics of the operations *append* and $\circ$ are defined by:

$$\Gamma(\textit{append}(x, q)) = q \,\hat{}< x >, \;\; \Gamma(q \circ q') = q \,\hat{} q'$$

Then the fault-tolerant version $[\mid c :: j : 1..4 :: c_j \mid]$ will give a result which may be weaker than the specification $\{q = q_0 \wedge x = x_0\}c\{q = q_0\,\hat{}< x_0 >\}$.

If the execution of c fails, the first alternative c_1 actually tries, by a different method than c (i.e. operations *append* and $\hat{}$ are different in the system) to append the item x onto the sequence q. If both of c and c_1 fail their execution, the rest of the alternatives will only give a part of the computation with appropriate warning messages to inform the user what have been lost. [Ran75]

Forward error recovery

Assume the execution of the command c or one of its alternatives, say c_j, results in an error state because the occurrence of faults. Sometimes the damage of the error state can be assessed and the cause of the error can be identified, by using the fault transformation $\mathcal{F}$ to the command c_j. In this case, instead of requiring the next recovery command $Recr(c, c_{j+1})$ to be a backward recovery command, it can be designed to be a forward recovery command of the form

$$Recr(c, c_{j+1}) = \textbf{if}\ \neg f \rightarrow \textbf{skip}[]f \rightarrow c_{j+1}\ \textbf{fi}$$

where f now should contain the information used for identifying the cause of the error. And the command c_{j+1} is commonly called an *exception handler*. It starts from the error state after the failed execution of c_j and tries to achieve a good state by making good use of the information maintained in the error state and continues from such a good state to get an acceptable result with respect to the desired result of c_j (of c if $j = 0$). The whole fault-tolerant version will succeed if this exception handler succeeds. If it fails, the next recovery command is then entered. Therefore, forward recovery and backward recovery approaches are both allowed in constructing a fault-tolerant version of a command.

Example 4.5.2 Let the commands c, c_j, $j = 1, 2, 3, 4$, and the fault environment be as the same as that in Example 4.5.1. Assume that the fault indicator f is equivalent to *lostsequence* $\vee$ *lostitem*. Let

(a) $Recr(c, c_1) = \textbf{if}\ \neg f \rightarrow \textbf{skip}[]f \rightarrow Restore_c; c_1\ \textbf{fi}$

(b) $Recr(c, c_2) = \textbf{if}\ \neg f \rightarrow \textbf{skip}$
$[]\ lostitem \rightarrow warning := (error : \text{“lost item”})\ \textbf{fi}$

(c) $Recr(c, c_3) = \textbf{if}\ \neg f \rightarrow \textbf{skip}$
$[]\ lostsquence \rightarrow q :=< x >;$
$warning := (error : \text{“lost squence”})\ \textbf{fi}$

(d) $Recr(c, c_4) =$ **if** $\neg f \rightarrow$ **skip**
[] *lostsquence* $\wedge$ *lostitem* $\rightarrow q :=<>$;
warning $:= (error$: *"lost squence and item"*$)$ **fi**

Then the fault-tolerant version $[\mid c :: j : 1..4 :: c_j \mid]$ becomes a version in which both backward and forward recovery approaches are used.

Intuitively, we can see that the fault-tolerant version in Example 4.5.2 is more economical than the fault-tolerant version in Example 4.5.1. This is because no restoring commands are needed for the forward recovery commands.

Multiple implementation

Again consider the fault-tolerant version $[\mid c :: j : 1..(2 \times k) :: c_j \mid]$ with $2 \times k$ alternatives $c_1, \ldots, c_{2k}$ such that $c \sqsubseteq c_j$ for each alternative c_j. If we can prove that at most k of the $2 \times k + 1$ commands $c, c_1, \ldots, c_{2\times k}$ may fail if they are executed from the same good state, this fault-tolerant version can be changed into the following form:

$$Save_c; \textbf{if } f \rightarrow c[]\neg f \rightarrow c; (Restore_c; c_1); \ldots; (Restore_c; c_{2k}); \ Vote \textbf{ fi}$$

where the execution of the command $Vote$ performs *majority voting*, taking the common result of the majority of the results (saved in different variables) of the executions of $2 \times k + 1$ commands as the final result. Such a variant of the fault-tolerant version of a command can therefore provide the *multiple implementation* (*or replication*) approach to the provision of fault-tolerance. It is noted that here we only described this invariant by taking the sequential compositions of the command c and its refined copies. There can be various implementations for this variant under different contexts. The parallel execution of the command c and its refined copies can be achieved by refining the atomicity of the action which contains the fault-tolerant command

$$Save_c; \textbf{if } f \rightarrow c[]\neg f \rightarrow c; (Restore_c; c_1); \ldots; (Restore_c; c_{2k}); \ Vote \textbf{ fi}$$

In this case, the occurrence of the saving and restoring command in this version can be removed by using refinement transformations.

4.6 Recovery in Asynchronous Communicating Systems

The fault-tolerant refinement rules described so far provide us with the approaches for constructing programs with fault-tolerant structures, i.e. fault-tolerant versions. These structures are proved to tolerate a certain kind of faults with the provision of *static* checkpointing and recovery that guarantee the consistency of the state from which the execution will continue in case faults occur. When considering parallel execution of the program, the fault-tolerant structures provide a means for coordinating the checkpointing and recovery in various processes to avoid the domino effect. However, communication between processes in parallel execution has so far been carried out by actions shared by the processes and the communication is synchronous.

In this section, we will describe asynchronous communicating systems using the action system formalism. We will then discuss the fault-tolerant issues that include both static and dynamic checkpointing and recovery. *Local state* preservation at checkpoints and *effectively founding* of a recovery line will be involved with these issues. This allows the *latest and smallest* solution to be found for checkpointing and recovery in asynchronous communicating systems.

4.6.1 Asynchronous Communicating Systems

Given a program $P = (Init_P, A_1[]\dots[]A_n)$, let us partition the actions of P into a set $Proc = \{p_1, \dots, p_k\}$ of k processes. Assume that each pair of processes p and q in $Proc$ share a variable ch_{pq} (called the *channel* from p to q) which is of type of sequences and through which process p sends messages to q. The *sending* process p sends a message to q by appending it to ch_{pq}; the *receiving* process q receives a message from p by removing the head of ch_{pq}. Let cs_{pq} be the sequence of messages actually sent by p to q and let cr_{pq} be the sequence of the messages actually received from p by process q.

For each process $p \in Proc$, let $var(p)$ be the set of variables (called *process variables*) which is the union of the following subsets:

(i) X_p: the set of all the *local* variables of p,

(ii) $From_p \triangleq \{cs_{pq} \mid q \in Proc \wedge q \neq p\}$: the set of sequence variables of messages sent from p to other processes,

(iii) $To_p \triangleq \{cr_{qp} \mid q \in Proc \wedge q \neq p\}$: the set of sequence variables of messages received by p from other processes.

Let $Ch_p \triangleq \{ch_{pq}, ch_{qp} \mid q \in Proc \wedge q \neq p\}$ be the set of channels through which process p communicates with other processes.

In an *asynchronous communicating system* with unbounded channels for the program P, each pair of processes p and q can only have the shared variables $Ch_{pq} = \{ch_{pq}, ch_{qp}\}$. These shared variables and the variables $var(p)$ are required to satisfy the following specification:

1. the sequence of messages sent by p to q never gets shorter, i.e. for any constant sequence cs_o,

$$\textbf{stable } cs_o \underline{\propto} cs_{pq}$$

2. the progress property of the sequence of messages sent by the sending process depends only on process variables, i.e. for any constant sequence cs_o,

$$(cs_o \underline{\propto} cs_{pq} \wedge \neg b_{pq}) \longmapsto (b_{pq} \vee cs_o \propto cs_{pq})$$

where b_{pq} is a predicate that does not name channels.

3. messages must be sent before they are received and must be received in the same order in which they are sent,

$$\textbf{invariant } cr_{pq} \underline{\propto} cs_{pq}$$

4. messages cannot be lost,

$$\textbf{invariant } ch_{pq} = cs_{pq} - cr_{pq}$$

5. messages sent by p to q must be eventually received by q,

$$(ch_{pq} \neq <>) \longmapsto (cr_{pq} = cr_{pq}\,\hat{}\,head(ch_{pq}))$$

6. termination of P implies termination of the communications,

$$Fixedpoint(P) \Rightarrow (cs_{pq} = cr_{pq})$$

For a pair of processes p and q, these properties can be implemented by an action $send_p$:

$$\neg b_{pq} \rightarrow cs_{pq},\ ch_{pq} := cs_{pq}\,\hat{}\,m,\ ch_{pq}\,\hat{}\,m$$

which is contained in process p and an action $receive_q$:

$$ch_{pq} \neq <> \rightarrow cr_{pq},\ ch_{pq} := cr_{pq} \hat{}\, head(ch_{pq}),\ rest(ch_{pq})$$

which is contained in process q.

Example 4.6.1 (Set-partition)

Given disjoint and non-empty sets of integers $A_1, \ldots, A_n$, $n > 1$, partition $A = A_1 \cup \ldots \cup A_n$ into n subsets $M_1, \ldots, M_n$ such that $|M_i| = |A_i|$ and for $i < n$ every element of M_i, is smaller than any element of M_{i+1}.

Let max_i, $1 \leq i < n$, and min_j, $1 < j \leq n$, be integer variables and M_i be a variable of sets of integers. A program to perform the partition can be written as:

Program *Partition*
  initially
    $[]_{ij} : 1 \leq i \leq n,\ 1 \leq j < n ::$
      $M_i,\ max_j,\ min_{j+1} = A_i,\ max(A_j),\ min(A_{j+1})$
  actions
    $[]_i : 1 \leq i < n ::$
      $max_i > min_{i+1} \rightarrow M_i := (M_i \cup \{min_{i+1}\}) - \{max_i\};$
        $M_{i+1} := (M_{i+1} \cup \{max_i\}) - \{min_{i+1}\};$
        $max_i := max(M_i);$
        $min_{i+1} := min(M_{i+1})$
End {*Partition*}

This program can be implemented by the following n communicating processes with the communication channels ch_{ij} for $i, j = 1, \ldots, n$.

process p_i
  initially
    $M_i, max_i, min_i = A_i, max(A_i), min(A_i)$
    $ch_{i-1i}, ch_{ii-1}, ch_{ii+1}, ch_{i+1i} = < max_{i-1} >, < min_i >, < max_i >, < min_{i+1} >$
  actions
    $min_i < head(ch_{i-1i}) \rightarrow M_i := M_i \cup \{head(ch_{i-1i})\} - \{min_i\};$
      $min_i := min(M_i);$
      $ch_{ii-1} := ch_{ii-1} \hat{}\, < min(M_i) >;$
      $ch_{i-1i} := rest(ch_{i-1i})$
    []
    $max_i > head(ch_{i+1i}) \rightarrow M_i := M_i \cup \{head(ch_{i+1i})\} - \{max_i\};$

$$\begin{aligned} max_i &:= max(M_i); \\ ch_{ii+1} &:= ch_{ii+1} \,\hat{}\, < max(M_i) >; \\ ch_{i+1i} &:= \textit{rest}(ch_{i+1i}) \end{aligned}$$

End $\{p_i\}$

where the sequential statements after and before [] are empty when respectively $i = 1$ and $i = n$.

4.6.2 Consistency of Local States

A global state of a program P described in the previous subsection can be partitioned into local states of its processes and the state of the channels. But the state of channels is determined by the local states of the processes. During the execution of the communicating processes of P, the invariants 3 and 4 given in the specification must be always true. These are the conditions for consistency of local states at this level. Such conditions at a lower level implement the condition for consistency at a higher level given in Section 2.4 in Chapter 2. In the sequel of this section, we always assume that the body of each action of the program P is a multiple assignment.

For each process $p \in Proc$, a *local state* of p is a state S_p over $var(p)$. For a subset $\mathcal{P} = \{p_a, \ldots, p_b\}$ of $Proc$ and a set $\Psi = \{S_{p_a}, \ldots, S_{p_b}\}$ of local states of these processes, we have:

$$Consistent(\Psi) \Leftrightarrow \forall\, p_i, p_j \in \mathcal{P} : i \neq j :: S_{p_j}(cr_{p_i p_j}) \underline{\propto} S_{p_i}(cs_{p_i p_j})$$

For processes $p, q \in Proc$ such that $p \neq q$ and local states S_p and S_q of p and q, the predicate $SemiCon(S_p, S_q)$ (i.e. *Semi-consistent*) is $true$ if

$$S_q(cr_{pq}) \underline{\propto} S_p(cs_{pq})$$

Obviously, $Consistent(\{S_p, S_q\}) \Leftrightarrow SemiCon(S_p, S_q) \wedge SemiCon(S_q, S_p)$.

4.6.3 Checkpointing

The backward recovery technique usually depends on the provision of *checkpoints*, i.e. a means by which the local state of a process can be recorded and later reinstated. In Sections 4.4 and 4.5, we have already described checkpoint commands (actions) as the saving commands (actions). However, there checkpointing must be done within a fault-tolerant structure. This subsection develops a solution for checkpointing that is inherently nondeterministic with no restriction on *when* or *where* a process takes its checkpoints.

For a program P with a set $\mathcal{P}$ of processes, assume that the fault indicator f is the disjunction of a set of $\{f_p \mid p \in Proc\}$ boolean variables whose values indicate the presence or absence of faults in the corresponding processes. Let $f_p = true$ when a fault occurs in the execution of process p and otherwise $f_p = false$.

From the specification given in Subsection 4.6.1, when a process p takes a checkpoint, it is sufficient for recovery to save the values of the process variables in $var(p)$. For each process p of P and $x \in var(p)$, let rec_x be a sequence variable which records some values of x. At any time, the checkpointing program can append the current value of each process variable x of a process p to rec_x.

Let $Rec_p = \{rec_x \mid x \in var(p)\}$ be the set of all the recording variables of p. A *checkpointing state* of process p is a state over Rec_p and denoted as CP_p. Rec_p is a set of sequence variables and the values of all the variables in $var(p)$ are required to be simultaneously appended into the corresponding variables in Rec_p when p takes checkpoint. Therefore, all the variables in Rec_p are of the same length in each state and this length is denoted as $\#CP_p$.

Given a positive number m such that $m \leq \#CP_p$, the function $CP_p(m)$ is defined as: for each variable $x \in var(p)$

$$CP_p(m)(x) = CP_p(rec_x)(m)$$

$CP_p(m)$ is a local state of p and called the m^{th} *checkpoint* of process p.

Let $\mathcal{P} = \{p_a, \ldots, p_b\}$ be a subset of $Proc$ and $\mathcal{CP}$ a set of checkpoints such that for each $p \in \mathcal{P}$, p has exactly one checkpoint $CP_p(m_p) \in \mathcal{CP}$, then $\mathcal{CP}$ is called a *consistent set of checkpoints* if

$$Consistent(\mathcal{CP}) = true$$

The *checkpointing program* for program P is required to satisfy the specification:

(i) for any $rec_x, rec_y \in Rec_p$

$$\textbf{invariant}\ \#rec_x = \#rec_y$$

(ii) for process $p \in Proc$ any $x \in var(p)$,

$$\textbf{invariant}\ rec_x \subseteq ob(x)$$

(iii) for any constant sequence c_o,

$$\textbf{stable}\ (c_o \underline{\propto} rec_x)$$

(iv) no checkpoint can be taken when a fault occurs in process p and a fault cannot modify variables used for checkpointing, i.e. for any constant sequence c_o,

$$\textbf{stable}\ f_p \wedge rec_x = c_o$$

(v) each process p will eventually take a new checkpoint unless a fault occurs in process p, i.e. for any constant sequence c_o,

$$(c_o \underline{\propto} rec_x) \longmapsto (c_o \propto rec_x) \vee f_p$$

For a process $p \in Proc$, we define the action $Save_p$ as

$$Save_p = \neg f_p \rightarrow \|_x \colon x \in var(p) :: rec_x := rec_x \, \hat{}\, x$$

Then we get the process p' as

$$p' = (Init_p \cup Init'_p, Act_p[]Save_p)$$

where $Init'_p = \{rec_x = x \mid x \in var(p)\}$. The program $P' = p'_1[]\ldots[]p'_k$ with n processes is thus the program P together with checkpointing actions.

Actually, we do not have to implement the checkpointing action of a process and the process itself into one process. We can write a checkpointing program C_P for the program P, which consists of the checkpointing actions. This means we impose no restriction on where these action will be executed. The checkpointing program C_P is written as

$$C_P = (\bigcup_{p \in Proc} Init'_p, Save_{p_1}[]\ldots[]Save_{p_k})$$

The program P' defined above is then a special implementation of the program $P[]C_P$.

4.6.4 Recovery Propagation

This subsection presents a way for determining a backward recovery line for a set of failed processes. For doing so, we shall define a linear order $\prec$ on the checkpoints of a process.

- We say that a checkpoint $CP_p(m)$ of p is *earlier than* a checkpoint $CP_p(i)$ of p, denoted $CP_p(m) \prec CP_p(i)$, if $m < i$. In this case $CP_p(i)$ is also said to be *later than* $CP_p(m)$
- We say that a checkpoint $CP_p(m)$ is *not later than* a checkpoint $CP_p(i)$, denoted $CP_p(m) \preceq CP_p(i)$, if $CP_p(m)$ is earlier than $CP_p(i)$ or $m = i$.
- Given a subset $\mathcal{P}$ of $Proc$, let $\mathcal{CP}$ and $\mathcal{CP}'$ be sets of checkpoints such that for each process $p \in \mathcal{P}$, there are exactly one checkpoint $CP_p(i_p)$ of process p in $\mathcal{CP}$ and exactly one checkpoint $CP_p(j_p)$ of process p in $\mathcal{CP}'$. Then,

  $$\mathcal{CP} \preceq \mathcal{CP}' \triangleq \forall p \in \mathcal{P} : CP_p(i_p) \preceq CP_p(j_p)$$

  and

  $$\mathcal{CP} \prec \mathcal{CP}' \triangleq (\mathcal{CP} \preceq \mathcal{CP}') \wedge \exists q \in \mathcal{P} : (CP_q(i_q) \prec CP_q(j_q))$$

Consider two processes p and q in $Proc$ such that $p \neq q$. Should process p have to restart from a checkpoint $CP_p(m)$ because of the occurrence of a fault, process q also has to restart from a checkpoint $CP_q(j)$ if the current state $\widehat{q}$ of q is not consistent with $CP_p(m)$.

$CP_p(m)$ is said to be a *recovery direct propagator* for $CP_q(j)$, that is denoted by $CP_p(m) \leadsto CP_q(j)$, if it satisfies the following conditions:

(i) $SemiCon(CP_p(m), CP_q(j))$

(ii) $\neg \exists CP_q(j') : (CP_q(j) \prec CP_q(j')) \wedge SemiCon(CP_p(m), CP_q(j'))$

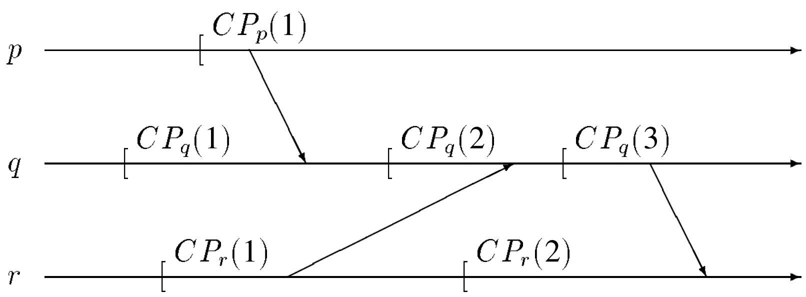
(iii) $\neg SemiCon(CP_p(m), \widehat{q})$

The *indirect propagator* relation $\leadsto^*$ between checkpoints is defined to be the reflexive and transitive closure of the direct propagator $\leadsto$ relation:

(i) $CP_p(m) \leadsto^* CP_p(m)$,

(ii) $CP_p(m) \leadsto CP_q(j) \Rightarrow CP_p(m) \leadsto^* CP_q(j)$,

(iii) $(CP_p(m) \leadsto^* CP_t(i) \leadsto^* CP_q(j) \Rightarrow CP_p(m) \leadsto^* CP_q(j)$.



$$CP_p(1) \rightsquigarrow CP_q(1) \rightsquigarrow CP_r(2) \quad CP_p(1) \rightsquigarrow^* CP_r(2)$$

Figure 4.1: Direct and indirect propagator relations

The direct and indirect propagator relations are illustrated in Figure 4.1, in which an arrow from one process to another represents message sending and receiving.

When a fault occurs, not all the processes of P have to recover starting from an earlier point. Let $\mathcal{P} \subseteq Proc$ and $\overleftarrow{\mathcal{P}}$ be a set of local states of processes in $\mathcal{P}$. For each process $p \in \mathcal{P}$, there is exactly one local state S_p of p in $\overleftarrow{\mathcal{P}}$. Then $\overleftarrow{\mathcal{P}}$ is a *recovery line* of $\mathcal{P}$ if

1. $Consistent(\overleftarrow{\mathcal{P}})$
2. $\forall\, p \in \mathcal{P}\ \forall\, q \notin \mathcal{P} : Consistent(S_p, \widehat{q})$

For a subset $\mathcal{P}$ of $Proc$, a recovery line $\overleftarrow{\mathcal{P}}$ of the process variables of the processes in $\mathcal{P}$ can be determined by the propagating relations. $\overleftarrow{\mathcal{P}}$ is also called a recovery line of $\mathcal{P}$.

In Figure 4.1, the following sets of checkpoints are recovery lines:

$\{CP_p(1), CP_q(1), CP_r(1)\}$ is a recovery line of $\{p, q, r\}$;

$\{CP_p(1), CP_q(1), CP_r(2)\}$ is a recovery line of $\{p, q, r\}$;

$\{CP_q(2), CP_r(1)\}$ is a recovery line of $\{q, r\}$;

$\{CP_q(2), CP_r(2)\}$ is a recovery line of $\{q, r\}$;

$\{CP_q(3), CP_r(2)\}$ is a recovery line of $\{q, r\}$.

But there are no other recovery lines in this figure.

Lemma 1 *A set of checkpoints $\overleftarrow{\mathcal{P}}$ is a recovery line of $\mathcal{P}$, iff*

(i) $\forall p \in \mathcal{P} : \exists! CP_p(m) \in \overleftarrow{\mathcal{P}}$

(ii) $\forall CP_p(i), CP_q(j) : ((CP_p(i) \in \overleftarrow{\mathcal{P}}) \wedge (CP_p(i) \rightsquigarrow^* CP_q(j)) \Rightarrow q \in \mathcal{P})$

(iii) *Consistent*$(\overleftarrow{\mathcal{P}})$

Proof: Directly from the definitions of the consistency and the relation $\rightsquigarrow^*$. □

The set of checkpoints propagated by $CP_p(m)$ can be termed the *recovery domain* of $CP_p(m)$ and is defined as

$$Z(CP_p(m)) \triangleq \{CP_q(i) \mid q \in Proc \wedge CP_p(m) \rightsquigarrow^* CP_q(i)\}$$

To obtain the lemmas, theorems and corollaries in this section, checkpointing and recovery are required to satisfy the following assumption[2].

Assumption 1 *If $CP_p(m)$ and $CP_p(i)$ are checkpoints of a process $p \in$ Proc, then for any process $q \in$ Proc such that $q \neq p$, then*

$$CP_p(m)(cs_{pq}) \underline{\propto} (\widehat{cs}_{pq}) \wedge CP_p(m)(cr_{qp}) \underline{\propto} \widehat{cr}_{qp}$$

and

$$(CP_p(m) \preceq CP_p(i))$$

implies that

$$CP_p(m)(cs_{pq}) \underline{\propto} CP_p(i)(cs_{pq}) \wedge CP_p(m)(cr_{qp}) \underline{\propto} CP_p(i)(cr_{qp})$$

[2]This assumption can be implemented in the following way: when process p recovers from checkpoint $CP_p(m)$, by deleting the checkpoints of process p that have been established after $CP_p(m)$, i.e. the checkpointing program also has to recover. The assumption is reasonable because that if process p recovers from $CP_p(m)$, the states of p from $CP_p(m)$ to the execution of the recovery operation are unusable.

The following results can be proved from the above assumption.

Lemma 2 *Given checkpoints $CP_p(i)$ and $CP_p(j)$ such that $CP_p(i) \preceq CP_p(j)$, then for any checkpoint $CP_q(m)$ of process q such that $q \neq p$,*

$$SemiCon(CP_q(m), CP_p(j)) \Rightarrow SemiCon(CP_q(m), CP_p(i))$$

and,

$$SemiCon(CP_p(i), CP_q(m)) \Rightarrow SemiCon(CP_p(j), CP_q(m))$$

and for any checkpoints $CP_p(m)$ and $CP_q(i)$ of processes p and q such that $p \neq q$,

$$SemiCon(CP_p(m), \widehat{q}) \Rightarrow SemiCon(CP_p(m), CP_q(i))$$

Proof: Directly from the definition of the predicate $SemiCon$ and Assumption 1. □

Lemma 3 *Given checkpoints $CP_p(i)$ and $CP_p(j)$, then for any checkpoints $CP_q(l)$ and $CP_q(m)$ of process q such that $q \neq p$,*

$$(CP_p(i) \preceq CP_p(j)) \wedge (CP_p(i) \rightsquigarrow CP_q(l)) \wedge (CP_p(j) \rightsquigarrow CP_q(m))$$

implies that $CP_q(l) \preceq CP_q(m)$.

Proof: From Lemma 2. □

Lemma 4 *Given processes p and q and checkpoints $CP_p(i)$, $CP_p(j)$ and $CP_q(m)$ such that $CP_p(i)$ is not later than $CP_p(j)$, then*

$$(CP_p(j) \rightsquigarrow CP_q(m)) \Rightarrow \exists\, CP_q(l) : (CP_q(l) \preceq CP_q(m)) \wedge CP_p(i) \rightsquigarrow CP_q(l)$$

Proof: From Lemma 2 we know that

$$\neg SemiCon(CP_p(i), \widehat{q})$$

Note that

$$SemiCon(CP_p(i), CP_q(0))$$

Let $CP_q(l)$ be the latest checkpoint of q which satisfies

$$SemiCon(CP_p(i), CP_q(l))$$

From the definition of $\rightsquigarrow$, we have

$$CP_p(i) \rightsquigarrow CP_q(l)$$

From Lemma 3, we have

$$CP_q(l) \preceq CP_q(m)$$

□

This lemma means that if by recovering at a checkpoint $CP_p(j)$, process p causes process q to recover from a checkpoint $CP_q(m)$, then by recovering at a checkpoint which is not later than $CP_p(j)$, p causes q to recover from a checkpoint which is not later than $CP_q(m)$.

Lemma 5 *The set of checkpoints*

$$\begin{aligned}\sum(CP_p(m)) = \{CP_q(i) \mid CP_q(i) \in Z(CP_p(m)) \wedge \\ \neg\,\exists\, CP_q(i') \in Z(CP_p(m)) : CP_q(i') \prec CP_q(i)\}\end{aligned}$$

is the recovery line of the set of processes

$$RL(CP_p(m)) \;\triangleq\; \{q \mid \exists\, CP_q(i) \in \sum(CP_p(m))\}$$

in which recovery propagation may appear.

Proof: It is necessary to show the following conditions:

1. $\forall\, q \in RL(CP_p(m)) : \exists! CP_q(j) \in \sum(CP_p(m))$
2. $\forall\, CP_q(i), CP_r(j) : (CP_q(i) \in \sum(CP_p(m)) \wedge (CP_q(i) \rightsquigarrow^* CP_r(j)) \Rightarrow$ $q \in RL(CP_p(m))$

3. $Consistent(\sum(CP_p(m)))$

Condition 1 is satisfied because for each process $q \in RL(CP_p(m))$, the set $\sum(CP_p(m))$ contains the earliest checkpoint of q that belongs to $\mathcal{Z}(CP_p(m))$.

Condition 2 can be derived directly from the definitions of the following sets, $\mathcal{Z}(CP_p(m))$, $\sum(CP_p(m))$ and $RL(CP_p(m))$.

Now to show Condition 3, for any checkpoints $CP_q(i), CP_r(j) \in \sum(CP_p(m))$ such that $CP_q(i) \neq CP_r(j)$, it is only required to prove

$$SemiCon(CP_q(i), CP_r(j))$$

If $\neg SemiCon(CP_q(i), CP_r(j))$, then from Lemma 2, $\neg SemiCon(CP_q(i), \hat{r})$ and there exists a checkpoint $CP_r(j')$ such that

$$(CP_r(j') \prec CP_r(j) \wedge CP_q(i) \rightsquigarrow CP_r(j')$$

Since

$$CP_p(m) \rightsquigarrow^* CP_q(i) \wedge CP_p(m) \rightsquigarrow^* CP_r(j')$$

we therefore have that

$$CP_r(j') \in \mathcal{Z}(CP_p(m))$$

This contradicts the fact that $CP_r(j)$ is the earliest checkpoint of process r in $\mathcal{Z}(CP_p(m))$. Thus,

$$SemiCon(CP_q(i), CP_r(j))$$

must be $true$. □

Therefore, the earliest checkpoints in the recovery domain of checkpoint $CP_p(m)$ represent the recovery line with respect to $RL(CP_p(m))$. This is illustrated in Figure 4.2.

It may be seen from the construction of the checkpointing program C_P that a great deal of storage may be occupied by checkpoints. There may be some *redundant*

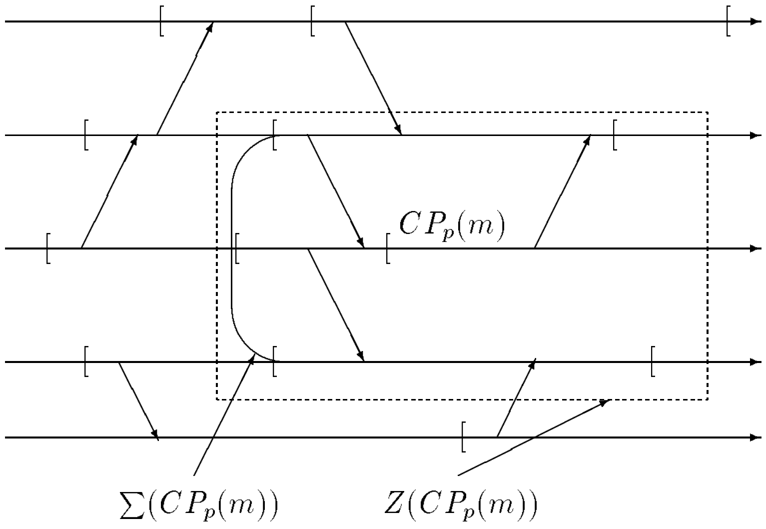


Figure 4.2: Illustration of Lemma 5

checkpoints that have been established up to an execution step that will not be used for any reasonable recovery. A checkpoint $CP_p(m)$ of p is the *active* checkpoint of p if there is no checkpoint $CP_p(j)$ of p later than $CP_p(m)$. We use $\widehat{rec}_p$ to denote the active checkpoint of p. It is easy to see that a checkpoint $CP_p(m)$ is redundant if

$$\forall\, q \in \mathit{Proc} : \neg(\widehat{rec}_q \rightsquigarrow^* CP_p(m))$$

Detection of redundant checkpoints is important for optimizing the use of retrieval storage occupied by checkpoints. To detect the redundand checkpoints and remove them, we can add some more program components in the fault-tolerant program by refinement transformations.

The following theorems characterize some important properties of checkpointing and recovery.

Theorem 4.6.1 *For each process $p \in \mathit{Proc}$ and checkpoints $CP_p(m)$ and $CP_p(i)$ of p,*

$$(CP_p(m) \preceq CP_p(i)) \Rightarrow RL(CP_p(i)) \subseteq RL(CP_p(m))$$

Proof: From Lemma 4. □

Theorem 4.6.2 *The recovery line determined in Lemma 5 is the* **latest** *recovery line for $RL(CP_p(m))$ $w.r.t.$ $CP_p(m)$, i.e. if $\mathcal{CP}$ is a set of consistent checkpoints* *such that for each process* $q \in RL(CP_p(m))$ *there is one and only checkpoint* $CP_q(n_q) \in \mathcal{CP}$ *and* $CP_p(n_p) \preceq CP_p(m)$*, then*

$$\mathcal{CP} \preceq \textstyle\sum(CP_p(m))$$

Proof: From Lemma 2, since $Consistent(\mathcal{CP})$, for any processes q and r in $RL(CP_p(m))$ and checkpoint $CP_q(i)$ such that $q \neq r$,

$$CP_q(n_q) \preceq CP_q(i) \Rightarrow SemiCon(CP_q(i), CP_r(n_r))$$

Particularly, $SemiCon(CP_p(m), CP_q(n_q))$ for each process $q \in RL(CP_p(m))$ such that $q \neq p$.

Using induction, we show that for each checkpoint $CP_q(i)$ of a process q, if $CP_p(m) \rightsquigarrow^* CP_q(i)$, then $CP_q(n_q) \preceq CP_q(i)$.

This is trivial for the case $CP_p(m) \rightsquigarrow^* CP_p(m)$.

If $CP_p(m) \rightsquigarrow CP_q(i)$, from Lemma 2 and the definition of the relation $\rightsquigarrow$, $CP_q(n_q) \preceq CP_q(i)$.

Suppose that $CP_p(m) \rightsquigarrow^* CP_r(j) \rightsquigarrow CP_q(i)$ and $CP_r(n_r) \preceq CP_r(j)$, it is required to prove that $CP_q(n_q) \preceq CP_q(i)$. Since $CP_r(n_r) \preceq CP_r(j)$, hence

$$SemiCon(CP_r(j), CP_q(n_q))$$

Again from Lemma 2 and the definition of $\rightsquigarrow$, we have $CP_q(n_q) \preceq CP_q(i)$.

Since $CP_p(m) \rightsquigarrow^* CP_q(m_q)$ for each checkpoint $CP_q(m_q) \in \sum(CP_p(m))$, therefore, we have $\mathcal{CP} \preceq \sum(CP_p(m))$. $\square$

Theorem 4.6.3 *The set of checkpoints* $\sum(CP_p(m))$ *determined in Lemma 5 satisfies: for each process* $q \in RL(CP_p(m))$*,*

$$\exists\, CP_r(i) \in \textstyle\sum(CP_p(m)) : \neg SemiCon(CP_r(i), \widehat{q})$$

Proof: From the definition of $\rightsquigarrow$. $\square$

Corollary 4.6.1 *The set of recovery processes $RL(CP_p(m))$ determined in Lemma 5 is the* **smallest** *set of processes that have to recover $w.r.t.$ $CP_p(m)$, i.e. for any recovery line $\overleftarrow{\mathcal{P}}$ of $\mathcal{P} \subseteq$ Proc that contains a checkpoint $CP_p(i)$ of process p such that $CP_p(i) \preceq CP_p(m)$,*

$$RL(CP_p(m)) \subseteq \mathcal{P}$$

Proof: From Theorem 4.6.1 and Theorem 4.6.3 □

For the detection of redundant checkpoints, it is enough to find for each process $p \in \mathcal{P}$, the recovery domain of the *latest checkpoint* of process p. The redundant checkpoints are the ones which do not belong to the set called the *active recovery domain*. This set is denoted by ARP and defined as

$$ARP \triangleq \bigcup_{p \in Proc} Z(\widehat{rec}_p)$$

Similar to Lemma 5, it is easy to show

Lemma 6 *The set*

$$Worst = \{CP_p(i) \mid CP_p(i) \in ARP \wedge (i = min\{j \mid CP_p(j) \in ARP\})\}$$

represents the recovery line of the set of processes

$$\mathcal{LP} = \bigcup_{p \in Proc} (RL(\widehat{rec}_p))$$

Proof: Similar to the proof of Lemma 5, it is necessary to show the following conditions:

1. $\forall\, p \in \mathcal{LP} : \exists! CP_p(m) \in Worst$,
2. $\forall\, CP_p(i), CP_q(j) : ((CP_p(i) \in Worst) \wedge (CP_p(i) \rightsquigarrow^* CP_q(j)) \Rightarrow q \in \mathcal{LP})$,
3. $Consistent(Worst)$.

Conditions 1 and 2 can be derived directly from the definitions of the sets $Worst$ and $\mathcal{LP}$.

To show the consistency of $Worst$, let $CP_p(i)$ and $CP_q(j)$ be checkpoints in $Worst$. If $\neg SemiCon(CP_p(i), CP_q(j))$, there must exist a checkpoint $CP_q(j')$ of process q such that $SemiCon(CP_p(i), CP_q(j'))$ (it is noted that $CP_q(1)$ is such a checkpoint). Let

$$m = max\{j' \mid SemiCon(CP_p(i), CP_q(j'))\}$$

Then $CP_q(m)$ is the latest checkpoint of q among those which are semi-consistent with $CP_p(i)$. Thus, $CP_q(m) \in ARP$.

On the other hand, from Lemma 2 and the fact that

$$\neg SemiCon(CP_p(i), CP_q(j))$$

we know that $CP_q(m) \prec CP_q(j)$ must be $true$. This contradicts the fact that

$$j = min\{i \mid CP_q(i) \in ARP\}$$

This contradiction implies that for any checkpoints $CP_p(i)$ and $CP_q(j)$ in $Worst$, $SemiCon(CP_p(i), CP_q(j))$ must be $true$. □

When a program recovers after detection of errors during an execution, the recovery line should be contained in the recovery domain of the latest checkpoint of the failed process. This motivates the *smallest* and *latest* backward recovery.

Corollary 4.6.2 *For each process* $p \in$ *Proc*, $\sum(\widehat{rec}_p)$ *is the* **smallest** *and* **latest** *recovery line* $w.r.t.$ *process* p, *i.e. for each recovery line* $\overleftarrow{\mathcal{P}}$ *such that* $p \in \mathcal{P}$,

$$RL(\widehat{rec}_p) \subseteq \mathcal{P}$$

and,

$$\forall q \in RL(\widehat{rec}_p) \cap \mathcal{P} :: (CP_q(m) \in \textstyle\sum(\widehat{rec}_p) \wedge CP_q(j) \in \overleftarrow{\mathcal{P}} \Rightarrow (CP_q(j) \preceq CP_q(m)))$$

Proof: From Theorem 4.6.2 and Corollary 4.6.1. □

Next we give the definition of the *concatenation operation* of recovery lines. This operation is necessary if more than one fault may simultaneously occur in different processes. The concatenation of two recovery lines is defined as follows:

$$\begin{aligned}
&\overleftarrow{\mathcal{P}}_1 \uplus \overleftarrow{\mathcal{P}}_2 = \\
&\quad \{CP_p(m) \mid CP_p(m) \in ((\overleftarrow{\mathcal{P}}_1 \cup \overleftarrow{\mathcal{P}}_2) - (\overleftarrow{\mathcal{P}}_1 \cap \overleftarrow{\mathcal{P}}_2)) \\
&\quad \vee (\exists\, CP_p(m_1) \in \overleftarrow{\mathcal{P}}_1, CP_p(m_2) \in \overleftarrow{\mathcal{P}}_2 \colon (m = min\{m_1, m_2\}))\}
\end{aligned}$$

Corollary 4.6.3 *The concatenation of two recovery lines* $\overleftarrow{\mathcal{P}}_1$ *and* $\overleftarrow{\mathcal{P}}_2$ *is a recovery line of the set* $\mathcal{P}_1 \cup \mathcal{P}_2$.

Corollary 4.6.4 *The concatenation* $\sum(CP_p(m)) \uplus \sum(CP_q(j))$ *for two checkpoints* $CP_p(m)$ *and* $CP_q(j)$ *is the latest and smallest recovery line* $w.r.t.$ $CP_p(m)$ *and* $CP_q(j)$*, i.e. any recovery line* $\overleftarrow{\mathcal{P}}$ *which contains* $CP_p(m)$ *and* $CP_q(j)$ *satisfies the following conditions.*

1. $\overleftarrow{\mathcal{P}} \preceq \sum(CP_p(m)) \uplus \sum(CP_q(j))$,

2. $\mathcal{P} \subseteq (RL(CP_p(m)) \cup RL(CP_q(j)))$.

The proofs of the above two corollaries are analogous to that of Lemma 6.

From the above discussions, a recovery program P_R for program P behaves in the following way. After a fault occurs in a process $p \in Proc$, a recovery line containing the process p has to be calculated by using the recovery propagation relation. The processes in this recovery line are then restored with the corresponding checkpoints of the recovery line. Also, the channels of these processes are restored with appropriate states that are determined by the checkpoints.

We present below a general recovery program P_R which makes the program P back up to a consistent state recorded by the checkpointing program.

For cleanly expressing the specification and program of the recovery, the following notations are needed. Let $\mathcal{P}$ be a subset of $Proc$ and

$$\mathcal{CP} = \{CP_p(m_p) \mid p \in \mathcal{P}\}$$

be a set of checkpoints such that for each process $p \in \mathcal{P}$ there is exactly one checkpoint $CP_p(m_p)$ of p in $\mathcal{CP}$. $\mathcal{CP}(x)$ is used to denote the value $CP_p(m_p)(x)$ for $x \in var_p$. For each pair of processes $p \in \mathcal{P}$ and $q \in Proc$ such that $p \neq q$,

(a) $ch_{pq}.rec(\mathcal{CP}) \triangleq CP_p(m_p)(cs_{pq}) - CP_q(m_q)(cr_{pq})$ if $q \in \mathcal{P}$,

(b) $ch_{pq}.rec(\mathcal{CP}) \triangleq CP_p(m_p)(cs_{pq}) - \widehat{cr}_{pq}$, if $q \notin \mathcal{P}$,

(c) $ch_{qp}.rec(\mathcal{CP}) \triangleq \widehat{cs}_{qp} - CP_p(m_p)(cr_{qp})$ if $q \notin \mathcal{P}$.

$ch_{pq}.rec(\mathcal{CP})$, $ch_{qp}.rec(\mathcal{CP})$ and $kc_p(\mathcal{CP})$ can also be simplified respectively as $v.ch_{pq}$, $v.ch_{qp}$ and kc_p when $\mathcal{CP}$ is explicitly given.

From Corollary 4.6.2, the smallest and latest recovery line $\sum(\mathcal{P})$ for the set $R(\mathcal{P})$ of processes can be defined as:

$$\sum(\mathcal{P}) = \biguplus_{p \in \mathcal{P}} \sum(\widehat{rec}_p)$$

and

$$R(\mathcal{P}) = \bigcup_{p \in \mathcal{P}} RL(\widehat{rec}_p)$$

For the set $\mathcal{P}$ of processes, let

$$f_{\mathcal{P}} = \forall\, p \in Proc : (p \in \mathcal{P} \Leftrightarrow f_p)$$

Program P_R:
$[]_{\mathcal{P}} : \mathcal{P} \subseteq Proc :: f_{\mathcal{P}} \rightarrow (\|_x : x \in Var(P) :: x := v_x).REC_P$
End $\{P_R\}$

where the predicate REC_P defines that there is a recovery line $\mathcal{CP}$ for the set $RL(\mathcal{CP})$ of processes such that the conjunction of the following predicates holds:

(i) $\mathcal{P} \subseteq RL(\mathcal{CP})$;
(ii) $\forall\, p \in RL(\mathcal{CP}) : x \in var(p) \Rightarrow v_x = \mathcal{CP}(x)$;
(iii) $\forall\, p \in RL(\mathcal{CP}) \forall\, q \in Proc : v_{ch_{pq}} = v.ch_{pq}$;
(iv) $\forall\, q \in RL(\mathcal{CP}) \forall\, p \in Proc : v_{ch_{pq}} = v.ch_{pq}$;
(v) $\forall\, p \notin \mathcal{P} : x \in var(p) \Rightarrow v_x = x$;
(vi) $\forall\, p, q \notin \mathcal{P} : v_{ch_{pq}} = ch_{pq}$.
(vii) $\forall\, p \in Proc : v_{f_p} = false$

Then we have

$$\mathcal{R}(P) \sqsubseteq_{T(F)} P[]C_P[]P_R$$

If in the above recovery program P_R, we choose $\mathcal{CP}$ as $\sum(\mathcal{P})$ and $RL(CP)$ as $R(\mathcal{P})$, P_R is refined to the latest and smallest recovery program $P_{R_{ls}}$. That is,

$$P[]C_P[]P_R \sqsubseteq_{T(F)} P[]C_P[]P_{R_{ls}}$$

It has proved in Corollary 4.6.2 that a recovery program which performs the smallest and latest recovery behaves in the manner described below. Should a fault occurs in a process $p \in Proc$, the recovery program simultaneously restores the processes in the recovery line $w.r.t.$ the active checkpoint $\widehat{rec}_p$ of process p with the corresponding checkpoints in this recovery line. It also restores the corresponding channels of the processes in $RL(\widehat{rec}_p)$ with appropriate states that are determined by the checkpoints. When faults occur in more than two processes and cause these processes to recover simultaneously, the restoration should be done with the concatenation of the recovery lines $w.r.t.$ the active checkpoints of these processes.

4.6.5 Implementations of the Checkpointing and Recovery

The fault-tolerant refinement rules described in the Sections 4.4 and 4.5 can also be applied to the refinement of $P[]C_P[]P_R$. If we use those rules, the recovery line is determined statically and the recording variables therefore do not have to be of type of sequences because just the last checkpoint of a process is active. Recovery propagation can also be restricted to lie within fault-tolerant structures, e.g. no recovery propagations between any process inside a conversation and any process outside the conversation. This subsection presents informal discussions on how to refine $P[]C_P[]P_R$ to allow dynamic checkpointing and recovery.

There are two approaches to dynamic checkpointing and recovery. With the first approach, processes take checkpoints independently and save them. Upon the presence of a fault, processes must find a consistent set of saved checkpoints which form a recovery line. The system is then rolled back and restarted from this recovery line. Techniques for the provision of fault-tolerance in communicating systems by using this approach, such as [ALS79, Rus80, Woo81, Had82], provide a class of refined versions of $P[]C_P[]P_R$. A sub-class of these refined version is the class of the refined versions of $P[]C_P[]P_{R_{ls}}$, which represent the latest and smallest recovery algorithms.

The direct and indirect indicator relations between checkpoints characterize the domino effect described in Section 4.5. These two relations are necessary for finding the recover lines. The independent checkpoints occupy a large amount of stable storage (even though the redundant checkpoints can be identified and discarded) and the domino effect will cause the recovery to take a lot of time. Therefore, the use of this approach is limited to the non-time-critical applications or applications with a very low frequency of occurrence of faults.

With the second approach, processes coordinate their checkpointing actions such that each process saves only one checkpoint, i.e. its most recent checkpoint, and the set of of checkpoints in the system is guaranteed to be consistent. When a fault occurs, the system restarts from these checkpoints [TS84].

From the definition of consistency of checkpoints, if each process takes a checkpoint after every sending of a message, and these two actions are done atomically, the set of the most recent checkpoints is always consistent. But creating a checkpoint after every send is expensive in a system which is mainly for carrying out communications between processes. Such an implementation is limited to the applications without many communications involved.

Coordinated checkpointing in asynchronous communicating systems can also be accomplished as follows. Processes are halted temporarily at arbitrary times; once halted, a process remains halted until the global state has been recorded. When all the processes are halted, their states are recorded. After all the process states have been recorded, the processes are restarted. Recording a global state in this way is usually termed as *global snapshots* [CL85, CM88, Dij85, FGL82]. We can introduce a new process, called a *checkpointing process* and refine the program $P[]C_P$ so that it will behave in the following way.

1. The checkpointing process sends a message to each process requesting it to halt the execution of the underlying computation.
2. After receiving the message, a process halts and sends an acknowledgment to the checkpointing process indicating that it has suspended the underlying computation.
3. After receiving acknowledgments from all the processes, the checkpointing process sends a message to each process requesting it to send its fault indicator.
4. After receiving the message, a process sends its fault indicator to the checkpointing process.
5. After receiving the fault indicators from all processes, if no fault indicator is $true$, the checkpoint processes sends sends a message to each process requesting it to record its state. If one or more of the fault indicators are $true$, the checkpointing process sends a message to each process requesting it await further instruction.
6. After receiving the message, if it is a waiting message, a process sends an acknowledgment to the checkpointing process. If it is a message for recording the state, the process records its state and sends the recorded state to the checkpointing process.

7. After receiving acknowledgment or the recorded states from all processes, the checkpointing process sends a message to each process instructing it to resume the execution of the underlying computation.

8. After receiving the message, a process resumes the execution of the underlying computation and sends an acknowledgment to the checkpointing process.

9. After receiving acknowledgments from all processes, the checkpointing process is ready to indicate another recording

In this program, the set of the most recent checkpoints is guaranteed to be consistent. Therefore, only the most recent checkpoint of each process has to be saved. We have to assume that the communications between the checkpointing process and the other processes are reliable, and recording of the local states in all the processes cannot fail.

After refining the program $P[]C_P$ in this way the fault-tolerant program $P[]C_P[]P_R$ can be refined further by introducing another process, called *recovery process*. The refined version of the program will behave in the following way.

1. After a fault occurs in a process, it sends a message to the recovery process to indicate the presence of the fault.

2. After receiving the message, the recovery process sends a message to each process (including the checkpoint process) requesting it to halt the execution of the underlying computation.

3. After the receiving the message, a process halts and sends an acknowledgment to the recovery process indicating that it has suspended the underlying computation.

4. After receiving acknowledgments from all the processes, the recovery process sends a message to each process (except the checkpointing process) requesting it to restore its variables with its recorded state.

5. After receiving the message, a process restores its state and sends the state to the recovery process.

6. After receiving restored process states from all processes, the recovery program computes the states of the channels from these process states and sends a message to each process (including the checkpointing process) instructing it to resume the execution from the restored state.

7. After receiving the message, a process resumes the execution and sends an acknowledgment to the recovery process.

8. After receiving acknowledgments from all processes, the recovery process is ready to receive another message indicating the presence of a fault.

We also have to assume that the communications between the recovery process and the other processes are reliable, and that the execution of restoring the local states in all the processes cannot fail.

Chapter 5

Example: A Resource Allocation Problem

5.1 Introduction

Considering an example of a resource allocation problem, this chapter demonstrates the use of the transformational approach developed in the previous chapters, together with stepwise refinement and modular design of fault-tolerant reactive systems.

Informal Description of the Problem

We are given a program $user$ with a set $Proc$ of processes which share a single resource. Associated with each process $p \in Proc$ is a variable $p.state$ whose value space is the set $\{d, r, u\}$. Informally, $p.state = d$, $p.state = r$, $p.state = u$ denote respectively that process p is *doing* computation without using the resource, *requesting* for the resource, *using* the resource in the computation. The only state transitions are from d to r, from r to u, and from u to d. It is given that every process which is using the resource will eventually release the resource and start *doing* computation without using the resource . Assume that only the transitions from d to r, and from u to d, are given in the program $user$. We are required to design a program os (for operating system) such that the union composition of $user$ and os satisfies the following conditions.

Informal Description of $user[]os$

Des1. Only one process can use the resource at a time.

Des2. A process which requests for the resource will eventually receive the resource.

From these two conditions, it follows that

Des3. A process which is using the resource will eventually release the resource.

The state transtion from r to u has to be decided during the development of program $user[]os$.

Organization of the Chapter

In Section 5.2, we start by giving the specification of the program $user$ and the specification of the composite program. We then construct a program specification os, for the given specification of $user$, such that the composite program specification $user[]os$ satisfies the given specification of composite program. This provides a nonfault-tolerant solution for the resource allocation problem which also appeared in [CM88].

Section 5.3 demonstrates how to find fault-tolerant solutions for $user[]os$. Probable faults of processes of $user$ are considered, and specifications of $\mathcal{F}(user)$, which defines the failure semantics of $user$, is used to derive the fault properties of $\mathcal{F}(user)[]os$. Based on such properties, os is refined to tolerate the faults of $user$. The program $user$ may also be refined. This refinement seeks to produce a better failure semantics to make the refinement of os easier and less expensive to tolerate the faults of the refined version of $user$.

We illustrate in Section 5.4 how, using a central scheduling process p_0, to refine $user[]os$ to $cuser[]p_0$. We show how to deal with various assumptions about the failure semantics of $cuser$, and to derive the failure properties of $cuser[]p_0$ from these assumptions. Based on the derived properties of $\mathcal{F}(cuser)[]p_0$, ways of refinement of $cuser$ and p_0 are suggested to meet the fault-tolerant requirements of $user[]os$.

In Section 5.5, a decentralized implementation of $user[]os$ is considered. Broadcast communication channels between processes are employed. Then fault-tolerant broadcasts are considered. We do not provide new solutions for the problems of fault-tolerant broadcasts. However, we shall discuss, in the framework presented in this thesis, how to use the existing solutions for the problem of *reliable broadcasts* [SG84], the problem of *reaching agreement in the presence of faults* [PSL80] and the *Byzantine General problem* [LSP82]. It is shown that the various solutions can be described and used in a single computational model so that the relations between different solutions become clear.

5.2 The Non-fault-tolerant Solution

Notation

For $p \in Proc$ and $s \in \{d, r, u\}$, let $p.s$ be a predicate which is *true* when process p is in state s.

1. $p.d \triangleq (p.state = d)$
2. $p.r \triangleq (p.state = r)$
3. $p.u \triangleq (p.state = u)$

Specification of *user*

The only transitions given in $user$ are from d to r and from u to d. No process uses the resource forever if the composite program guarantees that different processes do not use the resource at the same time. These properties are defined in the following specification $SpecUser$ of $user$.

$SpecUser$:

U1. $p.d$ **unless** $p.r$

U2. **stable** $p.r$

U3. $p.u$ **unless** $p.d$

U4. Conditional Property:

Hypothesis: $\forall\, p, q :: \neg(p.u \wedge q.u \wedge p \neq q)$

Conclusion: $\forall\, p :: p.u \longmapsto \neg p.u$

From these, for any process p, there is no transition from $\neg p.u$ to $p.u$ in $user$.

U5. **stable** $\neg p.u$

Specification of $user[]os$

In the composite program $user[]os$, different processes cannot use the resource simultaneously (UO1), and each requesting process receives the resource eventually (UO2). These two properties form of the specification $SpecUo$ of the composite program.

$SpecUo$:

UO1. **invariant** $\neg(p.u \wedge q.u \wedge p \neq q)$

UO2. $p.r \longmapsto p.u$

Constraints of os

The only variables shared between $user$ and os are $p.state$, for all processes. There are no transitions in os from d (O1), or from u (O2).

O1. **constant** $p.d$

O2. **stable** $p.u$

Derived Properties of os:

From O1 and O2 it follows that there are no transitions to r in os, and that the only transition in os from r is to u.

$$\textbf{stable}\ \neg p.r$$

$$p.r\ \textbf{unless}\ p.u$$

Derived Properties of $user[]os$

From U1 – U3 in $SpecUser$ and the constraints O1 and O2 on os, the following are derived properties of $user[]os$.

UO3. $p.d$ **unless** $p.r$

UO4. $p.r$ **unless** $p.u$

UO5. $p.u$ **unless** $p.d$

A specification of os

To meet the the specification $SpecUo$ for the given $SpecUser$, we propose a specification $SpecOs1$ which consists of O1 and O2 plus O3 and O4, given next.

$SpecOs1$: O1, O2

O3. **invariant** $\neg(p.u \wedge q.u \wedge p \neq q)$

O4. Conditional Property

Hypothesis: UO1, UO3, UO4, UO5, $\forall p :: p.u \longmapsto \neg p.u$

Conclusion: $\forall p :: p.r \longmapsto p.u$

5.2.1 Refinement Step: Introduction of a Total Ordering

Let us attempt to derive a program from the above specification. Obviously

$$
\begin{array}{l}
[]_p : p \in Proc :: \\
\quad p.r \wedge \forall q \in Proc : \neg q.u \rightarrow p.state := u
\end{array}
$$

satisfies O3, i.e. a requesting process uses the resource if no other process is using it. Unfortunately, the progress property O4 – requesting processes will eventually use the resource – may not hold.

Consider any arbitrary total ordering $\ll$ of the processes $Proc$, allowing os to select a single winner from a set of conflicting processes.

For each process $p \in Proc$, we define

$$p.top \triangleq \forall q : p \neq q \wedge q.r :: q \ll p$$

For $p \neq q$, let $prior[p, q]$ be p if $q \ll p$, q otherwise. The ordering $\ll$ is dynamic so that the priority of a requesting process will increase until it gets the resource. We refine $SpecOs1$ to $SpecOs2$.

$SpecOs2$: O1, O2, O3

O5. **invariant** $p.u \Rightarrow prior[p, q] = q$

O6. $(prior[p, q] = q)$ **unless** $q.u$

O7. **invariant** $\ll$ is total

O8. Conditional Property

Hypothesis: UO1, UO3 – UO5, O5 – O7, $\forall p :: p.u \longmapsto \neg p.u$

Conclusion: $\forall p :: p.r \wedge p.top \longmapsto \neg(p.r \wedge p.top)$

Correctness 1 *SpecOs2* $\Rightarrow$ *SpecOs1*

The program given below satisfies $SpecOs2$:

Program os
$[]_p : p \in Proc ::$
$p.r \wedge p.top \wedge \forall q : \neg q.u \rightarrow p.state := u; \|_{q \neq p} :: prior[q, p] := q$
End $\{os\}$

Therefore, if $user$ satisfies the $SpecUser$, then

$$user[]os \ \underline{Sat} \ SpecUo$$

5.3 Fault-tolerant Solutions

We have defined the conditions of the correctness of os in the previous section. However, if any process of $user$ suffers physical faults, then some conditions in the specification $SpecUser$ may be violated during the execution. And thus the specification of $SpecUo$ may not hold for $\mathcal{F}(user)[]os$. The specification of $\mathcal{F}(user)$ will give the assumptions about the effect of faults of $user$.

Given a variable f_p for each process p, let f_p be $true$ when process p fails. We assume that the failure semantics of $user$ satisfies the following specification.

$SpecFuser$:

FU1. In any process, faults cannot cause transitions from d or r to u:

$$\mathcal{F}(user) \ \underline{Sat} \ \textbf{stable} \ \neg p.u$$

FU2. If a process is in state d and has not failed, it stays d until it enters state r or failed:

$$\mathcal{F}(user) \ \underline{Sat} \ \neg f_p \wedge p.d \ \textbf{unless} \ p.r \vee f_p$$

FU3. If a process is in state u and has not failed, it stays in u until it enters d or failed:

$$\mathcal{F}(user) \ \underline{Sat} \ \neg f_p \wedge p.u \ \textbf{unless} \ p.d \vee f_p$$

FU4. If any different processes do not use the resource at the same time, a process which is using the resource and has not failed will eventually release the resource or fail:

Conditional Property
Hypothesis: $\forall p, q : p \neq q :: \neg(p.u \wedge q.u \wedge p \neq q)$
Conclusion: $\forall p : \neg f_p \wedge p.u \longmapsto f_p \vee \neg p.u$

It is noted that we have no assumption about whether a failed process can be restarted. The above specified failure semantics of $user$ is assumed to be guaranteed by the designers of $user$ and the hardware system.

Now let us discuss the properties of the fault-affected program $\mathcal{F}(user)[]os$.

Correctness 2 (Safety)

In the fault affected program $\mathcal{F}(user)[]os$, no two processes use the resource at the same time.

FUO1. $\mathcal{F}(user)[]os$ $\underline{Sat}$ **invariant** $\neg(p.u \wedge q.u \wedge p \neq q)$

Proof: Since,

$$\mathcal{F}(user)\ \underline{Sat}\ \textbf{stable}\ \neg p.u$$

$$os\ \underline{Sat}\ \textbf{invariant}\ \neg(p.u \wedge q.u \wedge p \neq q)$$

From the union theorem, we have

$$\mathcal{F}(user)[]os\ \underline{Sat}\ \textbf{invariant}\ \neg(p.u \wedge q.u \wedge p \neq q)$$

□

Unfortunately, the properties UO2 – UO5 are not satisfied by the fault-affected program $\mathcal{F}(user)[]os$. Even the property

FUO2. $\neg f_p \wedge p.r \longmapsto f_p \vee p.u$

is not satisfied by $\mathcal{F}(user)[]os$. This is because a process may fail when it is using the resource and prevent other requesting processes from entering their u states even though they may not fail. Also it is possible that a process enters state r from u or d because of a fault in the process. Such a fault may then be propagated to os and cause the transition of the state of the failed process from r to u.

However we have the following properties.

Correctness 3 (Fault Properties)

Program $\mathcal{F}(user)[]os$ satisfies:

FUO3. $\neg f_p \wedge p.d$ **unless** $f_p \vee p.r$

FUO4. $\neg f_p \wedge p.r$ **unless** $f_p \vee p.u$

FUO5. $\neg f_p \wedge p.u$ **unless** $f_p \vee p.d$

Proof: From FU1 – FU4 of $\mathcal{F}(user)$ and the properties of program os. □

To guarantee that the property FUO2 holds for the fault-affected composite program, we can refine $user$ into $user_1$ by using the refinement rules of commands so that, instead of FU4, $\mathcal{F}(user_1)$ satisfies the following property FU5.

FU5. Conditional Property

Hypothesis: $\forall\, p, q : \neg(p.u \wedge q.u \wedge p \neq q)$

Conclusion: $\forall\, p : p.u \longmapsto \neg p.u$

This means that $user_1$ guarantees that either: (1) a process cannot fail when it is using the resource, or (2) a process will eventually release the resource if it fails during using the resource.

Also we have to refine the program os into a program os_1 such that a failed process is not allowed to use the resource. Moreover, a failed requesting process cannot prevent other requesting processes from using the resource.

To do this, failed processes must be notified by os_1. Let $Procf$ be a variable local to os_1 which contains the names of the failed processes notified by os_1. Le os_1 be the following program.

Program os_1
actions
$[]_p : p \in Proc ::$
$p.r \wedge \forall\, q : \neg q.u \rightarrow Procf := \{p \mid f_p\}; changeorder;$
if $p.top \wedge p \notin Procf \rightarrow p.state := u; \|_{q \neq p} :: prior[q, p] := q []$
$\neg f.top \vee p \in Procf \rightarrow$ **skip fi**
End $\{os_1\}$

where the commands $changeorder$ satisfies

$$\{\ll\ is\ total\}\, changeorder\{\ll\ is\ total\}$$

and for all pairs p, q of processes, $p \neq q$,

$$\{p \ll q\}\, changeorder\{(\neg f_q \Rightarrow p \ll q) \wedge (f_q \wedge \neg f_p \Rightarrow q \ll p)\}$$

It is noted that a failed process can be allowed to restart in the execution of $user_1$. In this case, os_1 allows the restarted process to use the resource if it is on the top (according to the order $\ll$) of the requesting processes that have not failed.

We discuss below the properties of the program $\mathcal{F}(user_1)[]os_1$.

Correctness 4 *The program $\mathcal{F}(user_1)[]os_1$ is a refinement of $\mathcal{F}(user)[]os$:*

$$\mathcal{F}(user)[]os \sqsubseteq \mathcal{F}(user_1)[]os_1$$

Moreover, $\mathcal{F}(user_1)[]os_1$ satisfies the progress property FUO2*,*

$$\mathcal{F}(user_1)[]os_1\ \underline{Sat}\ \neg f_p \wedge p.r \longmapsto p.u \vee f_p$$

Proof: It is easy to see that os_1 satisfies the following properties.

FO1. For any process p, both $p.d$ and $\neg p.d$ are stable in os_1,

$$os_1\ \underline{Sat}\ \textbf{constant}\ p.d$$

FO2. For any process p, $p.u$ is stable in os_1,

$$os_1\ \underline{Sat}\ \textbf{stable}\ p.u$$

FO3. No two processes use the resource at the same time,

$$os_1\ \underline{Sat}\ \textbf{invariant}\ \neg(p.u \wedge q.u \wedge p \neq q)$$

FO4. A process which is using the resource has lower priority than other processes which have not failed,

$$os_1\ \underline{Sat}\ \textbf{invariant}\ p.u \Rightarrow prior[p, q] = q \vee q \in Procf$$

FO5. A process decreases its priority when it starts using the resource, or fails,

$$os_1\ \underline{Sat}\ prior[p, q] = q\ \textbf{unless}\ q.u \vee q \in Procf$$

FO6. The order $\ll$ is always total,

$$os_1 \ \underline{Sat} \ \textbf{invariant} \ (\ll \ is \ total)$$

FO7. A failed process cannot be allowed to use the resource,

$$os_1 \ \underline{Sat} \ \textbf{stable} \ (f_p \wedge \neg p.u)$$

FO8. The following conditional property holds,

Hypothesis: FUO1, FUO3 – FUO5, FO3 – FO7,
$\forall \, p :: p.u \longmapsto \neg p.u$

Conclusion: $\forall \, p, q :: (\neg f_p \wedge p.r \wedge (f_q \wedge q.r \Rightarrow q \ll p) \longmapsto p.u \wedge p \ll q)$

Correctness 2 and **Correctness** 3 imply that $\mathcal{F}(user)[]os$ satisfies FUO1 and FUO3 – FUO5. From these properties FO1 – FO8 and the specification of $user_1$, it is easy to prove that the program $\mathcal{F}(user_1)[]os_1$ also satisfies FUO1 and FUO3 – FU05.

These properties of $user_1$, os_1 and $\mathcal{F}(user_1)[]os_1$ imply that the progress property FUO2 holds in $\mathcal{F}(user_1)[]os_1$:

$$\mathcal{F}(user_1)[]os_1 \ \underline{Sat} \ \neg f_p \wedge p.r \longmapsto p.u \vee f_p$$

which is not satisfied by $\mathcal{F}(user)[]os$. $\square$

Corollary 5.3.1 *If no recovery actions are employed in user*$_1$*, i.e.*

$$\mathcal{F}(user_1) \ \underline{Sat} \ \textbf{stable} \ f_p$$

then,

$$\mathcal{F}(user_1)[]os_1 \ \underline{Sat} \ \textbf{stable} \ (f_p \wedge \neg p.u)$$

If the condition of this corollary holds, the failed processes notified by os_1 can be removed from the set of processes competing for the resource. Formally stated, os_1 can be refined to a program os'_1 described below.

Let $Procl$ and $p.ltop$ be variables local to os'_1, $Procl$ is a set variable which contains the names of the processes which have not been notified by os'_1 as failed processes,

$$Procl \triangleq Proc - Procf$$

and $p.ltop$ is a boolean variable defined as

$$p.ltop \triangleq \forall\, q \in Procl : p \neq q \wedge q.r :: p \in Procl \wedge prior[p,q] = p$$

Program os_1'
  actions
    $[]_p : p \in Proc ::$
      $p.r \wedge \forall\, q : \neg q.u \rightarrow Procl := Procl - \{q \mid q \in Procl \wedge f_q\};$
      if $p.ltop \rightarrow p.state := u; changeorderl(p)[]$
      $\neg p.ltop \rightarrow$ **skip fi**
 End $\{os_1'\}$

where the command $changeorderl(p)$ changes p to be the least element in $Procl$,

$$\forall\, q \in Procl :: p \ll q$$

Corollary 5.3.2 *If a process in* $user_1$ *never fails when it is using the resource, i.e.*

$$\mathcal{F}(user_1)\ \underline{Sat}\ \textbf{invariant}\ \neg(f_p \wedge p.u)$$

then,

$$\mathcal{F}(user_1)[]os_1\ \underline{Sat}\ \textbf{invariant}\ \neg(f_p \wedge p.u)$$

The Specification $SpecFuser$ does not guarantee the condition of the corollary

$$\mathcal{F}(\mathrm{user}_1)\ \underline{\mathrm{Sat}}\ \textbf{invariant}\ \neg(f_p \wedge p.u)$$

Therefore, program os_1 does not in general guarantee that

$$\mathcal{F}(\mathrm{user}_1)[]os_1\ \underline{\mathrm{Sat}}\ \textbf{invariant}\ \neg(f_p \wedge p.u)$$

Consider a failure semantics of $user\,1$ which satisfies both the specification $SpecFuser$ and the condition of the corollary. The corollary shows that such a 'better' failure semantics implies the 'better' failure semantics of the reactive program $user_1[]os_1$ which satisfies

$$\mathcal{F}(user_1)[]os_1 \ \underline{Sat} \ \textbf{invariant} \ \neg(f_p \wedge p.u)$$

Stated in another way, the better failure semantics of $user_1$ implies the better fault-tolerant properties of $user_1[]os_1$.

Using the techniques described in the previous chapters, recovery actions can be added to $user_1[]os_1$. A failed process can be restarted by executing such a recovery action. Execution of a recovery action will transform the state of $user_1[]os_1$ to a consistent state. The consistency of such a state when restarting a process p depends on the application of $user_1$. However, it must imply that

$$\forall q_1, q_2 : \neg(p_1.u \wedge q_1.u \wedge p_1 \neq q_1) \wedge \neg f_p$$

5.4 A Centralized Implementation

Now we discuss how to implement $user_1[]os_1$ by using a central scheduling process. We want to refine $user[]os$ into a program consisting of the refined versions of the processes in $Proc$ and a scheduling process p_0.

5.4.1 Non-fault-tolerant Solution

Let process p_0 have the following local variables:

- for each pair of processes $p, q \in Proc$, $prior[p, q]$ such that $prior[p, q] = p$ if $q \ll p$, q otherwise;
- a set variable $Procr$ whose value is a subset of the processes in $Proc$ that are notified by p_0 as requesting processes;
- a boolean variable $occu$ which is $true$ if p_0 has not notified that the resource is not being used (occupied) by a process.
- the boolean variables $p.top$ for all p,

$$p.top \triangleq p \in Procr \wedge \forall q \in Procr : (q \neq p \rightarrow prior[p, q] = p)$$

Processes communicate with each other through channels (synchronous or asynchronous). Channels are sequence variables shared by processes. A process p sends a process q a message m by appending m to the channel ch_{pq}. Process q receives a message from process q by removing the head of ch_{pq} and assigns m to a local variable m_{pq}. We assume that there are channels, ch_{pp_0} and ch_{p_0p}, between process p_0 and any other process $p \in Proc$.

The processes behave according to the following rules.

1. To enter r from d, a process p sends a message $p.request$ to p_0 through ch_{pp_0} and transforms $p.state$ from d to r and stays in this state. This is implemented by transforming the action of p which carries out the transition from d to r into the action

$$ch_{pp_0} =<> \rightarrow ch_{pp_0}, p.state :=< p.request >, r$$

2. When process p_0 receives the message $p.request$ from process p, it places the requesting process p into the ordered set $Procr$.

3. To go from u to d, a process p sends a message $p.release$ to p_0 through ch_{pp_0} and transforms its states from u to d. This is implemented by transforming the action of p which carries out the transition from u to d into the action

$$ch_{pp_0} =<> \rightarrow ch_{pp_0}, p.state :=< p.release >, d$$

4. When p_0 receives the message $p.release$ from p, it removes p from the set $Procr$.

5. When process p is in $Procr$ and $\neg occu \wedge p.top$, process p_0 sends process p a message $p.grant$ through ch_{p_0p}, assigns $prior[p, q]$ to be q for any process $q \in Proc$ and assigns $occu$ to be $true$.

6. When process p receives the message $p.grant$, it changes its state from r to u. For doing this, remove the action carrying out the transition of p from r to u from os and add to $user$ the action

$$ch_{p_0p} \neq <> \rightarrow p.state := u$$

While the rules (1), (3), (6) are implemented by refining $user$ to a program, denoted $cuser$, in the way described above, the rest of the rules are to be implemented by process p_0.

The refined version $cuser$ of $user$ satisfies the the properties U1 – U3 and the following properties about communications.

CU1. A channel ch_{pp_0} can hold only one message, and the message sent through a channel ch_{pp_0} can only be $p.request$ or $p.release$,

$$\textbf{invariant}\ (ch_{pp_0} =<> \vee ch_{pp_0} =< p.request > \\ \vee ch_{pp_0} =< p.release >)$$

CU2. A channel ch_{pp_0} contains a message $p.request$ to be delivered if and only if p is in state r,

$$\textbf{invariant}\ (p.r \Leftrightarrow p.request \in ch_{pp_0})$$

CU3. A channel ch_{pp_0} contains a message $p.release$ to be delivered if and only if p is in state d,

$$\textbf{invariant}\ (p.d \Leftrightarrow p.release \in ch_{pp_0})$$

CU4. A channel ch_{p_0p} can hold only one message which is $p.grant$,

$$\textbf{invariant}\ (ch_{p_0p} =<> \vee ch_{p_0p} =< p.grant >)$$

CU5. ch_{p_0p} is not empty only when p is in state r,

$$\textbf{invariant}\ (ch_{p_0p} \neq <> \Rightarrow p.r >)$$

CU6. A message $p.grant$ sent from p_0 to process p will be eventually received by p,

$$p.r \wedge ch_{p_0p} \neq <> \longmapsto ch_{p_0p} =<> \wedge \neg p.r$$

CU7. A process stays in $\neg p.u$ when ch_{p_0p} is empty,

$$\textbf{stable}\ (\neg p.u \wedge ch_{p_0p} =<>)$$

CU8. If no two processes are using the resource at the same time, and a message in the channel between a process of $cuser$ and p_0 will be eventually delivered to the receiving process, then a process which is using the resource will eventually release the resource. This is described below as a conditional property:

Hypothesis: $\forall\, p, q :: \neg(p.u \wedge q.u \wedge p \neq q)$,
$\forall\, p :: ch_{pp_0} \neq <> \longmapsto ch_{pp_0} =<>$

Conclusion: $\forall\, p :: p.u \longmapsto \neg p.u \wedge ch_{pp_0} =< p.release >$

For the program $cuser$ to satisfy the above specification, we give a central scheduling process p_0 below.

Process p_0

actions

$[]_p : p \in Proc ::$

$p.request \in ch_{pp_0} \rightarrow ch_{pp_0} :=<>;\ Procr := Procr \cup \{p\}$

$[]$

$ch_{p_0p} \neq<> \wedge \neg occu \wedge p.top \rightarrow ch_{p_0p} :=< p.grant >;$

$occu := true; \|_{q \neq p}: prior[p, q] := q$

$[]$

$p.release \in ch_{pp_0} \rightarrow ch_{pp_0} :=<>;\ Procr := Procr - \{p\};\ occu := false$

End $\{p_0\}$

We can prove that the process p_0 satisfies the following properties.

CO1. Both $p.d$ and $\neg p.d$ are stable in p_0,

$$\textbf{constant}\ p.d$$

CO2. $p.u$ and $\neg p.u$ are stable in process p_0,

$$\textbf{constant}\ p.u$$

CO3. Different processes cannot be granted the resource at the same time,

$$\textbf{invariant}\ \neg(ch_{p_0p} \neq<> \wedge ch_{p_0q} \neq<> \wedge p \neq q)$$

CO4. The process granted the resource is of the lowest priority,

$$\textbf{invariant}\ ch_{p_0p} \neq<> \Rightarrow prior[p, q] = q$$

CO5. The priority of a process cannot be reduced unless p_0 grants it the resource,

$$(prior[p, q] = q)\ \textbf{unless}\ (ch_{p_0q} \neq<>)$$

CO6. $\ll$ is always total,

$$\textbf{invariant}\ (\ll\ is\ total)$$

CO7. p_0 has the following conditional property,

Hypothesis: $\forall\, p :: p.u \longmapsto \neg p.u \wedge ch_{pp_0} =< p.release >$

Conclusion: $\forall\, p :: p \in Procr \longmapsto p \notin Procr$

From the properties of $cuser$ and p_0 described above, we can prove all the properties of $user[]os$ given in its specification. Therefore, $cuser[]p_0$ is a simulation (see Subsection 4.2.2 of $user[]os$.

5.4.2 Faults and Fault-tolerance

The problem that we consider below is how to refine $cuser[]p_0$, under the given fault properties of $\mathcal{F}(cuser)$, to another centralized program $cuser_1[]p_0'$ such that the fault-tolerant properties of $user_1[]os_1$ are satisfied. That is, under the given specification of $\mathcal{F}(cuser)$, refine $cuser[]p_0$ to $cuser_1[]p_0'$ so that $\mathcal{F}(cuser_1)[]p_0'$ refines $\mathcal{F}(user_1)[]os_1$.

We carry out this refinement in similar steps to the refinement from $user[]os$ to $user_1[]os_1$:

1. derive the fault properties of $\mathcal{F}(cuser)[]p_0$ from that of $\mathcal{F}(cuser)$;
2. failed processes must be notified by the central scheduling process p_0' or restarted in $user_1$ before p_0' sends a message to grant a process for using the resource;
3. refine p_0 to p_0' (and possibly $cuser$ to $cuser_1$) to avoid or correct the state transformations that $\mathcal{F}(cuser)[]p_0$ may carry out and violate properties in the specification of $\mathcal{F}(user_1)[]os_1$;
4. finally, verify that $\mathcal{F}(cuser_1)[]p_0'$ satisfies the specification of $\mathcal{F}(user_1)[]os_1$.

Fault Properties of $\mathcal{F}(cuser)[]p_0$

Now we discuss the failure semantics of $cuser$ and show how to derive the fault properties of $\mathcal{F}(cuser)[]p_0$.

A process p may fail and enter d from r. In this case, p may or may not send any information to p_0. If p sends a message to p_0, the message may be $p.request$ or $p.release$ (or anything else). When p fails in this way, the failure will cause the following fault properties of $\mathcal{F}(cuser)[]p_0$.

FR1. If p sends p_0 no message or any message different from $p.request$ or $p.release$, this failure of p does not prevent other good processes from using the resource if $p \notin Procr$. But if $p \in Procr$, the failure may cause p_0 to grant the failed process the resource and good processes to be starved.

FR2. If p sends $p.request$ to p_0, the failure may also cause p_0 to grant the failed process the resource and good processes to be starved.

FR3. If p sends $p.release$ to p_0, the failure may cause p_0 to assign $occu$ to $true$ while a process is using the resource and then to grant another process in $Procr$ the resource. This will destroy the safety condition of the system that different processes cannot use the resource at the same time.

To tolerate the faults described above, the central scheduling process p'_0 has to remove the failed process from $Procr$ before it selects the winner of the notified requesting processes. This can be done in the same way in os_1.

When a requesting process fails, it may stay in state r. But it still keeps sending messages to p_0. The fault properties in this case are the same as those described above corresponding to the messages sent to p_0. Therefore, p'_0 tolerates can tolerate this kind of faults if it tolerates the previously described one.

A process in a state r or d may fail and enter the state u without the permission of the central scheduling process. This will cause the problem that different processes may use the resource at the same time. This must be avoided by employing redundancy in $cuser$ and channels from the central scheduling processes. Broadcast communication can be employed so that processes can exchange information about whether there is any process using the resource. The agreement must achieved to permit a process to use the resource. Reaching such an agreement in the presence of process failures is the well known *Byzantine General Problem*. We shall discuss this in the next section where a decentralized implementation of $user[]os$ is concerned. Here, we just show that fault-tolerant refinement of $cuser$ is needed to ensure the better failure semantics that no process can use the resource without the permission from the central scheduling process.

Similarly, the fault properties of $cuser[]p_0$ can be discussed for the case when a process fails when it is in a state u or d. Based on the discussions, fault-tolerant strategies for different cases can then be provided.

Faults may also occur in channels. A message in a faulty channel may fail to be delivered to the receiving process, or may be corrupted or duplicated. If a message fails to be delivered, the effect of the failure is the same as if the sending process fails to send a message. A corrupted message is equivalent to the sending process sending a wrong message.

In Section 5.3 and in Subsection 5.4.1, we have provided the full formalization of the refinements of $user[]os$ to $user_1[]os_1$ and $cuser[]p_0$. From the approach used for such a formalization, we can see that the above discussion about the fault-tolerant refinement of $cuser[]p_0$ can be formalized (although such a formalization may not be simple or short).

5.5 A Decentralized Implementation

In this section we show how to decentralize the program $user[]os$ and then consider the fault-tolerant issues of the decentralized program.

5.5.1 Non-fault-tolerant Solution

We introduce a synchronous channel ch_{pq} between each pair p, q of processes in $user$, where p and q may or may not be different. Now let the variable $Procr$, $p.top$, $prior[p, q]$ and $occu$ be local to each process. A process behaves following the rules described below.

1. To request for the resource, a process p which is in state d broadcasts a message $p.request$ to each process, including itself. Therefore, we transform the action of $cuser$ which carries out the transition of p from d to r into the action:

$$\forall\, q : ch_{pq} = <> \rightarrow \|_q :: ch_{pq} := < p.request >; p.state := r$$

2. The processes will synchronously receive this message. We thus add to $cuser$ the action:

$$\forall\, q : (ch_{pq} \neq <> \wedge H(ch_{pq})) \rightarrow \|_q :: Procr, ch_{pq} := Procr \cup \{p\}, <>$$

   where $H(ch_{pq})$ be $true$ iff $ch_{pq} = p.request$.

3. To release the resource, an process p which is in state u broadcasts a message $p.release$ to each process, including itself. We transform the action of $cuser$ which carries out the transition of p from u to d into the action:

$$\forall\, q : ch_{pq} = <> \rightarrow \|_q :: ch_{pq} := < p.release >; p.state := d$$

4. Add to $cuser$ an action so that processes can synchronously receive this message by executing this action:

   $\forall\, q : (ch_{pq} \neq <> \wedge T(ch_{pq})) \rightarrow$
   $\|_q :: Procr, ch_{pq}, occu := Procr - \{p\}, <>, false$

   where $T(ch_{pq})$ is $true$ iff $ch_{pq} = < p.release >$.

5. A process p agrees that a process q can use the resource by sending q a message $p.grant$. So add to $cuser$ the action:

   $\neg occu_p \wedge ch_{pq} \neq <> q.top \rightarrow$
   $ch_{pq} :=< p.grant >; occu := true; \|_r\colon prior[q,r] := r$

6. When a process p receives $q.grant$ from all processes q, then it can enter state u. For this purpose, we transform the action of $cuser$ which carries out the transition of p from r to u into the action:

   $$\forall\, q : (ch_{qp} \neq <> \wedge G(ch_{pq})) \rightarrow \|_q\colon\colon ch_{qp}, :=<>; p.state := u$$

   where $G(ch_{pq})$ is $true$ iff $ch_{pq} =< p.grant >$.

After changing $cuser$ with these rules, the central scheduling process p_0 is no more needed. The program $cuser[]p_0$ is then transformed into a decentralized program $duser$ which is a simulation of $cuser[]p_0$ and thus satisfies the specification $SpecUo$ of $user[]os$.

5.5.2 Faults and Fault-tolerant Solutions

As in the case of centralized implementation, processes may fail and channels may be faulty in the decentralized program $duser$.

To make the synchronization program $duser$ fault-tolerant means to refine it so that the fault-tolerant properties specified by FUO1 – FUO5 are satisfied in the presence of process and communication failures. The refinement must be done based on the specification of $\mathcal{F}(duser)$ which defines the failure semantics of $duser$. Different specifications model different assumptions about the failure semantics. And different failure semantics determines different fault-tolerant refinement solutions.

For a given specification of $\mathcal{F}(duser)$, the problem of finding a fault-tolerant refinement solution to satisfy FU1 – FUO5 is a special case of the problem of *reliable broadcasts* [SG84], or the problem of *reaching agreement in the presence of faults* [PSL80] or the *Byzantine General Problem* [LSP82].

There have been many solutions [PSL80, LFF82, LSP82, Sch82b, SG84, Per85, Per86, Sri86] for this problem under various assumptions about the the faults that may occur. Different solutions and assumptions have been described by using different computational models, in most of which the effects of faults are described in rather informal terms.

Here, we are not going to provide new solutions for this problem. We want to show that, by using the transformational approach developed in the previous chapters, the different solutions and assumptions can be described in the model presented in this thesis. Moreover, each of the solutions can be applied to this framework in a different manner.

Consider a given specification of $\mathcal{F}(duser)$ which is derived from the specifications of $duser$ and the effect of the relevant fault actions. This specification defines the assumptions about the faults that may occur. A solution of the *Byzantine General* problem for the given assumptions can then be used for the program $duser$. Because any solution must behave correctly with respect to the specification of $duser$ if no fault occurs, it is therefore a simulation of $duser$. Also because it is still functioning in the presence of faults, it has to be proved as a fault-tolerant version of $duser$.

Reversing the argument described above is also important. Consider a given solution G of the *Byzantine General* problem which works only for the given assumptions about the fault. We can formalize the assumptions as a specification $SpecAs$ of $\mathcal{F}(duser)$. Assume that we have got a specification $SpecReal$ of $\mathcal{F}(duser)$, which may however be not equivalent to $SpecAs$. If $SpecReal$ is better than $SpecAs$ (i.e. the real failure semantics of $duser$ is better than what is assumed):

$$SpecAs \sqsubseteq SpecReal$$

The use of G for $duser$ with the failure semantics actually defined by $SpecReal$ can guarantee the required fault-tolerant properties.

For example, let $SpecReal$ specify that a process can fail only by stopping but without delivering corrupted messages or performing wrong state transformations. If $SpecAs$ specifies that a process may fail with arbitrary malicious behaviour, then any solution for $SpecAs$ (e.g. [Lam84]) works for $SpecReal$ although it may be very expensive.

If $SpecReal$ is not better than or at least as good as $SpecAs$, G cannot be used for $duser$ to tolerate the faults specified by $SpecReal$. In this case, we may refine $duser$ to another program $duser'$ which is of better failure semantics than $SpecAs$. Or if we know that the lower level system cannot support such a better failure semantics, then if possible, refine the lower level system to ensure this semantics. In each of this two cases, the solution G can be still used for the refined program. If both of the two cases cannot succeed, we can only come to the conclusion that G can only tolerate those actually existing faults which are in the class defined by $SpecAss$. The faults do not actually occur in the execution of $duser$, if they are in the class defined by $SpecAs$ and not in that defined by $SpecReal$.

As an example, we consider the case that $SpecAs$ specifies that a process may fail with arbitrary malicious behaviours, and $SpecReal$ specifies that a process can fail only by stopping but without delivery corrupted messages or performing wrong state transformations. We know that the solutions in [Sch82b, SG84] can tolerate the faults specified by $SpecAs$. But they cannot be used for $SpecReal$.

5.6 Conclusions

The example in this chapter shows that the approach supports formal stepwise development of fault-tolerant reactive systems. The framework provides a new way for understanding fault-tolerant properties. It is feasible to use and helpful to the designers at different level of a whole fault-tolerant system to communicate with each other, and make decisions together.

Chapter 6

Conclusions and Discussion

6.1 Conclusions

Assume that P_0 is a high level program specification which is constructed for a fault-free system and hence taking no account of faults. When physical faults of the system are introduced, the effect of the faults on the execution of P_0 is represented by a program specification F_0. Such a representation is given following the rule described in Section 1.3: never underestimate the damage made by a fault. The behaviour of P_0 in the fault environment represented by F_0 is defined by the failure semantics of P_0 with respect to F_0. The definition of this failure semantics is derived from the semantics of P_0 and the semantics of F_0 which are defined in the same semantic model of action systems. The properties of behaviours of P_0 in the fault environment F_0 are defined by the program specification

$$\mathcal{F}(P_0, F_0) = P_0 \oplus F_0$$

It has been shown that the semantics of $P_0 \oplus F_0$ equals the failure semantics of P_0 with respect to F_0, and there exists a F_0' such that

$$P_0 \oplus F_0 \equiv P_0 [] F_0'$$

Therefore, the properties of the behaviours of P_0 in the fault environment represented by F_0 are reasoned about by applying to the program specification $P_0 \oplus F_0$ the same technique for reasoning about programs in a fault-free environment. In this thesis, sequential programs are reasoned about by using a pre- and postcondition notation and concurrent programs by using a UNITY-like notation also using pre- and postconditions.

The program P_0 is to be transformed by a sequence of correctness preserving transformations, termed as refinements, into an executable program. For each refinement step $P \sqsubseteq P'$ of the original program $P_0 \sqsubseteq \ldots \sqsubseteq P \sqsubseteq P' \sqsubseteq \ldots$, a representation F' of the fault environment is obtained by providing more information about the use of the faulty resources such that

$$\mathcal{F}(P, F) \sqsubseteq \mathcal{F}(P', F')$$

Properties of P are what need to be preserved when executing program P'. This can be satisfied if P' is executed in a fault-free environment. However, the actual execution of P' in the fault environment F' is the same as the execution of $\mathcal{F}(P', F')$ and the following relation does not generally hold:

$$P \sqsubseteq \mathcal{F}(P', F')$$

To make P' tolerate the faults specified by F', it must be changed into a program P'' such that the following refinement is expected to hold:

$$P \sqsubseteq \mathcal{F}(P'', F')$$

This may not always be possible (or necessary) because behaviours of $\mathcal{F}(P'', F')$ include faults and corrections of faults. However, conditions for the corrective actions must be required to guarantee that, after occurrence of faults, a corrective action transforms the error state into a good state from which the execution can resume. These conditions are semantically defined as *consistency*. And based on this definition it is proved that a fault-tolerant program P'' satisfying the conditions described above can be obtained by applying the recovery transformation $\mathcal{R}$ to P':

$$\mathcal{R}(P') = P' [] P'_R$$

Based on the assumption that faults will not affect the execution of the recovery program P'_R, the fault-prone behaviour of $\mathcal{R}(P')$ is then defined by

$$\mathcal{F}(\mathcal{R}(P'), F') = \mathcal{F}(P' [] P'_R, F') = \mathcal{F}(P', F') [] P_R$$

The following refinement relation is then established:

$$\mathcal{F}(\mathcal{R}(P_0), F_0) \sqsubseteq \ldots \sqsubseteq \mathcal{F}(\mathcal{R}(P), F) \sqsubseteq \mathcal{F}(\mathcal{R}(P'), F')$$

This derives the fault-tolerant refinement relation:

$$\mathcal{R}(P_0) \sqsubseteq_{T(F)} \ldots \sqsubseteq_{T(F)} \mathcal{R}(P) \sqsubseteq_{T(F)} \mathcal{R}(P')$$

The fault-tolerant properties of $\mathcal{R}(P')$ for a program P are discussed at a high level in Chapter 2. In Chapter 3, we gave the definition and rules of the fault-tolerant refinement of commands and actions. Further refinement of $\mathcal{R}(P')$ by replacing one or more commands in $\mathcal{R}(P')$ with their fault-tolerant refined versions preserves and enhances the fault-tolerant properties of $\mathcal{R}(P')$. In such a refinement step, checkpoints and recovery methods, such as recovery blocks and conversations and dynamic recovery protocols as well, can be introduced and further refined.

In the informal scheme proposed for fault-tolerance, checkpoints were usually made for the entire state of a process. This was because there was no way to specify exactly what values must be saved. However, using our refinement rules, checkpoints can be made only for those variables that must be saved to guarantee the correctness of the refinement. As shown in Section 4.5, when the extent to which the state has been damaged can be assessed and the cause of the error can be identified, backward recovery can be refined into forward recovery and *vice versa*. The assessment and identification are made based on the specification of the fault-affected version of the program. Similarly, fault-tolerant blocks with alternatives and multiple implementation with majority voting actions can be refined each other.

Thus, a fault-tolerant program can be produced from its fault-free specification by using the transformations described above. The final step of these transformations produces an executable, efficient program which is fault-tolerant on a system with a specified set of faults. If no fault occurs in an execution of the fault-tolerant program which is obtained from nonfault-tolerant program by using the transformations, this execution must satisfy the specification of the nonfault-tolerant program. The fault-tolerant program is thus a refinement of the non-fault-tolerant program: if P' is a fault-tolerant program obtained from program P using the transformations,

$$P \sqsubseteq P'$$

But obviously not every refinement is a fault-tolerant refinement of the nonfault-tolerant program: $P \sqsubseteq P'$ does not imply

$$P \sqsubseteq \mathcal{F}(P')$$

or even

$$\mathcal{F}(\mathcal{R}(P)) \sqsubseteq \mathcal{F}(P')$$

We do not underestimate the difficulty of doing these transformations in practice, but the rules provided here do offer a formal means of verifying what is otherwise left to less precise methods of reasoning. Moreover, like the way in which we have solved the resource allocation problem in Chapter 5, the understanding of the failure semantics of a program and its fault-tolerant versions is also important for development of correct fault-tolerant programs, even if the refinement rules are not formally performed step-by-step.

Fault-tolerant refinement between commands and actions can also be used for improving the failure semantics of a program. In formal terms, replacing a command c in a nonfault-tolerant program P with a fault-tolerant refined version c' of c transforms P into $P[c/c']$. Since $c \sqsubseteq \mathcal{F}(c')$,

$$\mathcal{F}(P) \sqsubseteq \mathcal{F}(P[c/c'])$$

The better failure semantics of $P[c/c']$ make it easier and cheaper to make $P[c/c']$ fault-tolerant than to make P fault-tolerant. Chapter 4 extends fault-tolerant refinement between commands in sequential programs to parallel and reactive programs. This extension provides methods for improving the failure semantics of one part to make the design of the other easier to tolerate the faults in the first part, or to achieve a better failure semantics the composite program. Therefore, the approach developed in this thesis supports stepwise refinement and modular design of parallel and reactive fault-tolerant programs. Discussions and examples in Chapter 4 and Chapter 5 show that this approach, in addition to the way of formalizing the design and development of fault-tolerant programs, provides a new way of using and understanding the existing methods and solutions for solving fault-tolerant properties.

6.2 Related Work

A wide variety of formal methods have been developed for program specification, verification [Flo67, Hoa69, Gri81, OG76] and implementation [Bac78, Lam83, AL88, Mor90]. Programs developed using these methods are not capable of tolerating any physical fault: the correctness of the programs cannot be guaranteed if they run on a fault-prone system.

There is very little published work on formal approach for systematically developing fault-tolerant programs. There has been some work on extending existing formal methods to deal with fault-tolerant programming (such as Joseph, Moitra and Soundararajan's work on proof rules for fault-tolerant distributed programs [JMS87] and Herlihy and Wing's work on atomic objects [HW88]). These have been based on an *informal* description about the effect of faults. And hence, although a program with recovery actions can be described in the formal notation proposed in such work, the fault-tolerant properties are only proved based on an informal discussions about the effect of faults. For instance, the proof of the fault-tolerant properties of the bounded buffer program in [JMS87] was given based on the the assumption that a failure in the buffer process will cause the loss of information and the producer process and the printer process will never fail. Until such descriptions are formalized, the proof cannot be completely formal. Similarly, in the discussion of the fault-tolerant properties of programs in [HW88], a transaction either succeeds completely or has no effect. Within our framework, the effect of faults are formalized and simulated by a set of actions whose semantics are given in the same way as ordinary program actions. Therefore, fault-tolerant properties are precisely described with respect to a class of characterized faults: the properties of the program executions in the specified fault environment.

In Schlichting's thesis [Sch82a], an axiomatic approach has been proposed for verifying fault-tolerant programs. This approach has provided useful guidelines with formal rules for constructing fault-tolerant programs. A program constructed using this approach consists of fault-tolerant actions. The pre- and postcondition assertion $\{Q\}FTA\{R\}$ of a fault-tolerant action FTA are proved using six rules (denoted as $F1$-$F6$). But among these rules, two of them, $F3$ and $F6$, cannot be used in a formal way:

"$F3$: All program variables named in the precondition of the recovery protocol must be in stable storage;

$F6$: Variables in volatile storage may not be named in assertions appearing in programs executing on other processors."

The rules for proving the assertion $\{Q\}FTA\{R\}$ are based on an informal description of the semantics of the fault-tolerant action FTA in a fail-stop processor computational model: when a failure occurs, the fail-stop processor is halted, the internal state and contents of volatile storage are lost and the recovery protocol then is called to complete the state transformation and restore the storage to a well-defined state. Within such a semantic model, applications of the method is limited to backward recovery and each fault-tolerant action FTA corresponds to a retry version of a command presented as a special case in our fault-tolerant refinement in Section 3.7 and Section 4.3.

The idea that a physical fault is modelled as another kind of action (operation) that performs state transformations on the system has been described in [Cri85] through an example. In this example, processor crashes and faults that affect the physical storage medium are taken to be special operations, referred to *fault operations*, and described by axioms similar to those used to describe the semantics of ordinary operations. Combining the axioms about the fault operations and those of the ordinary operations, formal reasoning about the effect of a program suffering these faults becomes possible. It is realized that within the fail-stop processor model, what is left informal in [Sch82a] is formalized in [Cri85]. As we have seen in this thesis, developing this idea into a general method for systematically constructing fault-tolerant programs is far from trivial, especially when concurrent fault-tolerant programs are concerned. We have attempted to approach this by transformations of programs. Such a transformational approach enables us to formally discuss the fault-tolerant properties of a program at different level of abstraction.

6.3 Future Work

Fault-tolerant systems often also have real-time constraints. So is important that the timing properties of a program are refined along with the fault-tolerant and functional properties defined in a program specification. If we extend the model used in this thesis by adding timing properties with respect to some *time domain* [JG88], the recovery transformation and refinement can be defined with timing constraints. The specification and refinement of a fault-tolerant program can then be required to satisfy the condition that after a fault occurs, the system is restored to a consistent state within a time bound which includes the delay caused by the execution of a the recovery action. We thus believe that the method described in this thesis can be extended to take account of timing constraints. However, there are numerous problems still to be examined in making such a method practical, and these are the goals of further work.

For developing fault-tolerant programs and analyzing the reliability of programs, it is sometimes not sufficient to simulate the fault behaviour by a set F of actions. The presentation F of the faults should accompanied by some quantitative probabilistic measure specifying the probability for the occurrence of a fault in F. We would like to work on a probabilistic model and combine it with the model of this framework. Such a combined model should allow the specification and verification of the functional requirements and the relability requirements to be carried out together during the development of a fault-tolerant program. The functional and reliability verification shows that under a specified fault environment with a defined possibility of fault occurrence, the system will behave correctly with respect to the functional specification, with a probability no less than that specified by its reliability specification.

The logic used for reasoning about programs is based on a computational model with a built-in fairness condition, referred to *strong fairness* [MP83, GP89]. The logic itself cannot express such a fairness condition. We mentioned in Section 2.3, that finite and bounded error behaviour assumptions can be defined as fairness conditions [Fra86]. However, we have chosen an alternative way in which the finite and bounded error behaviour assumptions are modelled, at a program level, in the fault environment F. This makes the logic still suitable for dealing with these assumptions. However, it does not mean that it is less important to work on a more general logic in which various fairness conditions can be expressed and, to treat the finite and bounded error behaviour assumptions as fairness conditions. One benefit from this work is that fault-tolerant properties of fault-tolerant programs can be expressed more abstractly at a high level. For instance, assume that the finite error behaviour assumption is defined in a temporal logic TL as a fairness condition FE, than the stable property in Theorem 2.6.4

$P\ \underline{Sat}$ **stable** $Q \Rightarrow$
$P\ \underline{Sat}$ **stable**$((last(front(ob(bound))) = 1) \wedge Q)$

can be expressed as

$$P\ \underline{Sat}\ FE \wedge (\textbf{stable}\ Q) \Rightarrow P\ \underline{Sat}\ \textbf{eventually}\ (\textbf{stable}\ Q)$$

Similarly, we may express and prove the property in Theorem 2.6.5

$$\mathcal{F}(\mathcal{R}(P), F)\ \underline{Sat}\ ((last(front(ob(bound))) = 1) \wedge Q\ \textbf{ensures}\ R)$$

in a way like

$$\mathcal{F}(\mathcal{R}(P), F)\ \underline{Sat}\ \textbf{eventually}\ (Q\ \textbf{ensures}\ R)$$

Therefore, working out or using a more general logic for the approach given in this thesis is another goal of the further work. Lamport's temporal logic of actions (TLA) [Lam90, Lam91] is hopefully a good choice for this research. This should lead to a framework in which the ideas and methods of this thesis can be presented in a way which is less dependent of a programming language.

Bibliography

[AFR80] K. Apt, N. Francez, and W. de Roever. A proof system for communicating sequential processes. *ACM Transactions on Programming Languages and Systems*, 2(3):259–385, July 1980.

[AL81] T. Anderson and P.A. Lee. *Fault-tolerance: Principles and Practice*. Prentice-Hall International, 1981.

[AL88] M. Abadi and L. Lamport. The existence of refinement mapping. In *Proc. 3rd IEEE Sympoium on Logic and Computer Science*, 1988.

[ALS79] T. Anderson, P.A. Lee, and S. K. Shrivastava. System fault tolerance. In T. Anderson and B. Randell, editors, *Computing System Reliabilty*, pages 153–210. Cambridge University Press, 1979.

[Avi76] A. Avižienis. Fault-tolerant systems. *IEEE Transactions on Software Engineering*, C-25(12):1304–1312, December 1976.

[Bac78] R.J.R. Back. On the correctness of refinement in program development, Ph.D. thesis. Technical Report 4, Department of Computer Science, University of Helsinki, 1978.

[Bac80] R.J.R. Back. Correctness preserving program refinements: Proof theory and applications. Technical Report 131, Mathematical Centre, Amsterdam, 1980.

[Bac87] R.J.R. Back. A calculus of refinement for program derivations. Technical Report 54, Abo Akademi, 1987.

[Bac88] R.J.R. Back. Refining atomicity in parallel algorithms. Technical Report 57, Abo Akademi, 1988.

[Bac89] R.J.R. Back. Refinement calculus, Part II: Parallel and reactive programs. Technical Report 93, Abo Akademi, 1989.

[BC81] E. Best and F. Cristian. Systematic detection of exception occurrences. *Science of Computer Programming*, 1(1):115–144, 1981.

[BK83] R.J.R. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. In *Second Annual ACM Symposium on Principles of Distributed Computing*, pages 131–142, 1983.

[BS88] R.J.R. Back and K. Sere. Stepwise refinement of parallel algorithms. Technical Report 64, Abo Akademi, 1988.

[BS89] R.J.R. Back and K. Sere. Stepwise refinement of action systems. Technical Report 78, Abo Akademi, 1989.

[BW89] R.J.R. Back and J. von Wright. Refinement calculus, Part I: Sequential nondeterministic programs. Technical Report 92, Abo Akademi, 1989.

[CAR83] R.H. Campbell, T. Anderson, and B. Randell. Practical fault tolerant software for asynchronous systems. In *Proceedings of SAFECOM'83*, Cambridge, 1983.

[CL85] K.M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley Publishing Company, 1988.

[CR86] R.H. Campbell and B. Randell. Error recovery in asynchronous systems. *IEEE Transactions on Software Engineering*, SE-12(8):811–826, August 1986.

[Cri85] F. Cristian. A rigorous approach to fault tolerant programming. *IEEE Transactions on Software Engineering*, SE-11(1):23–31, January 1985.

[Cri90] F. Cristian. Understanding fault-tolerant distributed systems. Technical Report RJ 6980 (66517), IBM Research Division, Almaden Research Center, June 1990.

[Dij76] E.W. Dijkstra. *A Discipline of Programming*. Englewood Cliffs, New Jersey: Prentice-Hall, 1976.

[Dij85] E.W. Dijkstra. The distributed snapshot of K.M. Chandy and L. Lamport. In M. Broy, editor, *Control Flow and Data Flow*, pages 513–517. Berlin: Springer-Verlag, 1985.

[FF73] M. Fischler and O. Firschein. A fault tolerant multiprocessor architecture for real-time control applications. In *Proceedings of the First Annual Symposium on Computer Architecture*, pages 151–157, 1973.

[FGL82] M.J. Fischer, N. D. Griffeth, and N. A. Lynch. Global states of a distributed system. *IEEE Transactions on Software Engineering*, SE-8(3):198–202, May 1982.

[Flo67] R.W. Floyd. Assigning meanings to programs. In J.T. Schwartz, editor, *Mathematical Aspects of Computer Science, Proceedings of the Symposia in Applied Mathematics 19*, pages 19–32, Providence, 1967. American Mathematical Society.

[Fra86] N. Francez. *Fairness*. Springer-Verlag, New York, 1986.

[GP89] R. Gerth and A. Pnueli. Rooting UNITY. *In Proceedings of the 5th IEEE International Workshop on Software Specification and Design*, February 1989.

[Gra78] J.N. Gray. Notes on database operating systems. In *Lecture Notes in Computer Science 60*, pages 393–481. Springer-Verlag, 1978.

[Gri81] D. Gries. *The Science of Programming*. Springer-Verlag, New York Heidelberg Berlin, 1981.

[Had82] V. Hadzilacos. An algorithm for minimizing rollback cost. In *Proceedings of ACM Symposium on Principles of Database Systems.*, March 1982.

[HH87] J. He and C.A.R. Hoare. Algebraic specification and proof of a distributed recovery algorithm. *Distributed Computing*, 2:1–12, 1987.

[HLMR74] J.J. Horning, H.H. Lauer, P.M. Melliar-Smith, and B. Randell. A program structure for error detection and recovery. In *Proceedings of Conference on Operating Systems: Theoretical and Practical Aspects.*, pages 177–193, IRIA, April, 23-25 1974.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, October 1969.

[HW87] M.P. Herlihy and J.M. Wing. Reasoning about atomic objects. Technical Report 87-176, Computer Science Department, University of Carnegie Mellon, November 1987.

[HW88] M.P. Herlihy and J.M. Wing. Reasoning about atomic objects. In *Lecture Notes in Computer Science 331*, pages 193–208. Springer-Verlag, 1988.

[JC86] P. Jalote and R.H. Campbell. Atomic actions for fault-tolerance using CSP. *IEEE Transactions on Software Engineering*, SE-12(1):59–68, January 1986.

[JG88] M. Joseph and A. Goswami. What's 'real' about real-time systems? In *Proceedings of IEEE Real-time Systems Symposium*, pages 78–85, Huntsville, Alabama, December 1988.

[JM83] M. Joseph and A. Moitra. Cooperative recovery from faults in distributed programs. In R. W. Mason, editor, *Information Processing 83*, pages 481–486. North-Holland, 1983.

[JMS87] M. Joseph, A. Moitra, and N. Soundararajan. Proof rules for fault tolerant distributed programs. *Science of Computer Programming*, 8(1):43–67, February 1987.

[Kim82] K.H. Kim. Approach to mechanization of the conversation scheme based on monitors. *IEEE Transactions on Software Engineering*, SE-8(3):189–197, May 1982.

[KT87] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, January 1987.

[Lam83] L. Lamport. Reasoning about nonatomic operations. In *Proc. 10th ACM Conference on Principles of Programming Languages*, pages 28–37, 1983.

[Lam84] L. Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions on Programming Languages and Systems*, 6(2):254–280, April 1984.

[Lam90] L. Lamport. A temporal logic of actions. Technical report, Digital SRC, California, April 1990.

[Lam91] L. Lamport. The temporal logic of actions. Digital SRC, California, Januray 1991.

[LFF82] N.A. Lynch, M.J. Fischer, and R. Fowler. A simple and efficient Byzantine General algorithm. In *Proceedings of 2nd Symposium on Reliability in Distributed Software and Database Systems*, Pittsburgh, July 1982.

[LG81] G. Levin and D. Gries. Proof techniques for communicating sequential processes. *Acta Informatica*, 15:281–302, 1981.

[Lom77] D. Lomet. Process structuring, synchronization and recovery using atomic actions. In *Proceedings of the ACM Conference on Language Design for Reliable Software. Sigplan Notices 12*, pages 128–137, 1977.

[LRT78] P.A. Lee, B. Randell, and P.C. Treleaven. Reliable computing systems. In *Lecture Notes in Computer Science 60*, pages 289–391. Springer-Verlag, 1978.

[LSP82] L. Lamport, R. Shostak, and M. Pease. The Byzantine General problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[Min91a] Ministry of Defence, Glasgow, U.K. *Defence Standard: The Procurement of Safety Critical Software in Defence Equipment, PART 1: Requirements. 00-55(PART 1)/ Issue 1*, April 1991.

[Min91b] Ministry of Defence, Glasgow, U.K. *Defence Standard: The Procurement of Safety Critical Software in Defence Equipment, PART 2: Guidance. 00-55(PART 2)/ Issue 1*, April 1991.

[Min91c] Ministry of Defence, Glasgow, U.K. *Hazard Analysis and Safety Classification of the Computer and Programmable Electronic System Elements of Defence Equipment. 00-56 / Issue 1*, April 1991.

[Mor87] J.M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Science of Computer Programming*, 9:278–306, September 1987.

[Mor90] C. Morgan. *Programming from Specification*. Prentice Hall, 1990.

[MP83] Z. Manna and A. Pnueli. How to cook a temporal proof system for your pet language. In *Proceedings of 10th Annual ACM Symposium on Principles of Programming Languages*, Austin, Texas, 1983.

[Neu56] J. von Neumann. Probabilistic logics and the synthesis of reliable organisms from unreliable components. In C. E. Shannon and J. Macarthy, editors, *In Automata Studies*, pages 43–98, Princeton University Press, Princeton, 1956.

[OG76] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, July 1976.

[Per85] K.J. Perry. *Early Stopping Protocols for Fault-tolerant Distributed Agreement*. PhD thesis, Department of Computer Science, Cornell University, February 1985.

[Per86] K.J. Perry. Distributed agreement in the presence of processor and communication faults. *IEEE Transactions on Software Engineering*, SE-12(3):477–482, 1986.

[PSL80] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[Ran75] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220–232, June 1975.

[Ree83] D.P. Reed. Implementing atomic actions on decentralized data. *ACM Transactions on Computer Systems*, 1(1):3–23, February 1983.

[RLT78] B. Randell, P.A. Lee, and P.C. Treleaven. Reliability issues in computing systems design. *Computing Survey*, 10(2):123–165, 1978.

[RT79] D.L. Russell and M.J. Tiedeman. Multiprocess recovery using conversations. In *Digest of papers FTCS-9: The 9th Annual International Symposium on Fault-tolerant Computing*, pages 106–109, Madison, WI, 1979.

[Rus80] D.L. Russell. State restoration in systems of communication processes. *IEEE Transactions on Software Engineering*, SE-6(2):183–194, March 1980.

[Sch82a] R.D. Schlichting. *Axiomatic Verification to Enhance Software Reliability*. PhD thesis, Department of Computer Science, Cornell University, January 1982.

[Sch82b] F.B. Schneider. Fault-tolerant broadcasts. *ACM Transactions on Programming Languages and Systems*, 4(2):125–148, April 1982.

[SG84] F.B. Schneider and D. Gries. Fault-tolerant broadcasts. *Science of Computer Programming*, 4:1–15, 1984.

[SMR87] S.K. Shrivastava, L.V. Mancini, and B. Randell. On the duality of fault tolerant system structures. In *Lecture Notes in Computer Science 309*, pages 19–37. Springer-Verlag, 1987.

[Sri86] T.K. Srikanth. *Designing Fault-tolerant Algorithms for Distributed Systems Using Communication Primitives*. PhD thesis, The Faculty of the Graduate School, Cornell University, January 1986.

[SS83] R.D. Schlichting and F.B. Schneider. Fail-stop processes: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, 1983.

[Tay86] D.J. Taylor. Concurrency and forward recovery in atomic actions. *IEEE Transactions on Software Engineering*, SE-12(1):69–78, January 1986.

[TS84] Y. Tamir and C.H. Sequin. Error recovery in multicomputers using global checkpoints. In *Proceedings of 13th International Conference on Parallel Processing*, August 1984.

[Woo81] W.G. Wood. A decentralized recovery control protocol. In *Proceedings of the 11th Annual International Symposium on Fault-Tolerant Computing*, June 1981.