THE UNIVERSITY OF
WARWICK

Original citation:
Liu, Z. and Joseph, M. (1990) Transformation of programs for fault-tolerance. University of Warwick. Department of Computer Science. (Department of Computer Science Research Report). (Unpublished) CS-RR-165

Permanent WRAP url:
http://wrap.warwick.ac.uk/60860

Copyright and reuse:
The Warwick Research Archive Portal (WRAP) makes this work by researchers of the University of Warwick available open access under the following conditions. Copyright © and all moral rights to the version of the paper presented here belong to the individual author(s) and/or other copyright owners. To the extent reasonable and practicable the material made available in WRAP has been checked for eligibility before being made available.

Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge. Provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way.

A note on versions:
The version presented in WRAP is the published version or, version of record, and may be cited as it appears here.For more information, please contact the WRAP Team at:
publications@warwick.ac.uk

http://wrap.warwick.ac.uk/

Research report 165

TRANSFORMATION OF PROGRAMS FOR FAULT TOLERANCE

ZHIMING LIU, MATHAI JOSEPH
(RR165)

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 paper we shall describe how a program constructed for a *fault-free* system can be transformed into a *fault-tolerant* program for execution on system which susceptible to failures. We assume that the interference by a faulty environment F on the execution of a program P can be described as a *fault-transformation* F which transforms P into a program $F(P) = P \oplus F$. A recovery transformation R transforms P into a program $R(P) = P[]R$ by adding a set of *recovery actions* R, called a *recovery program.* If the system is *fail stop* and faults do not affect recovey actions, we have

$$F\ (R(P)) = F(P)[]R = (P \oplus F)[]R$$

We illustrate this approach to fault-tolerant programming by considering the problem of designing a protocol that guarantees reliable communication from a sender to a receiver in spite of faults in the communication channel between them.

Department of Computer Science
University of Warwick
Coventry CV4 7AL
United Kingdom

October 1990

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,BvW89,Bac89]. It has been usual to consider that the steps of refinement end with the production of the text of an executable program. But in many cases the program must be transformed to take account of the features (or limitations) of the particular architecture on which it is to be executed. In this paper we shall describe how such transformations can be performed for a program which is to be executed on a system which is susceptible to failures.

Our objective is to use transformations to construct a *fault-tolerant* program from a program constructed for a *fault-free* system. Let P be a program satisfying the specification Sp_P. Let the effect of each physical fault in the system on which P is executed be described as a *fault action* which transforms a *good* program state into an *error* state which violates Sp_P. Physical faults can be then modelled as the actions of a *fault program* F which interferes with the execution of P. A *failure* at any point during the execution of P takes it into an error state in which a boolean variable f is *true*. (F is assumed not to change an error state into a good state.)

In general a high level specification of a program is not sufficient to specify its behaviour in the presence of system faults or to transform it into a fault-tolerant program. It is also necessary to describe the hardware organisation of the system on which the program is to be executed, on its use of the resources of the system and the nature of the possible faults in the system, e.g. which processors and channels my fail, as all of these factors can affect the execution of the program. And very little can be said about the effects of a system fault on a program until it has been refined to the level where these effects can be observed.

After an informal overview of the method used for achieving fault-tolerance, in Section 2 a simple model for representing programs (specifications) is defined both syntactically and semantically. Based on the semantics of failure *w.r.t* a given failure-prone environment, the effect of faults on the original program is defined in terms of a program transformation in section 3. Section 4 provides an abstract definition of consistency which is used to define recovery transformations which permit both backward and forward recovery. Section 5 shows that the fault-tolerant properties of a program can be refined along with the other properties defined in a program specification. Finally, in Section 6 an example is given to illustrate fault-tolerant programming using the method described in this paper; 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.

2 A Simple Model and Specification Language

2.1 Background

The interference by a faulty environment F on the execution of program P can be described as a *fault-transformation* $\mathcal{F}$ which transforms P into a program $\mathcal{F}(P) = P \oplus F$. Assume that this transformation ensures that the execution of P under F is *fail stop* [SS83], i.e. that no further actions of P will be executed when a failure occurs. The execution of P in the faulty environment F is equivalent to the execution of $\mathcal{F}(P)$ in a *fault-free* environment.

The behaviour of $\mathcal{F}(P)$ will not usually satisfy Sp_P. To make the program fault-tolerant, P must be transformed by a *fault-tolerant transformation* into a program $\mathcal{T}(P)$ such that $\mathcal{F}(\mathcal{T}(P))$ is expected to satisfy the specification of P. Unfortunately, this may not always be possible 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*, based on a definition of *consistency*.

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 a failure. Let S be a state with is the union of substates of $A_m, \ldots, S_n$. If the execution of P can be restarted from state S and still satisfy Sp_P, then the state S is said to be *backward consistent* with the interrupted execution sequence. Alternatively, the execution of P can be continue from a *possible future* state S_{n+k}. If there exists an execution sequence $[S_n]A_{n+1}[S_{n+1}] \ldots A_{n+k}[S_{n+k}]$ of P which satisfies Sp_P, then S_{n+k} is said to be *forward consistent* with the interrupted execution sequence. A *consistent* state is a state which is either backward or forward consistent.

The recovery transformation $\mathcal{R}$ transforms a program P in to a program $\mathcal{R}(P) = P[]R$ by adding a set of *recovery actions* R, called a *recovery program*. The recovery actions of R are enabled only when a failure occurs and transform an error state into a good state which is consistent with the execution sequence interrupted by the fault. We assume that the recovery actions are not affected by the faulty environment F, i.e. that no failure occurs during the execution of a recovery action. Therefore, we have

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

Let $P_0 \sqsubseteq \ldots \sqsubseteq P_k$ be a sequence of program specifications such that P_{i+1} refines P_i and P_k contains enough information for specifying the fault environment F. From P_k we may be able to determine the number of processes in the program, those which may fail during execution, and the channel variables which are faulty. Based on F and the fault-transformation $\mathcal{F}$ on P_k, we can achieve the recovery transformation $\mathcal{R}$ on P_k and the *fault-tolerant program* $P_k[]R$. For example, fault-tolerant mechanisms such as *checkpointing*, *recovery blocks* and *conversations* [BR81,Ran75,MR78,KT87] can be then introduced by applying stepwise refinement to $P_k[]R$.

2.2 Commands and Actions

A *program* P is described as an *action system* 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 $= g \rightarrow c$, 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 $true \rightarrow c$ to c if there is no confusion.

A *command* c is defined as follows,

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

Here $\bar{x}_i$ and $\overline{x'}_i$ are lists of variables, $\bar{e}_i$ is a list of expressions, b is a boolean condition and Q is a condition over the values of program variables [BKS83].

The effect of the nondeterministic assignment command is to assign to the list of variables $\bar{x}_i$ some value(s) satisfying condition Q. This may introduce unbounded nondeterminism.

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: 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. Thus a program P can be partitioned into n processes by refinement mapping [CM88,Bac88,Bac89] and described as

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

As in Hoare logic, the specification $\{Q\}A\{R\}$ defines an action A which when executed in a state satisfying predicate Q terminates in a state satisfying 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 logical operators **unless, stable, invariant, ensures, leads-to** ($\longmapsto$) are used to describe the safety and progress properties of programs. As in Unity [CM88], these operators are defined for a program P as:

- $P \underline{Sat}\ (Q \textbf{ unless } R) \equiv \forall A : A \in Act_G :: \{Q \wedge \neg R\}A\{Q \vee R\}$
- $P \underline{Sat}\ (\textbf{stable } Q) \equiv G \underline{Sat}\ (Q \textbf{ unless } \textit{false})$
- $P \underline{Sat}\ (\textbf{invariant } Q) \equiv (Init_P \Rightarrow Q) \wedge P \underline{Sat}\ (\textbf{stable } Q)$
- $P \underline{Sat}\ (Q \textbf{ ensures } R) \equiv P \underline{Sat}\ (Q \textbf{ unless } R) \wedge (\exists A : A \in Act_P :: \{Q \wedge \neg R\}A\{R\})$

- $\dfrac{P\ \underline{Sat}\ (Q\ \textbf{ensures}\ R)}{P\ \underline{Sat}\ (Q \longmapsto R)}$, $\qquad \dfrac{P\ \underline{Sat}\ (Q \longmapsto R),\ \ P\ \underline{Sat}\ R \longmapsto R')}{P\ \underline{Sat}\ (Q \longmapsto R')}$,

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 program specification can be written by using either UNITY-like logic or an action system formalism. However, we will normally use the logic for the top level specification and the action system formalism for the refinement.

2.3 Semantics of Commands and Actions

We use a simple semantics for programs in which each action in a program is executed atomically. 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.

A *state* S of a program P is a function from the program variables $Var(P)$ to their value space. 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. The semantics of a command c is a function:

$$\Gamma(c) :\ \Psi_P \cup \{\perp, \textbf{abort}\} \rightarrow \mathbf{P}(\Psi_P \cup \{\perp, \textbf{abort}\})$$

where $\perp$ stands for nontermination or divergence and **abort** is a state such that $\{false\}c\{\textbf{abort}\}$ for any command c. This function can be extended to be a function over the powerset $\mathbf{P}(\Psi_P \cup \{\perp, \textbf{abort}\})$, i.e. for a set Ψ of states

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

We define $\Gamma(c)(\perp) = \{\perp\}$ and $\Gamma(c)(\textbf{abort}) = \{\textbf{abort}\}$ where $\perp$ and **abort** are treated as a states such that $\perp(x) = \perp$ and $\textbf{abort}(x) = \textbf{abort}$ for any variable $x \in Var(P)$.

Let **skip** be the identity function which leaves any state unchanged, i.e. $\textbf{skip}(S) = \{S\}$, for any state S. The semantics Γ of the execution of a command c in a state S is defined in the following way:

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

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. If c is the conditional composition **if** $A_1 [] \ldots [] A_n$ **fi**, $A_i = g_i \rightarrow c_i$ for $i = 1, \ldots, n$ and $gg = g_1 \vee \ldots \vee g_n$, then

$$\Gamma(c)(S) = \begin{cases} \bigcup_{g_i(S)=true} \Gamma(c_i)(S) & \text{if } gg(S) = true \\ \{\textbf{abort}\} & \text{if } gg(S) = false \end{cases}$$

5. For the iterative composition $c = \textbf{do } A_1 [] \ldots []A_n \textbf{ od}$, $A_i = g_i \rightarrow c_i$ for $i = 1, \ldots, n$, we first define

$$\Gamma(A_1 [] \ldots []A_n)(S) = \bigcup_{g_i(S)=true} \Gamma(c_i)(S)$$

Let Y be a set of states and $\Gamma F(S)$ be the least fixed point of the equation

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

$\Gamma(c)(S)$ is then

$$\Gamma(c)(S) = \begin{cases} \Gamma F(S) \cap M & \text{if } \Gamma F(S) \cap M \neq \\ \{\perp\} & \text{if } \Gamma F(S) \cap M = \end{cases}$$

where $M = \{S \mid \forall i : 1 \leq i \leq n : g_i(S) = false\} \cup \{\perp\}$

The semantics $\Gamma(A)$ of each action $A = g \rightarrow c$is defined by

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

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

Let V be the value space of program P and $V^\dagger$ be the set of all the finite and infinite sequences[1] over $V \cup \{\perp\} \cup \{\textbf{abort}\}$. An *observation* θ of program 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 a initial state of P,

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

[1] $< a, \ldots, b >$ is the sequence of elements $a, \ldots, b$; $<>$ is the empty sequence and $\wedge$ concatenates two sequences according to the standard definition: e.g. $< a > \wedge < b > = < a, b >$, $< a > \wedge <> = < a >$, etc.; $\sigma' \underline{\propto} \sigma$ denotes that σ' is a prefix of σ; $\sigma' \propto \sigma$ denotes that σ' is a proper prefix of σ; $\#\sigma$ is the length of the sequence σ and $\#\sigma = \infty$ if σ is infinite; and $\sigma(i-1)$ is the ith element of σ, $1 \leq i \leq \#\sigma$; $head(\sigma)$ and $last(\sigma)$ denote respectively the first and last elements of a non-empty sequence σ; $tail(\sigma)$ denotes the sequence obtained from σ by removing the first element of σ.

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, i.e. $\beta \preceq \theta$, if there is an observation θ' and a function $\gamma : Var(P) \longrightarrow V^\dagger$ satisfying Condition **OB2** such that

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

In this case, we write $pred_\theta(\beta) = \theta'$.

Obviously, given a command c and an action A, $\Gamma(c)$ and $\Gamma(A)$ can be extended to be functions over the set OB_P of the observations of P, e.g. $\Gamma(c)(\theta) = \{\theta\hat{}S \mid S \in \Gamma(c)(last(\theta))\}$.

The semantics of a program P is the set $\Gamma(P)$ of its executions. 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 $E(k+1) \in \Gamma(A)(E(k))$. Programs P and P' are said to be *equivalent* 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:

$$(P \ \underline{Sat}\ \textbf{stable}\ Init_P) \Rightarrow (P \equiv \textbf{skip})$$

and,

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

Therefore, this semantic model permits *finite stuttering*, i.e.in an execution, a state can be repeated consecutively at most a finite number of times before the execution terminates [Lam83,AL88,Bac89].

3 Transformations for Specified Faults

Given a program P, the effect of faults on program G is modelled by a program F which defines a set of atomic actions representing the faulty environment. The execution of the program P under the faulty environment specified by F is equivalent to the execution of P together with F on the fault-free system. Such a *failure execution* of P is defined by the *failure semantics*.

3.1 Failure Semantics

For a program P and a faulty 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. Each action $A \in F$ is called a *fault action* and is assumed to satisfy

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

A state S is *good* if $f(S) = \mathit{false}$ and an *error state* is a state which is not good.

Let a primitive command be a single assignemnt or a parallel composition of assignments. If c is a command of P, then the *failure semantics* of c *w.r.t* F is a function

$$\Gamma_F(c) : \Psi_P \cup \{\bot, \mathbf{abort}\} \to \mathrm{P}(\Psi_P \cup \{\bot, \mathbf{abort}\})$$

which satisfies the following conditions:

1. if c is an assignment command, then

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

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

3. if c is the conditional composition **if** $A_1 [] \ldots []A_n$ **fi**, $A_i = g_i \to c_i$, then

$$\Gamma_F(c)(S) = \begin{cases} \bigcup_{g_i(S)=\mathit{true}} \Gamma_F(c_i)(S) & \text{if } gg \wedge \neg f(S) = \mathit{true} \\ \{\mathbf{abort}\} & \text{if } \neg gg \wedge \neg f(S) = \mathit{true} \\ \{S\} & \text{if } f(S) = \mathit{true} \end{cases}$$

where $gg = g_1 \vee \ldots \vee g_n$.

4. for the iterative composition $c =$ **do** $A_1 [] \ldots []A_n$ **od**, $A_i = g_i \to c_i$, we first define

$$\Gamma_F(A_1 [] \ldots []A_n)(S) = \begin{cases} \bigcup_{g_i(S)=\mathit{true}} \Gamma_F(c_i)(S) & \text{if } f(S) = \mathit{false} \\ & \text{if } f(S) = \mathit{true} \end{cases}$$

Let $E(S)$ be the least fixed point of the equation

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

Then $\Gamma_F(c)(S)$ is defined as

$$\Gamma_F(c)(S) = \begin{cases} E(S) \cap M & \text{if } E(S) \cap M \neq \\ \{\bot\} & \text{if } E(S) \cap M = \end{cases}$$

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

For each action $A = g \to c$, $\Gamma_F(A)$ is defined by

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

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

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 a state of P defined as $\theta[i](x) = \theta(x)(i)$ for any $x \in Var(P)$,

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

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

A *failure execution* E of a program P *w.r.t.* F is an infinite failure observation of P. The *failure semantics* of the program P *w.r.t.* to F is the set $\Gamma_F(P)$ of the failure executions of P *w.r.t.* F.

From this definition we can see that the execution of program G is fail-stop. Therefore, the failure semantics of the program G can be described in terms of two functions *good* and *error* such that each failure observation θ *w.r.t.* to F can be written as:

$$\theta = good(\theta)\hat{}error(\theta)$$

where $good(\theta)$ is an observation of G and $error(\theta)$ is either empty or contains only error states. Obviously, if there are no faults, i.e. F is empty, each failure execution is the same as some execution of P:

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

Programs P and P' is 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.

3.2 Fault Transformation

Given a program $P = A_1[] \ldots []A_m$ and a faulty environment F, let $f = true$ if a fault occurs in the execution of P. First, transform P into a program $FS(P)$ such that

$$FS(P) = FS(A_1)[] \ldots []FS(A_m)$$

and each $FS(A_i)$ is obtained from A_i by changing every primitive command c which occurs in A_i into the command

$$FS(c) = \mathbf{if}\ \neg f \rightarrow c[]f \rightarrow \mathbf{skip}\ \mathbf{fi}$$

$FS(c)$ is the *primitive f-stop command* of c if c is a primitive command. For a command c of P, $FS(c)$ is a *f-stop command* which is obtained from c by changing every primitive command c' which occurs in c into the primitive f-stop command $FS(c')$. Given an action of P

$$A = g \rightarrow c$$

the *f-stop action* of A is

$$FS(A) = \neg f \wedge g \rightarrow FS(c)$$

The execution of $FS(P)$ on a system with the faults F is therefore fail-stop. Obviously, if f is invariantly *false*, $FS(P) \equiv P$ since P does not change the value of f.

For each f-stop command $FS(c)$ and each f-stop action $FS(A)$, a transformation $\mathcal{M}$ is defined as:

1. if c is a primitive command,

$$\mathcal{M}(FS(c)) = \textbf{if } \neg f \rightarrow c [] f \rightarrow \textbf{skip} [] F \textbf{ fi}$$

2. if $c = c_1;\ c_2$,

$$\begin{aligned}\mathcal{M}(FS(c)) &= \mathcal{M}(FS(c_1));\ \mathcal{M}(FS(c_2)) \\ &\equiv \textbf{if } (FS(c_1);\ FS(c_2) [] F [] FS(c_1);\ (\textbf{if } f \rightarrow \textbf{skip} [] F \textbf{ fi}) \textbf{ fi}\end{aligned}$$

3. for an action $A = g \rightarrow c$

$$\mathcal{M}(FS(A)) = \textbf{if } \neg f \wedge g \rightarrow \mathcal{M}(FS(c)) [] f \rightarrow \textbf{skip fi}$$

4. for a command $c = \textbf{if } A_1 [] \ldots [] A_n \textbf{ fi}$,

$$\mathcal{M}(FS(c)) = \textbf{if } \mathcal{M}(FS(A_1)) [] \ldots [] \mathcal{M}(FS(A_n)) \textbf{ fi}$$

5. if $c = \textbf{do } A_1 [] \ldots [] A_n \textbf{ od}$,

$$\mathcal{M}(FS(c)) = \textbf{do } \mathcal{M}(FS(A_1)) [] \ldots [] \mathcal{M}(FS(A_n)) \textbf{ od}$$

For the program $P = A_1 [] \ldots [] A_m$, we define

$$\mathcal{M}(FS(P)) = \mathcal{M}(FS(A_1)) [] \ldots [] \mathcal{M}(FS(A_m))$$

Given the faults F, the *fault transformation* is defined as:

$$\mathcal{F}(P) \triangleq \mathcal{M}(FS(P))$$

and $\mathcal{F}(P)$ is called the *fault affected program* of program P and denoted as $P \oplus F$.

Theorem 3.1 *Given a program P and its faulty environment F, the fault-transformation $\mathcal{F}$ satisfies*

$$\Gamma(P \oplus F) = \Gamma_F(P)$$

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

$$\Gamma_F(A)(S) = \Gamma(\mathcal{M}(FS(A)))(S)$$

Case 1 : if c is a primitive command, then

$$\begin{aligned}
\Gamma_F(A)(S) &= \begin{cases} \Gamma_F(c)(S) & \text{if } \neg f \wedge g(S) = \mathit{true} \\ \mathbf{skip}(S) & \text{otherwise} \end{cases} \\
&= \begin{cases} \bigcup_{a \in F} \Gamma(a)(S) & \text{if } f(S) = \mathit{true} \\ \bigcup_{a \in F} \Gamma(a)(S) \cup \Gamma(c)(S) & \text{if } \neg f \wedge g(S) = \mathit{true} \\ \mathbf{skip}(S) & \text{if } g(S) = \mathit{false} \end{cases} \\
&= \begin{cases} \Gamma(F)(S) & \text{if } f(S) = \mathit{true} \\ \Gamma(c)(S) \cup \Gamma(F)(S) & \text{if } \neg f \wedge g(S) = \mathit{true} \\ \mathbf{skip}(S) & \text{if } g(S) = \mathit{false} \end{cases} \\
&= \Gamma(\neg f \wedge g \rightarrow \mathbf{if}\ \neg f \rightarrow c [] f \rightarrow \mathbf{skip}\ \mathbf{fi})(S) \\
&= \Gamma(\mathcal{M}(FS(A)))(S)
\end{aligned}$$

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

$$\begin{aligned}
\Gamma_F(A)(S) &= \begin{cases} \Gamma_F(c_1)(\Gamma_F(c_2)(S)) & \text{if } \neg f \wedge g(S) = \mathit{true} \\ \mathbf{skip}(S) & \text{otherwise} \end{cases} \\
&= \begin{cases} \Gamma(\mathcal{M}(FS(c_1)))(\Gamma(\mathcal{M}(FS(c_2))(S)) & \text{if } \neg f \wedge g(S) = \mathit{true} \\ \mathbf{skip}(S) & \text{otherwise} \end{cases} \\
&= \begin{cases} \Gamma(\mathcal{M}(FS(c_1)); \mathcal{M}(FS(c_2)))(S) & \text{if } \neg f \wedge g(S) = \mathit{true} \\ \mathbf{skip}(S) & \text{otherwise} \end{cases} \\
&= \begin{cases} \Gamma(\mathcal{M}(FS(c_1; c_2))(S) & \text{if } \neg f \wedge g(S) = \mathit{true} \\ \mathbf{skip}(S) & \text{otherwise} \end{cases} \\
&= \Gamma(\neg f \wedge g \rightarrow \mathcal{M}(FS(c_1; c_2)))(S) \\
&= \Gamma(\mathcal{M}(FS(A)))(S)
\end{aligned}$$

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

$$\begin{aligned}
\Gamma_F(A)(S) &= \begin{cases} \bigcup_{g_i(S) = \mathit{true}} \Gamma_F(c_i)(S) & \text{if } \neg f \wedge g \wedge gg(S) = \mathit{true} \\ \{\mathbf{abort}\} & \text{if } \neg gg \wedge g \wedge \neg f(S) = \mathit{true} \\ \{S\} & \text{if } \neg f \wedge g(S) = \mathit{false} \end{cases} \\
&= \Gamma(\neg f \wedge g \rightarrow \mathbf{if}\ \mathcal{M}(FS(A_1)) [] \ldots [] \mathcal{M}(A_n) [] f \rightarrow \mathbf{skip}\ \mathbf{fi})(S) \\
&= \Gamma(\mathcal{M}(FS(A)))(S)
\end{aligned}$$

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

□

Corollary 1 *Given a program P and its faults F,*

$$F = \Phi \Rightarrow P \oplus F \equiv P$$

Therefore, a fault-free execution of the program $P \oplus F$ is an execution of program P and vice-versa, and $P \oplus F$ is equivalent to P if no fault occurs. Further, as shown in the following theorem, $P \oplus F$ is equivalent to the union composition of the f-stop program $FS(P)$ with a program F'.

Theorem 3.2 *Given P and F, there is a program F' such that*

$$P \oplus F \equiv FS(P)[]F'$$

Proof: Let $P = A_1[] \ldots []A_n$, $A_i = g_i \rightarrow c_i$ we prove that for each A_i,

$$\mathcal{M}(FS(A_i)) \equiv \textbf{if } FS(A_i)[]\neg f \wedge g_i \rightarrow F_i[]f \rightarrow \textbf{skip fi}$$

Case 1 : if c_i is a primitive command,

$$\begin{aligned}
\mathcal{M}(FS(A_i)) &= \textbf{if } \neg f \wedge g_i \rightarrow \mathcal{M}(FS(c_i))[]f \rightarrow \textbf{skip fi} \\
&= \textbf{if } \neg f \wedge g_i \rightarrow \textbf{if } \neg f \rightarrow c[]f \rightarrow \textbf{skip}[]F \textbf{ fi} \\
&\equiv \textbf{if } \neg f \wedge g_i \rightarrow \textbf{if } \neg f \rightarrow c[]f \rightarrow \textbf{skip fi } []\neg f \wedge g_i \rightarrow F[]f \rightarrow \textbf{skip fi} \\
&= \textbf{if } FS(A_i)[]\neg f \wedge g_i \rightarrow F[]f \rightarrow \textbf{skip fi}
\end{aligned}$$

Case 2 : if $c_i = c; c'$, then

$$\begin{aligned}
\mathcal{M}(FS(A_i)) &= \textbf{if } \neg f \wedge g_i \rightarrow \mathcal{M}(FS(c; c'))[]f \rightarrow \textbf{skip fi} \\
&\equiv \textbf{if } \neg f \wedge g_i \rightarrow (\textbf{if } FS(c);\ FS(c')[]F[]FS(c);\ (\textbf{if } f \rightarrow \textbf{skip}[]F \textbf{ fi}) \textbf{ fi})[]f \rightarrow \textbf{skip fi} \\
&\equiv \textbf{if } f \rightarrow \textbf{skip}[]\neg f \wedge g_i \rightarrow FS(c); FS(c') \\
&\quad []\neg f \wedge g_i \rightarrow (\textbf{if } F[]FS(c);\ (\textbf{if } f \rightarrow \textbf{skip}[]F \textbf{ fi})\textbf{fi})\textbf{fi} \\
&= \textbf{if } FS(A_i)[]\neg f \wedge g_i \rightarrow (\textbf{if } F[]FS(c);\ (\textbf{if } f \rightarrow \textbf{skip}[]F \textbf{ fi})\textbf{fi})\textbf{fi}
\end{aligned}$$

Case 3 : if $c_i = \textbf{if } A_{i1}[] \ldots []A_{in_i} \textbf{ fi}$, and $\mathcal{M}(FS(A_{ij})) \equiv FS(A_{ij})[]F_{ij}$,

$$\begin{aligned}
\mathcal{M}(FS(A_i)) &= \textbf{if } \neg f \wedge g_i \rightarrow \textbf{if } \mathcal{M}(FS(A_{i1}))[] \ldots []\mathcal{M}(FS(A_{in_i})) \textbf{ fi } []f \rightarrow \textbf{skip fi} \\
&\equiv \textbf{if } \neg f \wedge g_i \rightarrow \textbf{if } FS(A_{i1})[]F_{i1} \ldots []FS(A_{in_i})[]F_{in_i} \textbf{ fi } []f \rightarrow \textbf{skip fi} \\
&\equiv \textbf{if } \neg f \wedge g_i \rightarrow \textbf{if } FS(A_{i1})[] \ldots []FS(A_{in_i})[]F_{i1}[] \ldots []F_{in_i} \textbf{ fi } []f \rightarrow \textbf{skip fi} \\
&\equiv \textbf{if } FS(A_i)[]\neg f \wedge g_i \rightarrow \textbf{if } FS(A_{in_i})[]F_{i1}[] \ldots []F_{in_i} \textbf{ fi } []f \rightarrow \textbf{skip fi}
\end{aligned}$$

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

Therefore,

$$
\begin{aligned}
P \oplus F = & \ \mathcal{M}(FS(A_1))[] \ldots []\mathcal{M}(FS(A_n)) \\
\equiv & \ \textbf{if}\ FS(A_1)[]\neg f \wedge g_1 \rightarrow F_1\ \textbf{fi}[] \ldots []\textbf{if}\ FS(A_n)[]\neg f \wedge g_n \rightarrow F_n\ \textbf{fi} \\
\equiv & \ FS(A_1)[]\neg f \wedge g_1 \rightarrow F_1[] \ldots []FS(A_n)[]\neg f \wedge g_n \rightarrow F_n \\
\equiv & \ FS(A_1[] \ldots []A_n)[]\neg f \wedge g_1 \rightarrow F_1 \ldots []\neg f \wedge g_n \rightarrow F_n \\
= & \ FS(P)[]\neg f \wedge g_1 \rightarrow F_1 \ldots []\neg f \wedge g_n \rightarrow F_n
\end{aligned}
$$

□

The fault transformation defined above is based on the assumption that each fault action in F may interrupt the execution of any action in P. In general, an action in P can be interruped by only a subset of the fault actions in F.

For a program $P = A_1[] \ldots []A_n$, $A_i = g_i \rightarrow c_i$, let

$$FS(A_i) = \neg f_{A_i} \wedge g_i \rightarrow FS(c_i)$$

where $FS(c_i)$ is the f_i-stop command of c_i. A fault action $A_f \in F$ *interrupts* the action A_i if A_f transforms f_i from *true* to *false*, i.e.

$$A_f \bowtie A_i \triangleq \{\neg f_i\}A_f\{f_i\}$$

A fault-action $A_f \in F$ *stops* action A_i if it makes f_{A_i} *true*, i.e.

$$A_f \lhd A_i \triangleq \exists f_j : \{\neg f_j\}A_f\{f_j\} \wedge f_j \Rightarrow f_{A_i}$$

where $A_f \in F$ and $f_i, \{f_i\}A_f\{f_i\}$. Let

$$F_{A_i} \triangleq \{A_f \mid A_f \in F \wedge A_f \frown / A_i\}$$

Then $A_i \oplus F_{A_i}$ is defined as:

$$A_i \oplus F_{A_i} \triangleq \neg f_{A_i} \wedge g \rightarrow \mathcal{M}(FS(c_i))$$

where $\mathcal{M}(FS(c_i))$ is obtained by applying to the command c_i the transformations FS and $\mathcal{M}$ *w.r.t.* f_i and F_{A_i}. The fault affected program $\mathcal{F}(P)$ is then defined as:

$$\mathcal{F}(P) = P \oplus F \triangleq A_1 \oplus F_{A_1}[] \ldots []A_n \oplus F_{A_n}$$

It is easy to see that the transformation $\mathcal{F}$ defined in this way still satisfies **Theorem** 3.2. And if there is no fault action in F which interrupts or stops the action A_i, then

$$A_i \oplus \ \equiv A_i$$

In particular, let $P = p_1[] \ldots []p_n$ be a program with n processes and F_{p_i} be fault actions which stop and interrupt the actions of process p_i but not of any other process. The *fault transformation* $\mathcal{F}$ *w.r.t.* $\{F_{p_i} \mid i = 1, \ldots, n\}$ is defined as

$$\mathcal{F}(P) = p_1 \oplus F_{p_1}[] \ldots []p_n \oplus F_{p_n}$$

4 Consistency and Recovery Transformation

Let $P \oplus F$ be the fault affected version of a program P and $P_0 \sqsubseteq ... \sqsubseteq P_k = P$ be a sequence of refinements. When the execution of $P \oplus F$ reaches an error state it will stay in that state forever and no future action in P can be executed. Therefore, the behaviour of $P \oplus F$ will not in general satisfy the original specification P_0, i.e. $P \oplus F$ does not refine P_0. To make the execution of P *recoverable* from an error state, the system has to be restored to a good state from which the interrupted execution can resume. Such a good state can be described in terms of *consistency* with the execution of P.

4.1 Reachability and Consistency

For any state S of a program P, let $Reach(S)$ be the set of states which are reachable from S by executing P, i.e.

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

Let,

$$Reachable(S, S') \textbf{ in } P \triangleq S' \in Reach(S)$$

and, as an abbreviation,

$$Reachable(S) \textbf{ in } P \triangleq Reachable(init_P, S) \textbf{ in } P$$

Let θ be an 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, i.e. S is *forward consistent* (*ForwCon*) with θ:

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

Let $\mathcal{X}$ be a set of subsets of $Var(P)$ and $\Psi = \{S_X \mid X \in \mathcal{X}\}$, where S_X is a substate over X. We say that Ψ is *forward-consistent* with θ, if there exists a state S such that:

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

We may also have 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 this execution can continue from the state S.

Let $\mathcal{X} = \{X_0, ..., X_{n-1}\}$ 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, i.e. 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**:

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

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

$$\mathbf{BC1} \wedge \mathbf{BC2} \wedge \mathbf{BC3}$$

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. $pred_\theta(\beta)^\wedge\beta' \in OB_P$

If $BackwCon(\Psi, \theta)$, for β and β' satisfying these conditions, let $\theta' = pred_\theta(\beta)^\wedge\beta'$. 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, let Ψ_β be given for a subset $\mathcal{X} \subseteq \mathcal{P}(Var(P))$, i.e. $\Psi_\beta = \{S_{X_i} \mid X_i \in \mathcal{X}\}$, satisfies 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, $BackwCon(\Psi_\beta, \theta) \Rightarrow ForwCon(\Psi_\beta, pred_\theta(\beta))$

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

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

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

4.2 Recovery Transformation

To resume the execution of P after interruption by fault actions in F, P has to be transformed into a program $\mathcal{R}(P)$ by adding a set of *recovery actions* P_R called a *recovery program.*

Let *ob* be an auxiliary variable *ob* whoe value space is the set of observations of $P \oplus F$

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

where $\theta(i)$ is a good state for each $i \leq \#\theta$.

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

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

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

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

The predicate $ForwExt(\theta, \theta') = true$ if $ForwCon(last(\theta), \theta') \wedge \theta' \underline{\propto} \theta$.

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 $Var(P) \cup \{ob\}$ so that:

1. each $\{P \wedge \neg f\}A\{Q\}$ in P is transformed into

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

2. each $\{P\}A_f\{Q\}$ in F is transformed into

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

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

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

After the execution of $\mathcal{F}(P)$ reaches an error state, the recovery program P_R is invoked and restores the variables to a consistent state. P_R must satisfy the following conditions:

R1. any action of $FS(P)$ excludes each action $A_r \in P_R$: $\forall A \in P : \neg f \wedge gA \Rightarrow \neg gA_r$

R2. each action $A_r \in P_R$ excludes any action $FS(P)$: $\forall A \in P : gA_r \Rightarrow f \vee \neg gA$

R3. execution of the recovery program P_R cannot be interrupted or stopped by the fault program F.

R4. P_R transforms an error state into a good consistent state:

$$(f \wedge ob = \theta) \longmapsto (\neg f_{ob} \wedge ConsExt(ob, \theta))$$

The recovery program P_R can thus be given as:

Program P_R:
$\langle f \rightarrow X := X'.ConsExt(ob, ob_0)\ ;\ f := false\rangle$
End$\{P_R\}$

These conditions for P_R allow the recovery transformation $\mathcal{R}$ to take the form:

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

Condition **R2** implies that a recovery action A_r does not change a good state and thus

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

From Condition **R3** and Theorem 3.2 we have

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

$\mathcal{F}(\mathcal{R}(P))$ should ideally satisfy the specification P_0. Unfortunately, it is not usually possible to have such a strong transformation for an arbitrary program P and with faults F. However, we can often have a recovery transformation such that $\mathcal{F}(\mathcal{R}(P))$ is weaker than P_0 but acceptable in terms of its behaviour and

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

Two kinds of error recovery are used in practice [AL81]. With *backward error recovery*, the system recovers from a fault by starting from a state which is consistent with its previous states. Forward recovery is used when a program has to recover from an error whose effects can either not be overcome by backward recovery or (in a real-time program) because the time constraints do not permit backward recovery. As in the case of backward recovery, for forward recovery the variables $Var(P)$ have to be assigned values so that the state S is good (i.e. f is false) and forward consisten with the current observation ob.

The recovery transformation $\mathcal{R}$ can apply to both backward and forward recovery and this shows that in principle backward and forward recovery methods can be both used in one fault-tolerant system. Backward and forward recovery programs are special cases of P_R and specified as P_{BR} and P_{FR} respectively:

Program P_{BR}:
$\langle f \rightarrow X := X'.BackwExt(ob, ob_0)\ ;\ f := false\rangle$
End$\{P_{BR}\}$

Program P_{FR}:
$\langle f \rightarrow X := X'.ForwExt(ob, ob_0)\ ;f := false\rangle$
End$\{P_{FR}\}$

5 Using Refinement for Fault-tolerance in Programs

The recovery transformation $\mathcal{R}$ (or the recovery program P_R) describes *what* should be done for recovery but imposes no restrictions on *when* P_R is executed, *where* it is executed (e.g. on which processor) or *how* it is to be executed (e.g. how to find a consistent state). These restrictions can be introduced by using transformations on $P[]P_R$ which will be described in terms of *F-refinement.*

Given a faulty environment F, program P' is said to *F-refine* program P (denoted as $P \sqsubseteq_F P'$) if

$$(P \sqsubseteq P') \wedge (P' \ \underline{Sat}\ (f \longmapsto \neg f))$$

and for each execution E of $\mathcal{F}(P')$ in which there are finitely many error states of which $E[k]$ is the last,

$$E[k+1](ob)^{\wedge}last(E[k+2](ob))^{\wedge} \ldots {}^{\wedge}last(E[k+n](ob)) \ldots$$

is an execution of P'.

As for the equivalence of fault-prone programs, it is not necessary the case that $P \sqsubseteq_F P'$ if P' refines P. However, the following results are easily proved:

Theorem 5.1 *Given programs P, P', P'' and a faulty environment F,*

1. $P \sqsubseteq_F \mathcal{R}(P)$
2. $P \sqsubseteq_F P' \sqsubseteq_F P'' \Rightarrow P \sqsubseteq_F P''$
3. $P \sqsubseteq P' \Rightarrow \mathcal{R}(P) \sqsubseteq_F \mathcal{R}(P')$
4. $P \sqsubseteq P' \Rightarrow P \sqsubseteq_F \mathcal{R}(P')$

Proof: Directly from the definition of F-refinement and the specification of the transformation $\mathcal{R}$. □

The first two results show that fault-tolerance is introduced and preserved by F-refinement transformations. From the next two results, it can be seen that refinement transformations can be applied to the original program and fault-tolerance introduced after that using F-refinement.

Corollary 2 *Given a program P and a faulty environment F,*

1. $P_R \sqsubseteq P_{R'} \Rightarrow \mathcal{R}(P) \sqsubseteq_F P[]P_{R'}$
2. $P \sqsubseteq_F \mathcal{R}(P) \sqsubseteq_F P[]P_{BR}$
3. $P \sqsubseteq_F \mathcal{R}(P) \sqsubseteq_F P[]P_{FR}$

As an example of the first result in Corollary 2, let S_0 be the initial state of program P and

Program $P_{R'}$:
$\langle f \rightarrow X := S_0 \; ; \; f := \mathit{false} \rangle$
End$\{P_{R'}\}$

then

$$(P_R \sqsubseteq P_{R'}) \wedge (P \sqsubseteq_F P[]P_R \sqsubseteq_F P[]P_{R'})$$

Program $P[]P_{R'}$ is a fault-tolerant program such that whenever a fault occurs, the execution of P will recover by re-starting from its initial state [JMS84,JH87].

The following theorem introduces a useful F-refinement rule.

Theorem 5.2 *Given a program $P = A_1[] \ldots []A_n$ and a faulty environment F, let I be an non-empty subset of $\{1, \ldots, n\}$ and*

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

where for each $i \in \{1, \ldots, n\}$,

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

then

1. $\mathcal{R}(P) \sqsubseteq_F \mathcal{R}_I(P)$

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

The next theorem provides a rule to allow recovery action to be freely introduced at any point of the program.

Theorem 5.3 *For the program* $\mathcal{R}_I(P) = A'_1 [] \ldots []A'_n$ *given in Theorem 5.2,*

$$\mathcal{R}_I(P) \sqsubseteq_F \mathcal{R}_I(P)[(\textbf{if } \neg f \rightarrow \textbf{skip}[]P_R \textbf{ fi})/\textbf{skip}^1, \ldots, (\textbf{if } \neg f \rightarrow \textbf{skip}[]P_R \textbf{ fi})/\textbf{skip}^k]$$

where $\textbf{skip}^1, \ldots, \textbf{skip}^k$ *denote different occurrences of* **skip** *in* $\mathcal{R}_I$.

It may be noticed that Theorem 5.2 and Theorem 5.3 also hold for any program $P_{R'} \sqsubseteq_F P_R$.

A program P can be refined using rules of the form defined by Back[BS88], and then Theorem 5.1 can be used to add F-refinement. *Checkpointing actions* can then be added to P (or the refined version of P) by introducing new variables and assignments[BS88,Mor90]. Then, following Theorem 5.2 and Theorem 5.3, recovery actions can be introduced at appropriate points. The choice of recovery point may follow well-known practice, e.g. by using *recovery blocks* or *conversations*. It can be shown that the checkpointing and recovery protocol suggested by Koo and Toueg [KT87] can also be achieved by using F-refinement.

6 Example: A Protocol for Communication Over Faulty Channels

In this section consider an example of fault-tolerant programming using the methods of this paper. 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.

The *Sender* process produces an infinite sequence *ms* of data. The *Receiver* process reads in a sequence *mr* satisfying the following specification:

invariant $mr \precsim ms$	(i.e. *mr* is a prefix of *ms*)
$\#mr = n \longmapsto \#mr = n + 1$	(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:

	c , $ms := c\hat{}head(ms)$, $tail(ms)$	(*Sender*)
[]	$c \neq <> \rightarrow c$, $mr := tail(c)$, $mr\hat{}head(c)$	(*Receiver*)

The reliable FIFO channel c can be implemented as the following program:

$$C:: cs \neq <> \rightarrow cs\ ,\ cr := tail(cs)\ ,\ cr\hat{}head(cs)$$

Let program P be

$$cs := cs\hat{}head(ms) \parallel ms := tail(ms)$$
$$[] \quad cr \neq <> \rightarrow mr := mr\hat{}head(cr) \parallel cr := tail(cr)$$

Then,

$$\textit{Sender-Receiver} \sqsubseteq C[]P$$

Now assume that a faulty channel 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 specification for such a channel can be found in [CM88].

Program F behaves like a generator of faults.

declare b, f : *boolean*
initially
$b, f = \textit{false}, \textit{false}$
actions
$\neg b \wedge cs \neq <> \rightarrow cs, f := tail(cs), \textit{true}$ (loss)
$[]\ \neg b \wedge cs \neq <> \rightarrow cr, f := cr^{\wedge}head(cs), \textit{true}$ (duplication)
$[]\ b := \textit{true}$ (to guarantee the finiteness of consecutive loss and duplication)

Since,

$$C \sqsubseteq cs \neq <> \rightarrow b, cs, cr := \textit{false}, tail(cs), cr^{\wedge}head(cs)$$

the behaviour of a faulty channel can be simulated by a program $FC \triangleq \mathcal{F}(C)$, which is equivalent to:

declare b, f : *boolean*
initially
$b, f = \textit{false}, \textit{false}$
actions
$\neg b \wedge cs \neq <> \rightarrow cs, f := tail(cs), \textit{true}$ (loss)
$[]\ \neg b \wedge cs \neq <> \rightarrow cr, f := cr^{\wedge}head(cs), \textit{true}$ (duplication)
$[]\ \neg f \wedge cs \neq <> \rightarrow b, cs, cr := \textit{false}, tail(cs), cr^{\wedge}head(cs)$ (correct transfer)
$[]\ b := \textit{true}$ (to guarantee the finiteness of consecutive loss and duplication)

And $\mathcal{F}(\textit{Sender-Receiver})$ (or $\mathcal{F}(C[]P)$) is given as:

declare b, f : *boolean*
initially
$b, f = false, false$
actions
$\neg f \rightarrow cs := cs^{\wedge}head(ms) \parallel ms := tail(ms)$
$[]\neg f \wedge cr \neq <> \rightarrow mr := mr^{\wedge}head(cr) \parallel cr := tail(cr)$
$[]\neg b \wedge cs \neq <> \rightarrow cs, f := tail(cs), true$ (loss)
$[]\neg b \wedge cs \neq <> \rightarrow cr, f := cr^{\wedge}head(cs), true$ (duplication)
$[]\neg f \wedge cs \neq <> \rightarrow b, cs, cr := false, tail(cs), cr^{\wedge}head(cs)$ (correct transfer)
$[] b := true$ (to guarantee the finiteness of consecutive loss and duplication)

To design a recovery program for *Sender-Receiver*, let the type of cs and cr be sequence variables whose elements are pairs (*integer*, *data item*). Let $C[]P$ be refined to P_1:

declare ks, kr : *integer*,
initially
$ks, kr = 1, 0 [] cs, cr = <>, <>$
actions
$cs := cs^{\wedge}(ks, ms[ks]) \parallel ks := ks + 1$
$[] cr \neq <> \rightarrow mr := mr^{\wedge}head(cr).val \parallel cr := tail(cr) \parallel kr := kr + 1$
$[] cs \neq <> \rightarrow cs, cr := tail(cs), cr^{\wedge}head(cs)$

Here, $ks \leq head(cs).dex$ if some message is lost and $head(cr).dex < kr$ if a message is duplicated. Therefore, the fault affected program of P_1 can be refined to FP_1:

declare b : *boolean*
initially
$b = false$
actions
$ks = head(cs).dex + 1 \rightarrow cs := cs^{\wedge}(ks, ms[ks]) \parallel ks := ks + 1$
$[]kr = head(cr).dex \wedge cr \neq <> \rightarrow mr := mr^{\wedge}head(cr).val \parallel cr := tail(cr) \parallel kr := kr + 1$
$[]\neg b \wedge cs \neq <> \rightarrow cs := tail(cs)$ (loss)
$[] \neg b \wedge cs \neq <> \rightarrow cr := cr^{\wedge}head(cs)$ (duplication)
$[]cs \neq <> \rightarrow b, cs, cr := false, tail(cs), cr^{\wedge}head(cs)$ (correct transfer)
$[] b := true$ (to guarantee the finiteness of consecutive loss and duplication)

The recovery program for P_1 is P_{1R}, given below.

initially
$ks, kr = 1, 0$
actions
$head(cs).dex \leq ks \rightarrow ks := head(cs).dex + 1$
$[]$**do** $cr \neq <> \wedge head(cr).dex < kr \rightarrow cr := tail(cr)$ **od** ; **if** $cr = <> \rightarrow$ **skip**
$[]cr \neq <> \rightarrow mr := mr^{\wedge}head(cr).val \parallel cr := tail(cr) \parallel kr := kr + 1$ **fi**

In the following, we provide an informal outline of the proof to show that $FP_1[]P_{1R}$ satisfies the specification of the program *Sender-Receiver*.

Proof: To prove the invariant $mr \underline{\propto} ms$:

a) $mr \underline{\propto} ms$ is *true* initially,

b) each action in FP_1 and P_{1R} leaves $mr \underline{\propto} ms$ stable.

Hence $mr \underline{\propto} ms$ is invairiant.

To prove the progress property $\#mr = n \longmapsto \#mr = n + 1$:

a) it is easily seen that $\#mr = kr$ is invariant,

b) from both FP_1 and P_{1R}, each message in *ms* will eventually be transferred to *cs*,

c) each message in *cs* will be eventually be transferred to *cr* or lost,

d) if a message in *cs* is lost, it will be re-sent again because of the execution of the recovery action,

e) FP_1 also guarantees that the *correct transfer action* will eventually be executed,

f) by the second recovery action in P_{1R} and the receiving action in FP_1, each message transferred into *cr* by the execution of the *correct transfer action* in FP_1 will eventually be transferred to *mr*.

Therefore the progress property is guaranteed. □

$P_1[]P_{1R}$ is thus a version of *Sender-Receiver* which can tolerate message loss and duplication; it can of course be refined further for a different implementation[CM88].

7 Discussion

Assume that $P_0 = Sp$ is the top level specification of a program in a UNITY-like notation. The top level specification of the faulty environment F_0 can be given as

$$\neg f \longmapsto f$$

P_0 can be then refined into an action system P_1 while F_0 can be simulated (or refined) by an action system F_1 consisting of one action

$$true \rightarrow f := true$$

Based on P_1 and F_1, the fault and recovery transformations can then be applied to program P_1 in the way described in Section 4.2. For each refinement step $P_k \sqsubseteq P_{k+1}$ of the original program P_0, a refinement step $F_k \sqsubseteq F_{k+1}$ of the faulty environment is derived by providing more details about the system and its possible faults. The fault and recovery transformations can be applied again to P_{k+1} and F_{k+1}. During the refinement of the original program $P_0 \sqsubseteq ... \sqsubseteq P_k \sqsubseteq ...$, checkpoints and recovery methods such as recovery blocks and

conversations can be introduced. Thus, a fault-tolerant program can be produced from its fault-free specification by using F-refinement rules. We do not underestimate the difficulty of doing this in practice, but the rules provided here do offer a formal means of verifying what is otherwise left to less precise methods of reasoning.

Each step in the refinement $P_0 \sqsubseteq ... \sqsubseteq P_k \sqsubseteq ...$ provides more information about the system on which the program is to be executed and similar information about the faults of the system is used for refining the fault specification. For certain programs and faults, e.g. *Sender-Receiver* and the faulty channels, the boolean variable f which indicates the presence of faults in F_O can be derived from the state predicates of program P_k at a suitable stage in the refinement. But it is an open question whether this is always possible for any program and any set of possible faults.

Fault-tolerant systems often also have real-time constraints. So it 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 paper by adding timing properties *w.r.t.* some *time domain* [JG88], the recovery transformation can be defined with timing constraints. The specification and refinement of the recovery actions 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 the recovery action. Thus the method described in this paper 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.

Acknowledgment

We would like to thank our colleague, Asis Goswami, for helpful discussions on the definition of consistency. This work was supported by research grant GR/D 11521 from the Science and Engineering Research Council.

References

[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 Symp. on LICS*, 1988.

[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 54, Abo Akademi, 1988.

[Bac89] R.J.R. Back. Refinement calculus, Part I: Sequential nondeterministic programs. Technical Report 92, Abo Akademi, 1989.

[BKS83] R.J.R. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. In *Second Annual ACM Symposium on PoDC*, pages 131--142, 1983.

[BR81] E. Best and B. Randell. A formal model of atomicity in asynchronous systems. *Acta Informatica*, 16:93--124, 1981.

[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.

[BvW89] R.J.R. Back and J. von Wright. Refinement calculus, Part II: Parallel and reactive programs. Technical Report 92, Abo Akademi, 1989.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley Publishing Company, 1988.

[GP89] Rob Gerth and Amir Pnueli. Rooting unity. *In Proceedings of the 5th IEEE International Workshop on Software Specification and Design*, February 1989.

[JG88] Mathai Joseph and Asis Goswami. What's 'real' about real-time systems? In *Proceedings of IEEE Real-time Systems Symposium*, pages 78--85, December 1988.

[JH87] He Jifeng and C.A.R. Hoare. Algebraic specification and proof of a distributed recovery algorithm. *Distributed Computing*, 2:1--12, 1987.

[JMS84] Mathai Joseph, Abha Moitra, and Neelam Soundararajan. Proof rules for fault tolerant distributed programs. *Science of Computer Programming*, 8(1):34--68, October 1984.

[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. A simple approach to specifying concurrent systems. *Communications of the ACM*, 32(1):32--45, January 1983.

[Mor90] Carroll 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*, 1983.

[MR78] P.M. Merlin and B. Randell78. State restoration in distributed systems. In *Proceedings of 8th Ann. Int. Symp. on Fault-Tolerant Comput.*, pages 129--134, 1978.

[Ran75] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1(2):220--232, Jule 1975.

[SS83] Richard D. Schlichting and Fred B. Schneider. Fail-stop processes: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222--238, 1983.