Modelling Checkpointing and Recovery Within UNITY

Zhiming Liu*
University of Warwick

Abstract

Following the method for developing programs proposed in UNITY [**?**], this paper presents a general model for checkpointing and recovery within which the *consistency* of checkpoints, *error propagation* and the *domino-effect* of recovery from faults are formally characterized. Based on this model and UNITY, a solution is developed for checkpointing and recovery which describes *what* should be done for checkpointing and recovery but imposes no restrictions on *when* it should, *where* it should be done (e.g. on which processors), or *how* it should be done. This supports the idea that the introduction of checkpointing and recovery can be treated systematically as transformations which convert a program into a *fault-tolerant* program. Such *fault-tolerant* transformations and their correctness are defined in this paper.

*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.

1 Introduction

The demand for reliable computer systems has led to the development of techniques for the construction of fault-tolerant systems. The aim of these techniques is to ensure that a system provides an intended service despite possible software or hardware faults. The techniques depend on the use of *error recovery*, i.e. a method of transforming the failed state of the system into a well defined and error free state. Methods for achieving this transformation can sometimes make good use of the information retained in the erroneous state, but it can be more secure to simply discard the erroneous state and reset the system either to some prior state) or to an error-free state computed by an exception-handling mechanism. A particular form of error recovery restore the state of the system to a prior value stored as a checkpoint and is called *backward recovery*. Much of the current work in error recovery is focused on backward error recovery in distributed systems [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**].

If the individual processes in a distributed system take checkpoints at unconnected points in their execution, it is difficult to ensure that the system as a whole can recoverfrom errors by making each process resume execution from its latest checkpoint. So an important approach to fault-tolerance is based on structuring the communication between a group of processes into *atomic actions* [**?**, **?**]. The activity of a group of processes constitutes an atomic action if there are no interactions between that group and the rest of the the system for the duration of the activity. This must also ensure of course that there is no error propagation to the rest of the system during the performance of the atomic action. Another method [**?**, **?**, **?**, **?**, **?**] limits the exchange of information between processes to *conversation structures*. A conversation permits the exchange of information between processes in such a way that processes may independently enter the conversation region but all communicating processes must leave it at the same time after the establishment of their individual checkpoints. A common feature of these methods is the requirement that recovery is performed from a consistent set of checkpoints. Recent work [**?**, **?**] concerns the design of protocols to take a *consistent* set of checkpoints.

Thus, some of the methods determine the consistency statically from the structure of the program, and the others do so dynamically by protocols. In general, since no formal definition is given for the *correctness* of *fault-tolerant programs*, it is not possible to formally discuss the correctness of these methods. Formal specification for fault-tolerant programs and the correctness of the defined fault-tolerant programs are required for evaluating specifications and proof systems of fault-tolerant systems.

Following the method for developing programs proposed in UNITY [**?**], this paper presents a general model for checkpointing and recovery in distributed systems within which the *consistency* of checkpoints is formally defined. Based on this model and UNITY, a solution is developed for checkpointing and recovery which describes *what* should be done for checkpointing and recovery but imposes no restrictions on *when* it shoul be done, *where* it should be done (i.e. on which processors), or *how* it should be done. This supports the idea that the introduction of checkpointing and recovery can be treated systematically as transformations which convert a program into a *fault-tolerant* program. Such *fault-tolerant transformations* is defined which for a given specified *fault-prone* system, transforms a program into a program that can tolerate the specified faults. A simple and formal definition for the correctness of the fault-tolerant transformations is then achieved. In this paper, we assume that all the processes are *fail-stop* [**?**] and this sumption can be defined within the UNITY program model.

After a summary of the basic features of UNITY in Section **??**, a model of UNITY programs is described in Section **??**. In Section **??**, a method of checkpointing is specified and implemented by using UNITY; checkpoints and the requirements for consistency are also defined in this section. Section **??** presents the modelling of backward error recovery. In Section **??**, a general scheme of recovery and a solution for *smallest* and *latest* recovery are developed. The *fault-tolerant transformation* is introduced at the end of this section. The correctness of the fault-tolerant transformation is discussed in Section **??**.

2 Summary of UNITY

UNITY is a formal system recently proposed by Chandy and Misra [**?**] to provide a foundation for an approach to designing 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.

The model of computation is very simple and can be described by a UNITY program. A program consists of a declaration of variables, a specification of their initial values, and a set of multiple-assignment statements. Program execution starts in any state satisfying the initial specification, and proceeds by repeatedly selecting and executing nondeterministically one of the assignment statements. This continues forever, with nondeterministic selection constrained by appropriate fairness conditions[**?**, **?**]. A UNITY program describes *what* should be done but not *when* an assignment is executed, *where* it is executed (i.e. on which processor), or *how* it is to be executed. These details are given by mapping from programs to architectures.

The logic used for the specification, design and verification of UNITY programs is based on Hoare-style assertions of the form $\{p\}s\{q\}$, where s is universally or existentially quantified over the statements of the program. These form properties that hold for the program as a whole. For example, if for program F we have $< \forall s : s \; in \; F :: \{x = k\}s\{x \geq k\} >$ (where k is implicitly universally quantified) then the value of x does not decrease during the execution of F. The following notation is used in UNITY:

- p **Unless** $q \;\; \equiv \;\; < \forall s : s \; in \; F :: \{p \wedge \neg q\}s\{p \vee q\} >$
- **Stable** $p \;\; \equiv \;\; p$ **Unless** $false$
- **Invariant** $p \;\; \equiv$ (initial condition $\Rightarrow p) \wedge$ **Stable** p
- p **Ensures** $q \;\; \equiv \;\; p$ **Unless** $q \wedge < \exists s : s \; in \; F :: \{p \wedge \neg q\}s\{q\} >$

Progress properties are usually specified using *leads-to* ($\longmapsto$). For a given program $p \longmapsto q$ iff this property can be derived by a finite number of applications of the following three inference rules:

$$\frac{p \; \boldsymbol{Ensures} \; q}{p \longmapsto q}, \qquad \frac{p \longmapsto q, \quad q \longmapsto r}{p \longmapsto r},$$

for any set W:

$$\frac{< \forall m : m \in W :: p(m) \longmapsto q >}{< \exists m : m \in W :: p(m) > \longmapsto q}.$$

A set of theorems about these terms is used in proving the properties of programs. Another useful notion is that of a *fixed point*. A fixed point of a program is a program state such that execution of any statement in that state leaves the state unchanged. A predicate (usually termed $Fixedpoint(F)$) defining that state can be deduced from the program text.

There is no special UNITY notations for processes and channels. However, processes can be represented by UNITY programs (i.e. sets of assignments) with shared variables, and channels by shared variables. Two forms of composition of UNITY programs are defined: *union* and *superposition*. The union of two programs is a program formed by appending their commands together. It acts like parallel composition and is suitable for building networks of communicating processes. Various properties of a union can be derived from the properties of the components. Superposition, on the other hand, is a transformation on programs which adds new variables and assignments, but without altering the assignments of the original program. Thus superposition preserves all the properties of the original program and is useful for building programs in layers.

3 Model of Programs

The UNITY approach fits in well with our aim of generally modelling checkpointing and backward error recovery. Let G be a program with a set $Proc$ of n processes. For each pair of processes p and q in $Proc$ such that $p \neq q$, let ch_{pq} be a shared sequence variable (called the *channel* from p to q) through which p sends a message 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 $\overline{cs}_{pq}$ be the sequence of messages actually sent by p to q and let $\overline{cr}_{pq}$ be the sequence of the messages actually received from p by process q. The required properties of these variables can be specified by using UNITY programming logic and implemented as UNITY program statements within different parts of G. This section presents the specification and implementation with necessary explaintations.

For each process $p \in Proc$, let V_p be the set of variables (called *process variables*) which is the union of the following subsets:

- X_p: the set of all the *local* variables of p,
- $From_p =_{df} \{\overline{cs}_{pq} | q \in Proc \wedge q \neq p\}$: the set of sequence variables of messages sent from p to other processes,

- $To_p =_{df} \{\overline{cr}_{qp} | q \in Proc \wedge q \neq p\}$: the set of sequence variables of messages received by p from other processes.

Let $Ch_p =_{df} \{ch_{pq}, ch_{qp} | q \in Proc \wedge q \neq p\}$ be the set of channels through which process p communicates with other processes.

An element in a sequence Seq can be accessed by its index: thus, $Seq[k]$ is the k^{th} element in Seq. Let λ denote the empty sequence, $e \in Seq$ mean that e is an element of Seq, $First(Seq)$ be the first element of Seq, $First(k, Seq)$ be the sequence of the first k elements of Seq and $Tail(Seq)$ be the sequence obtained from Seq by removing the first element. Let $Seq' \preceq Seq$ denote that sequence Seq' is a prefix of sequence Seq and if Seq' is a prefix of Seq, then $Seq - Seq'$ is the sequence obtained from Seq by removing the prefix Seq'. A sequence of elements is defined by enclosing it within $<<$ and $>>$, $Seq ^\frown m$ denotes the concatenation of sequence Seq with element M and $\#Seq$ is the length of sequence Seq.

If there is no fault, then for any p and q in $Proc$ such that $p \neq q$, channel ch_{pq} and the variables in V_p are required to satisfy the following properties that can be implemented as UNITY statements:

1. the sequence of messages sent by p to q never gets shorter, i.e. for any constant sequence cs_o,

   $$\textbf{Stable } cs_o \preceq \overline{cs}_{pq},$$

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,

   $$(cs_o \preceq \overline{cs}_{pq} \wedge \neg B_{pq}) \longmapsto (B_{pq} \vee cs_o \prec \overline{cs}_{pq}),$$

   where B_{pq} is a predicate that does not name channels.

3. the message in ch_{pq} must be an element in $\overline{cs}_{pq}$, i.e. for any message e,

   $$\textbf{Invariant } e \in ch_{pq} \Rightarrow e \in \overline{cs}_{pq},$$

4. messages must be sent before they are received and must be received in the same order that they are sent, i.e.

$$\textbf{Invariant } \overline{cr}_{pq} \preceq \overline{cs}_{pq}$$

5. messages cannot be lost, i.e.

$$\textbf{Invariant } ch_{pq} = \overline{cs}_{pq} - \overline{cr}_{pq}.$$

6. messages sent by q must be eventually received by q, i.e.

$$(ch_{pq} \neq \lambda) \longmapsto (\overline{cr}_{pq} = \overline{cr}_{pq} \frown First(ch_{pq})),$$

7. termination of G implies termination of the communications, i.e.

$$Fixedpoint(G) \Rightarrow \exists\, k : (\overline{cs}_{pq} = \overline{cr}_{pq} \wedge \#\overline{cs}_{pq} = k \wedge ch_{pq} = \lambda).$$

To satisfy these properties, process p must contain the following statement:

$$\overline{cs}_{pq},\ ch_{pq} := \overline{cs}_{pq} \frown m,\ ch_{pq} \frown m \ \ \textbf{if} \ \ B_{pq};$$

and process q must contain

$$\overline{cr}_{pq},\ ch_{pq} := \overline{cr}_{pq} \frown First(ch_{pq}),\ Tail(ch_{pq}) \ \ \textbf{if} \ \ ch_{pq} \neq \lambda.$$

Thus, G must be written as the union of such n processes so that Properties **??**–**??** can be implied from the original specification of program G

4 Checkpointing

We shall assume that fault-tolerance is provided by using a *backward error recovery* [**?**] technique. This depends on the provision of *checkpoints*, i.e. a means by which a state of a process can be recorded and later reinstated. Various techniques can be used for checkpointing. These techniques can be, however, essentially classified into two kinds, i.e. *static* checkpointing and *dynamic* checkpointing. Static checkpointing can be achieved by introducing some program constructs such as *recovery blocks* and *conversations* [**?**]. Dynamic checkpointing can be implemented as a protocol to take a *consistent* set of checkpoints [**?**]. Whatever kind of checkpointing, the *consistency* of checkpoints is important. In this section, a solution is developed for checkpointing that is inherently nondeterministic with no restriction on *when*, *where* or *how* a process takes its checkpoints. By doing so, a simple model of checkpointing and recovery can be achieved which is good for understanding *what* checkpointing is. With the definition of additional program constructs in UNITY, special techniques for checkpointing, such as those given in [**?**], [**?**] and [**?**], can be taken as deterministic solutions that might be achieved by refining the solution given in this paper.

The model for checkpointing in this paper is based on the semantic model of UNITY given in [**?**]. But for simplicity, executions of UNITY programs are defined in terms of *states* instead of *observations*. Formally speaking, for a set of variables X and its value space V, a *state* over X is a function S from X to V. A state of a given UNITY program F is a function S from the set $Var(F)$ of all the variables of F to its value space, i.e. a state over $Var(F)$. And no distinction is made between the function and its values when talking about a state of a program. For defining the checkpointing, some notations are defined as follows.

From the model of programs given in Section **??**, when process p takes a checkpoint, it is sufficient for recovery to save the values of the process variables in V_p. Let $V(G)$ be the set of the process variables of G, and $CV(G)$ be a set of sequence variables with an one-one correspondence to $V(G)$ such that for each variable $x \in V(G)$, $v.x \in CV(G)$ is a sequence variable that records values of x. For each process $p \in Proc$, let CV_p be the subset of $CV(G)$ with the restriction of this one-one correspondence to V_p. Let $v.p$ denote the subset of CV_p corresponding to the subset X_p of V_p and for each pair of processes p and q such that $p \neq q$, $v.\overline{cs}_{pq}$ and $v.\overline{cr}_{pq}$ are sequence variables that respectively record the contents of $\overline{cs}_{pq}$ and $\overline{cr}_{pq}$. Note that $v.\overline{cs}_{pq}$ and $v.\overline{cr}_{qp}$ are variables of sequences of sequences. At any time, the checkpointing program can append the current values of all the corresponding variables of a process p to $v.p$, $v.\overline{cs}_{pq}$ and $v.\overline{cr}_{qp}$.

Let X be a set of variables and S be a state over X, if Y is a subset of X, $S|_Y$ is the restriction function of S to Y. For a variable $x \in X$, $S|_x$ is the restriction function of S to x (i.e. the value of x in state S). For each process $p \in Proc$, a **local state** of p is a state over V_p and can be denoted as $S.p$ and let $init_p$ be the initial state of p. A **checkpointing state** of process p is a state over CV_p and denoted as CP_p. Since CV_p is a set of sequence variables and the values of all the variables in V_p are required to be simultaneously appended into the corresponding variables in CV_p when p takes checkpoint, all the variables in CV_p are of the same length in each state and this length is denoted as $\#CV_p$. Given $k \leq \#CV_p$, the function $CP_p[k]$ is defined as: for each variable $v.x \in CV_p$

$$CP_p[k](v.x) = CP_p(v.x)[k].$$

$CP_p[k]$ is called the k^{th} **checkpoint** of process p. In the following, the notation $[S_p]\boldsymbol{v}_p[S'_p]$, where S_p and S'_p are states over $V_p \cup Ch_p$ and $\boldsymbol{v}_p$ is a finite sequence of statements in process p, means that starting in state S_p execution of the sequence of statements $\boldsymbol{v}_p$ results in state S'_p. We use a **bold** lower-case letter indexed with a process to represent a sequence (possible empty) of statements in the indexing process. The *checkpointing program* for program G is required to satisfy the specification:

- for any $v.x, v.y \in CV_p$,

  Invariant $\#v.x = \#v.y$,

- for any checkpoint $CP_p[k]$ of a process $p \in Proc$,

  Invariant $\exists\, S'_p, \boldsymbol{u}_p, \boldsymbol{v}_p :: \forall\, S_p : ([init_p]\boldsymbol{u}_p[S'_p] \wedge [S'_p]\boldsymbol{v}_p[S_p] \Rightarrow ([CP_p[k] \cup S'_p|_{Ch_p}]\boldsymbol{v}_p[S_p]))$,

- for any constant sequence c_o,

  Stable $(c_o \preceq v.x)$,

- for any constant sequence c_o,

  $(c_o \preceq v.x) \longmapsto (c_o \prec v.x)$.

In what follows, for each $x \in V_p$, $init\ x$ denotes the initial value of x defined in the initially section of G. The checkpointing specification can be implemented as a UNITY program:

Program C_G:
  initially:
  $\langle [] p \in Proc, v.x \in CV_p :: v.x = << init\ x >> \rangle$
  assign:
  $\langle [] p \in Proc ::< \|\ v.x \in CV_p :: v.x := v.x \frown x \rangle$
End$\{C_G\}$

Let $C(G) =_{df} G[]C_G$; since C_G does not modify any variable in G, $C(G)$ is correct with respect to the original specification of G, and thus $C(G)$ preserves all the *invariants*, *stable* and *progress* properties of G. Also, $C(G)$ satisfies the specification of the checkpointing program. However, from the progress properties in the specification of the checkpointing program for G, it is obvious that program $C(G)$ cannot reach any fixed point even though program G always reaches its fixed point $Fixedpoint(G)$. To guarantee that no checkpoint is taken after G reaches its fixed point, it is necessary and sufficient to transform each statement s in the checkpointing program C_G into

$$s \textbf{ if } \neg Fixedpoint(G).$$

It is assumed that such a transformation has been made when discussion or verification is required for the termination or correctness of the composed *fault-tolerant* program, i.e. the program with checkpoints and recovery. In the following discussion, we use C to simply denote C_G.

Let S be the state at an execution step of $C(G)$. For each process $p \in Proc$ the restriction of S to CV_p, $S|_{CV_p}$ is a checkpoint state of p which contains the history of state saving that C has made for p up to S. $S|_{CV_p}$ is called the **checkpointing state of** p **at** S and denoted as $S|_p^C$. The function $S|_p^C[k]$ defined by

$$S|_p^C[k](v.x) =_{df} S(v.x)[k]$$

for any variable $v.x \in CV_p$ is therefore a checkpoint of process p, and said to be the k^{th} **checkpoint** (**recovery point**) of p **up to** S. In these two definitions, $S(x)$ is the value of x in state S. In this paper, we shall only consider a model for checkpointing and recovery: the question of how to get the *global state* of $C(G)$ within UNITY has been presented in [**?**, **?**] as the problem of taking global snapshots.

The following definitions and notations will be used:

- given a global state S of G, $S|_p^C[k]$ is the k^{th} checkpoint of process p up to S,
- for each $p \in Proc$ and a state S of $C(G)$, the restriction of S to V_p, $S|_{V_p}$ is called the **local state** of p at S,
- For $\mathcal{P}' = \{p_a, \ldots, p_b\} \subseteq Proc$ and the local states, $\mathcal{S} = \{S_a.p_a, \ldots, S_a.p_b\}$ in an execution of G, $\mathcal{S}$ is said to be **consistent** if for all $p_i, p_j \in \mathcal{P}'$ such that $i \neq j$,

$$S_j.p_j(\overline{cr}_{p_ip_j}) \preceq S_i.p_i(\overline{cs}_{p_ip_j})),$$

the predicate $Consistent(\{S_a.p_a, \dots, S_b.p_b\})$ is $true$ iff $\{S_a.p_a, \dots, S_b.p_b\}$ is consistent,

- For processes $p, q \in Proc$ such that $p \neq q$ and local states $S.p$ and $S.q$, the predicate $SemiCon(S.p, S.q)$ (i.e. *Semi-consistent*) is $true$ if

$$S.q(\overline{cr}_{pq}) \preceq S.p(\overline{cs}_{pq}),$$

obviously, $Consistent(\{S.p, S.q\}) \equiv SemiCon(S.p, S.q) \wedge SemiCon(S.q, S.p)$,

- For a process p and checkpoints $CP_p[k]$ and $CP_p[i]$ of p, the predicate $Before(CP_p[k], CP_p[i])$ is $true$ if $k < i$, and

$$NotLater(CP_p[k], CP_p[i]) =_{df} Before(CP_p[k], CP_p[i]) \vee k = j.$$

- for two sets of checkpoints $\mathcal{CP}$ and $\mathcal{CP}'$ such that for each checkpoint $CP_p[i] \in \mathcal{CP}$ of process p, there is one and only one checkpoint $CP_p[j] \in \mathcal{CP}'$ of process p, $NotLater(\mathcal{CP}, \mathcal{CP}')$ is $true$ if for each checkpoint $CP_p[i] \in \mathcal{CP}$ and its corresponding checkpoint $CP_p[j] \in \mathcal{CP}'$, $NotLater(CP_p[i], CP_p[j])$ is $true$; and

$$Before(\mathcal{CP}, \mathcal{CP}') =_{df} NotLater(\mathcal{CP}, \mathcal{CP}') \wedge \exists\, CP_q[i] \in \mathcal{CP}, CP_q[j] \in \mathcal{CP}' : Before(CP_q[i], CP_q[j])$$

For the validity of these definitions, the following theorem is needed.

Theorem 4.1 *A checkpoint of a process is a record of a local state of this process reached at the establishment of this checkpoint, i.e. let E be an execution of $C(G)$ and S be the global state in E, say, $E(i)$, and Let $S|_p^C[k_i]$ be the k^{th} checkpoint of p up to S for $p \in Proc$, then there is some state S' of $C(G)$ which is not later than S in E (i.e. $S' = E(i')$ for some $i' \geq i$) such that for each $x \in V_p$,*

$$S|_p^C[k](v.x) = S'|_p(x).$$

Proof: Directly from the specification of $C(G)$.

□

Based on this theorem, the consistency of a set of checkpoints can be defined, i.e. if $\mathcal{P}' = \{p_a, \ldots, p_b\} \subseteq Proc$ and $\mathcal{CP}$ is a set of checkpoints such that for each $p \in \mathcal{P}'$, p has one and only one checkpoint $CP_p[k_p] \in \mathcal{CP}$, then there exists a set $\mathcal{LS}$ of local states such that for each $p \in \mathcal{P}'$, there is one and only one local state $S.p$ of process p in $\mathcal{LS}$ and for each $x \in V_p$, $S.p(x) = CP_p[k_p](v.x)$. $\mathcal{CP}$ is called a **consistent set of checkpoints** if

$$Consistent(\mathcal{LS}) = true.$$

5 Modelling Recovery

In what follows, let $Current(x)$ denote the current value of variable x, and $Current(p)$ denote the current local state of process p during an execution of program $C(G)$. For two checkpoints $CP_p[k]$ and $CP_q[j]$ in an execution E of $C(G)$ such that $p \neq q$, $CP_p[k]$ is said to be a **direct propagator** of the recovery operation for $CP_q[j]$ if it satisfies the following conditions:

1. $SemiCon(CP_p[k], CP_q[j])$,
2. $\neg \exists\, CP_q[j'] : (Before(CP_q[j], CP_q[j']) \wedge SemiCon(CP_p[k], CP_q[j']))$,
3. $\neg SemiCon(CP_p[k], Current(q))$.

Condition **??** implies the consistency between messages sent and received while Condition **??** and Condition **??** jointly guarantee that $CP_q[j]$ is the latest checkpoint of q that is propagated by $CP_p[k]$. The relation "$CP_p[k]$ is a direct propagator for $CP_q[j]$" is denoted as $CP_p[k] \leadsto CP_q[j]$. This definition implies that recovery from the checkpoint $CP_p[k]$ requires recovery from the checkpoint $CP_q[j]$.

Checkpoint $CP_p[k]$ is said to be an **indirect propagator** of checkpoint $CP_q[j]$, denoted as $CP_{pk} \leadsto^* CP_q[j]$, if

1. $\exists\, CP_t[i] : (CP_p[k] \leadsto CP_t[i]) \wedge (CP_t[i] \leadsto^* CP_q[j])$, and
2. $CP_p[k] \leadsto^* CP_p[k]$.

For a process p and checkpoints of $CP_p[k]$ and $CP_p[i]$ of p, the predicate $Active(CP_p[k])$ is $true$ if $k = \#Current(CV_p)$. Let $\mathcal{P}'$ be a subset of $Proc$, then a **recovery line** for $\mathcal{P}'$, denoted as $RL(\mathcal{P}')$, is defined to be a set of checkpoints satisfying

1. $\forall p \in \mathcal{P}' : \exists! CP_p[k] \in RL(\mathcal{P}')$,
2. $\forall CP_p[i], CP_q[j] : ((CP_p[i] \in RL(\mathcal{P}')) \wedge (CP_p[i] \leadsto^* CP_q[j]) \Rightarrow q \in \mathcal{P}')$,
3. $Consistent(RL(\mathcal{P}'))$.

The definitions of *indirect propagator* and *recovery line* can be used to show the *domino-effect* in recovery from faults in fault-tolerant systems [**?**]. In the case of an error arising in pits execution, process p may have to restore a checkpoint $CP_p[k]$ that has been established. Because of the domino-effect, a set of checkpoints propagated by $CP_p[k]$ must restored as well. This set can be termed the **recovery domain** of $CP_p[k]$ and is defined as

$$Z(CP_p[k]) =_{df} \{CP_q[i] | q \in \mathcal{P} \wedge CP_p[k] \leadsto^* CP_q[i]\}.$$

To obtain the lemmas, theorems and corollaries in this section, the checkpointing and recovery are required to satisfy the following conditions.

Assumption 1 *If $CP_p[k]$ and $CP_p[i]$ are checkpoints of a process $p \in Proc$, then for any process $q \in Proc$ such that $q \neq p$,*

$$CP_p[k](v.\overline{cs}_{pq}) \preceq Current(\overline{cs}_{pq}) \wedge CP_p[k](v.\overline{cr}_{qp}) \preceq Current(\overline{cr}_{qp}), \text{ and}$$

$$\begin{aligned} NotLater(CP_p[k], CP_p[i]) \Rightarrow & (CP_p[k](v.\overline{cs}_{pq}) \preceq CP_p[i](v.\overline{cs}_{pq}) \\ & \wedge CP_p[k](v.\overline{cr}_{qp}) \preceq CP_p[i](v.\overline{cr}_{qp})). \end{aligned}$$

This assumption can be implemented , when process p recovers from checkpoint $CP_p[k]$, by deleting the checkpoints of process p that have been established after $CP_p[k]$, i.e. the checkpointing program also has to recover (see Section **??**). The assumption is reasonable because that if process p recovers from $CP_p[k]$, the states of p from $CP_p[k]$ to the execution of the recovery operation are unusable.

From this assumption, the following results can be proved.

Lemma 1 *Given checkpoints $CP_p[i]$ and $CP_p[j]$ such that $NotLater(CP_p[i], CP_p[j])$, then for any checkpoint $CP_q[k]$ of process q such that $q \neq p$,*

$$SemiCon(CP_q[k], CP_p[j]) \Rightarrow SemiCon(CP_q[k], CP_p[i]), \textit{ and}$$

$$SemiCon(CP_p[i], CP_q[k]) \Rightarrow SemiCon(CP_p[j], CP_q[k]);$$

and for any checkpoints $CP_p[k]$ and $CP_q[i]$ of processes p and q such that $p \neq q$,

$$SemiCon(CP_p[k], Current(q)) \Rightarrow SemiCon(CP_p[k], CP_q[i]).$$

Proof: Directly from the definition of the predicate $SemiCon$ and **Assumption ??**.

□

Lemma 2 *Given checkpoints $CP_p[i]$ and $CP_p[j]$ such that $NotLater(CP_p[i], CP_p[j])$, then for any checkpoints $CP_q[m]$ and $CP_q[n]$ of process q such that $q \neq p$ and $CP_p[i] \rightsquigarrow CP_q[m] \wedge CP_p[j] \rightsquigarrow CP_q[n]$,*

$$NotLater(CP_q[m], CP_q[n]).$$

Proof: From Lemma **??**, if $Before(CP_q[n], CP_q[m])$, then

$$(CP_p[i], CP_p[j]) \wedge SemiCon(CP_p[i], CP_q[m])$$

implies $SemiCon(CP_p[i], CP_q[m])$ that contradicts $CP_p[j] \rightsquigarrow CP_p[n]$. Thus, $NotLater(CP_q[m], CP_q[n])$.

□

Lemma 3 *Given checkpoints $CP_p[i]$ and $CP_p[j]$ such that $NotLater(CP_p[i], CP_p[j])$ and $CP_p[j] \rightsquigarrow CP_q[n]$ for process q, then there exists a checkpoint $CP_q[m]$ of process q such that*

$$NotLater(CP_q[m], CP_q[n]) \wedge CP_p[i] \rightsquigarrow CP_q[m].$$

Proof: From Lemma **??**,

$$\neg SemiCon(CP_p[j], Current(q)) \Rightarrow \neg SemiCon(CP_p[i], Current(q)).$$

Therefore, there must be a checkpoint $CP_q[m]$ of process q such that $CP_p[i] \rightsquigarrow CP_q[m]$. From Lemma **??**, we have $NotLater(CP_q[m], CP_q[n])$.

□

This lemma means that if by recovering at a checkpoint $CP_p[j]$, process p causes process q to recover from a checkpoint $CP_q[n]$, then by recovering at a checkpoint that is not later than $CP_p[j]$, p causes q to recover from a checkpoint that is not later than $CP_q[n]$.

Lemma 4 *If $RL(\mathcal{P}')$ is a recovery line for $\mathcal{P}' \subseteq Proc$, then processes not in $\mathcal{P}'$ are not required to recover due to the recovery of processes in $\mathcal{P}'$ from $RL(\mathcal{P}')$, i.e. for any checkpoint $CP_p[k] \in RL(\mathcal{P}')$ and any process $q \notin \mathcal{P}'$, $Consistent(CP_p[k], Current(q))$.*

Proof: See Appendix **??**.

□

Lemma 5 *The set of checkpoints*
$\mathcal{X}(CP_p[k]) =$
$\{CP_q[i] | CP_q[i] \in Z(CP_p[k]) \wedge \neg \exists CP_q[i'] \in Z(CP_p[k]) : Before(CP_q[i'], CP_q[i])\}$,
is the recovery line for the set of processes

$$P(CP_p[k]) =_{df} \{q | \exists CP_q[i] \in \mathcal{X}(CP_p[k]))\},$$

in which error propagation may appear.

Proof: See Appendix **??**.

□

Therefore, the earliest checkpoints in the recovery domain of the recovery point $CP_p[k]$ represent the recovery line with respect to $P(CP_p[k])$.

It may be seen from the construction of the checkpointing program C_G that a great deal of storage may be occupied by checkpoints. There may be some *redundant checkpoints* that have been established up to an execution step of $C(G)$ which will not be used for any recovery. It is easy to see that a checkpoint $CP_p[k]$ is redundant if

$$\forall q \in Proc :: \neg \exists CP_q[j] : (Active(CP_q[j]) \wedge CP_q[j] \leadsto^* CP_p[k]).$$

Detection of redundant checkpoints is important for optimising the use of retrieval storage occupied by checkpoints. However, such detection must depend on the way of determining determination of the recovery line for the failed processes. The above condition is with respect to *recovering from the latest and smallest recovery line*.

The following theorems formally express some important properties of checkpointing and recovery that are usually expressed and proved in informal ways [**?**].

Theorem 5.1 *For each process $p \in Proc$ and checkpoints $CP_p[k]$ and $CP_p[i]$ of p,*

$$NotLater(CP_p[k], CP_p[i]) \Rightarrow P(CP_p[i]) \subseteq P(CP_p[k]).$$

Proof: From Lemma **??**.

$\square$

Theorem 5.2 *The recovery line determined in Lemma* **??** *is the* **latest** *for $P(CP_p[k])$ with respect to $CP_p[k]$, i.e. for any consistent set of checkpoints $\mathcal{CP}$ such that for each process $q \in P(CP_p[k])$ there is one and only checkpoint $CP_q[n_q] \in \mathcal{CP}$ and $NotLater(CP_p[n_p], CP_p[k])$, $NotLater(\mathcal{CP}, \mathcal{X}(CP_p[k]))$*

Proof: See Appendix **??**.

$\square$

Theorem 5.3 *The set of checkpoints $\mathcal{X}(CP_p[k])$ determined in Lemma* **??** *satisfies:*

1. *for each process $q \in P(CP_p[k])$,*

$$\exists\, CP_t[i] \in \mathcal{X}(CP_p[k]) : \neg SemiCon(CP_t[i], Current(q)),$$

2. *for any $q \notin P(CP_p[k])$,*

$$\forall\, CP_t[i] \in \mathcal{X}(CP_p[k]) : Consistent(CP_t[i], Current(q)).$$

Proof: From the definition of $\leadsto$ and the closure property of $\leadsto^*$.

$\square$

Corollary 1 *The set of recovery processes $P(CP_p[k])$ determined in Lemma* **??** *is the* **smallest** *set of processes that have to recover with respect to $CP_p[k]$, i.e. for any recovery line $RL(\mathcal{P})$ of a set of processes $\mathcal{P} \subseteq Pro$ that containes a checkpoint $CP_p[i]$ of process p such that $NotLater(CP_p[i], CP_p[k])$,*

$$P(CP_p[k]) \subseteq \mathcal{P}.$$

Proof: From Theorem **??** and Theorem **??**.

□

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 determination of the *smallest* and *latest* recovery in Section **??**. For the detection of redundant checkpoints with respect to this kind of recovery, it is enough to find for each process $p_i \in Proc$, the recovery domain of the *latest checkpoint* of process p_i. A checkpoint $CP_{p_i}[k_i]$, of process p_i is said to be the **active** (**latest**) checkpoint if $Active(CP_{p_i}[k_i]) = true$. As stated before, a checkpoint is redundant if after error detection in any process it will not be needed for a recovery operation. Formally speaking, 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

$$ARP = \bigcup_{CP_p[i] \in AR} Z(CP_p[i]),$$

where $AR =_{df} \{CP_{p_i}[k_i] | Active(CP_{p_i}[k_i])\}$, and AR is called the set of *active checkpoints*.

The reader should note that ARP and AR change dynamically during an execution of $C(G)$. The active recovery domain ARP contains the checkpoints which have to be restored in the worst case when all the processes in G fail simultaneously and then recover. Actually finding AR and ARP requires the use of techniques of designing protocols that are beyond the scope of this paper.

Similar to Lemma **??**, it is easy to show

Lemma 6 *The set*

$$Worst = \{CP_p[i] | CP_p[i] \in ARP \wedge (i = min\{j | CP_p[j] \in ARP\})\}$$

represents the recovery line for the set of processes

$$\mathcal{LP} = \bigcup_{CP_p[i] \in AR}(P(CP_p[i])).$$

Proof: See Appendix **??**.

$\square$

For each process $p \in Proc$, let Ar_p denote the active checkpoint of p, then we have the following corollary.

Corollary 2 *For each process $p \in Proc$, $\mathcal{X}(Ar_p)$ is the* **smallest** *and* **latest** *recovery line with respect to process p, i.e. for each recovery line $RL(\mathcal{P}')$ such that $p \in \mathcal{P}'$,*

$$P(Ar_p) \subseteq \mathcal{P}' \quad \textit{and}$$

$$\forall\, q \in P(Ar_p) \cap \mathcal{P}' :: (CP_q[k] \in \mathcal{X}(Ar_p) \wedge CP_q[j] \in RL(\mathcal{P}') \Rightarrow NotLater(CP_q[j], CP_q[k])).$$

Proof: From Theorem **??** and Corollary **??**.

$\square$

Next we give the definition of the *concatenation operation* of recovery lines. This operation is necessary if more than one error may simultaneously occur in different processes. The concatenation of two recovery lines is defined as follows:

$$\begin{aligned} RL(\mathcal{P}') \uplus RL(\mathcal{P}'') = {} & \\ & \{CP_p[k] | CP_p[k] \in ((RL(\mathcal{P}') \cup RL(\mathcal{P}'')) - (RL(\mathcal{P}') \cap RL(\mathcal{P}''))) \\ & \vee (\exists\, CP_p[k_1] \in RL(\mathcal{P}'), CP_p[k_2] \in RL(\mathcal{P}'') : (k = min\{k_1, k_2\}))\}. \end{aligned}$$

Corollary 3 *The concatenation of two recovery lines $RL(\mathcal{P}')$ and $RL(\mathcal{P}'')$ is the recovery line for set $\mathcal{P}' \cup \mathcal{P}''$, i.e.*

$$RL(\mathcal{P}') \uplus RL(\mathcal{P}'') = RL(\mathcal{P}' \cup \mathcal{P}'').$$

Corollary 4 *The active recovery line during any execution of G is the concatenation of recovery lines for sets of processes associated with the recovery domains of all active checkpoints of G, i.e.*

$$RL(Proc) = \biguplus_{CP_p[i] \in AR} RL(P(CP_p[i])).$$

The proofs of these last two corollaries are analogous to that of Lemma **??**.

□

From what has been achieved above, the *recovery function* can be implemented as a UNITY program R, which behaves in the following way. After the detection of an error in a process $p \in Proc$, the recovery program simultaneously restores the processes in the recovery line with respect to a checkpoint $CP_p[k]$ of process p with the corresponding checkpoints in this recovery line. It also restores the corresponding channels of the processes in $P(CP_p[k])$ with appropriate states that are determined by the checkpoints. When errors are detected in more than two processes and cause these processes to recover simultaneously, the restoration should be done with the concatenation of the recovery lines with respect to selected checkpoints of these processes. A good solution for this problem is to find the recovery lines with respect to the active checkpoints relevant to the failed processes. This solution has been proved to be the *smallest* and *latest* recovery in Corollary **??** and will be implemented as a UNITY program in the next section.

6 Recovery Program and Fault-tolerant Transformation

In this paper, it is assumed that all processes are *fail-stop* and that the occurrence of faults in the underlying system can be simulated by a UNITY program F_G which is called the *faulty program* and developed following the specification SP_F of the faults [**?**, **?**]. The faulty program is not considered in this paper.

In the specification SP_F of the faults of the system, there are predicates $\{F_p | p \in Proc\}$ such that $F_p = true$ (due to being modified by the faulty program F_G) when an error appears in process p. To fulfill the fail-stop assumption, each statement s_p in process p should be transformed to

$$s_p \quad \textbf{if} \quad \neg F_p,$$

and only the faulty program F_G can assign F_p to be $true$. Thus, for each process $p \in Proc$, $F_p = false$ is stable in the recovery program R and the underlying program G. Being $true$, F_p eventually causes R to restore a recovery line $RL(\mathcal{P}')$ to the corresponding processes in $\mathcal{P}' \subseteq Proc$ such that $p \in \mathcal{P}'$ and assign F_p to be $false$. Various techniques have been proposed for achieving the recovery line $RL(\mathcal{P}')$ with respect to a failed process and for the design of the recovery program [**?**, **?**, **?**]. This section gives a general scheme of backward recovery and a program for restoring the smallest and latest recovery line. Both of these are expressed by using UNITY. By doing this, the *fault-tolerant* transformation is introduced.

For cleanly expressing the specification and program of the recovery, the following notations are introduced. Let

$$\mathcal{CP} = \{CP_p[k_p] | p \in \mathcal{P}' \subseteq Proc\},$$

be a set of checkpoints such that for each process $p \in \mathcal{P}'$ there is exactly one checkpoint $CP_p[k_p]$ of p in $\mathcal{CP}$. $\mathcal{CP}|_{v.x}$ is used to denote the value $CP_p[k_p](v.x)$ for $x \in V_p$. For each pair of processes $p \in \mathcal{P}'$ and $q \in Proc$ such that $p \neq q$,

- $v.ch_{pq}(\mathcal{CP}) =_{df} CP_p[k_p](v.\overline{cs}_{pq}) - CP_q[k_q](v.\overline{cr}_{pq})$ if $q \in \mathcal{P}'$,
- $v.ch_{pq}(\mathcal{CP}) =_{df} CP_p[k_p](v.\overline{cs}_{pq}) - Current(\overline{cr}_{pq})$, if $q \notin \mathcal{P}'$, and
- $v.ch_{qp}(\mathcal{CP}) =_{df} Current(\overline{cs}_{qp}) - CP_p[k_p](v.\overline{cr}_{qp})$ if $q \notin \mathcal{P}'$.

For each process $p \in \mathcal{P}'$, $kc_p(\mathcal{CP}) =_{df} k_p$. $v.ch_{pq}(\mathcal{CP})$, $v.ch_{qp}(\mathcal{CP})$ and $kc_p(\mathcal{CP})$ can also be simplfied respectively as $v.ch_{pq}$, $v.ch_{qp}$ and kc_p when $\mathcal{CP}$ is explicitly given. Now a general program scheme of rocovery can be developed by using UNITY.

If for each subset of processes $\mathcal{P}' \subseteq Proc$, a recovery line $RL(R(\mathcal{P}'))$ can be found for a set of processes $R(\mathcal{P}') \subseteq Proc$ such that $\mathcal{P}' \subseteq R(\mathcal{P}')$, then after all the processes in $\mathcal{P}'$ fail (i.e. $F_p = true$, for each $p \in \mathcal{P}'$), the recovery program R restores the states saved in the recovery line $RL(R(\mathcal{P}'))$ to the corresponding processes in $R(\mathcal{P}')$ and sets F_p to be $false$ for each $p \in \mathcal{P}'$. Also, to guarantee **Assumption ??** in Section **??**, for each $p \in R(\mathcal{P}')$ and $CP_p[k] \in RL(R(\mathcal{P}'))$, the recovery program R has to restore each $v.x \in CV_p$ with $First(k, v.x)$. Therefore, the specification of the recovery program for G can be given as:

1. for any process $p \in Proc$, **Stable** $\neg F_p$,

2. for any subset $\mathcal{P}' \subseteq Proc$,
   $(\forall p \in \mathcal{P}' :: F_p) \longmapsto$

   (a) $(\forall p \in R(\mathcal{P}') :: (\forall x \in V_p : x = RL(R(\mathcal{P}'))|_{v.x}))$
   (b) $\wedge\ (\forall p \in R(\mathcal{P}'), q \in Proc : q \neq p :: ch_{pq} = v.ch_{qp})$
   (c) $\wedge\ (\forall p \in R(\mathcal{P}'), q \in Proc : q \neq p :: ch_{qp} = v.ch_{qp})$
   (d) $\wedge(\forall p \in R(\mathcal{P}') :: (\forall v.x \in CV_p : v.x = First(kc_p, v.x))$
   (e) $\wedge\ (\bigwedge_{p \in \mathcal{P}'} \neg F_p)$.

Condition **??** implies the restoration of the process variables of the recovering processes; Condition **??** and Condition **??** guarantee the consistency of the channel programs relevant to the recovering processes; Condition **??** implies **Assumption ??** and Condition **??** means the errors have been corrected and the failed processes resume their computation. This specification can be implemented as a UNITY program R_G: for a subset of processes $\mathcal{P}' \subseteq Proc$, let $F_{\mathcal{P}'} \equiv \forall p \in \mathcal{P}' : F_p$.

Program R_G:
$\langle [] \mathcal{P}' \subseteq Proc ::$
$\langle\langle \| \ p \in R(\mathcal{P}') :: \langle \| \ x \in V_p :: x := RL(R(\mathcal{P}'))|_{v.x}$ **if** $F_{\mathcal{P}'}\rangle\rangle$
$\| \ \langle \| \ p \in R(\mathcal{P}'), q \in Proc : q \neq p :: ch_{pq} := v.ch_{pq}$ **if** $F_{\mathcal{P}'}\rangle$
$\| \ \langle \| \ p \in R(\mathcal{P}'), q \in Proc : q \neq p :: ch_{qp} := v.ch_{qp}$ **if** $F_{\mathcal{P}'}\rangle$
$\| \ \langle \| \ p \in R(\mathcal{P}') :: \langle \| \ v.x \in CV_p :: v.x := First(kc_p, v.x)$ **if** $F_{\mathcal{P}'}\rangle\rangle$
$\| \ \langle \| \ p \in \mathcal{P}' :: F_p := false$ **if** $F_{\mathcal{P}'}\rangle\ \rangle$
$\rangle$
End$\{R_G\}$

Therefore, for each concrete technique of finding the recovery line for a set of failed processes, a UNITY program can be obtained from the above scheme for implementing the recovery function. Two cases are given as examples in the following.

- A trivial case is that the checkpointing program only records the initial state of each process $p \in Proc$. If an error occurs in a process, the recovery line is $RL(Proc)$ for $Proc$ that contains all the checkpoints taken by the processes. R_G makes the program G restart from the very begining after an error detection.

- Another case is that the checkpoint program has all the processes in G take checkpoints simultaneously each time. For each failed process, the recovery line is the set of the latest checkpoints of all the processes. R_G makes program G recover from the latest checkpoints. In this case, it is sufficient to save only the latest checkpoint for each process.

Techniques, such as *recovery blocks*, *conversations*, the checkpointing protocol and recovery protocol, for checkpointing and recovery in [**?**, **?**] could be fit into this scheme. For instance, in the case of *conversations*, the checkpointing program has a process of a *conversation* take a checkpoint whent it takes part in the conversation. When a process in a conversation failed, the recovery line in the set of checkpoints taken at the begining of it for all the proceses of the conversation. In the following, a solution for recovering from the *smallest* and *latest* recovery line is achieved.

Let $\mathcal{P}'$ be a subset of $Proc$, a recovery line $\mathcal{X}(\mathcal{P}')$ for the set $R(\mathcal{P}')$ of processes can be defined as:

$$\mathcal{X}(\mathcal{P}') = \biguplus_{p \in \mathcal{P}'} \mathcal{X}(Ar_p)$$

and

$$R(\mathcal{P}') = \bigcup_{p \in \mathcal{P}'} P(Ar_p).$$

Thus, a specification can be obtained as:

1. for any process $p \in Proc$, **Stable** $\neg F_p$,

2. for any subset $\mathcal{P}' \subseteq Proc$,
   $(\forall\, p \in \mathcal{P}' :: F_p) \longmapsto$

   (a) $(\forall\, p \in R(\mathcal{P}') :: (\forall\, x \in V_p : x = \mathcal{X}(\mathcal{P}')|_{v.x}))$
   (b) $\wedge\ (\forall\, p \in R(\mathcal{P}'), q \in Proc : q \neq p :: ch_{pq} = v.ch_{pq})$
   (c) $\wedge\ (\forall\, p \in R(\mathcal{P}'), q \in Proc : q \neq p :: ch_{qp} = v.ch_{qp})$
   (d) $\wedge(\forall\, p \in R(\mathcal{P}') :: (\forall\, v.x \in CV_p : v.x = First(kc_p, v.x))$
   (e) $\wedge\ (\bigwedge_{p \in \mathcal{P}'} \neg F_p)$.

And this specification can be implemented as a UNITY program R'_G:

Program R'_G:

$\langle [] \mathcal{P}' \subseteq Proc ::$
$\langle \langle \| \ p \in R(\mathcal{P}') :: \langle \| \ x \in V_p :: x := \mathcal{X}(\mathcal{P}')|_{v.x} \ \textbf{ if } \ F_{\mathcal{P}'} \rangle \rangle$
$\| \ \langle \| \ p \in R(\mathcal{P}'), q \in Proc : q \neq p :: ch_{pq} := v.ch_{pq} \ \textbf{ if } \ F_{\mathcal{P}'} \rangle$
$\| \ \langle \| \ p \in R(\mathcal{P}'), q \in \mathcal{P} : q \neq p :: ch_{qp} := v.ch_{pq} \ \textbf{ if } \ F_{\mathcal{P}'} \rangle$
$\| \ \langle \| \ p \in R(\mathcal{P}') :: \langle \| \ v.x \in CV_p :: v.x := First(kc_p, v.x) \ \textbf{ if } \ F_{\mathcal{P}'} \rangle \rangle$
$\| \ \langle \| \ p \in \mathcal{P}' :: F_p := false \ \ \textbf{ if } \ F_{\mathcal{P}'} \rangle \ \ \rangle$
$\rangle$

End$\{R'_G\}$

Viewing what has been achieved, given a program G within the model in Section **??**, let F_G be the program, called the *faulty program* with respect to program G, that simulates the faults of the underlying *fault-prone* system. The effect of the specified faults can be treated as a transformation $\mathcal{F}$ on programs defined as:

$$\mathcal{F}(G) =_{df} (G', F_G),$$

where G' is the program obtained by transforming each statement s_p in any process p of G into

$$s_p \quad \textbf{if} \quad \neg F_p.$$

As stated in Section **??**, this transformation makes the underlying program fail-stop. $\mathcal{F}(G)$ means that the execution of G' on the specified *fault-prone* system is equivalent to the execution of the composed program $G'[]F_G$ on a corresponding *fault-free* system. Similarly, the effect of the checkpointing can be treated as a transformation $\mathcal{C}$ with a fail-stop program and the specified faults as parameters, and the recovery can be treated as a transformation $\mathcal{R}$ taking a program with specified checkpoint program and the specified faults as parameters. These two transformations are defined respectively as:

$$\mathcal{C}(\mathcal{F}(G)) =_{df} (G'[]C_G, F_G),$$

and

$$\mathcal{R}(\mathcal{C}(\mathcal{F}(G)) =_{df} (G'[]C_G[]R_G, F_G).$$

$\mathcal{C}(\mathcal{F}(G))$ means that the execution of program $G'[]C_G$ on the specified *fault-prone* system is equivalent to the execution of $G'[]F_G[]C_G$ on the corresponding *fault-free* system. $\mathcal{R}(\mathcal{C}((\mathcal{F}(G), F_G)))$ means that the execution of program $G'[]C_G[]R_G$ on the specified *fault-prone* system is equivalent to the execution of $G'[]F_G[]C_G[]R_G$ on the corresponding *fault-free* system. From the specification of C_G and R_G, $G'[]C_G[]R_G$ satisfies **Assumption ??** and therefore satisfies all the results obtained in Section **??**. Program $G'[]C_G[]R_G$ is called the **fault-tolerant program** of G. It is easy to see that the transformation $\mathcal{C}$ does not depend on the specification of the faults (the faulty program), but the transformation $\mathcal{R}$ should be given with respect to the specified faults and the checkpoint program as well. The composition of the three transformations, $\mathcal{FT} =_{df} \mathcal{R} \circ \mathcal{C} \circ \mathcal{F}$, can be defined as follows:

$$\mathcal{FT}(G) = \mathcal{R}(\mathcal{C}(\mathcal{F}(G))),$$

and $\mathcal{FT}$ is called the **fault-tolerant transformation**.

To summarize the main ideas of this section, we simplify the notations of the above transformation by omitting the second components F_G on the right hand side of their definitions. Given a program G of the proposed model and the specified (physical) faults (the faulty specification or program) F_G, G is firstly transformed into G' given in the definition of $\mathcal{F}$. The execution of G' on the specified *fault-prone* system is equivalent to the execution of the program $G'[]F_G$ on the corresponding *fault-free* system. However, execution of $G'[]F_G$ on a fault-free system may not be correct with respect to the specification of G because of the effect of F_G. F_G may violate the progress properties of G. Thus a function which corrects the effect of F_G is needed. This function is implemented as recovery based on restoration of the system with states previously reached and saved. For specified faults, the checkpointing can be treated systematically as a transformation with a program in this model as a parameter, whereas recovery can be treated systematically as a transformation with a program in this model, specified faults and checkpointing program as parameters. Actually, given G in the model and specified faults F_G, the program obtained after the transformations $\mathcal{F}$, $\mathcal{C}$ and $\mathcal{R}$ should be

$$\mathcal{FT}(G) =_{df} G'[]C_G[]R_G.$$

The execution of $\mathcal{FT}(G)$ on the specified *fault-prone* system is equivalent to the execution $G'[]F_G[]C_G[]R_G$ on the corresponding *fault-free* system. Given a program G with original specification, we say that the fault-tolerant program

$G'[]C_G[]R_G$ can tolerate the specified faults F_G, if $G'[]F_G[]C_G[]R_G$ can provide the *same service* as G on fault-free system. Therefore, the argument of the correctness of fault-tolerant programs has been converted to that of usual programs.

7 Correctness of the Fault-tolerant Transformantion

From the semantic model given in [**?**, **?**] and the definition of the fault-tolerant transformation in Section **??**, the faulty program F_G has no effect on the checkpoint program C_G and the recovery program R_G, i.e. F_G cannot interrupt the execution of the programs C_G and R_G. This means that *we have no guards for the guards*. Obviously, after being refined to fit the model, the fault-tolerant programs can be transformed further by the fault-tolerant transformation. This introduces *hierachical fault-tolerance*. In this section, the correctness of the fault-tolerant transformation is defined under this assumption.

Let $Init_G$ and $Init_{\mathcal{FT}(G)}$ be the initial states of G and $\mathcal{FT}(G)$. The correctness of the fault-tolerant transformation $\mathcal{FT}$ is defined as follows: for any state S of program G, and finite statement sequence $\boldsymbol{u}$ in G, there exists a state S' of program $G'[]F_G[]C_G[]R_G$ and a finite satement sequence $\boldsymbol{u}'$ in $G'[]F_G[]C_G[]R_G$ such that $S'|_{Var(G)} = S$ and $S'(F_p) = false$ for each $p \in Proc$, and

$$[Init_G]\boldsymbol{u}[S] \Rightarrow [Init_{\mathcal{FT}(G)}]\boldsymbol{u}'[S'].$$

Conversely, for any state S' of program $G'[]F_G[]C_G[]R_G$ such that $S'(F_p) = false$ for each $p \in Proc$, and finite statement sequence $\boldsymbol{u}'$ in $G'[]F_G[]C_G[]R_G$, there exist a state S of program G and a finite satement sequence $\boldsymbol{u}'$ in G such that $S'|_{Var(G)} = S$, and

$$[Init_{\mathcal{FT}(G)}]\boldsymbol{u}'[S'] \Rightarrow [Init_G]\boldsymbol{u}[S].$$

The proof of correctness is based on the assumption that faults can only happen finitely often, i.e. statements in F_G can be selected to execute only finitely many times. This assumption can be defined as the **bounded fairness** condition in [**?**]. Therefore, the semantics of the fault-tolerant program $G'[]F_G[]C_G[]R_G$ is defined as the set of the executions that are *fair* with respect to $G'[]C_G[]R_G$ and *bounded* with respect to F_G. From the construction of $\mathcal{FT}$, no checkpoint can be taken after G reaches its fixed point and no recovery action takes place if no fault occurs. Therefore, based on lemmas, theorems and corollaries given in Section **??**, the correctness of the fault-tolerant transformation $\mathcal{FT}$ and the termination of the fault-tolerant program can be easily proved with this semantics.

8 Discussion

Channel faults and recovery from channel faults were not mentioned in this paper. Faults of the system were implicitly assumed not to affect the execution of the channel programs in the underlying program G. Since channel faults, e.g. message lost, message duplication and message corruption, often occur in distributed systems, techniques for dealing with them are required. To cope with recovery from channel faults within the model given in this paper, channel faults can be treated as faults in the processes connected by the the faulty channels. This mainly involves with the specification of faults. The model of recovery, however, is not required to be changed or extended. For example, a lost message requires the sending process to re-send the message by recovering from a checkpoint. Therefore, message loss can be treated as fault in the sending process.

The correctness of the fault-tolerant transformation was formally defined in Section **??**. From what we have got, we can see that the UNITY program model is powerful enough to express hwat shoulbe done by checkpointing and recovery. However, because the UNITY logic is based on *justice* condition, it is not powerful enough for specifying and verifying the properties of fault-tolerant programs. The recovery program may change the progress properties of the underlying program and in general, it my not be possible to achive any stable property. Progress properties, p **Ensures** q and $p \longmapsto q$, of the underlying program G should be converted to p **Ensures** $(q \vee Recovers)$ and $p \longmapsto (q \vee Recovers)$. These are not progress properties without any constraints on faults. The predicate $Recovers$ denotes that the recovery program is initiated. With the assumption that faults can occur only finitely often, property that p *will be eventually stable* is required but cannot be specified with the UNITY logic. Therefore, it will be required to extend the UNITY logic to cope with the specification and verification of fault-tolerant programs. It should be noted that such extention is not the problem of UNITY but that of the semantics and implementation.

Acknowledgment
I wish to thank my supervisor, Prof. Mathai Joseph, for his constant encouragement, guidance, and his comments on earlier versions of this report as well. My

thanks also go to my colleagues, A. Goswami and A.M. Lord, for the helpful discussion during this work.

Appendix:

A The proof of Lemma 4:

It is sufficient to show that for each process $q \notin \mathcal{P}'$ and any checkpoint $CP_p[k] \in RL(\mathcal{P}')$, $SemiCon(CP_p[k], Current(q))$. If $\neg SemiCon(CP_p[k], Current(q))$ for some $CP_p[k] \in RL(\mathcal{P}')$, there must be a checkpoint $CP_q[i]$ (e.g. $CP_q[1]$) of process q such that $SemiCon(CP_p[k], CP_q[i])$. Let $j = max\{i | SemiCon(CP_p[k], CP_q[i])\}$, then $CP_p[k] \rightsquigarrow CP_q[j]$ and thus, $q \in \mathcal{P}'$. This contradicts $q \notin \mathcal{P}'$.

B The proof of Lemma 5:

It is necessary to show the following conditions:

1. $\forall p \in \mathcal{P}' : \exists! CP_p[k] \in RL(\mathcal{P}')$,
2. $\forall CP_p[i], CP_q[j] : ((CP_p[i] \in RL(\mathcal{P}')) \wedge (CP_p[i] \rightsquigarrow^* CP_q[j]) \Rightarrow q \in \mathcal{P}')$,
3. $Consistent(RL(\mathcal{P}'))$.

Condition **??** is satisfied because that for each process $q \in P(CP_p[k])$, $\mathcal{X}(CP_p[k])$ contains the earliest checkpoint of q that belongs to $\mathcal{Z}(CP_p[k])$.
Condition **??** can be derived directly from the definitions of the following sets, $\mathcal{Z}(CP_p[k])$, $\mathcal{X}(CP_p[k])$ and $P(CP_p[k])$.
Now we show Condition **??**, for any checkpoints $CP_q[i], CP_t[j] \in \mathcal{X}(CP_p[k])$ such that $CP_q[i] \neq CP_t[j]$, it is only required to prove $SemiCon(CP_q[i], CP_t[j])$. If $\neg SemiCon(CP_q[i], CP_t[j])$, then from Lemma **??**, $\neg SemiCon(CP_q[i], Current(t))$ and there exists a checkpoint $CP_t[j']$ such that $Before(CP_t[j'], CP_t[j])$ and $CP_q[i] \rightsquigarrow CP_t[j']$. Since $CP_p[k] \rightsquigarrow^* CP_q[i]$, $CP_p[k] \rightsquigarrow^* CP_t[j']$. Therefore, $CP_t[j'] \in \mathcal{Z}(CP_p[k])$. This contradicts the fact that $CP_t[j]$ is the earliest checkpoint of process t in $\mathcal{Z}(CP_p[k])$. Thus, $SemiCon(CP_q[i], CP_t[j])$ must be $true$ and Lemma **??** holds.

C The proof of Theorem 5.2:

From Lemma **??**, since $Consistent(\mathcal{CP})$, for any processes $q, t \in P(CP_p[k])$ and checkpoint $CP_q[i]$ such that $q \neq t$,

$$NotLater(CP_q[n_q], CP_q[i]) \Rightarrow SemiCon(CP_q[i], CP_t[n_t]).$$

Particularly, $SemiCon(CP_p[k], CP_q[n_q])$ for each process $q \in P(CP_p[k])$ such that $q \neq p$.
Using induction, we show that for each checkpoint $CP_q[i]$ of a process q, if $CP_p[k] \leadsto^* CP_q[i]$, then $NotLater(CP_q[n_q], CP_q[i])$.
It is trivial for the case that $CP_p[k] \leadsto^* CP_p[k]$.
If $CP_p[k] \leadsto CP_q[i]$, from Lemma **??** and the definition of the relation $\leadsto$, $NotLater(CP_q[n_q], CP_q[i])$.
Suppose that $CP_p[k] \leadsto^* CP_t[j] \leadsto CP_q[i]$ and $NotLater(CP_t[n_t], CP_t[j])$, it is required to prove that $NotLater(CP_q[n_q], CP_q[i])$. Since $NotLater(CP_t[n_t], CP_t[j])$, so $SemiCon(CP_t[j], CP_q[n_q])$. Again from Lemma **??** and the definition of $\leadsto$, we have $NotLater(CP_q[n_q], CP_q[i])$.
Since $CP_p[k] \leadsto^* CP_q[m_q]$ for each checkpoint $CP_q[m_q] \in \mathcal{X}(CP_p[k])$, therefore, we have $NotLater(\mathcal{CP}, \mathcal{X}(CP_p[k]))$.

D The proof of Lemma 6:

Similar to the proof of Lemma **??**, it is necessary to show the following conditions:

1. $\forall p \in \mathcal{LP} : \exists! CP_p[k] \in Worst$,
2. $\forall CP_p[i], CP_q[j] : ((CP_p[i] \in Worst) \wedge (CP_p[i] \leadsto^* CP_q[j]) \Rightarrow q \in \mathcal{LP})$,
3. $Consistent(Worst)$.

Conditions **??** and **??** 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 $SemiCon(CP_p[i], CP_q[j'])$ ($CP_q[1]$ is such a checkpoint). Let $CP_q[k]$ be the latest checkpoint of q such that $SemiCon(CP_p[i], CP_q[k])$, i.e. $k = max\{j' | SemiCon(CP_p[i], CP_q[j'])\}$, then $CP_p[i] \leadsto CP_q[k]$. Thus,

$CP_q[k] \in ARP$. On the other hand, from $\neg SemiCon(CP_p[i], CP_q[j])$ and Lemma **??**, $Before(CP_q[k], CP_q[j])$ must be $true$. This contradicts the fact that $j = min\{i|CP_q[i] \in ARP\}$. 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 $true$.