Partial Order Based Runtime Recovery Intended For Highly Available Distributed Applications

Ching Wei Su

A Thesis

In The Department of

Computer Science

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Computer Science at

Concordia University

Montreal, Quebec, Canada

June 2011

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: Ching Wei Su

Entitled: Partial Order Based Runtime Recovery Intended

For Highly Available Distributed Applications

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____Chair

_____Examiner

_____Examiner

_____Supervisor

Approved by _____
Chair of Department or Graduate Program Director

_____2011 _____
Dean of Faculty

Abstract

Partial Order Based Runtime Recovery

Intended For Highly Available Distributed Applications

Ching Wei Su

This thesis develops a checkpoint based runtime rollback recovery technique intended to be used with highly available distributed applications whose correct expected behavior is specified by the application developer through a partially ordered multiset (POMSET) of application events.

Checkpoint based rollback recovery techniques for distributed applications typically store the state of all the application processes and application events (called the global checkpoints) in persistent store at periodic time intervals. When a runtime failure is detected, the application is rolled back to an appropriate correct past state in its execution using the saves checkpoints. Such techniques do not know the correct application behavior and hence have to store large amount of state requiring significant amount of persistent storage and recovery time. The idea behind this thesis is that knowing the correct expected behavior of the application, only the checkpoints necessary to ensure runtime recovery of the application can be identified and stored thereby making the recovery much more efficient.

The application developer specifies the correct expected behavior of the distributed application through a POMSET of application events which is stored as a tree. The developed runtime recovery technique identifies the nodes of the POMSET tree at

which a checkpoint must be taken to ensure recovery. In addition, instead of storing the states of all the processes in the application (the global state), this technique only stores the states of the processes that are necessary to recover from a potential failure as a collection of local checkpoints (called group checkpoints). Furthermore, unnecessary checkpoints are avoided by appropriate analysis of the POMSET tree before execution (called static checkpoint reduction) and by predicting the necessary checkpoints for iterative executions at runtime (called dynamic checkpoint reduction) based on past execution.

A prototype implementation of the runtime recovery technique is developed and it shows that the technique has very little performance impact on the application. The technique is illustrated with a practical example application for online travel agency. Experimental results from the prototype implementation on this example application show that the developed technique saves about 65% of the persistent store to save the checkpoints. Qualitative analysis of the developed technique shows that the required recovery time is significantly reduced in comparison with the traditional recovery technique.

Acknowledgements

It is a pleasure to express my gratitude to many people who made this thesis possible.

First, it is so difficult to overstate my gratitude to my master supervisor, Dr. Jayakumar, for his enthusiasm, his inspiration, and his great efforts to help me in so many ways. Throughout my thesis-writing period, he provided me great encouragement, advice, and lots of good ideas. I would have been lost without him.

Also, I would like to thank my sweet family for providing a loving environment for me. My boyfriend and my mom have been particularly supportive in these years. To them I dedicate this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1 : INTRODUCTION

The goal of this thesis is to develop a runtime rollback recovery technique to be used with highly available distributed applications based on the partially ordered multi-set model and a prototype implementation of the technique. This chapter motivates the research and highlights the contributions.

1.1 Motivation

With the advent of the internet, distributed applications integrating a collection of autonomous services (processes) running on geographically distributed hosts (computers) into a single unified software system have become a practical reality. Such applications should be highly available (with little or no down time) in order to be accessed and used around the clock. Furthermore, these highly available distributed applications may not be of any value if they cannot guarantee correct operation at all times without any human intervention. Thus, approaches for the design and implementation of such reliable and highly available distributed applications have significant practical importance.

Highly available distributed applications usually employ two approaches: failure masking or failure recovery. An application employing failure masking is normally designed to produce correct result in the presence of incorrect operation (failure) by concurrently producing the result in multiple ways (using replicated processes) and then selecting and returning the correct result (using a voting process) [RKSC06, SMNTWB02, RR06, OFG07]. Thus, such applications require multiple replicas of the application processes running in different hosts thereby increasing the resource

requirements, cost and complexity of the application. On the other hand, an application employing failure recovery is normally designed to detect the occurrence of an incorrect result (failure) and correct the result by restoring the application to a previous correct point (called checkpoint) in its execution and then replaying it by applying all the inputs since that correct point [CR72, CR92, EX92, CY96, EA02]. Thus, such applications need to keep track of the execution and maintain the checkpoints and the necessary information for replay.

Failure detection and correction in distributed applications can be achieved by properly modeling the correct execution of the application. Two commonly used such models are the state space model [LHLL08] and the partially ordered multiset (POMSET) model [AM94, MG03, RG04]. In the state space model, the execution of an application is modeled by the state of the application (events that have happened) during the execution. The size of such states (and hence the information maintained by the application in persistent store) can grow exponentially in a distributed system leading to the well-known state explosion problem [YUA88, BCMDH 90]. In the POMSET model, on the other hand, the execution is modeled by the order in which events could happen in the application. By properly defining the events of interest (for example, by grouping a number of events that should all occur or none should occur into an atom of events), it is possible to significantly reduce the amount of information necessary to be maintained [LM07, LMG07] thereby alleviating the state explosion problem.

An approach for runtime verification of distributed applications based on the POMSET model has recently been developed [GAO10]. In this approach, the correct execution of the application is specified as a POMSET of atoms of events (atoms in

short) that should happen within the processes. As the application is running, the processes report the atoms happening within themselves (by properly modifying or instrumenting the source code of the processes to do so) to a monitor that compares them with the specified POMSET. When the monitor notices that an atom violates the given specification, it flags it as a failure in that execution. This thesis extends that runtime verification approach to a runtime recovery approach using the checkpoint recovery mechanism.

Typical checkpoint based rollback recovery techniques take and maintain checkpoints at periodic time intervals and the system is rolled back to the nearest correct checkpoint when a failure is detected. Quite a few of these periodic checkpoints may not be used during the recovery and maintaining them uses large amount of persistent storage. The proposed runtime recovery approach, on the other hand, maintains only the necessary checkpoints (that will be necessary for roll back) based on the POMSET specification and restoring the application to the last checkpoint when a runtime failure happens. This is done by first analyzing the correct expected behavior of the application as modeled in the given POMSET specification and identifying all the required checkpoints. As the application executes, only the necessary checkpoints (from the identified set) are maintained so that if and when a runtime failure happens, the system is rolled back to the last checkpoint and replayed with the necessary inputs to correct the error. The number of such group checkpoints taken and maintained are further reduced by avoiding some of the unnecessary ones based on the priorities of the operator nodes and their positions in the POMSET tree (static checkpoint reduction) and by forecasting whether a checkpoint is necessary or not in a future iteration based on what happened in

the past iterations (dynamic checkpoint reduction). Thus, the proposed runtime recovery approach is more efficient as it maintains only the necessary checkpoints (thereby reducing the amount of persistent storage required) and easily performs the roll back to the last checkpoint (thereby improving the roll back time).

The developed technique is implemented as a proof of concept prototype. The design and implementation of the prototype will be presented in detail explaining many of the issues and problems faced. The developed runtime recovery technique is also highlighted using a highly available online travel agent application. This application will be introduced in Chapter 2 and used throughout the thesis to illustrate the important concepts, issues and their solutions.

1.2 Contribution

This thesis develops an automatic runtime recovery technique using the checkpoint based rollback recovery technique. The developed technique is also implemented as a proof of concept monitor that automatically handles the runtime recovery. The major contributions of this thesis are as follows.

- First, since POMSET specifies the correct behavior of processes in the distributed application, an approach is developed to analyze the POMSET specification and identify the required checkpoints for the given application.
- Secondly, since not all the required checkpoints may be necessary, two algorithms to identify and avoid the unnecessary checkpoints are developed – the static checkpoint reduction algorithm and the dynamic checkpoint reduction algorithm.

- Furthermore, a proof of concept implementation of the runtime recovery monitor that follows the POMSET based model checking algorithm is developed. This monitor verifies the ordering of atoms and the correctness of execution by comparing the execution atoms received from the processes with the expected execution specified in the POMSET specification. Once a failure is detected, the runtime recovery monitor automatically and immediately rolls back the application to the last checkpoint. In addition to reducing the size of the state space to be searched in partial-ordered checking, the runtime monitor only allows the last checkpoint to be kept by discarding previous checkpoints during execution runs.

1.3 Outline

This thesis is organized as follows.

Chapter 2 discusses related previous works on rollback recovery protocols, including checkpoint-based recovery and log-based recovery. It also describes the runtime verification approach based on the partial-order model.

Chapter 3 explains the runtime recovery technique based on the POMSET model and introduces the properties of operators in POMSET specification. It also presents the static checkpoint reduction algorithm and the dynamic checkpoint reduction algorithm.

Chapter 4 shows an implementation of the runtime recovery technique, including the deployment environment, the instrumented program, atoms, events, and status. It also describes how to persistently store messages, checkpoints and execution runs. The developed runtime recovery approach is illustrated using an online travel agency

example. The various steps in the runtime recovery process are clearly described for this highly available application.

Chapter 5 concludes the thesis by pointing out issues and future work to improve the runtime recovery technique.

CHAPTER 2 : PARTIAL ORDER MODEL AND CHECKPOINT BASED RECOVERY

The objective of this chapter is to describe the partially ordered multi-set (POMSET) specification of the correct behavior of distributed applications and the runtime verification technique using that model. Traditional checkpoint based runtime recovery approaches and their characteristics are also presented.

2.1 Partial Order Model

The execution of highly available distributed applications involves parallel/ concurrent processes that cooperate to perform assigned functionality. Due to the concurrent execution, comprehensive specification and analysis of the correct behavior in distributed applications consisting of multiple processes/threads is a challenge. While the execution of sequential programs can be described by a vector of events, the correct behavior of a concurrent distributed application should be properly presented using a partially ordered set of events (the partial order model) [AM94, MG03, RG04], which is adopted from the distributed event model and allows unrelated/independent events to occur concurrently in distinct processes.

In the partial-ordered multi-set (POMSET) model [LM07, LMG07], an event is an instance of action corresponding to a statement of execution and the low-level events can be compressed into a fixed number of atoms corresponding to abstract events for higher granularity. An atom consists of a set of events within a process that should occur atomically (that is, all should occur or none should occur). The POMSET model specifies

the partial order in which these atoms should occur during the execution of the distributed application.

Consider an online travel agent application as an example. This is a highly available distributed application that sells and provides information about travel, transportation and accommodation and it can be treated as a virtual agent of multiple flight, rail, coach companies, and accommodation agencies. Moreover, it is accessible by multiple customers who are interested in all kinds of travel information about different flights or buying tickets online.

There are four processes in the online travel agent application and each of them represents one role as listed in Table 1. Each process in this application has its own role functionality, such as users, virtual agent, hotel and transportation reservation services. When a virtual agent receives a user request, it asks the hotel and the transportation services for booking information, and gathers and reports this information as multiple journey choices back to the user. When the user chooses one preferred journey from among these choices, the virtual agent atomically performs the reservation of both hotel and transportation, and then notifies the user the success/failure of the reservation.

A process may be divided into multiple atoms according to its functionality. All primitive events in this travel agent application are grouped into 21 atoms (*A* to *U*) as listed in Table 2. For instance, atom *A* sends a user request for journey choices, while atom *B* represents a user already having a preferred journey so the virtual agent does not need to gather booking information. Note that an atom is classified as a critical atom if it computes and stores some information thereby changing the state of the application; otherwise, it is a non-critical atom.

Table 1: Role List of the Online Travel Agent Application

Roles	Description
Customer	A customer sends a booking request to the agency and waits for response.
Agency	An agency receives requests from a customer, gathers booking information from the hotel and the transportation services, and reserves/cancels booking.
Hotel	There are two hotel services for reservation and cancellation – H1 and H2.
Transportation	There are two transportation services for reservation and cancellation – T1 and T2.

Table 2: Atom List of Travel Agency

Atom	Critical Atom?	Role	Description	Input Message	Output Message
A	No	Customer	A customer sends a request for available hotel and transportation reservation information.	N/A	M1 – Customer request with travel information

B	No	Agency	An agency receives the request and asks the hotels and the transportation services for booking information.	M1	M2, M3, M4, M5 – Agency requests with customer travel information
C	No	Hotel H1	Hotel services query database and provide current booking list.	M2	M6 – Hotel H1 booking list
D	No	Hotel H2		M3	M7 – Hotel H2 booking list
E	No	Transportation T1	Transportation services query database and provide current booking list.	M4	M8 – Transportation T1 booking list
F	No	Transportation T2		M5	M9 – Transportation T1 booking list
G	No	Agency	Gather all booking lists from hotel and transportation services and return those to customer as choices.	M6, M7, M8, M9	M10 – gathered booking list

H	Yes	Customer	A customer chooses one hotel with transportation as travel routines.	M10	M11 – Customer's choice of hotels and transportation
I	No	Customer	A customer already has preferred hotel with transportation as travel routines.	N/A	M12 – Customer's preferred choice of hotels and transportation
J	Yes	Agency	An agency receives customer's choice tagged with one unique number and records it in database.	M11 or M12	M13 – booking request for hotel M14 – booking request for transportation
K	Yes	Hotel H1	Hotel services book the room for the customer and returns the result of booking.	M13	M15 – result of booking (Yes for success, No for failure)
L	Yes	Hotel H2			

M	Yes	Transportation T1	Transportation services book the vacancy for the customer and returns the result of booking.	M14	M16 – result of booking (Y for success, N for failure)
N	Yes	Transportation T2			
O	Yes	Agency	An agency receives the results of booking from hotel and transportation services, and check whether all bookings for the customer are done.	M15, M16	2 messages (M13 and M14 to repeat reservation) or 1 message (M17 for reservation success) or 3 messages (M18, M19, M20 for reservation failure) M17 – reservation successfully

					with reservation number M18 – reservation failed with failure messages M19 – cancellation request for hotel M20 – cancellation request for transportation
P	No	Customer	Notify Customer that all travel routines, including hotels and transportation, have done successfully. A customer should	M17	N/A

			receive the unique number of reservation.		
Q	No	Customer	Notify customer that the travel routines, including hotels and transportation, have failed of reservation with fail messages.	M18	N/A
R	Yes	Hotel H1	Hotel services receive cancellation messages from agency, check booking list, and cancel the room, if that has been booked for the customer.	M19	N/A
S	Yes	Hotel H2			
T	Yes	Transportation T1	Transportation services receive cancellation messages from agency, check booking list, and cancel the vacancy, if that has been booked for the customer.	M20	N/A
U	Yes	Transportation T1			

The correct behavior of this application can be specified by the regular expression *((A ; B ; (C || D || E || F) ; G ; H) + I) ; J ; (((K + L) || (M + N)) ; O) * ; (P + (Q || (R + S) || (T + U))* which is the partially ordered multiset (POMSET) of atoms. This specification says that the customer first sends the request (*A*), the agency receives the request (*B*), then the agency concurrently queries the hotel services (*C*, *D*) and the transportation services (*E*, *F*), gathers the result and report to the customer (*G*) who then selects a preferred choice (*H*), and so on. The POMSET specified in this regular expression can be visually shown (and stored in memory at runtime) as the POMSET tree illustrated in Figure 1. In the POMSET tree, an alphabetic node (a leaf node) is an abstract event (atom); while an operation node (an internal node) specifies the ordering of two/multiple atom sets. For example, a concatenation operation node (;) denotes the partial order relationship between two atom sets: the left atom set must occur before the right atom set. So *A*;*B* specifies that atom *A* should occur before atom *B*.

While the POMSET indicates the order in which atoms should occur at runtime, the correctness of the computation performed by the individual atoms also needs to be specified. This is typically done by specifying a program invariant (called a predicate) for an atom, and the variables computed by the atom (called predicate variables) that are needed to check the invariant. Thus, knowing the predicates of individual atoms, the correctness of their computation can be verified at runtime using the predicate variables. For example, if atom *J* does not receive the predicate variable (a preferred journey choice containing hotel and transportation information) this behavior does not satisfy the global correctness property.

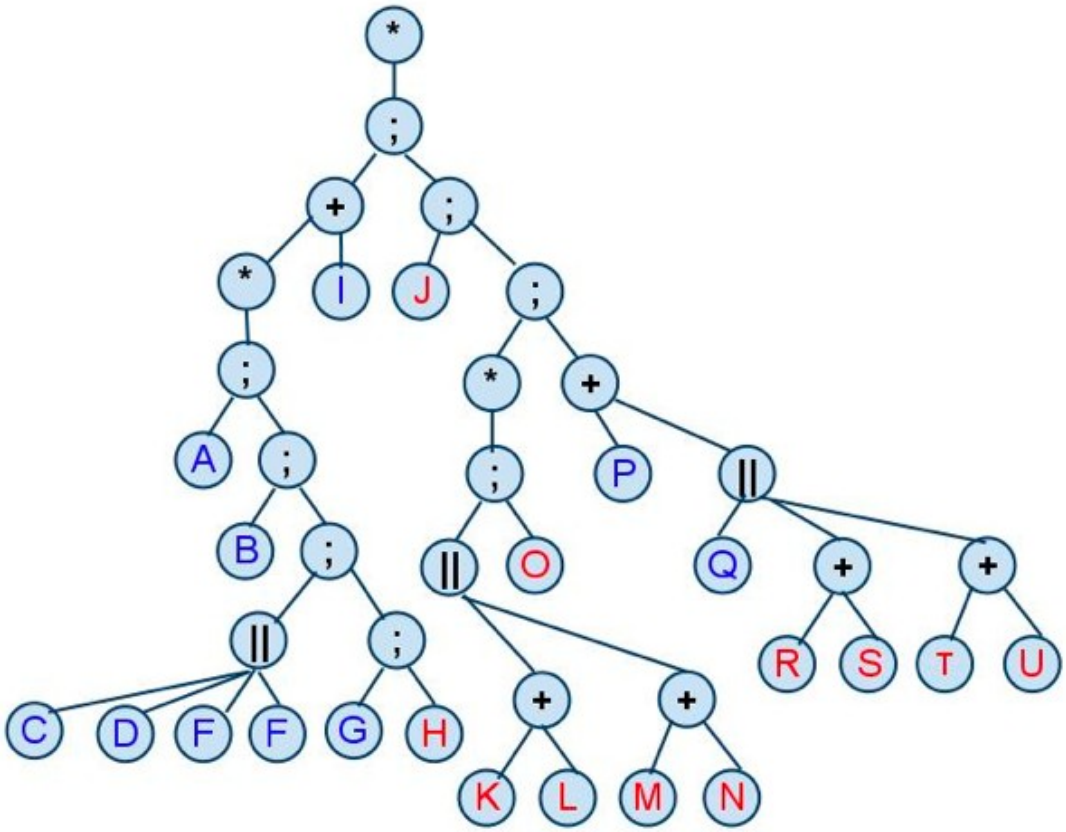


Figure 1: The POMSET tree of the online Travel Agent Application

2.2 Runtime Verification

The POMSET specification can be used to verify the correct execution of a distributed application as it runs. This runtime verification is done by checking if all the atoms occur according to the partial order specified by the POMSET. In order to do that, the application processes should be modified (instrumented) by adding the necessary program code to report the occurrence of the program events to a monitor which will check them with the POMSET specification. For example, during a run of the travel agent application, if atom *J* occurs before atom *A* or *I*, the runtime verification monitor flags a failure. Also, if *J* does not receive a customer preference of journey (predicate

variable), a failure would be detected because of the unsatisfied predicate. Thus, partial order based runtime verification concerns two fundamental requirements – the ordering among atoms and the correctness of computation performed by each atom. Notice that the necessary state space of this partial order model based verification can be dramatically reduced from the number of low-level program events to the number of granular atoms.

This runtime verification approach has been recently introduced [LM07, LMG07] and implemented as a partial order based runtime verification tool [GAO10]. In that tool, the application programmers are required to specify the proper set of atoms, the POMSET specification of the correct behavior of the application, the atom predicates and predicate variables for checking compatible ordering among atoms and correct execution of each atom. With sufficient data gathered from the instrumented program code, the tool can compare the current execution of atom slices with the given specification and check the atom properties using the predicate variables. The goal of this thesis is to extend this partial order based runtime verification approach into a runtime recovery approach so that when a failure (runtime error) is detected, the system can automatically correct itself thereby guaranteeing application reliability and high availability. With the correct behavior specified by a POMSET, this approach identifies the necessary set of checkpoints formed by local checkpoints of atoms in the application. At runtime, it avoids unnecessary checkpoints using appropriate reduction algorithms, and it only needs to save the last global consistent state in persistent memory in order to reduce the resources necessary for recovery.

2.3 Checkpoint Based Rollback Recovery

The first fundamental issue for the partial order based runtime recovery falls on approaches for taking and maintaining global consistent states in distributed applications. A global consistent state, which is also called a global checkpoint, consists of a collection of individual states of all participating processes and communication channels [LAMP78, CL85, PAH08]. The occurrence of an event may change the global state. For example, the consistent line/cut of process events illustrated in Figure 2 represents a global consistent state in which message *m1* has been sent through the channel of Process 1 and is traveling in the network. In contrast, Process 3 in the inconsistent line/cut has delivered the message *m2* that has not yet been sent by Process 2.

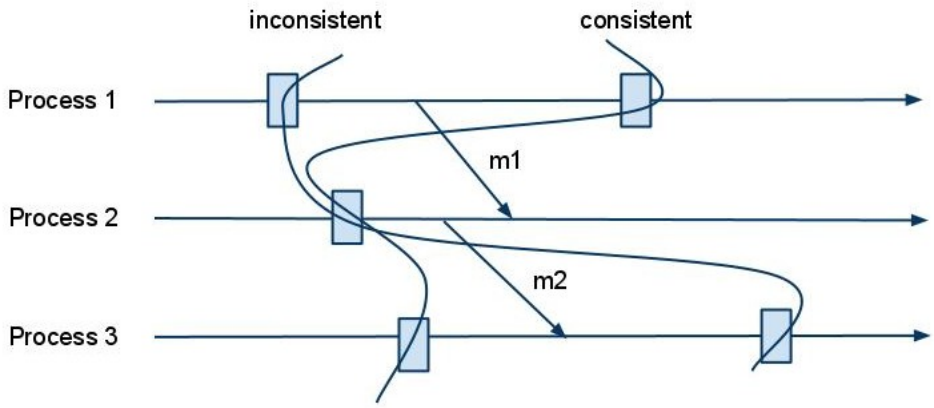


Figure 2: Global Consistent System State

Rollback recovery protocols [CR72, CW92, EZ92, CY96, EA02] in highly available distributed applications store/maintain consistent states of the application in a persistent store during failure-free execution at periodic time points/intervals. When a failure is detected at run time, the application can restart itself with an appropriate saved consistent state (or rolled back) from the persistent store to reduce the loss of

computation. Depending on the information being saved in the persistent store, these protocols employ two different approaches: checkpoint based [KT87, EJZ92, BBHMR95, EP04] or log based. The former only relies on periodic checkpoints to save the necessary recovery information while the latter relies on the piecewise deterministic assumption which identifies the nondeterministic events and logs the necessary information for replay.

Moreover, based on how processes cooperate to take/maintain checkpoints, checkpoint-based rollback recovery can be classified as: independent, coordinated, or communication-induced. Every process in independent checkpoint-based rollback recovery [BL88, TRI96] is allowed to take checkpoints independently. However, this technique leads to the serious domino effect problem [RAN75, BCS84], which may result in all the processes rolling back to their initial states thereby losing all the work they have done because of message dependencies among them. As illustrated in **Figure 3**, Process 2 detects a failure after receiving message *M5* and rolls back to checkpoint *C23*. Meanwhile, this pushes Process 1to forget sending message *M5* and to roll back to checkpoint *C12*. As a consequence, both Process 1 and Process 2 eventually roll back to their initial states.

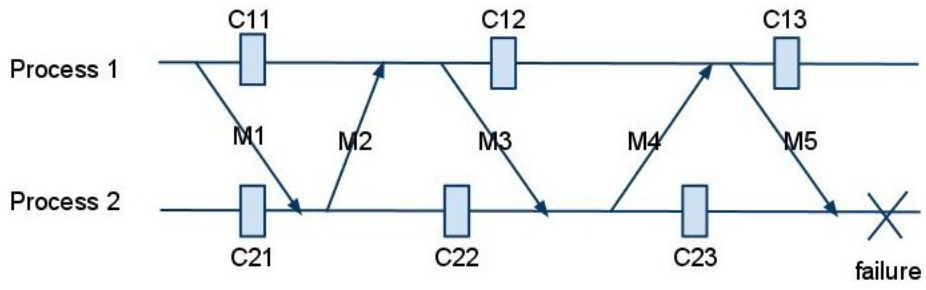


Figure 3: The Domino Effect

All the processes in coordinated checkpoint-based rollback recovery [BLKC03, CLG05, LPN05, BGR06] are required to cooperate their local checkpoints in order to form a global consistent state, thereby alleviating the domino effect problem. Moreover, this technique allows each process in the application to maintain only one local checkpoint to reduce storage overhead and the need for garbage collection.

In communication-induced checkpoint-based rollback recovery, there are two kinds of checkpoints: local checkpoints and forced checkpoints [HMNR97, AER99, BG00, BG01]. Each process can take local checkpoints while forced checkpoints must be taken based on the information piggybacked on the application messages received from other processes to guarantee a global consistent state. However, the number of checkpoints is changeable depending on the number of messages passing through the application. This technique incurs overhead in piggybacking the information, making it difficult in practice.

In contrast to checkpoint-based rollback recovery, log-based rollback recovery [JOH90, AHM95, ALV96, AM98, AV98] takes a piecewise deterministic approach which assumes that all nondeterministic events can be identified and the necessary information for replay can be properly logged. It guarantees that the processes of the system are "orphan-free", where an orphan process's state depends on a nondeterministic event and cannot be reproduced during recovery. According to how nondeterministic events are logged, log-based rollback recovery can be classified as: pessimistic, optimistic, or casual.

A process in pessimistic log-based rollback recovery [EZ92, BCHKLM03] always logs a message before delivering it, so orphan processes are never created.

Therefore, it is straightforward to reconstruct the state of a failed process. On the other hand, log-based rollback recovery requires blocking a process for all messages it receives, and as a consequence, system performance would be slowed down even when no failure occurs.

Compared to pessimistic log-based rollback, optimistic log-based rollback recovery [SY85, HW95] takes a smaller risk to have orphans for better system performance, because it does not require the application to be blocked when receiving messages. However, this advantage results in complicated recovery algorithms and garbage collection.

Taking a balance between optimistic log-based rollback and pessimistic log-based rollback, the causal log-based rollback recovery [AM96, BMA98, LPYC98, MG98, BCHLC05] prevents orphans and allows simple failure recovery. Furthermore, it has advantages of non-blocking run-time scheme and low failure-free overhead.

Like applications combining checkpoint-based and message logging recovery techniques, such as coordinated checkpoints with sender-based message logging [RN08], the partial order based runtime rollback recovery also takes both approaches from checkpoint-based rollback recovery and message log-based recovery. This technique allows each process in the application to make local checkpoints individually and organizes these local checkpoints to form a global consistent state. In addition, this technique also applies casual logging for all input/output messages through the channels, allowing all processes in the application to be orphan-free.

In summary, using the POMSET specifying the correct behavior of the application, runtime verification can detect failure occurrence by checking the ordering of

atoms and their predicate properties thereby reducing the state space required. The runtime rollback recovery is developed based on the runtime verification. By adding some properties to the nodes of the POMSET tree, this technique can identify the necessary checkpoints keeping only the last one in persistent storage and rollback the system to that state when a failure occurs. All properties, rules and the checkpoint reduction algorithms will be presented in Chapter 3 and illustrated using the example travel agent application.

CHAPTER 3 : THE RUNTIME RECOVERY TECHNIQUE

This chapter develops the partial order based runtime recovery technique that restores a highly available distributed application using a limited number of checkpoints (resources) upon a failure and continues its normal execution.

As described in Chapter 2, the traditional checkpoint-based recovery approaches store checkpoints at periodic intervals. Also because the behavior of the distributed application is unknown, the traditional approach needs to store the state of every process in the application as the global consistent state (called global checkpoint), thereby using significant resources (time for blocking system and space to store states of all processes). The partial order based runtime recovery differs from the traditional checkpoint-based approach in many ways. First of all, given a POMSET tree specifying the expected behavior of the application, this runtime recovery technique identifies the nodes of the POMSET tree (operators) at which a checkpoint must be taken to ensure recovery (because they impact application execution). Secondly, instead of storing the states of all the processes in the application to form a global consistent state (global checkpoint), this technique only stores the states of the processes that are necessary to recover from a potential failure as a collection of local checkpoints (called group checkpoint). This collection of local checkpoints (that is, group checkpoint) represents the global consistent state in the partial order based runtime recovery.

The approaches and algorithms for this partial order based runtime recovery technique are presented in the following sections.

3.1 Checkpoint Identification

In order to identify the necessary checkpoints from the POMSET specification composed of atoms and operators, the first step is to find out which atoms should store their local checkpoints to form a group checkpoint.

Atoms in processes compute their tasks and cooperate to perform the designed functionality in a distributed application. A **critical atom** computes and changes the value of a relevant variable that affects the correct progress of the execution, marked as red in Figure 4. A non-critical atom, shown in blue, does not change the value of any relevant variable and hence need not be restored. The computed variable value should be saved so that a critical atom can restart itself with the recovered data if a failure occurs in the future. In other words, those atoms that should avoid to be re-executed upon failures are critical atoms.

Figure 4: Atom Classification

For the online travel agent example application listed in Table 2, most critical atoms perform the following operations: reserving/cancelling a room, update vacancy information (because the booking information for a hotel/transportation service should be kept). When a failure occurs, the booking data should be restored and the services can continue their normal execution.

Besides atoms, operator nodes are also involved in checkpoint identification because they specify the correct ordering among their sub-trees. As presented in Figure 5, each operator node has its unique ordering requirement described in the following.

A **concatenation operator** "**;**" denotes before–after relationship between two children sub-trees. For example, *(A;B)* represents that atom *A* should occur before atom *B*.

An **alternation operator** "+" denotes the exclusive choice from more than one possible execution paths (sub-trees). For example, *(A+B)* represents that only atom *A* or atom *B* can occur, but never both in strictly partial ordered modeling.

A **concurrency operator** "||" denotes the concurrency of more than one execution paths (children sub-trees). For example, *(A||B)* represents that atom *A* and atom *B* can occur without any ordering constraints.

A **recurrence operator** "*****" denotes the recurrence of the children sub-tree. For example, $(A)^*$ represents that atom *A* can occur zero or more times.

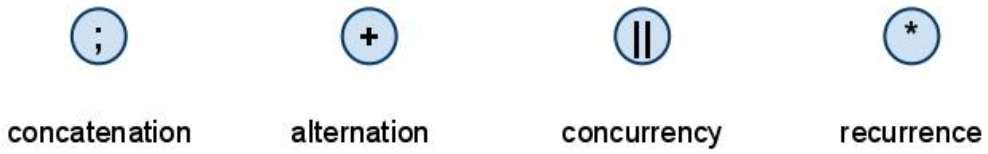


Figure 5: Operation Node Classification

Even with critical atoms and well-defined operator information, checkpoints cannot be identified by intuition; but adding a property to the nodes of the POMSET tree can help this process. This **potential checkpoint** property indicates the checkpoint decisions for the nodes. The value of a potential checkpoint can be True, False, or

Unknown. A **True** potential checkpoint implies that a checkpoint must be taken when the execution is at the node; while a **False** potential checkpoint implies there is no need to take a checkpoint when the execution is at that node. An **Unknown** potential checkpoint is the special case with unpredictable execution paths that can be recognized only at runtime.

For atoms, since their potential checkpoint values influence checkpoint decisions, it is straightforward to set it as True for critical atoms to avoid re-execution and False for non-critical atoms, as illustrated in Figure 6.

Figure 6: Potential Checkpoint Values of Atoms – True/False

The potential checkpoint value of an operator node represents the intention of checkpoint taken at the last child/children nodes that will be invoked later. Different operators have different potential checkpoint values according to the operator type and the potential checkpoint values of its children nodes.

An alternation operator "+" denotes multiple choices from more than one possible execution paths (sub-trees), but there will be only one path to be executed at runtime. In other words, the result of an executed path should be unpredictable until runtime. Therefore, as Case 3 in Figure 7, the potential checkpoint value of an alternation operator should be set as Unknown instead of True or False, because the real execution path cannot be predicted until execution time. However, when all the children nodes have the

same potential checkpoint values (any execution path results in the same intention for checkpoints), an alternation operator has the potential checkpoint value other than Unknown. In short, an alternation operator has total agreement of potential checkpoint values among its children. For example, for the POMSET specification *(A+B)* in Figure 7, in Case 1 and Case 2 where atoms *A* and *B* have the same potential checkpoint values, the alternation operator has total agreement of *A* and *B* – True and False, respectively. If *A* and *B* both have True potential checkpoint values, potential checkpoint value of the alternation operator should be set as True for representing necessary future checkpoint taken for invocation of either *A* or *B*. However in Case 3 where atoms *A* and *B* have different potential checkpoint values, the alternation operator has Unknown potential checkpoint value because of unpredictable execution paths.

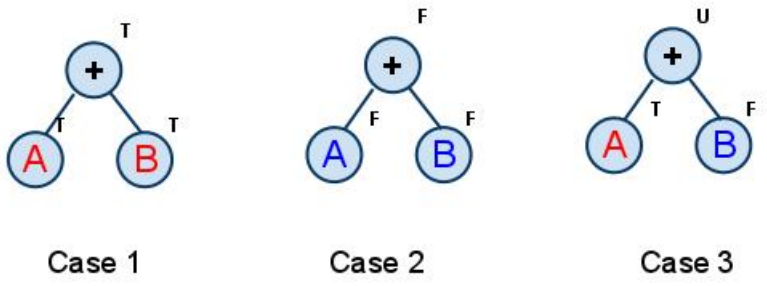


Figure 7: Potential Checkpoint values of Alternation Operator – True/False/Unknown

A concatenation operator ";" denotes the partial-order sequence of children nodes. So its potential checkpoint value should be the same as its right child (since the left child has already occurred in the execution) to represent the intention of checkpoint to be taken at the last child (right node). Potential checkpoint property of the concatenation operator

intends there exist a checkpoint that can be used for recovery, if a failure has occurred after the last child invocation of the operator. As illustrated in Figure 8, for the POMSET specification *(A;B)*, in Case 1 where *A* and *B* are both critical atoms, the concatenation operator has True potential checkpoint value. It may be the concatenation operator itself or its upper operators with a checkpoint that can be used for recovery, so that B can be rolled back instead of re-execution. In Case 2 where atom *B* is a non-critical atom, the concatenation operator has False potential checkpoint value. However, under certain conditions such as in *(A;(B+C))*, the concatenation operator sets its potential checkpoint value as Unknown, because the last right child node has Unknown potential checkpoint value.

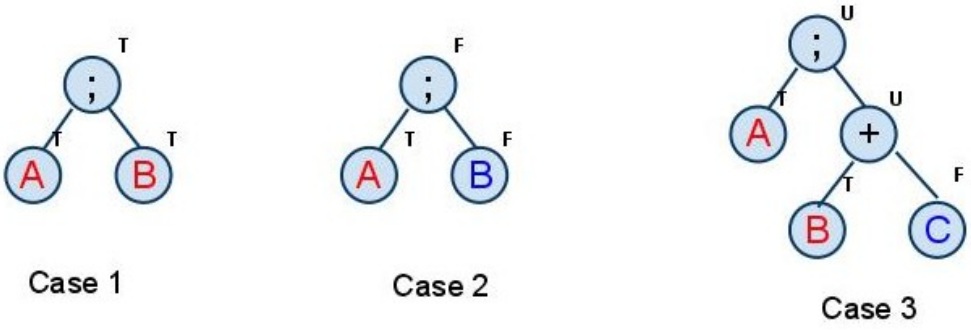


Figure 8: Potential Checkpoint Value of Concatenation Operator – True/False/Unknown

A concurrency operator denotes that more than one path can be run without any ordering constraints, so its potential checkpoint value should represent the intention of any checkpoint taken by the last children of each execution paths. Basically, if any of the last children of the execution paths should take a checkpoint, this operator has True potential checkpoint value; otherwise it should have False checkpoint value. Under some

conditions where its operand children only have False and Unknown potential checkpoint values, a concurrency operator node should set its potential checkpoint value as Unknown instead of False for checkpoint decisions. If any last children of the concurrently executing subtree have True/Unknown potential checkpoint properties that will cause the concurrency operator itself or its ancestor operators taking a checkpoint which can be used if a failure has occurred after invoking the last children of the operator. Thus, for the POMSET specification *(A||B)* in Figure 9, in Case 1 and Case 2 where atom *A* has True potential checkpoint value, the concurrency operator should set its potential checkpoint value as True. In Case 3 where atoms *A* and *B* have False potential checkpoint values, the concurrency operator should set its potential checkpoint value as False. Given another POMSET specification *(A||(B+C))* illustrated in Figure 10, the concurrency operator should set its potential checkpoint value as Unknown, because its operand children only have False and Unknown potential checkpoints.

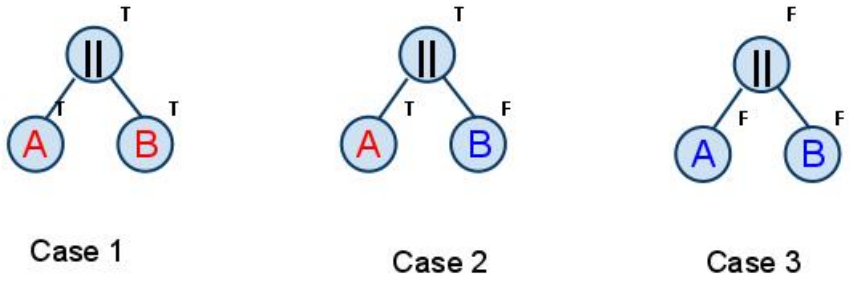
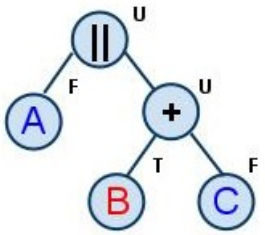


Figure 9: Potential Checkpoint Value of Concurrency Operator – True/False

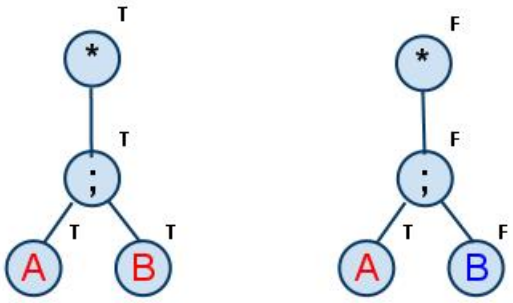


Case 4

Figure 10: Potential Checkpoint Value of Concurrency Operator – Unknown

A recurrence operator denotes multiple iterations of its sub-tree; so its potential checkpoint value should be judged by its only child, because the potential checkpoint value of the child also represents the last child of the subtree of recurrence operator. For the POMSET specification *(A;B)** illustrated in

Figure **11**, in Case 1 and Case 2, the recurrence operator has the same potential checkpoint value as its only child – True/False, respectively. However, for another POMSET specification *((A;B)+(C;D))** illustrated in Figure 12, the recurrence operator sets its potential checkpoint value as Unknown because its child has Unknown potential checkpoint value.



Case 1 Case 2

Figure 11: Potential Checkpoint Value of Recurrence Operator – True/False

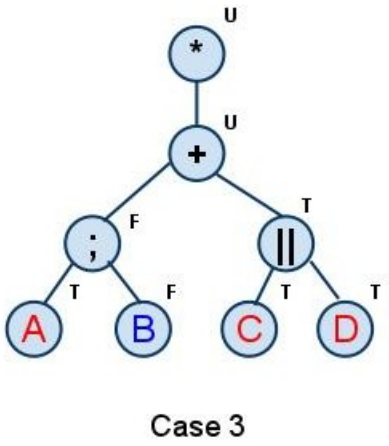


Figure 12: Potential Checkpoint Value of Recurrence Operator - Unknown

In most practical distributed applications, such as the online travel agent, there are multiple alternation operators for multiple choices resulting in most nodes in the POMSET tree with Unknown potential checkpoint values as illustrated in Figure 13.

With the potential checkpoint values, which show the intention of taking a checkpoint, checkpoints of operator nodes presented in Figure 14 can be identified by the following rules. In this thesis, a checkpoint taken by an operator represents that the critical atoms controlled by the operator should participate in forming a group checkpoint which is the global consistent state for the distributed application.

No checkpoint is taken when the execution is at an alternation, because this operator denotes exclusive choice from more than one execution paths. In other words, there is only one path that can be executed at runtime. Once the execution path is selected, this operator has fulfilled its task, so it is not meaningful to take a checkpoint.

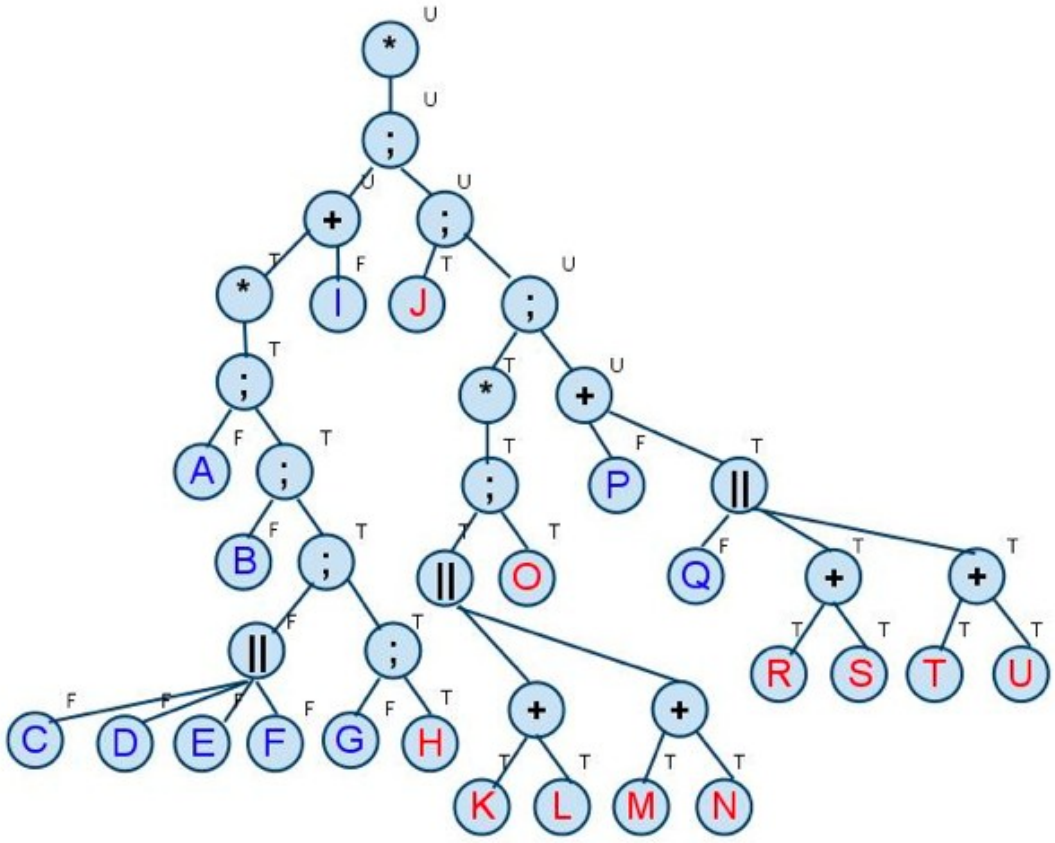


Figure 13: Potential Checkpoint Values for the Online Travel Agent Application

Figure 14: Checkpoints of Operators

A checkpoint is taken at a concatenation operator if its left child has True/ Unknown potential checkpoint value. Because it denotes the before-after relationship between the left sub-tree and the right sub-tree, if the left sub-tree has True potential checkpoint value, a checkpoint should be taken when the execution is at that node in order to avoid re-execution of the left sub-tree. Instead, if the left sub-tree has False

potential checkpoint value, then a checkpoint need not be taken when the execution is at that node. However, under some conditions where the left sub-tree has Unknown potential checkpoint value, which can be confirmed only at runtime, a checkpoint must be taken at the concatenation operator that may be avoided by later static or dynamic checkpoint reduction analysis. For example, given the POMSET specification *(A;B)* illustrated in Figure 15, in Case 1 where atom *A* has True potential checkpoint value, a checkpoint is taken at the concatenation operator to avoid re-executing atom *A*. In case 2, because atom *A* has False potential checkpoint value, a checkpoint is not taken at the concatenation operator. In Case 3 where the left child of the concatenation operator has Unknown potential checkpoint value, a checkpoint is still necessary.

A checkpoint is always taken at a concurrency operator. This operator denotes concurrent execution of more than one execution paths. Before splitting one mainstream execution into multiple concurrent executions, a checkpoint is taken to record the current stages of system.

A checkpoint is taken at a recurrence operator if its only child node has True/Unknown potential checkpoint value in order to avoid re-executing the only child. However, under some conditions where the recurrence operator has a child with Unknown potential checkpoint value, a checkpoint is still taken that may be avoided by static or dynamic checkpoint reduction analysis. For the POMSET specification *(A;(B||C))** illustrated in Figure 16, in Case 1 where the child node of the recurrence operator has True potential checkpoint value, a checkpoint is taken at the recurrence operator. In Case 2, the child node has a False potential checkpoint value, so a checkpoint is not taken at the recurrence operator. However, for the POMSET specification

*(A+(B||C)*** in Case 3 where the recurrence operator has a child node with Unknown potential checkpoint value, a checkpoint is still taken.

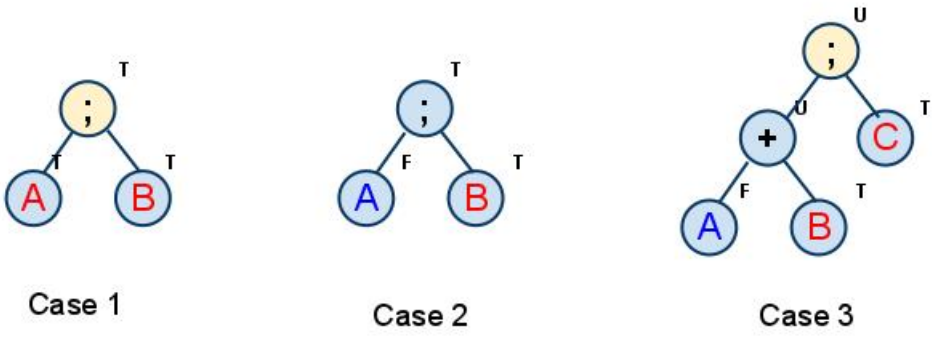


Figure 15: Checkpoint of Concatenation Operator

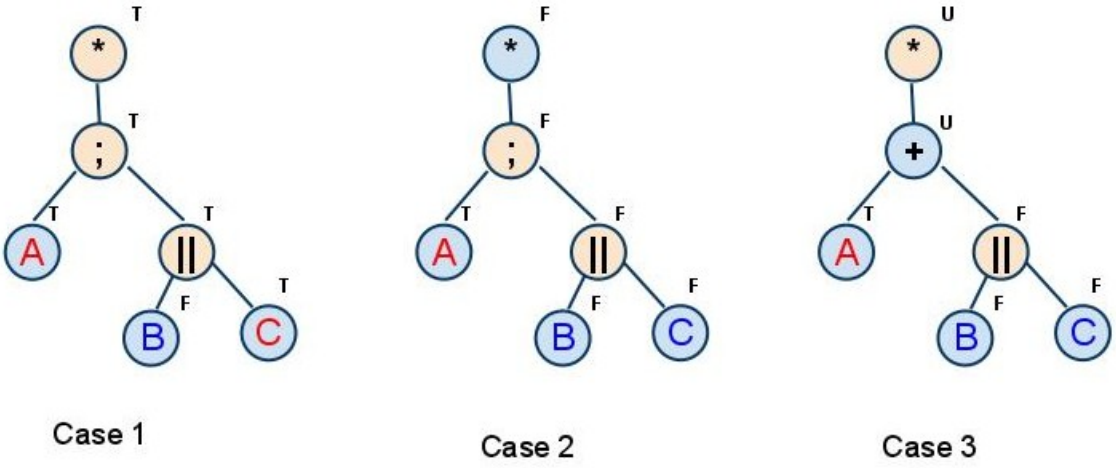


Figure 16: Checkpoint of Recurrence Operator

As illustrated in **Figure 17**, applying the checkpoint rules on the online travel agent application identifies ten checkpoints for the given POMSET specification. However, two of these checkpoints belong to the special cases caused by unpredictable alternation operators that will be re-evaluated by checkpoint reduction analysis. In fact,

the more alternation operators in the POMSET specification, the more special cases occur in execution.

Unlike a global checkpoint in traditional checkpoint-based recovery (which is formed by the local checkpoints of all the processes in the distributed application), the partial order based runtime recovery only requires critical atoms in the processes to take local checkpoints and organize these local checkpoints as a **group checkpoint** (a collection of local checkpoints). This approach thereby reduces the time and space required for a checkpoint. Note that checkpoints taken at the operators in the POMSET tree mentioned above are group checkpoints.

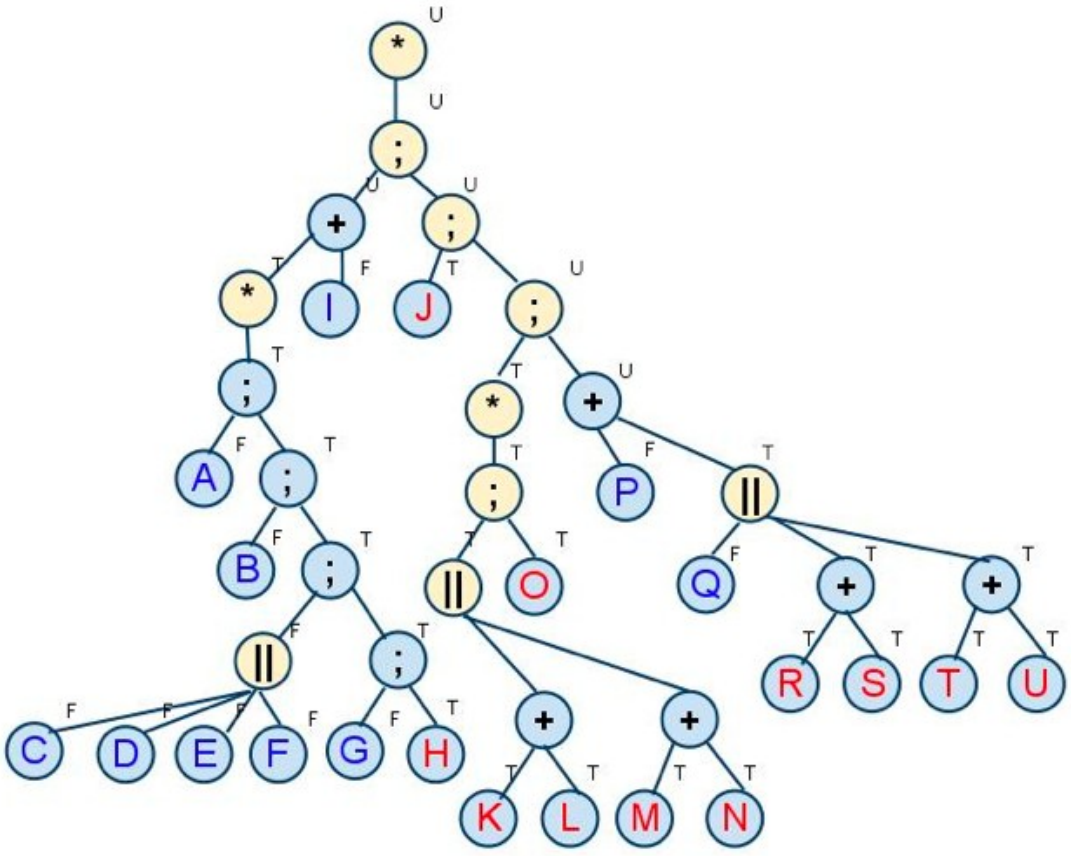


Figure 17: Checkpoint Identification on Online Travel Agent Application

As illustrated in Figure 18, a critical atom can be divided into three parts: message receiving, atom execution, and message sending. First, an atom waits to be invoked by receiving requests or messages from other atoms. Then, it performs its assigned functionality during atom execution. At the last stage, it sends output messages to other atoms. Hence, as illustrated in Figure 19, a critical atom takes local checkpoints to record the different stages of its operation: LC1 for the initial stage, LC2 for the execution end stage before sending messages, and LC3 for the end stage of the atom.

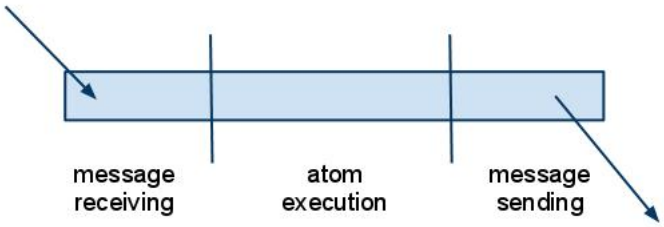


Figure 18: Stages of an Atom

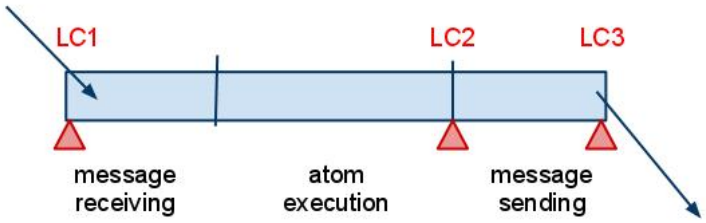


Figure 19: Local Checkpoints of an Atom

Based on the POMSET tree, which specifies the correct behavior of the distributed application, the partial order based runtime recovery approach decides whether or not to take a group checkpoint (which is the collection of local checkpoints of the executing critical atoms) at an operator node. For example, given the POMSET

specification *(A;(B||C))* illustrated in Figure 20 and Figure 21, a critical atom *A* sends a messages *m1* to a non-critical atom *B* and sends another message *m2* to another critical atom *C*. As illustrated in Figure 22, each consistent cut represents different stages in the execution of the application: *GC1* represents the initial stage in which all the atoms are ready to receive input messages, *GC2* represents that messages are on their way where *A* sends the messages, but *B* and *C* have not received the messages yet, and *GC3* represents the end stage in which all the atoms complete their execution.

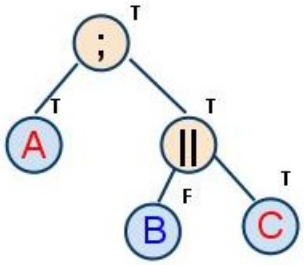


Figure 20: Group Checkpoint Example (A;(B||C)) as a POMSET Tree

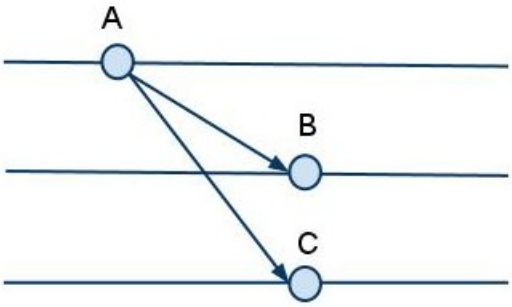


Figure 21: Group Checkpoint Example (A;(B||C)) as Time Lines

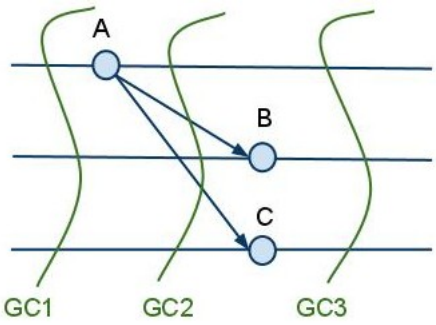


Figure 22: Group Checkpoint Example (A;(B||C)) as Consistent Cuts

Note that a group checkpoint represents a global consistent cut of execution. Using the example *(A;(B||C))* where *A* and *C* are critical atoms, *GC1* representing the initial stage of application execution can be organized as the combination of the initial stages of all the atoms – *A.LC1*, *B.LC1* and *C.LC1*. Similarly, *GC3* representing the end stage of application execution can be organized as the combination of the end stages of all the atoms – *A.LC3*, *B.LC3*, and *C.LC3*. Note that all atoms in the distributed application, including critical atoms and non-critical atoms, should participate in forming initial and end stage checkpoint. During execution, while messages are on their way as illustrated in Figure 23, *GC2* can be organized as the combination *A.LC3*, *B.LC1*, and *C.LC1*. If any failure happens after *GC2* (but before *GC3*), the application should then be rolled back to *GC2* for recovery. Notice that atom *B* does not take a new local checkpoint because it is a non-critical atom and no data needs to be kept for rollback recovery.

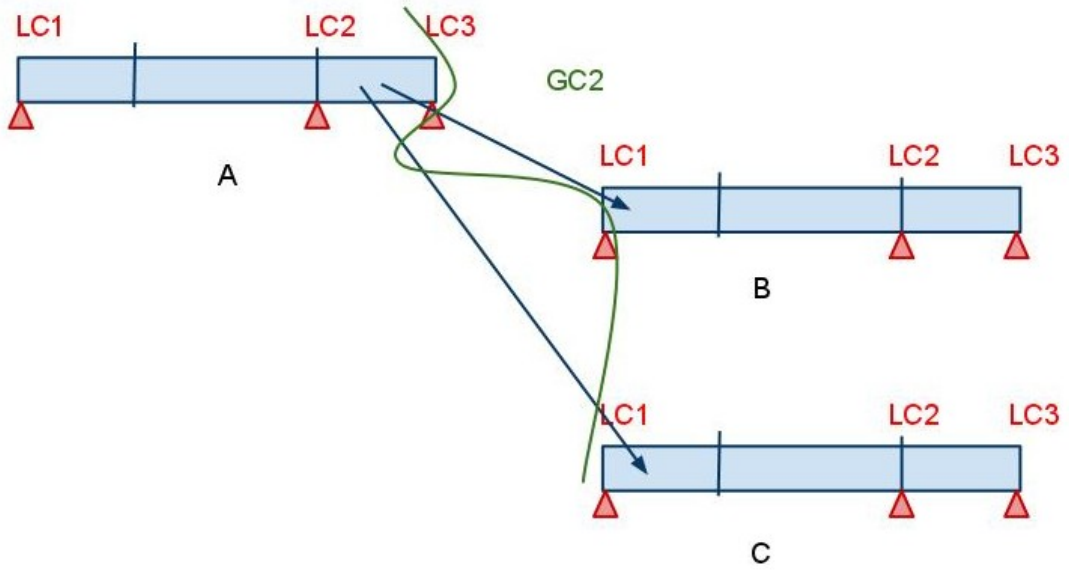


Figure 23: Group Checkpoint Example (A;(B||C)) – Messages On Delivery

However, not all of the identified group checkpoints may be necessary. Global checkpoints of the concatenation operator and the concurrency operator represent the same group checkpoints (*GC2*). In addition, the alternation operator may result in group checkpoints taken at its parent operators. To avoid such unnecessary checkpoints, checkpoint reduction algorithms are developed next.

3.2 Checkpoint Reduction Algorithms

Given a POMSET tree specifying the correct behavior of a distributed application, the partial order based runtime rollback recovery identifies the group checkpoints (a collection of local checkpoints taken by the critical executing atoms) of operators in the POMSET tree. However, some of these group checkpoints may be unnecessary and can be avoided. In order to identify such avoidable group checkpoints, two checkpoint

reduction algorithms are developed: the static checkpoint reduction algorithm and the dynamic reduction algorithm.

3.2.1 Static Checkpoint Reduction

As illustrated for the specification *(A;(B||C)* in Figure 20, sometimes group checkpoints are to be taken at two operators with overlapping sub-trees. In fact, these group checkpoints mean that the same operation occurs upon the same sub-trees. In order to solve this issue, the static checkpoint reduction algorithm is developed that reduces the group checkpoints by (1) the priority of the operators and (2) the height/position of the operator in the POMSET tree.

The priority of operators is one of the fundamental rules in checkpoint reduction analysis. According to the coverage of atom sets by the operator, the priority of an operator is set as: recurrence operator (*) = concurrency operator (||) > concatenation operator (;) > alternation operator (+) as explained below.

An alternation operator has the lowest priority since no checkpoint is taken at that node. A concatenation operator has lower priority compared to a concurrency operator and a recurrence operator since it only denotes the partial-order sequence of two sub-trees. Also, both the recurrence operator and the concurrency operator have the same priority since they denote multiple iterations or split execution paths, respectively.

Applying these priority rules on the specification *(A;(B||C))* in Figure 24, the group checkpoints taken at both the concatenation operator and the concurrency operator are avoided. The group checkpoint taken by the concatenation operator is evaluated as unnecessary because the concurrency operator has higher priority. A similar condition

occurs in another case *(A;(B;C)*)* illustrated in Figure 25 where the group checkpoint at the concatenation operator is avoided because the recurrence operator has higher priority.

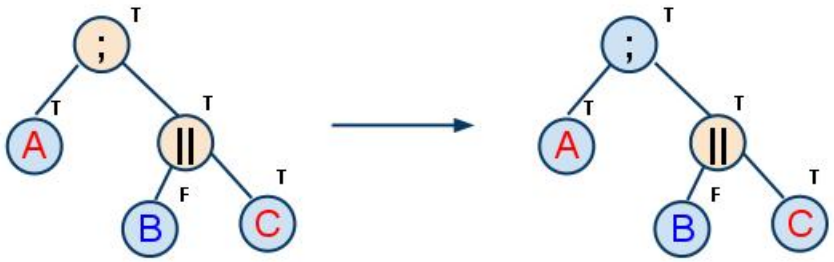


Figure 24: Static Checkpoint Reduction Algorithm (Priority) on (A;(B||C))

However, there is another issue that group checkpoints are taken at both the recurrence operator and the concurrency operator. Avoiding the group checkpoints taken at operators with the same priority also involves the static checkpoint reduction algorithm. From the POMSET tree point of view, the group checkpoint taken at the operator at higher position should be kept. For example, given the POMSET specification *(A||B)** illustrated in Figure 26, the group checkpoint taken by the concurrency operator is identified as unnecessary because it is under the control of the recurrence operator. That is, the recurrence operator's position is higher in the POMSET tree.

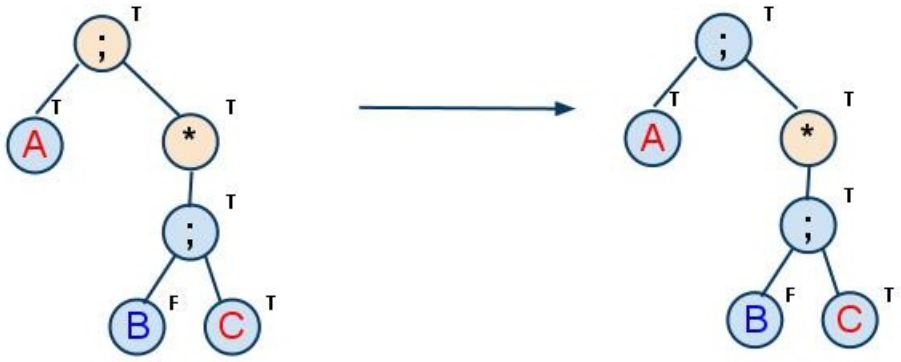


Figure 25: Static Checkpoint Reduction Algorithm (Priority) on (A;(B;C)*)

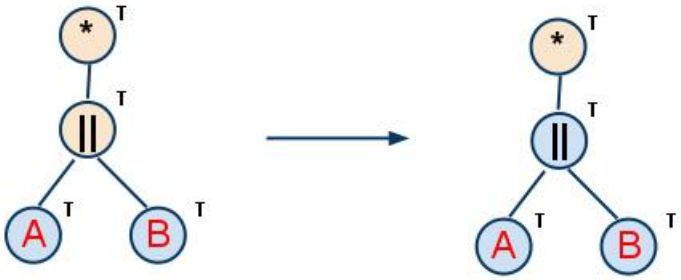


Figure 26: Static Reduction Rule – Depth

The pseudo code of this static checkpoint reduction algorithm is presented below.

```
function setProperties (node, level)
        node.level := level
        if (node is a critical atom)
                node.potential_checkpoint := True
        else if (node is a non-critical atom)
                node.potential_checkpoint := False
```

```
else if (node is a concatenation operator )
        node.potential_checkpoint := node.rightchild.potential_checkpoint
        if (node.leftchild.potential_checkpoint = True)
                node.group_checkpoint := True
        node.priority := 1
else if (node is an alternation operator)
        if (node's all children have the same potential checkpoint value)
                node.potential_checkpoint := node.firstchild.potential_checkpoint
        else
                node.potential_checkpoint := Unknown
        node.group_checkpoint := False
        node.priority := 0
else if (node is a concurrency operator)
        if (node's any last child has True potential_checkpoint)
                node.potential_checkpoint := True
        else if (node's last child has Unknown potential_checkpoint)
                node.potential_checkpoint := Unknown
        else
                node.potential_checkpoint := False
        node.group_checkpoint := True
        node.priority := 2
else if (node is a recurrence operator)
        node.potential_checkpoint := node.child.potential_checkpoint
```

```
        if (node.child.potential_checkpoint is True or Unknow)
            node.global_checkpoint := True
        else
            node.global_checkpoint := False
        node.priority := 2

function reduceCheckpoint (node)
    if (both nodes have group_checkpoints and they control overlapped sub-trees)
        if (node.priority > node.child.priority)
            node.child.group_checkpoint := False
        else if (node.priority < node.child.priority)
            node.group_checkpoint := False
        else
            if (node.level > node.child.level)
                node.child.group_checkpoint := False
            else
                node.group_checkpoint := False
```

When applying the static checkpoint reduction algorithm on the online travel agent application as illustrated in Figure 27, two group checkpoints are identified as unnecessary. One is the checkpoint taken at the concatenation operator, because of its lower priority than the recurrence operator. The other is the checkpoint taken by the

concurrency operator, because its position is lower than the recurrence operator in the POMSET tree.

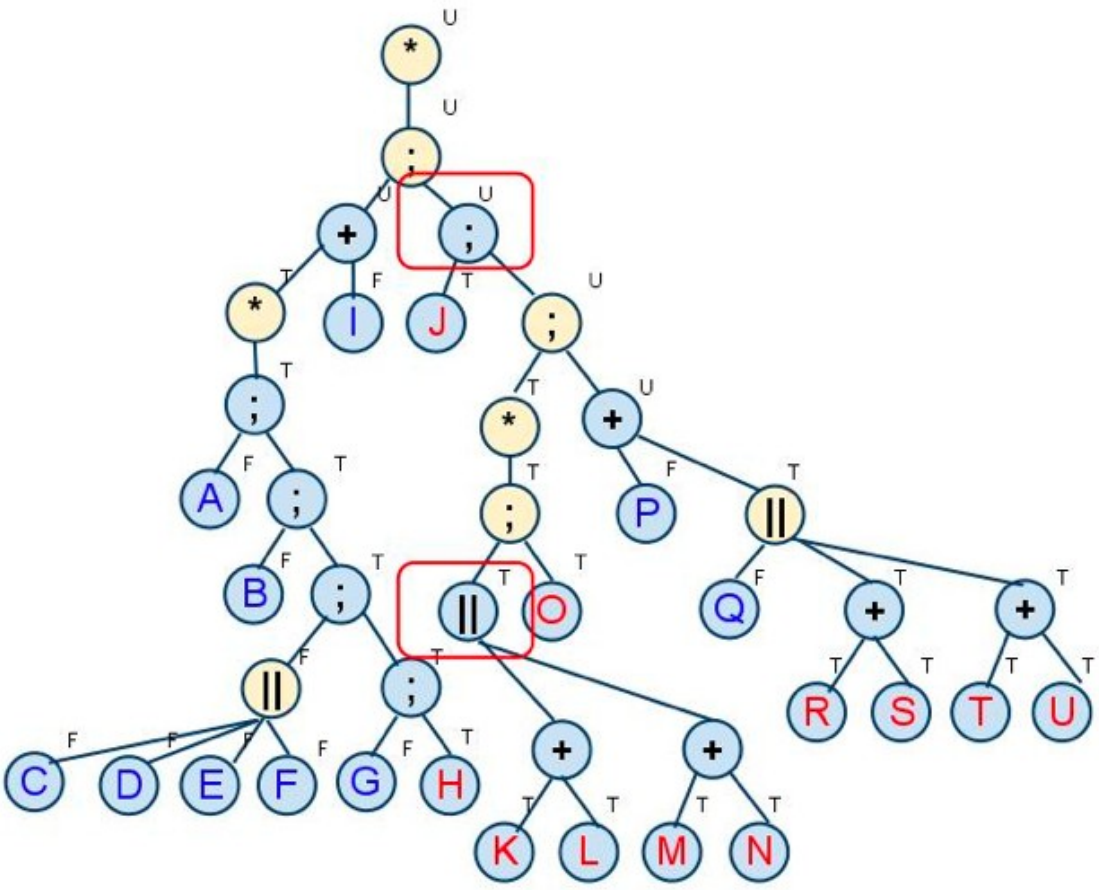


Figure 27: Static Checkpoint Reduction Algorithm on the Online Travel Agent Application

3.2.2 Dynamic Checkpoint Reduction

Like the operators controlling overlapping sub-trees, alternation operators with unpredictable execution paths may result in unnecessary checkpoints in the POMSET tree. During different iteration runs, the executed path of the alternation operator must be

confirmed because the alternation operator denotes that exactly one path will be executed. By remembering the confirmed executing path with relevant data events, next few iterations of the execution can be forecasted according to what has happened in past. Once the unpredictable execution paths caused by the alternation operator can be forecasted, group checkpoints can be re-identified without Unknown potential checkpoint values.

Figure 28: Dynamic Checkpoint Reduction Algorithm on (A+B)

Consider Case 2 in Figure 7 as an example. The execution paths happened in the previous iterations can be presented as in Figure 28. In Execution 1, with the relevant data event *E1*, the alternation operator has True potential checkpoint value because the invoked atom *A* has True potential checkpoint value. Otherwise, with the relevant data event *E2*, the alternation operator has False potential checkpoint value because the invoked atom *B* has False potential checkpoint value. After two runs, this alternation operator can be exactly forecasted– True for event *E1*, False for event *E2*.

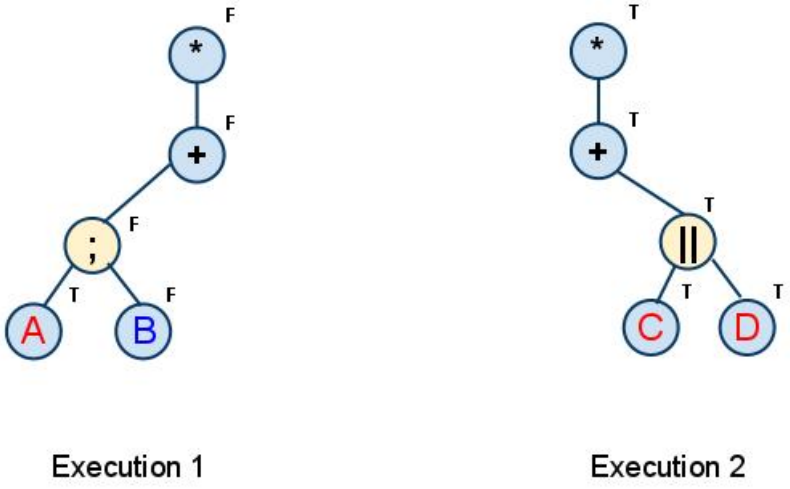


Figure 29: Dynamic Checkpoint Reduction Algorithm on ((A;B)+(C||D))*

Furthermore, forecasting the unpredictable execution paths of alternation operators also helps confirm potential checkpoints of its ancestor operators. This approach solves most exceptional cases for unpredictable paths. Consider Figure 12 as an example. The two possible execution paths illustrated in Figure 29 include the sub-tree of the alternation operator and its parent recurrence operator after two runs.

The pseudo code of the dynamic checkpoint reduction algorithm is presented in the following.

```
(execution pool: a collection of execution paths with relevant data events)
function forecast (node, events)
    if (node is an alternation operator with Unknown potential checkpoint)
        if (received variant events map compatibly with events in execution pool)
            predicatedExecution := remembered execution in execution pool
            apply predicatedExecution into POMSET tree
```

```
        for (node in predicatedExecution)
            setProperty(node)
    else
        add this execution path beginning from node into execution pool
        with received relevant data events
```

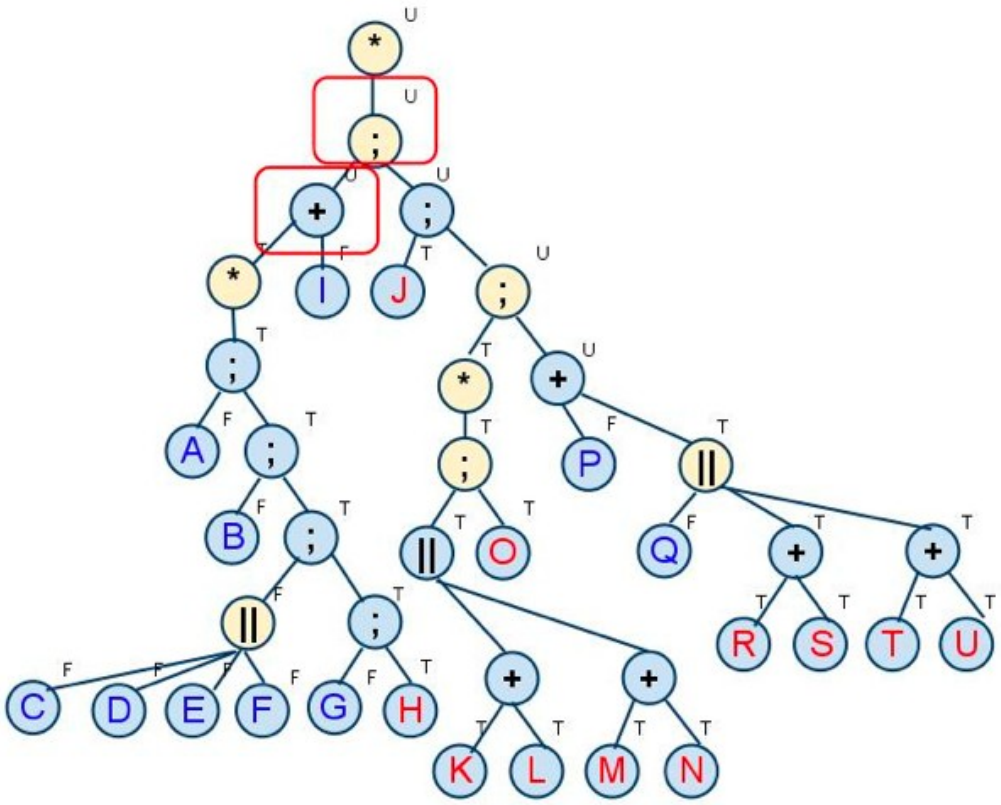


Figure 30: Dynamic Checkpoint Reduction Algorithm on Online Travel Agent Application Case 1

Applying this dynamic checkpoint reduction algorithm on the online travel agent application, it is obvious that the two alternation operators make most of their ancestor nodes in the POMSET tree to have Unknown potential checkpoint values. One of the

operators is in the left side of the POMSET tree, as illustrated in **Figure 30**, and a group checkpoint is taken at its parent concatenation operator. The other operator is in the right side of the POMSET tree as illustrated in **Figure 31**, and a group checkpoint us taken at its parent recurrence operator (also the root of the POMSET tree).

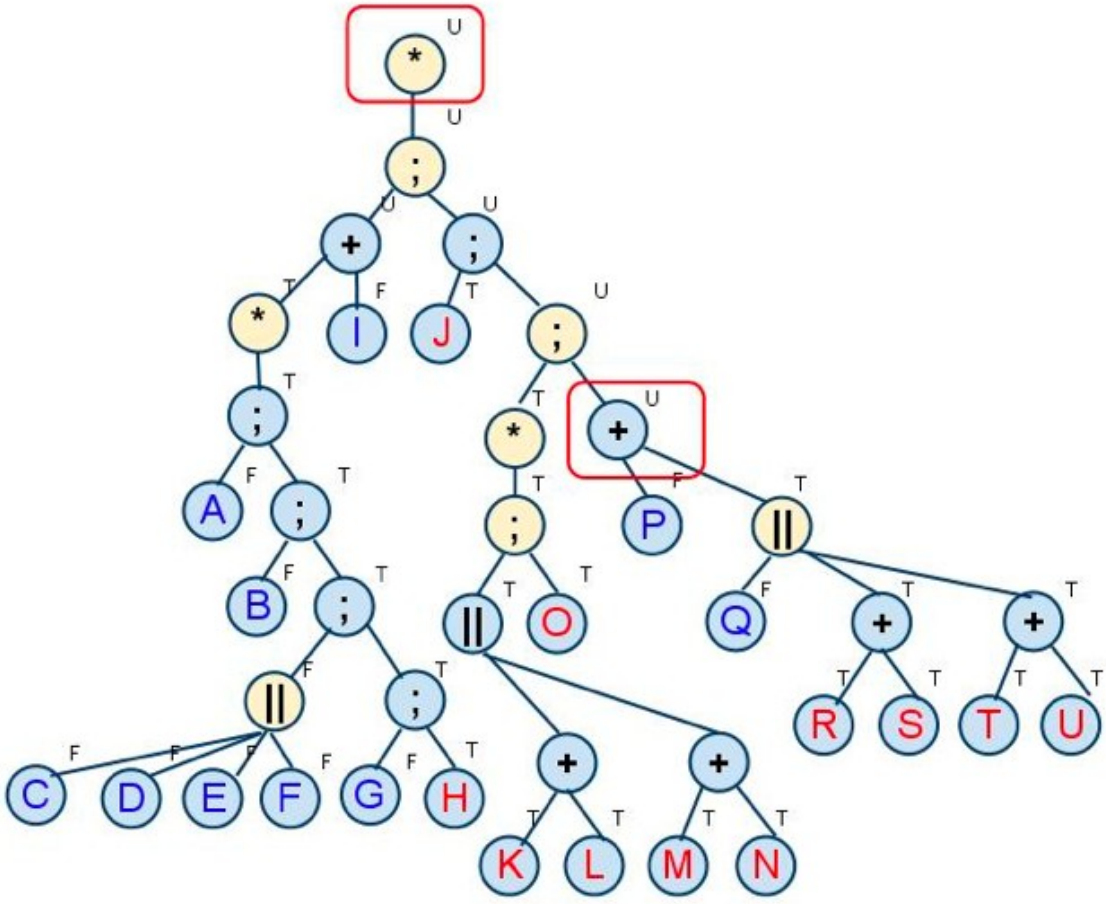


Figure 31: Dynamic Checkpoint Reduction Algorithm on the Online Travel Agent Application Case 2

In order to precisely forecast the execution paths in the former case, the two execution paths would be remembered as illustrated in Figure 32. During the execution runs, if event *E1* occurs, the alternation operator with True potential checkpoint value allows a group checkpoint to be kept at the concatenation operator. Otherwise, if event

E2 occurs, the group checkpoint is discarded at the concatenation operator because the alternation operator has False potential checkpoint value.

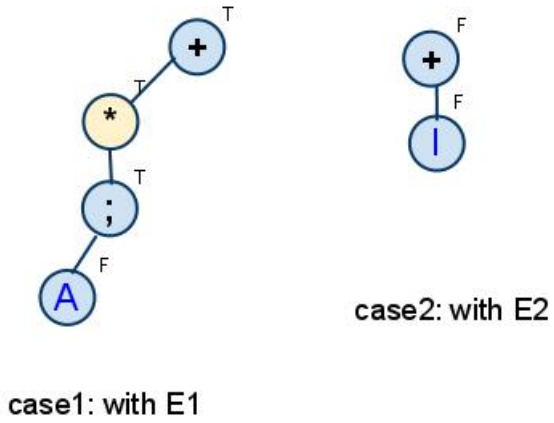


Figure 32: Dynamic Checkpoint Reduction Algorithm on the Online Travel Agent Application Case 1 – Remembered Execution Paths

Similar to the previous example, another two execution paths should also be remembered for the latter alternation operator as illustrated in Figure 33. If event *E3* occurs, the alternation operator changes its potential checkpoint property and those of its parent operator. At the end, it makes the root of the POMSET tree, the recurrence operator, to discard the group checkpoint. Otherwise, if event *E4* occurs, the alternation operator changes the potential checkpoint value as True and a group checkpoint is still taken at the recurrence operator. Moreover, because the concurrent operator has higher priority than the concatenation operator, the group checkpoint of the upper concatenation operator would be identified as unnecessary during the reduction.

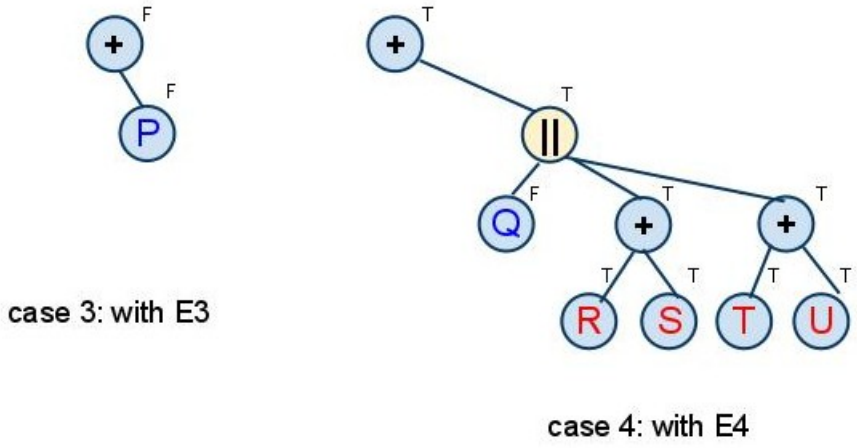


Figure 33: Dynamic Checkpoint Reduction Algorithm on the Online Travel Agent Application Case 2 – Remembered Execution Paths

3.3 Summary

The partial order based runtime recovery approach, which extends the partial order based runtime verification idea, identifies the necessary group checkpoints to be taken at the operators in a POMSET tree that specifies the correct behavior of a given distributed application. Unlike a global checkpoint in traditional checkpoint-based approaches, a group checkpoint represents a collection of local checkpoints of only the necessary executing critical atoms (not all processes), thereby reducing the amount of time and storage space to maintain a global consistent state.

In addition, some unnecessary checkpoints can be avoided by the static and the dynamic checkpoint reduction algorithms, thereby reducing the number of necessary group checkpoints. During execution runs, only the last group checkpoint are kept for rollback recovery, thereby reducing the required persistence space.

A prototype implementation of all the approaches introduced in this chapter will be presented in Chapter 4.

CHAPTER 4 : PROTOTYPE IMPLEMENTATION

This chapter presents a prototype implementation of the partial order based runtime recovery technique developed in Chapter 3. This prototype requires the developer to specify the atoms within the processes by inserting appropriate statements in (thereby instrumenting) the application program. This application-level instrumented implementation [JLSU87, SG91, SBFMPS04] is similar to embedded codes [XR96] or compiler-based programs [LF90]. The prototype implements a monitor that does the following using the POMSET tree:

1. When the execution is at an operator node, identify whether a group checkpoint should be taken at that node and instruct the participating (descendant) critical atoms of that operator node to take a group checkpoint by calling the checkpoint() function provided by the application developer.
2. Receive event information from the atoms in the instrumented application program, compares the events with the POMSET specification of the correct behavior to detect failures, and recovers the system when a failure happens by calling the recover() function provided by the application developer.

Thus, the monitor handles most of the runtime recovery.

4.1 Deployment Environment

Processes in a highly available distributed application cooperate to perform their assigned functionality and communicate with each other by messages, as illustrated in Figure 34. Each such process is developed with instrumented code, which sends relevant

events to a central monitor. These various events trigger the runtime monitor to verify the behavior of application against the POMSET specification. When the runtime monitor detects a failure, it automatically and immediately sends specific system level events to the processes for checkpoint-based recovery.

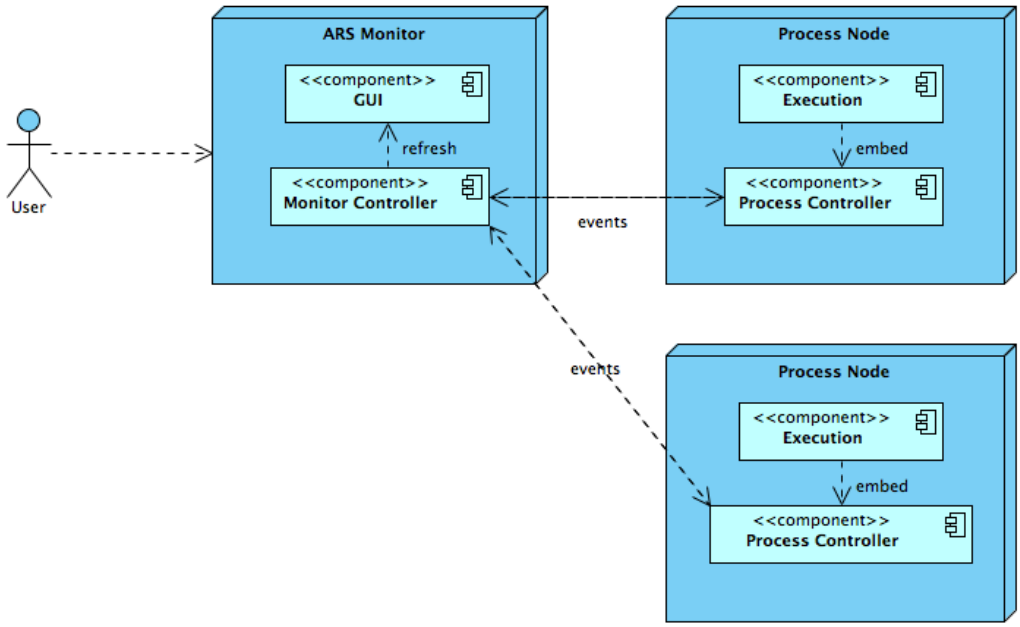


Figure 34: Deployment Environment

4.2 Monitor Prototype

Given a POMSET specification, the runtime monitor identifies the necessary group checkpoints to be taken at operator nodes in the POMSET tree and takes/maintains these group checkpoints (consisting of the states of the involved critical atoms). To achieve this goal, several managers are classified as: monitor side or atom side. As illustrated in Figure 35 and Figure 36, according to their place of deployment, these managers have different responsibilities. For example, the Event Manager in the monitor side listens to atom level events and sends system level events. On the other hand, the

Event Manager in the atom side listens to system level events designed for checkpoint rollback recovery. All these details in the prototype design for managers will be described in the following sections.

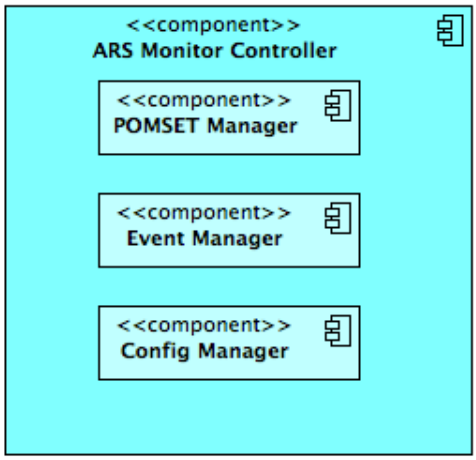


Figure 35: Monitor Prototype in Central Monitor

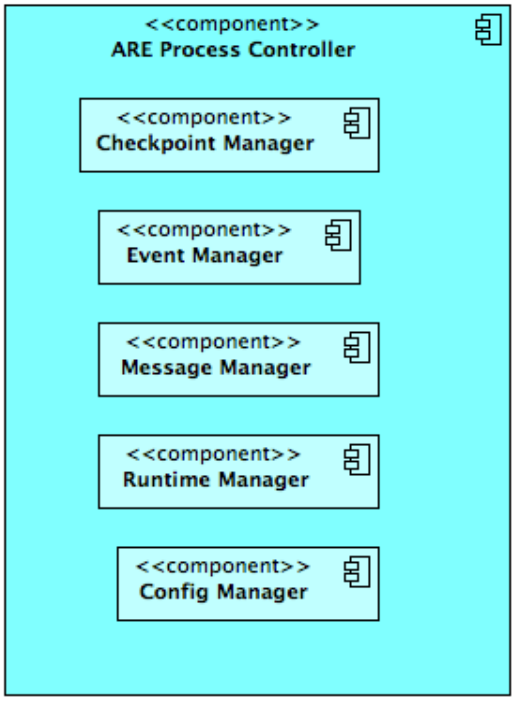


Figure 36: Monitor Prototype in Atoms

4.3 Configuration Manager

The configuration Manager allows the central monitor and the instrumented codes in the process to manage configurations without modifying programs, such as UI display format, event storage link. etc.

4.4 Event Manager

The Event Manager is mainly developed to manage event sending, receiving and remembering.

In the atom side, the Event Manager is responsible to send events corresponding to execution in atoms, and to listen to system level events for rollback recovery notification. In the monitor side, the Event Manager has an event storage that receives events sent by atoms. It also allows the central monitor event listeners to receive notification of atom level events. In fact, this event storage is assumed to be failure free and there is no omission failure occurring on event delivery. In addition to the event storage, this Event Manager sends specified system level events to instruct the appropriate atoms in the application to recover in case of a failure.

Information about various events and listeners in the runtime monitor is described in the following.

4.4.1 Event Type

Adopted from the distributed event model, the application consists of various events. Based on where they happen, these events can be classified as: atom level event or system level event, as listed in Table 3. As illustrated in Figure 37, all events extend

from *BasicEvent* that implements *Serializable* allowing it to be delivered through the network.

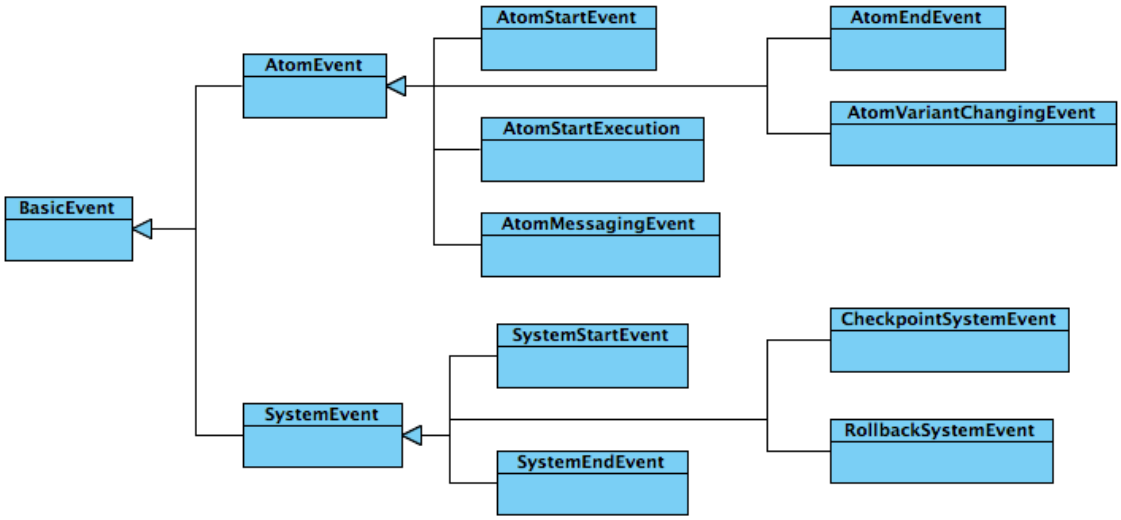


Figure 37: Event Type Relationship

In the atom side, each atom sends corresponding atom level events for predicate variable (by variant event) or different stages. In the monitor side, when the central monitor receives these atom level events, compares the current event slices (including the received events) with the given POMSET specification and checks the correct execution of the atoms by invariants. In execution runs, the runtime monitor sends system level events for two purposes. One is to notify the atoms to take a global checkpoint. The other is for failure notification, and the atoms should be recovered to the specified global consistent state and continue their normal execution.

Table 3: Event Type

Event	Level	Description
AtomStartEvent	Atom	The atom starts to receive message via channels. This event contains local checkpoint id.
AtomStartExecutionEvent	Atom	The atom starts to execute its tasks.
AtomMessagingEvent	Atom	The atom sends messages via channels. This event contains local checkpoint id.
AtomEndEvent	Atom	The atom ends after sending messages. This event contains local checkpoint id.
AtomVariantChangingEvent	Atom	Data variant changes in the atom.
SystemStartEvent	System	The system starts monitoring.
SystemEndEvent	System	The system ends monitoring.
SystemGlobalCheckpointEvent	System	The system confirms a global checkpoint composed of local checkpoints id.
SystemRollbackEvent	System	The system rollbacks to specified global checkpoint that composed of local checkpoints id.

4.4.2 Event Sender and Event Listener

In the atom side, the Event Sender sends atom level events corresponding to different stages. As illustrated in Figure 38, an atom sends *AtomStartEvent* before receiving messages via channels, *AtomStartExecutionEvent* to prepare tasks execution, *AtomMessagingEvent* to start sending messages to channels and finally *AtomEndEvent* for end status. In a critical atom, local checkpoints are stored with unique identifications. And these local checkpoints might be discarded later or become part of a group checkpoint.

The Event Listener looks for system level events for taking a group checkpoint or for performing rollback recovery. When a critical atom is notified to form a group checkpoint with its local checkpoint *LC*, it asks the Checkpoint Manager to discard all previous local checkpoints *LC'* taken before *LC*.

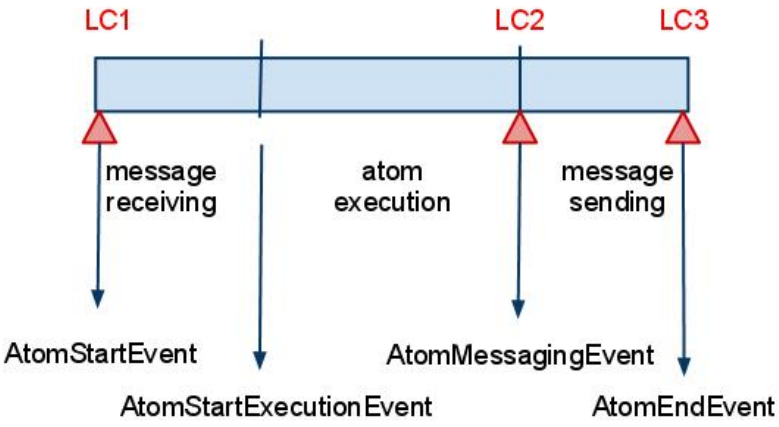


Figure 38: Event Sender

In the monitor side, when the Event Listener receives notification of atom level events, it passes the atom status to the POMSET Manager for partial order based runtime

verification. During checkpoint evaluation, if the POMSET Manager decides to take a group checkpoint, the Event Sender sends *SystemCheckpointEvent* with local checkpoint identifications of the involved critical atoms to discard previous local checkpoints. But once the POMSET Manager detects a failure, the Event Manager sends a *SystemRollbackEvent* with the last group checkpoint to notify those critical atoms to recover the execution by rolling back.

4.5 POMSET Manager

The POMSET Manager is mainly responsible for managing the partial order execution model in the monitor side, including runtime verification and runtime recovery. It maintains the specification as the POMSET tree structure, identifies the necessary checkpoints, verifies runtime failures and invokes rollback recovery, if necessary. In other words, it plays the important role of runtime monitor.

4.5.1 Node Type

As illustrated in Figure 39, a POMSET specification that describes the correct behavior of a given application consists of more than one nodes and each node represents an atom or an ordering operator between multiple atom sets.

All nodes are listed in Table 4 with node type, representation character and whether it can be shown in the POMSET tree. An atom node can be a critical node or a non-critical node depending on whether it computes and updates relevant data variable, but naming of atoms is required to be well defined. A critical atom should be named as *C_XXX* while a non-critical atom as *N_XXX* for simple identification.

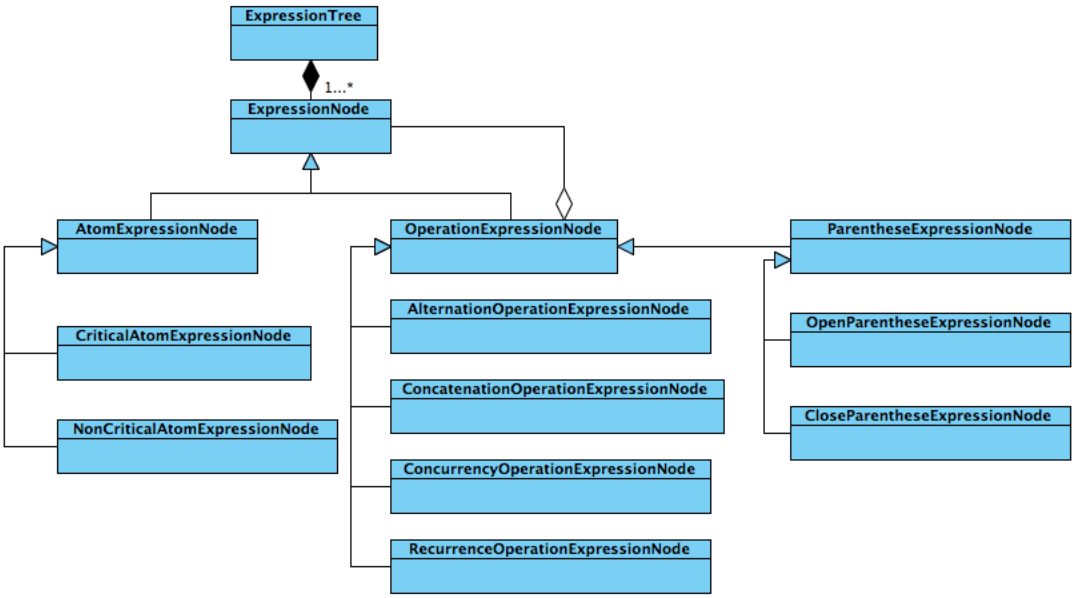


Figure 39: POMSET Tree Structure

As for operators, naming is pre-defined. An operator node can be an alternation node (+), concatenation node (;), concurrency node (||), or recurrence node (*). Most nodes can be shown in the POMSET tree, except for parentheses that only are designed to avoid ambiguity in the specification.

4.5.2 Operation Node Specification

Every operation node has its rules of specification that describe the partial-ordered sequence among multiple atom sets. The specification beginning with an open parenthesis "(" should be closed with a close parenthesis ")". A recurrence operation node "*" should only have one child node. A concatenation operation node ";" should have exactly two children nodes. An alternation operator "+" and a concurrency operator "||" should have more than one children node.

Table 4: POMSET Node List

Node	Inherited Node	Char	Show
CriticalAtomExpressionNode	AtomExpressionNode	C_XXX	Yes
NonCriticalAtomExpressionNode	AtomExpressionNode	N_XXX	Yes
AlternationOperationExpressionNode	OperationExpressionNode	+	Yes
ConcatenationOperationExpressionNode	OperationExpressionNode	;	Yes
ConcurrencyOperationExpressionNode	OperationExpressionNode	||	Yes
RecurrenceOperationExpressionNode	OperationExpressionNode	*	Yes
OpenParentheseExpressionNode	ParentheseExpression	(	No
CloseParentheseExpressionNode	ParentheseExpression	)	No

4.5.3 Node Status

During the execution runs, every node changes its status to represent its current stage. As listed in Table 5, each status has its meaning in the atom/operator.

For an operator, the beginning status is NOT_STARTED as illustrated in Figure 40. When it starts its own tasks, that is, the runtime monitor receives corresponding atom level event and compare current execution with predication, then the status changes as EXECUTING. After the tasks are completed (no failure/conflict occurs), the status

changes as ENDED. However, during runtime executions, nodes may rollback to previous statuses because of failures.

Table 5: POMSET Node Status Type

Status	For Atom	For Operator	Description
NOT_STARTED	Yes	Yes	The node does not start yet.
STARTED	Yes	No	The node is ready for message receiving.
EXECUTING	Yes	Yes	The node is executing its tasks.
MESSAGING	Yes	No	The node sends messages to other atoms.
ENDED	Yes	Yes	The node completes its tasks, including message sending.

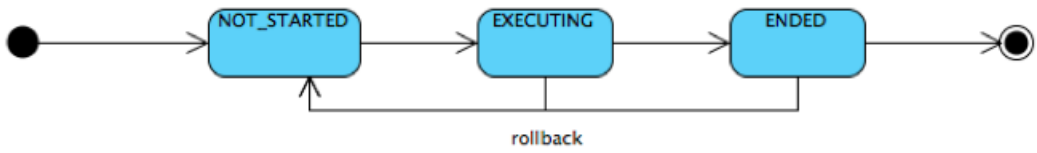


Figure 40: Operation Node Status Lifecycle

An atom has the beginning status NOT_STARTED as illustrated in Figure 41. Later, when it is ready to receive messages, it sends an event to the monitor and changes its status to STARTED. As an atom node can be divided into three parts—message

receiving, execution, and message sending, it has different status to represent its progress. When it receives messages and starts execution, the status should be changed to EXECUTING. When it starts to sending message to other atoms, its status should be changed to MESSAGING. Finally, the status changes to ENDED after sending messages. However, during execution runs, nodes may rollback to STARTED (waiting for messages) and MESSAGING (re-sending messages).

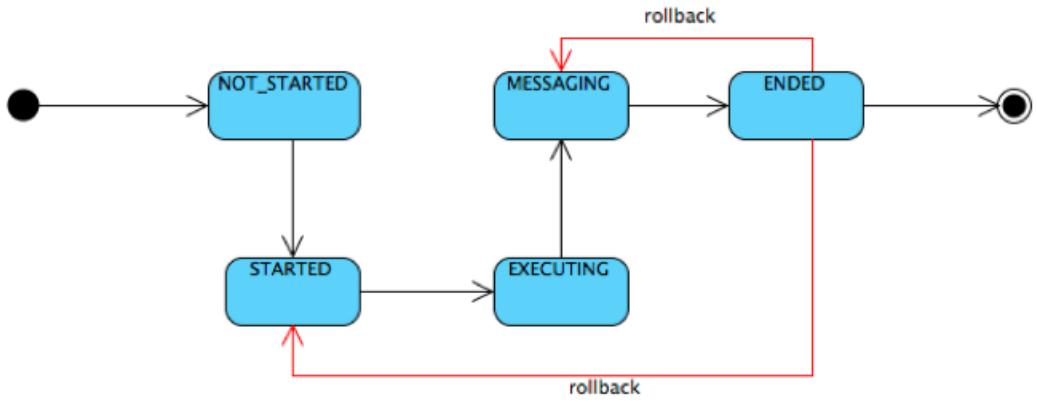


Figure 41: Atom Node Status

4.5.4 Checkpoint Evaluator

Given a POMSET specification, the Checkpoint Evaluator sets potential checkpoint values of each node: a critical atom node as True, a non-critical atom node as False, an operator node based on its operator definition and children atom sets. Then it identifies group checkpoints of operator nodes according to potential checkpoints values. Since some of the global checkpoints may not be necessary, the Static Checkpoint Reducer identifies the unnecessary global checkpoints.

4.5.5 Static Checkpoint Reducer

The Static Checkpoint Reducer implements the static checkpoint reduction algorithm. This object identifies the unnecessary checkpoints by using the priorities of the operators: recurrence operator "*" = concurrency operator "||" > concatenation operator ";" > alternation operator "+". In addition, if two operation nodes with overlapping atom sets both have global checkpoints, it compares the position of the two operators in the POMSET tree and avoids taking the group checkpoint for the lower level operator node. With this object, most unnecessary checkpoints can be avoided, except for those caused by unpredictable execution paths.

4.5.6 Dynamic Checkpoint Reducer

The Dynamic Checkpoint Reducer is designed to solve issues of unpredictable execution paths. In order to avoid those unnecessary group checkpoints that cannot be identified by the Static Checkpoint Reducer, the Dynamic Checkpoint Reducer implements the dynamic checkpoint reducing algorithm in two parts: remembering the past execution paths and forecasting the future execution paths.

It is straightforward to forecast the execution of a future iteration by remembering what happened in the past iterations. However, it costs significant resources and time to remember the whole execution in a repository. Therefore, only partial execution paths caused by alternation operators are remembered as logic rules rather than the whole execution paths. This approach comes with advantages and disadvantages. It beneficially reduces the amount of time and persistent storage, but it is difficult to identify the levels of the partial execution paths to be remembered. For the online travel agent example,

unnecessary group checkpoints are taken at two alternation operators in the POMSET specification. One only affects its parent operator as illustrated in Figure 30. The other affects the root of the POMSET tree as illustrated in Figure 31. The remembered partial execution paths are of no use if they cannot sufficiently cover the subtree for forecasting. This problem is solved by using a configuration parameter that specifies the maximum level of the POMSET specification to remember.

The Dynamic Checkpoint Reducer forecasts the next possible iteration execution according to what happened in past iteration executions. Once the execution paths are confirmed, the Dynamic Checkpoint Reducer asks the Static Checkpoint Reducer to re-evaluate the potential checkpoint values, and identify the necessary group checkpoints. This time, all the identified group checkpoints should be necessary because of the absence of unpredictable execution paths.

4.5.7 Runtime Specification Verifier

The Runtime Specification Verifier mainly implements the partial-ordered verification and checks only two requirements: the ordering among atom sets and the correct behavior of each atom. First, it maintains the current atom slices (including the received atom events) and checks the compatibility between atom slices and the given POMSET specification. Secondly, it compares the predicate variables with received relevant data events from the atoms promising global property match. In case of mismatch of any of the requirements, it flags a failure and asks the Checkpoint Manager roll back to the last group checkpoint for recovery.

4.6 Instrumented Program Codes

The runtime monitor provides a framework for instrumented program codes for the atoms, and the application programmers are required to declare well-formed atoms as in Figure 42. An atom implements the *Execution* interface and implements three predefined functions: *execute*, *checkpoint*, and *rollback*. First, all tasks implementation should be described in *execute*, necessary data to be recovered should be stored in a map inside *checkpoint*, and *rollback* specifies how to roll back the atom to a previous correct state.

```
AtomMonitor monitor = AtomMonitor.getInstance("S_C", this);
monitor.start();

monitor.startAtom();

Serializable obj = monitor.receiveMessage("I_B");

monitor.startExecution();

AccountInfo account = (AccountInfo) obj;
this.accounts.put(account.getUserName(), new Integer(account.getMoney()));
System.out.println(this.accounts.toString());

monitor.endExecution();

monitor.sendMessage("S_A", "OK");

monitor.endAtom();

monitor.stop();
```

Figure 42: Runtime Manager for Instrumented Program Codes

More than that, an atom has to be declared by a unique id and it activates its message manager for receiving messages. In different execution stages, application programmers have to command the Runtime Manager to do corresponding actions, such as *startAtom*, *startExecution*, *endExecution*, *endAtom* described in Table 6..

In brief, the instrumented program codes control background events/messages handling by cooperating with the Event Manager and manage checkpoint instances with the Checkpoint Manager.

Table 6: Runtime Manager for Instrumented Program Codes Function List

Function	Description
startAtom	Atom is ready for message receiving, and its status should be changed from NOT_STARTED to STARTED
startExecution	Atom starts executing its tasks described in execute(), and its status should be changed from STARTED to EXECUTING
endExecution	Atom completes tasks execution to send message via channels and its status should be changed from EXECUTING to MESSAGING.
endAtom	Atom ends with status changed from MESSAGING to ENDED.

4.6.1 Message Manager

The Message Manager is mainly responsible for message sending and receiving via channels. All message types should implement *Serializable* for network transmission. An atom can receive serialized objects from the other atoms and cast them into other classes, because the Message Manager captures all sending and receiving messages with checkpoint id by another thread handling system I/O despite the main thread. All message life period depends on the checkpoint. If the local checkpoint is to be discarded, meaning

that local checkpoint is no longer used, all messages with the local checkpoint id should also be discarded. With this approach, the Message Manager can provide flexible object message sending and receiving functions without affecting the main thread performance.

4.6.2 Checkpoint Manager

The Checkpoint Manager mainly controls the local checkpoints of the atoms, including unique id and data mapping. First of all, when an atom is started, the Checkpoint Manager makes an initial checkpoint as *LC1*. When the atom ends its execution and starts to send messages, it makes another checkpoint *LC2*, and the final checkpoint *LC3* for the end stage. However, a group checkpoint is composed by local checkpoints of the involved critical atoms. The local checkpoint is meaningful only when it is part of the latest group checkpoint. In other words, when an atom is notified that there is another group checkpoint G1 composed of *LC2* by a *SystemCheckpointEvent*, the Checkpoint Manager should discard all local checkpoints before *LC2*, such as *LC1*. The Checkpoint Manager also provides rollback recovery invocation, when an atom is notified to roll back to a specific local checkpoint; the Checkpoint Manager retrieves the stored data map to the Runtime Manager for rollback recovery.

4.7 Observation on the Online Travel Agent Application

The runtime rollback monitor initially identifies 10 group checkpoints for the online travel agent application as illustrated in Figure 17, and two of them are avoided because they are identified as unnecessary by the static checkpoint reduction algorithm as illustrated in Figure 27. Another two group checkpoints may be avoided by precise forecasting by the dynamic checkpoint reduction algorithm during iteration runs as

illustrated in Figure 30 and Figure 31. Most detail about how the static/dynamic checkpoint reduction algorithms applied on the online travel agent application are described in Chapter 3.

In addition, the performance penalty due to the instrumented program codes in the atoms is acceptable. As illustrated in Figure 43, the difference between average execution time without monitor (the blue line) and with monitor (the green line) is quite subtle. Only atoms *G* and *O* have obvious but tolerable differences, because both of these atoms rely on message inputs and outputs more than other atoms. The instrumented program codes in these atoms require more time to recover a number of messages.

However, the average recovery time for critical atoms is tolerable but not perfect, as illustrated in Figure 44. The recovery time not only depends on the size of restored data but also on the number of messages.

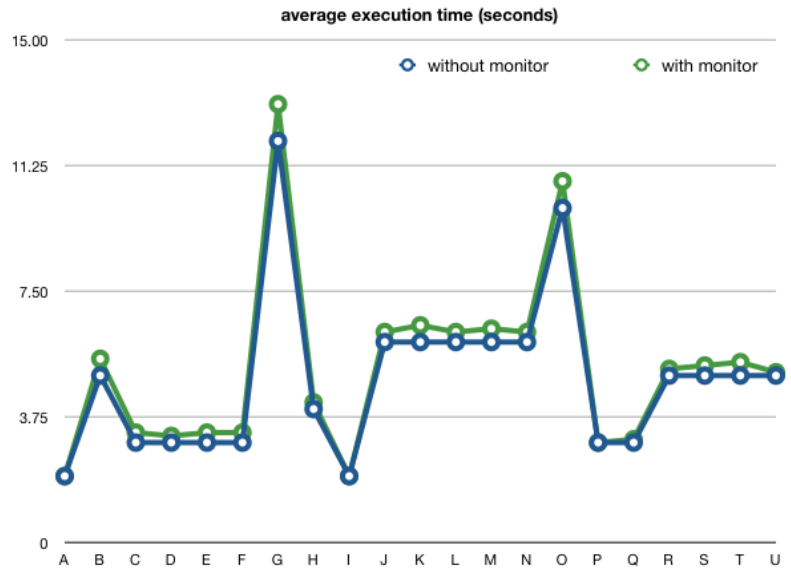


Figure 43: Average Execution Time for Instrumented Program Codes

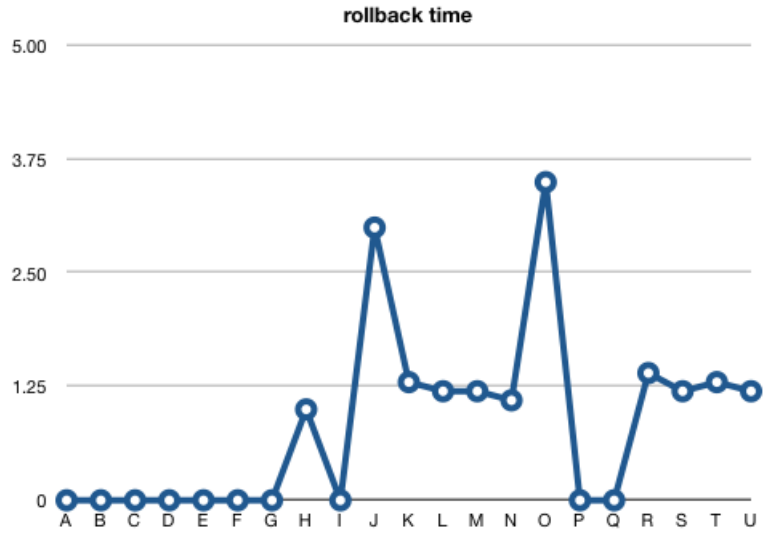


Figure 44: Average Recovery Time for Atoms

From the discussions so far, it should be clear that the partial order based runtime recovery technique significantly improves the recovery efficiency by reducing both the amount of persistent store used and the time required for recovery. In comparison with traditional checkpoint-based rollback recovery (assume taking a checkpoint per minute) shown in Table 7, the partial order based runtime recovery reduces 65% of stored space as listed in Table 8. This technique also reduces the recovery time because it can specify which last checkpoint should be used for recovery without domino effect. Note that this improvement is application dependent and it changes with the checkpoint frequency, number and granularity of atoms, etc.

4.8 Summary

This chapter detailed a prototype implementation of the runtime recovery monitor that extends the runtime verification for partial-ordered multi-set model and manages events, status, and checkpoints for rollback recovery. Instrumented program codes in

each atom/process send responsible relevant events to a central monitor, which compares the received events with the expected execution specified by the POMSET tree. From the correct behavior of a distributed application, the runtime monitor identifies the group checkpoints to be taken by operators in the POMSET tree, and avoids unnecessary group checkpoints by implementing the static checkpoint reduction algorithm and the dynamic checkpoint reduction algorithm. This monitor also manages the local checkpoints of critical atoms and keeps the last group checkpoint by discarding the previous ones. When the runtime monitor detects a failure, it automatically and immediately rolls the system back to the last stored state and continues the normal execution of system.

The operation of this prototype is also explained on the example highly available distributed application – the online travel agent – along with the result of observation on the runtime rollback recovery monitor.

Table 7: Amount of Storage Space for Traditional Checkpoint Rollback Recovery

Process Name	Amount Of Storage Space For Checkpoint (mb)
User	0.2
Agent	5.3
Hotel Service	4.1
Transportation Service	4.6
Subtotal	14.2
Total Of 2 Checkpoints	28.4

Table 8: Amount of Storage Space for Partial Order Based Runtime Recovery

Atom Name	Amount Of Storage Space For Checkpoint (mb)
H	0.2
J	0.4
K	1.5
L	0
M	0
N	1.7
O	1.4
R	2.0
S	0
T	0
U	2.4
Total	9.6

CHAPTER 5 : CONCLUSION AND FUTURE WORK

This thesis developed a partial order based runtime recovery technique that can be used with highly available distributed applications. Given a partially ordered multiset (POMSET) specifying the correct behavior of the distributed application, this technique can identify at which operator nodes in the POMSET tree should group checkpoints (global consistent states) be taken. First, the atoms in the distributed application are categorized as critical atoms and non-critical atoms according to whether the atom changes the invariant variable that impacts the progress of application. Based on the potential checkpoint properties of operators, that indicates the intended checkpoint decisions, this technique identifies when group checkpoints (the collection of local checkpoints in atoms) should be taken. Furthermore, two algorithms are developed to avoid unnecessary checkpoints – one by priority and position of operators (static checkpoint reduction) and the other by predicting future execution paths according to what has occurred in the past iteration runs (dynamic checkpoint reduction).

In the developed proof-of-concept prototype, the application developer specifies the POMSET specification with program atoms (set of events) and the atom themselves with instrumented code to send the application events to the runtime recovery system. The runtime recovery system checks the occurrence of the events reported by the program atoms with the POMSET specification (stored in memory as the POMSET tree) and detects a failure when an event does not satisfy the specification and recovers the application by rolling it back to a past correct execution point using the saved checkpoint. The runtime recovery system only maintains the necessary checkpoints (instead of

periodic checkpoints) and further reduces the number of checkpoints maintained by avoiding unnecessary checkpoints using the developed static and dynamic checkpoint reduction algorithms.

As explained in Chapter 3 and illustrated in Chapter 4, the partial order based runtime recovery technique significantly improves the efficiency on the required storage space (space efficiency) and the recovery time (time efficiency) while ensuring the application is highly available. Unlike traditional checkpoint based rollback recovery techniques, in which all processes participate to form a global consistent state (global checkpoint), each group checkpoint in the runtime recovery is a collection of the local checkpoints of only the necessary atoms. Given a POMSET tree specifying the correct behavior of the application, this technique identifies the group checkpoints to be taken by operator nodes. In addition, the two checkpoint reduction algorithms developed identify and avoid the unnecessary checkpoints. The developed proof of concept prototype shows that this technique does not impact the performance of the application in any significant manner.

However, this technique can be further improved by solving the following issues in the future. Given a POMSET specification, this technique can figure out the optimal (the least) number of group checkpoints necessary. In order to find out the optimal solution, appropriate algorithms/properties should be developed. For example, in Figure 45, if the concurrent operator contains all non-critical atoms, it is unnecessary to take a

group checkpoint (now the concurrent operator always takes a group checkpoint).

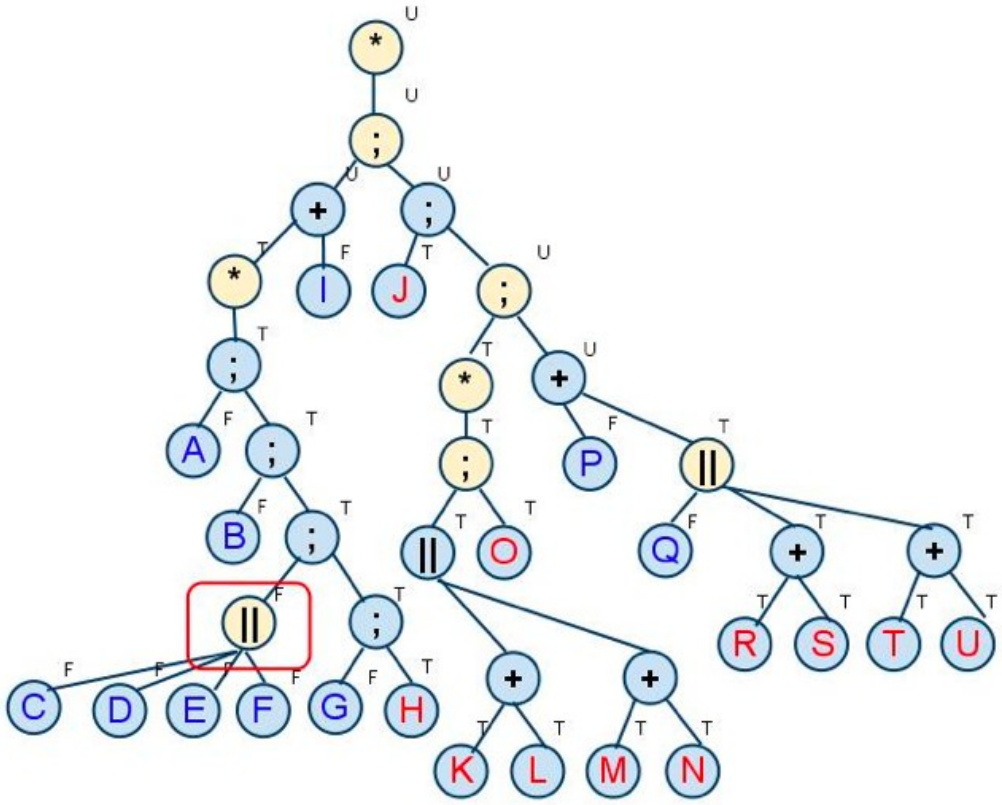


Figure 45: Issues to be solved – Group Checkpoint of Concurrent Operator

In addition, the central runtime monitor used in the developed technique (that extends the partial order based runtime verification into runtime recovery and manages the checkpoints/messages/events) is supposed to be failure-free. If a failure occurs, it rolls the monitored application back to the last stored group checkpoint so that the application can continue its normal execution. If a prototype of this technique is implemented using a distributed monitor, it can further improve the availability and reliability of the application.

As described in Chapter 4, this technique relies on application programmers to provide correct information in order to consider the distributed application as a black box.

However, this requires that the application programmers have sufficient skills and experience with the POMSET model and specification, and program instrumentation. Therefore, developing techniques to automatically instrumenting the source code of the target application, which can organize events as well-formed atoms and send data variant events to the monitor by analyzing the original application code, will be more practical.

REFERENCES

Replication

[SMNTWB02] D. Sames, B. Matt, B. Niebuhr, G. Tally, B. Whitmore and D. Bakken, "Developing a heterogeneous intrusion tolerant CORBA system," in Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on, 2002, pp. 239-248.

[RR06] L. Rodrigues and M. Raynal, "Atomic broadcast in asynchronous crash-recovery distributed systems and its use in quorum-based replication," Knowledge and Data Engineering, IEEE Transactions on, vol. 15, pp. 1206-1217, 2003.

[RKSC06] K. Ravindran, K. A. Kwiat, A. Sabbir and B. Cao, "Replica voting: A distributed middleware service for real-time dependable systems," in Communication System Software and Middleware, 2006. Comsware 2006. First International Conference on, 2006, pp. 1-7.

[OFG07] J. Osrael, L. Froihofer and K. M. Goeschka, "Availability/Consistency balancing replication model," in Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International, 2007, pp. 1-8.

State Space Explosion

[YUA88] M. C. Yuang, "Survey of protocol verification techniques based on finite state machine models," in Computer Networking Symposium, 1988., Proceedings of the, 1988, pp. 164-172.

[BCMDH 90] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill and L. J. Hwang, "Symbolic model checking: 1020 states and beyond," in Logic in Computer Science, 1990. LICS '90, Proceedings., Fifth Annual IEEE Symposium on e, 1990, pp. 428-439.

State-Space Model

[LHLL08] Lixian Liu, Bingxin Han, Jinbo Li and Xinling Li, "A globally optimized state-space model identification method," in Intelligent Control and Automation, 2008. WCICA 2008. 7th World Congress on, 2008, pp. 4741-4744.

Partial-Order Model

[AM94] M. Ahuja and S. Mishra, "Units of computation in fault-tolerant distributed systems," in Distributed Computing Systems, 1994., Proceedings of the 14th International Conference on, 1994, pp. 626-633.

[MG03] N. Mittal and V. K. Garg, "Software fault tolerance of distributed programs using computation slicing," in Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on, 2003, pp. 105-113.

[RG04] H. F. Li, J. Rilling and D. Goswami, "Granularity-Driven Dynamic Predicate Slicing Algorithms for Message Passing Systems," Automated Software Engg., vol. 11, pp. 63-89, January, 2004.

[LM07] H. F. Li and E. Al Maghayreh, "Checking distributed programs with partially ordered atoms," in Software Engineering Conference, 2007. APSEC 2007. 14th Asia-Pacific, 2007, pp. 518-525.

[LMG07] H. F. Li, E. Al Maghayreh and D. Goswami, "Using atoms to simplify distributed programs checking," in Dependable, Autonomic and Secure Computing, 2007. DASC 2007. Third IEEE International Symposium on, 2007, pp. 75-83.

[GAO10] Xiangyu Gao “Design and Implementation of A Partial-Order Semantics Based Runtime Verification Toolset for Distributed Java Programs” Master Of Computer Science Department, 2010

Global Consistent State

[LAMP78] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," Commun ACM, vol. 21, pp. 558-565, 1978.

[CL85] K. M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," ACM Trans.Comput.Syst., vol. 3, pp. 63-75, 1985.

[PAH08] S. Pourmahmoud, S. Asbaghi and A. T. Haghighat, "A new way of calculating the recovery line through eliminating useless checkpoints

in distributed systems," in Computer and Information Sciences, 2008. ISCIS '08. 23rd International Symposium on, 2008, pp. 1-4.

Domino Effect

[RAN75] B. Randell, "System structure for software fault tolerance," in Proceedings of the International Conference on Reliable Software, Los Angeles, California, 1975, pp. 437-449.

[BCS84] D. Briatico, Augusto Ciuffoletti, Luca Simoncini. A Distributed Domino-Effect free recovery Algorithm. In Proceedings of Symposium on Reliability in Distributed Software and Database Systems'1984. pp.207~215.

Application Level Implementation

[JLSU87] J. Joyce, G. Lomow, K. Slind and B. Unger, "Monitoring distributed systems," ACM Trans.Comput.Syst., vol. 5, pp. 121-150, 1987.

[SG91] L. Strigini and F. Di Giandomenico, "Flexible schemes for application-level fault tolerance," in Reliable Distributed Systems, 1991. Proceedings., Tenth Symposium on, 1991, pp. 86-95.

[SBFMPS04] M. Schulz, G. Bronevetsky, R. Fernandes, D. Marques, K. Pingali and P. Stodghill, "Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for MPI programs," in SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing, 2004, pp. 38.

Other Implementation

[XR96] Jie Xu and B. Randell, "Roll-forward error recovery in embedded real-time systems," in Parallel and Distributed Systems, 1996. Proceedings., 1996 International Conference on, 1996, pp. 414-421.

[LF90] C. -. J. Li and W. K. Fuchs, "CATCH-compiler-assisted techniques for checkpointing," in Fault-Tolerant Computing, 1990. FTCS-20. Digest of Papers., 20th International Symposium, 1990, pp. 74-81.

Rollback

[CR72] K. M. Chandy and C. V. Ramamoorthy, "Rollback and Recovery Strategies for Computer Programs," Computers, IEEE Transactions on, vol. C-21, pp. 546-556, 1972.

[CW92] J. Cao and K. C. Wang, "An abstract model of rollback recovery control in distributed systems," SIGOPS Oper.Syst.Rev., vol. 26, pp. 62-76, 1992.

[EZ92] E. N. Elnozahy and W. Zwaenepoel, "Manetho: transparent roll back-recovery with low overhead, limited rollback, and fast output commit," Computers, IEEE Transactions on, vol. 41, pp. 526-531, 1992.

[CY96] Ge-Ming Chiu and Cheng-Ru Young, "Efficient rollback-recovery technique in distributed computing systems," Parallel and Distributed Systems, IEEE Transactions on, vol. 7, pp. 565-577, 1996.

[EA02] E. N. Elnozahy, L. Alvisi, Y. Wang and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," ACM Comput.Surv., vol. 34, pp. 375-408, 2002.

Checkpoint-Based Rollback

[KT87] R. Koo and S. Teoug, "Checkpointing and Rollback-Recovery for Distributed Systems," Software Engineering, IEEE Transactions on, vol. SE-13, pp. 23-31, 1987.

[EJZ92] E. N. Elnozahy, D. B. Johnson and W. Zwaenepoel, "The performance of consistent checkpointing," in Reliable Distributed Systems, 1992. Proceedings., 11th Symposium on, 1992, pp. 39-47.

[BBHMR95] R. Baldoni, J. Brzezinski, J. M. Helary, A. Mostefaoui and M. Raynal, "Characterization of consistent global checkpoints in large-scale distributed systems," in Distributed Computing Systems, 1995., Proceedings of the Fifth IEEE Computer Society Workshop on Future Trends of, 1995, pp. 314-323.

[EP04] E. N. Elnozahy and J. S. Plank, "Checkpointing for peta-scale systems: a look into the future of practical rollback-recovery," Dependable and Secure Computing, IEEE Transactions on, vol. 1, pp. 97-108, 2004.

Independent Checkpoint Recovery

[BL88] B. Bhargava and Shu-Renn Lian, "Independent checkpointing and concurrent rollback for recovery in distributed systems-an optimistic

approach," in Reliable Distributed Systems, 1988. Proceedings., Seventh Symposium on, 1988, pp. 3-12.

[TRI96] P. Triantafiliou, "Independent recovery in large-scale distributed systems," Software Engineering, IEEE Transactions on, vol. 22, pp. 812-826, 1996.

Coordinated Checkpoint Recovery

[BLKC03] A. Bouteiller, P. Lemarinier, K. Krawezik and F. Capello, "Coordinated checkpoint versus message log for fault tolerant MPI," in Cluster Computing, 2003. Proceedings. 2003 IEEE International Conference on, 2003, pp. 242-250.

[CLG05] Jiannong Cao, Yinghao Li and Minyi Guo, "Process migration for MPI applications based on coordinated checkpoint," in Parallel and Distributed Systems, 2005. Proceedings. 11th International Conference on, 2005, pp. 306-312 Vol. 1.

[LPN05] O. Laadan, D. Phung and J. Nieh, "Transparent checkpoint-restart of distributed applications on commodity clusters," in Cluster Computing, 2005. IEEE International, 2005, pp. 1-13.

[BGR06] X. Besseron, S. Jafar, T. Gautier and J. -. Roch, "CCK: An improved coordinated Checkpoint/Rollback protocol for dataflow applications in kaapi," in Information and Communication Technologies, 2006. ICTTA '06. 2nd, 2006, pp. 3353-3358.

Communication Induced Checkpoint Recovery

[HMNR97] J. -. Helary, A. Mostefaoui, R. H. B. Netzer and M. Raynal, "Preventing useless checkpoints in distributed computations," in Reliable Distributed Systems, 1997. Proceedings., the Sixteenth Symposium on, 1997, pp. 183-190.

[AER99] L. Alvisi, E. Elnozahy, S. Rao, S. A. Husain and A. de Mel, "An analysis of communication induced checkpointing," in Fault-Tolerant Computing, 1999. Digest of Papers. Twenty-Ninth Annual International Symposium on, 1999, pp. 242-249.

[BG00] R. Belhassine-Cherif and A. Ghedamsi, "Diagnostic tests for communicating nondeterministic finite state machines," in Computers and Communications, 2000. Proceedings. ISCC 2000. Fifth IEEE Symposium on, 2000, pp. 424-429.

[BG01] R. Belhassine-Cherif and A. Ghedamsi, "Multiple fault diagnostics for communicating nondeterministic finite state machines," in Computers and Communications, 2001. Proceedings. Sixth IEEE Symposium on, 2001, pp. 661-666.

Message Logged-Based Rollback

[JOH90] D. B. Johnson, "Distributed system fault tolerance using message logging and checkpointing," 1990.

[EZ94] E. N. Elnozahy and W. Zwaenepoel, "On the use and implementation of message logging," in Fault-Tolerant Computing, 1994. FTCS-24.

Digest of Papers., Twenty-Fourth International symposium on, 1994, pp. 298-307.

[AHM95] L. Alvisi, B. Hoppe and K. Marzullo, "Nonblocking and orphan-free message logging protocols," in Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'., Twenty-Fifth International Symposium on, 1995, pp. 229.

[ALV96] L. Alvisi, "Understanding the message logging paradigm for masking process crashes," 1996.

[AM98] L. Alvisi and K. Marzullo, "Message logging: pessimistic, optimistic, causal, and optimal," Software Engineering, IEEE Transactions on, vol. 24, pp. 149-159, 1998.

[AV98] S. Rao, L. Alvisi and H. M. Vin, "The cost of recovery in message logging protocols," in Reliable Distributed Systems, 1998. Proceedings. Seventeenth IEEE Symposium on, 1998, pp. 10-18.

Optimistic Message Logging Recovery

[SY85] R. Strom and S. Yemini, "Optimistic recovery in distributed systems," ACM Trans.Comput.Syst., vol. 3, pp. 204-226, 1985.

[HW95] Yennun Huang and Yi-Min Wang, "Why optimistic message logging has not been used in telecommunications systems," in Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on, 1995, pp. 459-463.

Pessimistic Message Logging Recovery

[EZ92] E. N. Elnozahy and W. Zwaenepoel, "Manetho: transparent roll back-recovery with low overhead, limited rollback, and fast output commit," Computers, IEEE Transactions on, vol. 41, pp. 526-531, 1992.

[BCHKLM03] A. Bouteiller, F. Cappello, T. Herault, G. Krawezik, P. Lemarinier and F. Magniette, "MPICH-V2: A fault tolerant MPI for volatile nodes based on pessimistic sender based message logging," in SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing, 2003, pp. 25.

Casual Message Logging Recovery

[AM96] L. Alvisi and K. Marzullo, "Trade-offs in implementing causal message logging protocols," in PODC '96: Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, Pennsylvania, United States, 1996, pp. 58-67.

[BMA98] K. Bhatia, K. Marzullo and L. Alvisi, "The relative overhead of piggybacking in causal message logging protocols," in Reliable Distributed Systems, 1998. Proceedings. Seventeenth IEEE Symposium on, 1998, pp. 348-353.

[LPYC98] Byoungjoo Lee, Taesoon Park, H. Y. Yeom and Yookun Cho, "An efficient algorithm for causal message logging," in Reliable

Distributed Systems, 1998. Proceedings. Seventeenth IEEE Symposium on, 1998, pp. 19-25.

[MG98] J. R. Mitchell and V. K. Garg, "A non-blocking recovery algorithm for causal message logging," in Reliable Distributed Systems, 1998. Proceedings. Seventeenth IEEE Symposium on, 1998, pp. 3-9.

[BCHLC05] A. Bouteiller, B. Collin, T. Herault, P. Lemarinier and F. Cappello, "Impact of event logger on causal message logging protocols for fault tolerant MPI," in Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International, 2005, pp. 97-97.

Combination of Checkpoint and Message Logging Example

[RN08] C. D. V. Rao and M. M. Naidu, "A new, efficient coordinated checkpointing protocol combined with selective sender-based message logging," in Computer Systems and Applications, 2008. AICCSA 2008. IEEE/ACS International Conference on, 2008, pp. 444-447.