INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

A Modular Approach to Formal Specification and Verification of Dependable Distributed Protocols

Da Qi Ren

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Applied Science at

Concordia University

Montréal, Québec, Canada

July 2002

ABSTRACT

A Modular Approach to Formal Specification and Verification of Dependable Distributed Protocols

Dependable distributed system typically utilize a hierarchy of protocols to provide for reliable and timely services. Such protocols have both dependability and real-time attributes, and the analysis of these protocols is a problem of growing complexity. The development of precise and accurate formal specifications of these protocols and their subsequent formal verification to gain assurance have been a great challenge. Exploiting the inherent modularity in the design of most dependable protocols, in this thesis, we present our modular approach to specification composition and verification of dependable distributed protocol. In particular, we consider redundancy management protocols that are needed to manage redundant resources used in the system for dependability purposes. Utilizing building-block protocols inherently used in redundancy management protocols, we perform compositional specification and verification of a checkpointing and recovery protocol based on them. The key idea is that if a library of these basic components, like the primitives and sub-protocols are being formulated, then these elements aid in systematic and hierarchical development of dependable distributed protocols. The main contribution of this thesis to illustrate the fact that by defining *a priori* validated building-blocks for dependable distributed protocols, larger and more complex protocols can be easily specified and verified. For a mechanical support in formal verification process, we use formal tools such as Specware and PVS.

Dedicated to my dearest parents .....

ACKNOWLEDGMENTS

I would like to thank my academic advisor, Dr. Purnendu Sinha, for invaluable supports, guidance and encouragement that he has provided over the past years.

I would like to thank Vasudevan Janarthanan for his very good suggestions. I would like to thank all my friends who have provided me all kinds of help in the past years.

Most of all, I would like to thank my parents for their love and encouragement. I could not have completed my degree without their continuous and immeasurable support.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

In this chapter, we first introduce some basic concepts and terminology regarding dependable distributed computing system, and then describe the precise definitions of the notions of faults which are the potential source of system undependability. Following we present the protocols for managing redundancy, the difficulties in analyzing these protocols and the use of formal method to alleviate these difficulties. In the last section, we conclude with detailing the contribution of this thesis.

1.1 Dependable Distributed System Overview

Distributed Computing System (DCS) is a collection of autonomous computers interconnected through a communication network to satisfy the information processing needs of modern enterprises. Technically, the computers do not share the main memory so that the information cannot be transferred through global variables. The information between the computers is exchanged only through messages over a network. The restriction of no shared memory and information-exchange through messages is of key importance because it distinguishes distributed computing systems from shared memory multiprocessor computing systems. This definition requires that the DCS computers are connected through a network which is responsible for

the information exchange between computers. The definition also requires that the computers to work together and cooperate with each other to satisfy enterprise needs [29].

1.1.1 Dependable Distributed System

The dependability for computer system is defined as the basic trustworthiness of a computer system such that reliance can justifiably be placed on the service it delivers [6]. Dependability is thus a global concept that subsumes the usual attributes of reliability, availability, safety, and security. Not only are computers becoming more pervasive, they are also being used in critical applications where failures resulting in deviation from specified service can have disastrous consequences. For example, air traffic control, banking, and nuclear reactor control are all applications that fit into this category. Moreover, there are many, well-documented instances where problems in hardware and/or software have in fact caused system failures resulting in the loss of life or substantial economic disruptions.

We give definitions of the notions of failures, errors, faults as follow:

- *A system failure* is said to occur when the service delivered by the system no longer complies with its specification.
- *An error* is that part of the system state that is liable to lead to failure, An error affecting the service is an indication that a failure occurs or has occurred.
- *A fault* is the adjudged or hypothesized cause of an error.

An error is thus the manifestation of a fault in the system, and a failure is the effect of an error on the service.

A distributed system often encompasses software architecture to implement functionalities to provide dependability in the system. To accomplish this, the software must be constructed as fault-tolerant software, that is, software that can

continue to provide service despite some number and type of failures. Fault tolerance - the ability to provide service complying with the specification in spite of faults - may be seen as complementary to fault-prevention techniques aimed at improving the quality of components and procedures to decrease the frequency at which faults occur or are introduced into the system. Fault tolerance is achieved by error processing and by fault treatment. Error processing is aimed at removing errors from the computational state before a failure occurs, if possible, fault treatment is aimed at preventing faults from being activated again.

1.1.2 The Structure of Dependable Distributed System

The hardware basis for a distributed system consists of a collection of processors connected by a communication network. Each processor has its own local memory, but there is typically no shared memory between processors. This property implies that the only means for processes executing on different machines to communicate is by message exchange.

The actual configuration of the network can vary within these constraints, ranging from, for example, local-area broadcast networks like an Ethernet, to store-and-forward networks like those commonly used for wide-area communication. The software found on such systems varies widely in its overall structure and organization, but can generally be divided into application software and system software.

The software consists of the standard operating system services providing abstractions such as processes and virtual memory, a fault-tolerance support layer realizing the fault-tolerant services, and finally the application software. As is standard in such a level-structured organization, each layer uses the abstractions defined by the level below it to implement its own services. The purpose of the fault-tolerance support layer is to implement abstractions i.e., fault-tolerant services that simplify the programming of distributed applications requiring resilience to failures. For many of these abstractions, the implementation requires that software components

on two or more machines to communicate and cooperate, so the most accurate way to view this support layer is as a single logical entity that spans multiple machines [29].

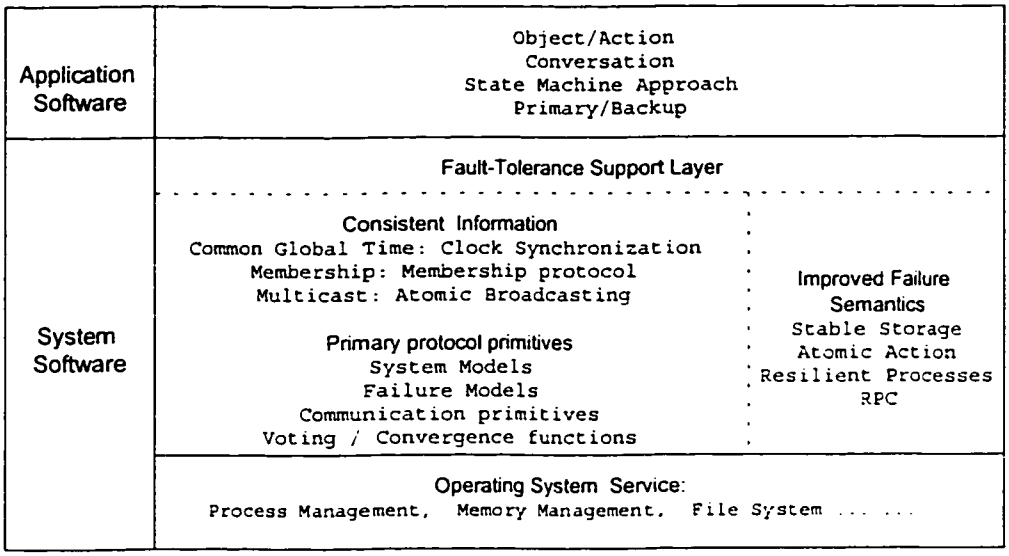

Figure 1.1: Distributed System Model

Protocols are the set of rules that software components on different machines use to realize a given abstraction. We also use this term informally to refer to the actual software components implementing these rules, so each fault-tolerant service can be thought of as being implemented by one or more protocols on each machine. These protocols are typically identical on all machines.

The abstractions implemented by the fault-tolerance support layer can be classified into two general categories based on the kind of functionality they provide. The first contains those abstractions that are similar to features found in standard systems, but with improved failure semantics. Examples of such abstractions in Figure 1.1 include stable storage, atomic actions, resilient processes, and certain types of remote procedure call (RPC). Stable storage is data storage that suffers no failures itself and is not affected by the failure of other components; thus, stable storage is similar to standard memory or disk storage, but with better semantics in the face

of failures. Atomic actions are sequences of instructions potentially spanning multiple machines that are guaranteed to either execute completely or not at all despite failures; again, this makes atomic actions similar to standard sequential execution sequences, but with better behavior when failures occur. Resilient processes are processes that can continue executing correctly even if interrupted by failure and then restarted; the similarity here is, of course, to regular processes, but with the ability to tolerate failures. Finally, RPC refers to a collection of interprocess communication protocols commonly used in distributed systems that attempt to provide semantics similar to procedure calls; while most of these protocols simply terminate a call abnormally when failures occurs, a few actually provide strong fault-tolerance guarantees.

The second category encompasses abstractions whose purpose is to provide consistent information to processes executing on different machines in a distributed system. Examples of such abstractions in Figure 1.1 include common global time, membership, and multi-cast. Common global time provides a consistent time base for all machines despite the lack of a single physical clock; this service is especially useful for consistently ordering events in a distributed system. Membership is a service that provides a consistent view of which processors are functioning and which have failed at any given moment in time.

Finally, multi-cast is a communication service that allows a message to be transmitted asynchronously to a group of processes rather than just a single process; properties often associated with multi-cast primitives designed for fault-tolerant systems include atomicity and various ordering properties, which ensure that messages are delivered to all processes in some sort of consistent order. The collection of fault-tolerant services needed for a given application and the exact way in which they are implemented depend on a multitude of factors. Of these, three can be identified as especially important: the programming paradigm used for the application, the failure model assumed, and the synchrony of the system. The first refers to the way

in which the application software is organized; as mentioned in the Introduction, several canonical structuring techniques have been identified, so we concentrate our attention on these. The failure model is the type of failure a component in the system is assumed to suffer; common failure models used for fault-tolerant distributed systems include fail-stop.

1.2 Redundancy in Dependable Distributed System

Redundancy is a well established approach for providing dependable services. In error detection and recovery, the fact that the system is in an erroneous state must first be ascertained. An error-free state is then substituted for the erroneous one. This error-free state may be some past state of the system or some entirely new state. In error compensation, the erroneous state contains enough redundancy for the system to be able to deliver an error-free service from the erroneous state.

Redundancy can be manifested in two ways, spatial or temporal. Spatial redundancy uses extra hardware or functional modules to guard against the effect of failures. Temporal redundancy involves extra executions of the same calculation (probably by different methods) and comparisons of the results to determine if any discrepancies exist.

The basic concept of temporal redundancy is to perform the same computation two or more times and compare the results to detect failures or discrepancies. Temporal or time redundancy attempts to reduce the amount of extra units at the expense of using additional time. In case an error is detected, the computation is performed again to see whether the discrepancies exist or not. This approach can detect errors resulting from transient faults. Time redundancy can be used for implementing backward error recovery. The simplest form of backward error recovery is retry, where the failed instruction is repeated again. Other forms include rolling

back the failed computation to a previous checkpoint (or recovery point) and continuing from there, or (may be) restarting the computation all the way from the beginning.

1.3 Protocols for Managing Redundancy

Protocols for the redundancy management define the nature of redundancy and the associated recovery process. We list the protocols which are required in formulating redundancy management procedures in this section.

Redundancy requires some form of synchronization among the independent data sources. The objective of clock synchronization primitive is to establish and maintain a consistent, system-wide time base among the various processors in the system. The basic way to achieve fault-tolerant synchronization is for each processor to periodically execute a protocol that involves exchanging clock values with the other processors, computing a reference value, and appropriately adjusting its local clock to reflect the consensus.

Membership protocol is one of the most fundamental services in fault-tolerant distributed systems to maintain a consistent, system-wide view of which processes are functioning at any given moment. Atomic Broadcasting protocol is one of the multi-cast communication mechanism which can provide agreement, integrity, termination and total order properties has to be employed for supporting membership protocol services. Both of the protocols need synchronized clock values, which presented by clock synchronization protocol.

Checkpointing is a commonly used technique to provide for sustained operations in a distributed system in the presence of transient faults, without incurring the high performance cost of restarting processes from scratch as transients are encountered. Over the process flow, periodic or aperiodic, consistent or inconsistent checkpoints are set up, till where the consistency of a task execution is assured.

In case of a transient, the process needs to only roll back to the past consistent checkpoint and restart rather than rolling back to the initial stage of the process.

1.4 Analyzing Dependable Protocols Using Formal Method

Formal methods have been extensively used for proving the correctness of fault-tolerant system design and implementation because of its supporting to the formal specification and verification of dependable distributed protocols, especially it support automated and exhaustive state explorations based on the analysis of operations of the protocol.

The formal system consists of a formal language and a deductive system. The language may be introduced either informally or using a metalanguage. the syntax of the language provides representational abstraction for objects in the conceptual domain. The semantics of the language characterizes those statements in the language that are valid for the conceptual domain. A deductive system is a machinery for conducting inferences in the construction of proofs for logical assertions. A sound deductive approach must include only correct representations and correct mathematical reasoning. The problem description, the specification of domain objects and their interdependence, and the program satisfying the specification represent models at different levels of abstraction.

The redundancy management protocols are typically used for delivering reliable and timely services in dependable system. Obviously, the specification and verification of each different distributed protocol requires an individualistic effort, because they have to guarantee the correctness of the description of the protocol and its desired properties. The kinds of system properties might include functional behavior, timing behavior, performance characteristics, or internal structure.

The analysis of these dependable protocols involves investigating the extremely

large operational state spaces. This operational state space issue and the arbitrary large number of possible execution paths to explore in these protocols constrains the effectiveness of conventional specification and verification techniques.

1.5 Related Work

There has been considerable effort in the development of the modularization technique for simplifying complex real-time and fault-tolerant systems. Most approaches to modularize dependable/distributed protocols focus on developing an implementation of the protocol by combining selected components or micro-protocols. Some of these approaches provide for a formal framework to ascertain configurations against system specifications.

We list the related research works on formal method and dependable distributed protocol analysis in following section.

1.5.1 Related Researches on Formal Method and Dependable Distributed Protocol Analysis

Mocha: Modularity in Model Checking [2] is a joint project of the U.C. Berkeley etc. It describes a new interactive verification environment for the modular and hierarchical verification of heterogeneous systems. Mocha supports the heterogeneous modeling framework of reactive modules. Instead of employing system requirements written in traditional temporal logic, Mocha checks module requirements written in Alternating Temporal Logic (ATL), which is designed for specifying collaborative as well as adversarial interactions between the components of a system. And to support hierarchical design and verification, Mocha combines model checking with automatic and semiautomatic refinement checking based on assume-guarantee reasoning.

Approach for specification composition based on MOKA: MOKA [28] provides a methodology to capture and formalize engineering knowledge and reuse it. For Knowledge-Based Engineering (KBE) systems, used in the aeronautical and automotive industries, long term risk can be reduced by employing a systematic methodology that covers the development and maintenance of such systems.

Closure and Convergence: [3] gives a formal definition of what it means for a system to " tolerate" a class of "faults". The definition consists of two conditions: One, if a fault occurs when the system state is within a set of "legal" states, the resulting state is within some larger set and, if faults continue occurring, the system state remains within that larger set (Closure). And two, if faults stop occurring, the system eventually reaches a state within the legal set (Convergence). The authors demonstrate the applicability of definition for specifying and verifying the fault-tolerance properties of a variety of digital and computer systems. Further, using the definition, the authors obtain a simple classification of fault-tolerant systems and discuss methods for their systematic design.

Building blocks Approach Based on Automata: The approach provided in [9] is both mathematically rigorous and closely tied to real systems. They have developed a significant general theory, based on I/O automata, that enables rigorous reasoning about distributed algorithms and systems. The usefulness of the framework has been demonstrated by many case studies working with system developers, They give a clean and simple specification of a view-synchronous group communication service, similar to the virtually synchronous group communication services of Isis, Transis, Totem and Ensemble.

Approach based on event-driven software framework: [15] provide an approach to modularizing fault-tolerant protocols such as reliable multicast and membership. The approach is based on implementing a protocol's individual properties

as separate micro-protocols, and then combining selected micro-protocols using an event-driven software framework; a system is constructed by composing these frameworks with traditional network protocols using standard hierarchical techniques. In addition to simplifying the software, this model helps clarify the dependencies among properties of fault-tolerant protocols, and makes it possible to construct systems that are customized to the specifics of the application or underlying architecture.

Other approaches [14] introduces an approach based on assume-guarantee reasoning. Two methodologies are introduced in their project: (1) decomposing proof obligations in system verification, (2) refinement mappings (homomorphisms) for solving the language-inclusion problem in practice.

[16] presents a solution to solve the problem of building systems from components for the ensemble communication architecture by showing the configurations being checked against specifications. Optimized code can be synthesized from these configurations. The performance results show that the approach reduces end-to-end latency in the already optimized ensemble system.

1.5.2 Discussion

Over our recent studies in applying formal methods for specification and verification, we have observed that formal analysis typically requires intensive effort for both specifying and verifying a specific protocol. Most of the formal specifications and proof constructs cannot easily be reused directly to verify other protocols which employ similar basic concepts. These facts limit wide acceptance of formal techniques in the design and development of dependable distributed systems.

Also we observed that most protocols that provide for dependable services required people to model just a few basic functional primitives. From the formal viewpoint, we are investigating the reusability of these functional primitives and sub-protocols. Aiming to identify these functional blocks for their subsequent reuse

in formulating generic protocols, we are trying to construct formal library routines for these identified building blocks, and further more, specify and verify it.

An important distinction of our approach from other approaches to modularization is that we iteratively add implementation or parametric details to an abstract specification, and establish the proof of correctness. Our modular composition of dependable distributed protocol is specifically based on formal specification of building blocks (or functional primitives) and with a consideration to guide and supplement specification and verification process . The main benefit of our approach is that the proof of correctness of the composite protocol specifications can be readily established utilizing the proof-constructs being developed for these basic building blocks. Instead of defining formal theories from scratch each time a new protocol is tackled, we could utilize reusable and parameterized theories.

1.6 Contribution

The main contribution of this thesis is in the area of formal specification and verification of dependable distributed protocols. This thesis realizes the methodology for specifying and verifying complex fault tolerant protocols in dependable distributed system. The thesis extends our modular and compositional approaches for formal analysis of dependable distributed protocols [22] [23].

Specifically, this thesis makes a contribution to the area of formal specification and verification of dependable distributed protocols by:

- Presenting the category-theory based framework for specification composition and verification of Checkpointing Protocol in Dependable Distributed System.
- Illustrating how category-based formalization of building-block protocols permit re-usability of the basic formal modules.

- Showing how to simplify specification and verification for a spectrum of protocols by utilizing modular approach.

1.7 Organization of the Thesis

The organization of the thesis is as follows. Chapter 2 introduce the formal analysis of dependable distributed protocols, where we describe the selection of sub-protocols and our approaches for formal analysis. We present the compositional specification of checkpointing protocol based on category theory by using Specware in Chapter 3, and following that in Chapter 4 we present the formal specification and verification of checkpointing protocol using PVS. Chapter 5 concludes with discussions and outlining future research directions.

Chapter 2

Formal Analysis of Dependable Protocols

In this chapter, we identify the basic building blocks of redundancy management protocols, and outline a formal framework of these primitives and sub-protocols which are constituent to the development of higher level protocols. We start with describing the basic primitives. Next, based on them we present the sub-protocols for combining a group communication protocol which supporting the hierarchical protocol development. We introduce a checkpointing protocol based on all these primitives and sub-protocols. After the overview of all these modules, we highlight two approaches for the composition of checkpointing protocol using these building-block protocols, namely a category-theoretic-based and assertions-based approaches.

2.1 Overview of Redundancy Management Protocols

As we mentioned in Chapter 1, a number of protocols providing distributed and dependable services can often be formulated using very few basic functional primitives or their variations. The typical building blocks for constructing fault-tolerant protocols include System Model, Failure Model, Communication Primitives, and voting/convergence functions. Also the Clock Synchronization, Atomic Broadcast and Group Membership are essential fault tolerant services for developing most group communication protocols.

Figure 2.1 depicts our general approach [22] for modular composition and verification of dependable distributed protocols utilizing these basic primitives. We note that the building blocks mentioned on the top most level in Figure 2.1 could be substituted by other building -block protocols for a more complex protocol composition.

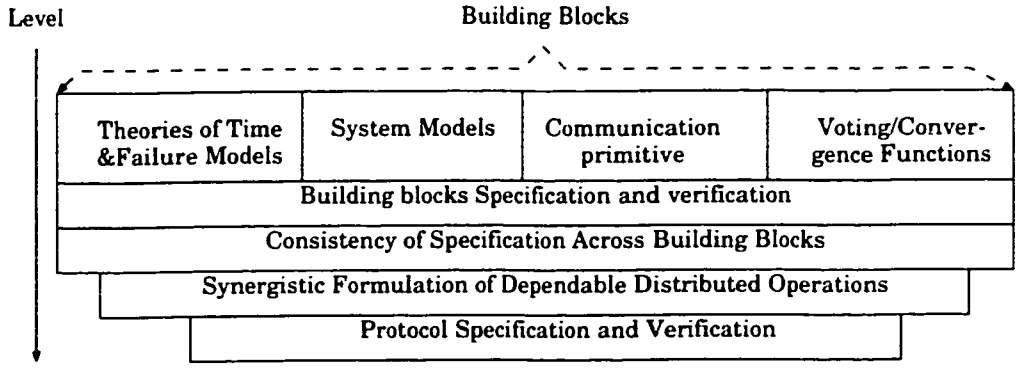

Figure 2.1: Building blocks Model [22]

The formalization of these different building blocks in formal languages can make it easier for us to formalize the higher level protocols in the particular semantic framework. It also would be beneficial to develop a methodology which would allow the formal constructs of these building blocks to be utilized efficiently, and guide

the process of establishing the proof of correctness of the generated protocol.

Our objectives are to develop a modular approach for the specification and verification of dependable protocols by identifying building blocks within the class of redundancy management protocols, highlighting and addressing the block interactions and inter-dependencies, developing guidelines for constructing libraries of formal specifications and their associated verifications of these categorized building blocks protocols that can be utilized in establishing the overall correctness of the composite protocol.

In the thesis, we give a case study of a Checkpointing Protocol to present our motivation and outlook towards resuability of formal theories. The checkpointing protocol employs operations for fault isolation and resource reconfiguration, it can be composed utilizing building blocks of synchronization, atomic broadcast and Membership.

2.2 Basic Functional Primitives

According to the structure in Figure 2.1, we give a brief introduction for each basic functional primitives in following subsections.

2.2.1 Time

In a standard distributed architecture, each processor has its own clock, but there is no global physical clock that can be accessed by all processors. We assumed that processor clocks are reliable and monotonic, running within a linear envelope of real-time, and at any real-time, the values are always within a known constant maximum deviation [26]. Synchronized clocks allow us to reason in terms of global system clock time (Logical Time) rather than real time(Physical Time).

A global time base is the most important to determining the causal relation among various events on different processors. This causality relation holds between

two events a and b on the same or different processors if the execution of a could possibly have affected execution of b. This property is useful for ensuring that messages multi-cast among a group of processes are received in the same order by all processes and in an order that reflects causality.

The specifications of the time model are possible:

- *A bounded time applies to dense (discrete) model when a finite subset of real numbers (natural numbers) is chosen to model time;*
- *An interval time model has a discrete time structure, so that the next and previous intervals can be referred to. Within each interval, a property may hold in every subinterval or only at specific discrete points. In addition, time modeled as intervals may be bounded or unbounded.*

The notion of time required in the the reliable dependable distributed protocols in our model. The time base can be absolute, corresponding to a physical clock, or relative, based on specific events [26].

2.2.2 Failure Models

When a specification of a component's acceptable behavior is available, it provides a standard against which the behavior of that component can be judged. The specification may prescribe both the component's response for any initial state and input sequence, and the real-time interval within which the response should occur.

A component is correct if, in response to inputs, it behaves in a manner consistent with the specification; if it behaves otherwise, it has failed. A failure model is a way for precisely specifying assumptions about how a component behaves when it fails. A number of such failure models have been defined; although we state these in terms of generic components, they are most often applied to processors. In the fail-stop failure model, it is assumed that the component fails by ceasing execution

without undergoing any incorrect state transition and that this failure is detectable by other components. In a crash model, a component is assumed to fail in the same way, but without the guarantee of detect ability. This model has also been termed fail-silent [4].

A process is faulty in an execution if its behavior deviates from that prescribed by the algorithm it is running, otherwise it is correct. A model of failure specifies in what way a faulty process can deviate from its algorithm. The following is a list of models of failures that have been studied in the thesis.

- *crash*: a faulty process stops prematurely and does nothing from that point on. Before stopping, however, it behaves correctly.
- *send omission*: A faulty process stops prematurely, or intermittently omits to send messages it was supposed to send, or both.
- *Receive omission*: A faulty process stops prematurely, or intermittently omits to receive messages sent to it, or both.

Particularly, timing failures is a model that is only pertinent to synchronous systems. A process subject to timing failure can fail in one or more of the following ways:

- *it commits general omission failures,*
- *its local clock drift exceeds the specified bound (a clock failure)*
- *it violates the bounds on the time required to execute a step (a performance failure)*

2.2.3 Communications

We assume the datagram service: a node p diffuses a message to another node q by sending diffusion packets in parallel on all physically independent routes between p

and q. We assume that there is some arbitrary, but fixed bound F on the number of communication components (nodes and links) that can be faulty during a diffusion, and that the network possesses enough redundant independent physical links between any two pairs of nodes (p, q), so that a node q always receives at least one copy of each packet diffused by another node p despite up to F faulty communication components.

A synchronous communication network also enables the implementation of a synchronous atomic broadcast communication service. Synchronous atomic broadcast protocols ensure, for some time constant D that depends on N, the following properties: (a) if a node attempts to broadcast a message m at time T, then at time $T + D$ either all correct nodes deliver m or none of them delivers m (atomicity). (b) all messages delivered are delivered in the same order at each correct node (order). (c) if the sending node of m is correct, then all correct nodes deliver m at $T + D$(termination). Since in the thesis atomic broadcast is the only primitive used for broadcasting information, when we say "broadcast", we mean "atomic broadcast".

Also, we give out the communication failure as below:

- *Crash*:A faulty link stops transporting message. Before stopping, however, it behaves correctly.
- *Omission*: A faulty link intermittently omits to transport messages sent through it.
- *Timing failures*: A faulty link transports messages faster or slower than its specification.

2.3 Building-Block Protocols

2.3.1 Clock Synchronization

Synchrony refers to assumptions that are made about the execution bounds on components. A hardware or software component is synchronous if it always performs its intended function within a finite and known time bound, and asynchronous otherwise. This bound on the execution time of the synchronous component must hold whenever the component is correctly operating, and in particular, under all operational conditions within its specification. Synchrony can be defined for communication channels, communication networks, processors, and protocols. For example, in a synchronous communication channel, the transmission delay of a unit of data across the link is known and bounded. Similarly, a synchronous processor is a processor in which the time to execute a unit of work is known and bounded. The success of a real-time system in meeting its service requirements depends upon the correctness and timeliness of its responses and its resilience to faults [26].

Synchrony is an attribute of both processes and communication. We say that a system is synchronous if it satisfies the following properties:

There is a known upper bound δ on message delay; This consists of the time it takes for sending transporting, and receiving a message over a link. Every process p has a local clock Cp with known bounded rate of drift $\rho \geq 0$ with respect to real-time. That is , for all ρ and all t$\geq$t',

$$(1+\rho)^{-1} \leq \frac{C_p(t)-C_p(t')}{(t-t')} \leq (1+\rho)$$

Where $C_p(t)$ is reading of C_p at real-time t.

There are known upper bounds on the time required by a process to execute a step.

In synchronous systems it is possible to measure message timeouts, and this

provides a mechanism for failure detection to implement approximately synchronized clocks, i.e. clocks that, in addition to the bounded rate of drift property. also satisfy the following condition: There is such that for all t, and any two processes p and q,

$$|C_p(t)-C_q(t)|\leq\rho$$

In fact, such clocks can be implemented even in the presence of failures . Approximately synchronized clocks have many applications, for instance, real-time process control, file management, cache consistency, authentication, etc.

2.3.2 Atomic Broadcasting Protocol

An important objective in distributed system is to provide consistent information to multiple processors in the system. Reliable broadcast is a broadcast that satisfies the following three properties:

- *Validity*:If a correct process broadcast a message m , then all correct processes eventually deliver m.
- *Agreement*: if a correct process delivers a message m, then all correct processes eventually deliver m.
- *Integrity*: For any message, every correct process delivers m at most once, and only if m was previously broadcast by sender(m).

Formally, reliable broadcast is defined in terms of two primitives: broadcast(m) and deliver(m), where m is a message from a set M of possible messages. When a process invokes broadcast(m), we say that it broadcasts m. Similarly, when a process executes deliver(m), we say that it delivers m.

Since every process can broadcast several messages, it is important to be able to determine the identity of a message's sender, and to distinguish the different

messages broadcasted by a particular sender. Thus, we assume that every message m includes the following fields: The identity of its sender, denoted sender(m) and a sequence number, denoted seq#(m). If sender(m)=p and seq#(m)=i, then m is the ith message broadcast by p. These fields make every message unique.

It is important to realize that if a process p fails during the broadcast of a message. Reliable Broadcast allows two possible outcomes; either the message is delivered by all correct processes or by none. For example, if p invokes broadcast(m) and then immediately crashes, correct processes will never be executed according to p's intention to broadcast that message, and thus cannot deliver anything. On the other hand, if p fails during the broadcast, but after having sent enough information, then correct processes may be able to deliver m.

Atomic broadcast requires that all correct processes deliver all messages in the same order. This total order on message delivery ensures that all correct processes have the same "view" of the system, hence they can act consistently without any additional communication. Formally, an atomic broadcast is a reliable broadcast that satisfies the following requirement:

- *Total order*: if correct processes p and q both deliver messages m and m', then p delivers m before m' if q delivers m before m'.

The agreement and total order requirements of atomic broadcast imply that correct processes eventually deliver the same sequence of messages.

A synchronous communication network also enables the implementation of a synchronous atomic broadcast communication service. Synchronous atomic broadcast protocols ensure, for some time constant D that depends on the following properties. If a node attempts to broadcast a message m at time T, then at time $T + D$ either all correct nodes deliver m or none of them delivers m (atomicity). All messages delivered are delivered in the same order at each correct node (order). If the sending node of m is correct, then all correct nodes deliver m at $T + D$ (termination).

2.3.3 Membership Protocol

The membership problem is a fundamental problem of distributed computing, like routing, clock synchronization, atomic broadcast, or atomic commit, in the sense that once solved, it allows easy solutions to other important problems encountered when designing fault-tolerant distributed applications.

In this thesis, the membership protocol which we employed shows how a processor membership service helps solve the server-group membership problem, the problem of ensuring high availability of computing services in a distributed system, and the problem of precisely defining the scope of server-group communication.

The protocol considers a system consisting of distributed server processes running on processors linked by a physical network. Each processor consists of hardware and software. The operating system supports process execution. The communication subsystem accepts messages from, and delivers messages to, processes, manages message queues, and drives the physical network links. Collectively, the communication subsystems provide the distributed processes that run at various processors the abstraction of a communication network. The communication subsystems are the nodes of this network. Atomic broadcast protocols for point-to-point and broadcast networks.

When a local client process c declares an interest in knowing the processor membership, p gives c the membership of the current processor group and then notifies c of any subsequent membership changes in a timely manner. Such notifications are sent to c until either c declares that it is no longer interested in receiving them or the failure of c is reported to p by the underlying operating system.

2.4 Checkpointing Protocol : An Overview

Checkpointing is a well know technique for providing fault-tolerance in distributed systems. It allows long-running distributed system to save state at regular interval

so that they may be restarted after interruptions avoiding total loss of work.

- *A checkpoint* is the saved state of a single process stored in a form such that the process can restart its execution from the point in time when the checkpoint was created.
- *Checkpointing* is the process of saving process states into checkpoints.
- *Rollback :* The restoring of the state of one or more processes to their state previously stored in a checkpoint is called rollback.
- *Recovery :* Systems which incorporate checkpoints and rollback are called checkpoint and rollback recovery systems since they recover to a previous state of a system.

In distributed checkpointing different processes coordinate the checkpoint of their local states. A distributed checkpointing protocol ensures that the checkpoints taken by the different processes form a consistent system state. In this section, we will see some approaches to distributed checkpointing.

We now describe a synchronized checkpointing and recovery technique that takes a consistent set of checkpoints and avoids live lock problems during recovery. The algorithm's approach is said to be synchronous, as the processes involved coordinate their local checkpointing actions such as the set of all recent checkpoints in the system is guaranteed to be consistent.

The checkpoint algorithm assumes the following characteristics for the distributed system: (*a*) Processes communicate by exchanging messages through communication channels. (*b*) Channels are FIFO in nature. (*c*) End-to-end protocols are assumed to cope with message loss due to rollback recovery and communication failure. (*d*) Communication failures do not partition the network.

2.4.1 Problems Related to Checkpointing

To specify and verify a checkpointing protocol is a complicated problem because the inherent characteristics of distributed systems make any operation which requires global state information difficult. Checkpointing are no exception to this rule. There are many problems related to the specification and verification of checkpointing protocol. We list them below:

Global Time Synchronization One problem is the lack of global time. Distributed processes suffer from the problem of having no global time scale. Events occurring in a traditional sequential program will always be totally ordered by physical time. Unfortunately this is not the case in a distributed system. The protocol assume a approximately synchronized global system for the checkpointing. Therefore in the verification in this thesis, the global time synchronization is described in the assumption of SYNC.

Communication Delay The communication delay is another problem to be concerned by the protocol. In multiple process applications the fact that there is no global time scale, implies that it is impossible for system wide states to be saved instantaneously. Therefore, in order to save states which are consistent across several processes, some communication protocol must be employed. And in distributed systems, there is an inherent delay between the time that a message is sent and the time at which it is received at some remote process.

Checkpoint Period How often checkpoints should be created really depends on the application. Two things to consider when deciding on the checkpoint frequency are the need to minimize the amount of computation to be rolled back, and the overhead of the actual checkpointing operation. If checkpoints are taken often, the system performance will degrade but recovery time will be decreased. On the other hand, if checkpoints are taken less often, system performance will increase

but a penalty will be incurred at the time of rollback and recovery. The design of checkpoint algorithms must weigh the advantages and disadvantages of checkpoint frequency on the basis of the likelihood of rollback occurrence. In this thesis, we assume that the checkpoint interval is much greater that the communication delay upper bound plus the Clock skew [7].

Orphan Message and Domino Effect The communication delay inherent in distributed system introduces another obstacle to rollback and recovery. When a process has rolled back its state, some messages which it has sent prior to its rollback may still be in transit (i.e. the messages may not have been received at their destination yet). These messages would no longer be valid since they were sent before the sender had changed its state.

Following example illustrates a scenario where a message could become an orphan message. Consider the system activity in Fig 2.2. P, Q and R are three processes that cooperate by exchanging information (shown by the arrows). Each symbol "[" marks a recovery point to which a process can be rolled back in the event of a failure.

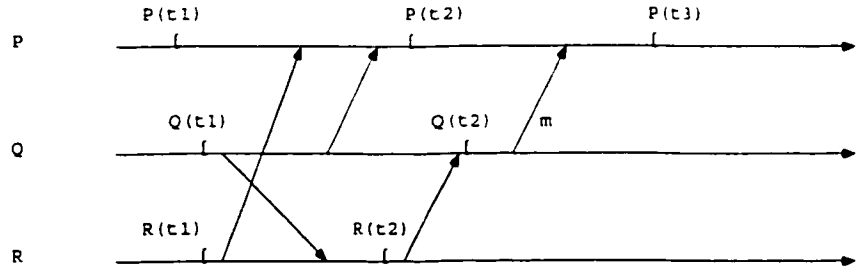

Figure 2.2: Orphan Message

If process P is to be rolled back, it can be rolled back to the recovery point $p(t3)$ without affecting any other process. Suppose that R fails after sending message m and is rolled back to $R(t2)$, In this case the receipt of $m$is recorded in $P(t3)$ but the sending of m is not recorded in $R(t2)$. Now we have a situation where P has

received m from Q, but Q has no record of sending it. which corresponds to an inconsistent state. Under such circumstances, m is referred to as an orphan message and process P must also roll back. P must also rollback because Q interacted with P after establishing its recovery point $R(t2)$. When Q is rolled back to $R(t2)$, the event that is responsible for the interaction is undone. Therefore, all the effects at P caused by the interaction must also be undone. This can be achieved by rolling back P to recovery point $P(t2)$. Likewise, it can be seen that, if R is rolled back, all three processes must roll back to their very first recovery points, namely $P(t1), Q(t1)$ and $R(t1)$. This effect, where rolling back one process cause one or more other processes to roll back, is known as the domino effect and orphan messages are the cause.

In distributed systems, several processes cooperate by exchanging information to accomplish a task. They are constantly communicating with each other through messages. Because of this, if one of the cooperating processes fails and resumes execution from a recovery point, then the effects it has caused at other processes due to the information it has exchanged with them after establishing the recovery point will have to be undone. To undo the effects caused by a failed process at an active process, the active process must also rollback to an earlier state. Thus, in concurrent systems, all cooperating processes need to establish recovery points.

An error which occurs in one process may contaminate the state of another process through the sending of incorrect information in a message. In order to avoid the domino effect, checkpoint creation must be coordinated such that the checkpoints from consistent recovery lines. Planned checkpoint could guarantee a domino free situation.

Lost Message Suppose that checkpoints $P(t1)$ and $Q(t1)$ are chosen as the recovery points for processes P and Q, respectively. In this case, the event that sent message m is recorded in $P(t1)$, while the event of its receipt at Q is not recorded in $Q(t1)$. If Q fails after receiving message m, the system is restored to state {P(t1),

$Q(t1)\}$, in which message m is lost as process P is past the point where it sends message m. This condition can also arise if m is lost in the communication channel and processes P and Q are in state $P(t1)$ and $Q(t1)$, respectively. Both the above conditions are indistinguishable.

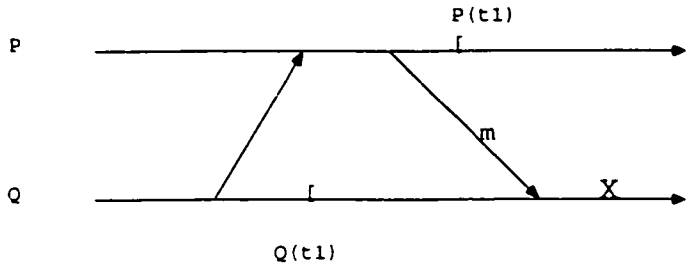

Figure 2.3: Lost Message

2.5 A Time-stamped Checkpointing Protocol

Now we consider one particular approach to distributed checkpointing and the corresponding method for recovering state. This approach has been proposed by Christain and Jahanian [7].

The protocol employs synchronized clocks. It is based on periodic checkpointing of local process states and logging of incoming messages during a short bounded interval. The processor clocks are synchronized within a known maximum deviation δ. i.e.at any global time, the lock reading of any two different processes in the system is within δ of each other.

The basic approach is that processes checkpoint periodically, each with the same period π. It is assumed that π is much larger than δ+BroadcastBound. If the clocks were perfectly synchronized, then the set of checkpoints taken at some time kπ may suffice. Since clocks do not work in perfect synchrony, something more needs to be done to avoid orphan and lost messages. Orphan messages are avoided in this approach by making the checkpointing time flexible such that the set of

kth checkpoints of the processes does not contain any message that is sent by a process after establishing its *kth* checkpoint. This approach also explicitly handles lost messages rather than leaving it to the communication protocol to handle by using message logging. Times are bounded by using the clock and delay bounds.

2.5.1 Time-stamp

The protocol is based on periodic checkpointing of local process states and logging of incoming messages during a short bounded interval in such a way that time-stamps increase within each process and the sending and the delivery events associated with each message have the same time-stamp. So, there is a logical time frame in which for each message, the send event and the corresponding delivery events occur simultaneously.

Whenever a message is sent, the event of sending a message occurs before the event of receiving the message. We define the communication relation, denoted by send and receive as follows: for any message m, send(m) $\rightarrow$ receive(m) (where send(m) is the event of sending the message and receive(m) the receiving event)

Each process P_i maintains a logical clock at physical time T , C(P_i,T). Each site time-stamps events with the value of its logical clock. For each send and state transforming operation: time-stamp equal with the current value of the local logical clock. Let m be the event of sending a message: then the message piggyback the Time-stamp. On receiving m in the destination, a process P_j computes max (C(P_j), Time-stamp) to identify if the message m is over due.

2.5.2 System Model

The distributed system consists of a collection of processes P_i, i=1, 2, 3, ... belongs to the set of processors, MemAll. Each process executes on a single processor. A process can checkpoint its own local state, and can log the messages it sends

or receives. Information exchange only occurs through message exchange. Each process P_i has a state S_i(i.e., the content of all application variables). The state changes according to the operations the process performs. Each process executes a sequence of events/actions, each of which is either a send, receive or an operation that transforms its state.

Assume all processors in MemAll have hardware clocks that run within a linear envelope of real-time and message delays between processors are random and unbounded. The delay between the moment a message is sent by a source process and the moment the message is received by its target is presented by *BroadcastDelay*. We assume there exists a time-out delay *BroadcastBound* such that interprocess communication delays are smaller than BroadcastBound with high probability. Since messages delays are unbounded, a small percentage of messages may need more than the time-out delay to travel between processes. If this happened, they will be handled by the communication protocols.

A state of P is called consistent if it could have been seen at some instant by an outside observer during the execution of P from its initial state, regardless of the relative speed of the processes. A notion of a consistent P state is important since after recovery from a failure, the set of processes in P must be restored to a consistent state.

The approaches can be summarized with the following rules to establish the kth global checkpoint:

Suppose P is a distributed process consisting of n process $P_1,...,P_n$.

1 The initial global checkpointing of P , denoted by S(0) is defined as $< S_1(0), S_2(0), ...,S_n(0) >$ where $S_i(0)$ is the initial local state of process P_i.

2 The *kth* global checkpoint of P, denoted by $S(k)$, for $k>0$, is defined as

a.$< S_1(k), S_2(k),...,S_n(k) >$ where $S_i(k)$ is the local state of process P_i at time $T_i^k=\min(T, R_i^k)$, $T=k\pi$, and $R_i^k$is the time P_i receives the first message sent by some other process P_j after it has locally checkpointed its state $S_j(k)$, and

b. $<m_{i,j}(k)>$, $1 \leq i,j \leq n$, where $m_{i,j}(k)$ is the fifo ordered set of messages with time-stamps in $[T-\delta-\varepsilon, T_i^k]$ sent by P_i to P_j and not consumed by P_j until after it checkpoints its local state $S_j(k)$.

2.5.3 Bounds on the Checkpointing Time Interval

For the system synchronization, the protocol assumes that the non-faulty clocks are initially synchronized to some constant quantity, and should not drift by a rate greater than the specified drift-rate. The services are essentially required from clock synchronization functional block.

Throughout this thesis, our formalization of different protocols refers to the global time interval as depicted in figure 2.4:

As discussed in [7], the scheduled checkpoint is on processor's local clock S. Due to clock skew, different processor may has it's local checkpoint within the peroid of $(S(1-\rho), S(1+\rho))$. The overall checkpointing peroid, as defined in the protocol, will be $(S(1-\rho)-BroadcastBound-\varepsilon, S(1+\rho)+BroadcastBound+\varepsilon)$.

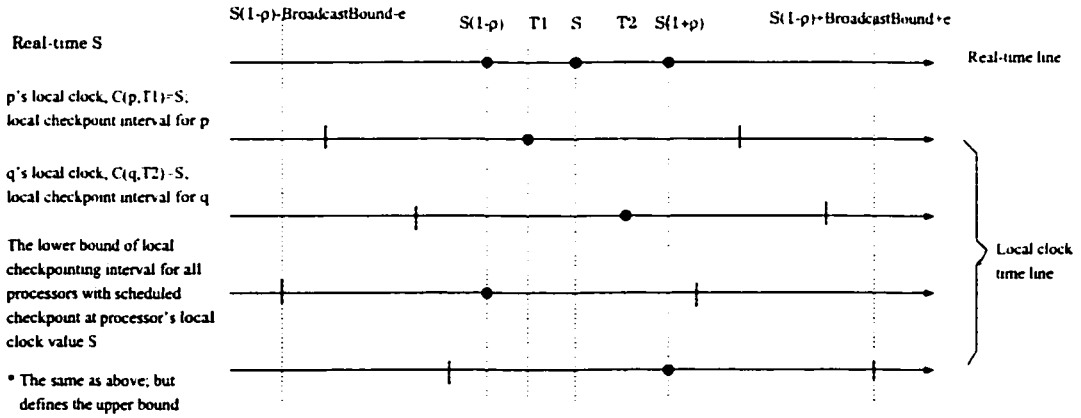

Figure 2.4: Bounds on the Checkpointing Time Interval

2.5.4 Dealing With Orphan Messages

There exist two methods to prevent the occurrence of orphan messages. one method is to prevent a process from sending any messages during the time ε after establishing a checkpoint. Time-stamped checkpointing approach employ the alternate method which is to establish the checkpoint earlier, i.e., when a processor p receives a message m before its scheduled current round of checkpoint, and if the message m was sent by the sender after the sender has established its checkpoint of the current round, then the processor p will make checkpoint right at the time of the message arrival.

2.5.5 Dealing With Lost Messages

To avoid missing messages, message logging can be used. One way to do this is to have the senders log the messages they send during the interval$[T-\delta-\varepsilon,T]$ on stable storage. If a rollback is performed, then these messages are retransmitted after rollback,. The message can also be logged at the receiver. A receiver logs any messages it receives in the interval$[T,T+\delta+\varepsilon]$, these messages are added to the queue, if rollback is done.

In the absence of failures, a message sent during the interval $[T-\delta-\varepsilon]$ will be delivered to destination no later than $T+\varepsilon+\delta$.

We define the *kth* local checkpoint P process P_i to be successful if the local state $S_i(k)$ is saved on stable storage and the message in each set $m_{j,i}(k)$ for $1 \leq j \leq n$ are logged by $T+\delta+\varepsilon$, where $T=k\pi$. The *kth* global checkpoint of P is successful if all the local checkpoints are successful. It can be shown that if all processes successfully take their kth local checkpoint, then no domino effect occurs [7].

With synchronized clocks and bounds on message will be received by the receiver process at the latest by its local clock time $T+\delta+\varepsilon$. It shows that if a message is sent by a process in the time period (according to local clock) $[T-\delta-\varepsilon,T]$, then it will be delivered to the destination by the time $T+\delta+\varepsilon$ (according to local clock).

2.6 Modular Composition of Checkpointing Protocol

After having identified the protocol building blocks and having given an introduction to the checkpointing protocol being considered, we now illustrate how the building-block protocols help to compose the Checkpointing Protocol.

We now relate these functions to protocol-building blocks which are constituent to the hierarchical composition of the checkpointing protocol.

2.6.1 Choosing Building-block Protocols

Synchronized checkpointing requires a clocks synchronization protocol. It also essentially requires broadcast and membership which build up the group communication services together with Synchronization.

- *Synchronization* is used to synchronize various activities of all processors, and implement atomic broadcast primitive.
- *Atomic broadcast* is used to achieve the atomicity, order and bounded communication of message exchanges.
- *Membership* is essentially needed to identify members in a group to form a consistent global state.

These building blocks inherently assume the presence of synchronous communication network primitives for bounded communication between two processes.

2.6.2 Outlining the Block Interactions

We have identified the building blocks and pointed out the role of each building block in achieving the overall objectives of the checkpointing operation. Now we present

the hierarchical composition of the checkpointing protocol using the building blocks of group communication services

In order to establish the consistency of specifications across the various constituent building blocks, we identify the inter-dependencies [22] of these basic building blocks as shown in Figure 2.4.

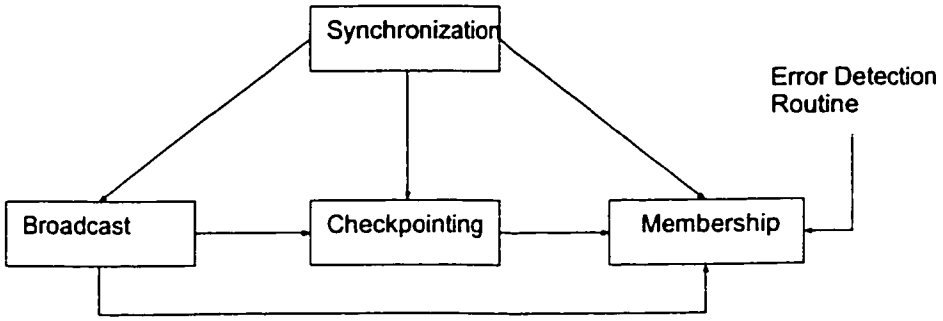

Figure 2.5: Inter-dependencies of Building Blocks [23]

After having outlined the interactions of the building-block protocols in Figure 2.5, next we identify the exact conditions that a particular block imposes on another blocks. This helps ascertain the exact requirements that need to be observed across the blocks. As the modularization approach help us identify the specific attributes of each block, we summarize the conditions and requirements for each block interactions in Figure 2.6.

In the figure, the interactions between building blocks are depicted: such as the path Synchronization→Broadcast→Membership→Checkpointing depicts a form of interactions among these four building blocks. We illustrate how modular composition assist in addressing these dependencies and identifying any governing conditions. For example, the synchronous membership protocol has the properties of Agreement and Recognition: Each processor has an up-to-date state information of all group members. In case of any failure is detected in a process by a processor P, as a result of the operations of the protocol, P must timely broadcast this event to all other processes to stop all execution and initiate a roll back to a validated

Block Interactions	Primary Attributes
Sync. -> Broadcast Sync. -> Membership Sync. -> Checkpoint	Synchronized clocks Bounded drift rate Bounded clock skew
Broadcast -> Membership Broadcast -> Checkpoint	Synchronous system Bounded delays Synchronized clocks
Membership -> Checkpoint	Unpartitioned network synchronized processors bounded communication atomicity and ordering

Figure 2.6: Inter-Block Requirement [23]

checkpoint. These facts can be asserted by the atomicity and termination properties of atomic broadcast. The influence of atomic broadcast block on the membership block is that the underlying system model must be synchronized. This constraint is satisfied by the properties of synchronization block.

2.6.3 Interpreting The Building Block Interaction

To construct modular composition of a protocol, we need not only to identify and separate the required functionality into modules, but also to define the interfaces across the modules. The capabilities to identify underlying dependencies between building blocks is very important to the effectiveness of the process of protocol decompositions. We note that the interfaces of each building block should be well-defined even if defining dependencies between functions can complicate the process of defining the modules.

In our model of this thesis, we consider the properties of atomic broadcast and clock synchronization that can influence the working of checkpointing operation. Because of the clocks of all processors are tightly synchronized within a maximum

skew and there is no lower bound imposed on broadcast delay, it is possible that a set of checkpoints taken by a processor may contain an orphan message in a checkpoint. The checkpointing operation ensures that a process does not consume any message delivered during the clock skew before establishing its checkpoint. To achieve this, all messages sent by a process between its Kth checkpoint and $K+1th$ checkpoint are tagged with k to indicate the interval in which they are sent. Only messages with tag $K-1$ are include in the Kth checkpoint. From this example we can conclude that clock skew in clock synchronization protocol is also one important parameter in checkpointing protocol. The correctness of the Checkpointing Protocol is guaranteed by existence of a common time base by synchronizing the clocks of all processors in the system, and also guaranteed by the satisfaction of the properties of the atomic broadcast. These dependencies of building blocks can be highlighted by modularizing and defining morphisms according to these specific requirements.

2.7 Specification and Verification of Checkpointing Protocol

The Checkpointing Protocol is implemented with the hierarchy of building-block primitives / protocols. By composing the checkpointing protocol via such a protocol stack, it is important to establish its correctness. On the other hand, the combination of distribution, local clocks, real-time, and fault-tolerance makes it difficult to guarantee that the protocol indeed lead to the required service. Following we introduce two approaches for the specification and verification of the Checkpointing Protocol.

2.7.1 Specification and Verification with Specware

We first introduce the approach to specify and verify the Checkpointing Protocol based on a category of axiomatic specifications and specification morphisms. The composition of these specifications is based on the colimit calculation of Category Theory.

We use Specware to support the systematic construction. In Specware, the specification structure is expressed via specification diagrams, directed multi-graphs whose nodes are labeled with specifications and arcs with specification morphisms. Specification diagrams are useful both for composing specification from pieces and for designing structure of a system.

In Specware the design process proceeds by stepwise refinement of an initial specification into executable code. The unit of refinement is an interpretation, a theorem-preserving translation of the vocabulary of a source specification into the terms of a target specification. Each interpretation reduces the problem of finding a realization for the source specification to finding a realization for the target specification. The overall result of the design process is to refine an initial specification into a program module.

Using Specware on our protocol stack, we manage the steps for composition by a idea of "putting specifications together". That is, the primitives and sub-protocols which are the primary component of our model are defined as the category of specifications and specification morphisms. The diagrams which are built in this category describe the system structure of our protocol stack. Specifications can be put together via colimits to obtain the generated specification of the operations and services of our Checkpointing Protocol. SNARK, the external prover of Specware, will finally complete the theorem proving and the protocol verification . We describe these steps in Chapter 3.

2.7.2 Specification and Verification with PVS

The second approach that we introduce is an assertional method to specify and verify the modular composition of Checkpointing Protocol. Assertional method means using the logical formulas (assertions) characterize the required service, the protocols operations, and assumptions about the primitives and underlying group communication protocols.

To obtain mechanical support, we use the interactive proof checker PVS (Prototype Verification System). The PVS specification language is a higher-order typed logic and specifications can be structured into a hierarchy of (parameterized) theories. The tool contains a proof checker to construct proofs interactively and to rerun proofs automatically after small changes.

PVS is a general tool with a rich specification language that can be used for a large number of applications. To apply PVS to our system model, we have a clear sequence of steps to be performed. Also we define a number of reusable theories and illustrative paradigms. We formulate the clock synchronization in terms of time primitive and the relation between local clocks and real-time. Further, we formulate the failures of processors to define failure primitives. Based on these primitives we setup all the sub-protocols. We discuss the verification of the Checkpointing Protocol using PVS in Chapter 4.

Chapter 3

Specification Composition Based On Category Theory

Now we start introducing specification composition of Checkpointing Protocol. This chapter is organized as follows. Section 1 describes the basic concepts of Category Theory. Section 2 introduces the modular specification framework. In Section 3 we explain procedures for combining specifications of the identified basic primitives. We describe the Category Theory based formalization of the Checkpointing Protocol in Section 4. In Section 5 we conclude a discussion.

3.1 Category Theory Based Composition

In [22], [23], a formal framework is introduced which utilizes concept of category theory to facilitate a rigorous and consistent composition out of system building block protocols. Here, we first provide a brief overview of the proposed category theory based modular composition framework prior to examining the overall approach.

We have adapted calculus of modules based on categorical concepts [22], [23] for simplifying formal specifications. We first define a few general terms.

- *Signature*: A signature $SIG = (S, OP)$ consists of a set S, the set of sort, and

a set OP,the set of constant and operation symbols.

- *Specification*:A specification $SPEC = (SIG, AX)$ consistent of two parts: the signature SIG and a set of axioms AX which describes the behavior of the system as well as constraints on the environment.

- *Specification Morphism*: A specification morphism m: $SPEC1 \rightarrow SPEC2$ is a map from the sorts and operations of one specification to the sorts and operations of another such as that (a) axioms are translated to theorems, and (b) source operations are translated compatibly to target operations.

In order to define module specifications, we utilize the notion of push-out operation from category theory. Given specifications A and B, and a specification R describing syntactic and semantic requirements along with two morphisms f and g, the push out operation gives specification P which contains A and B.

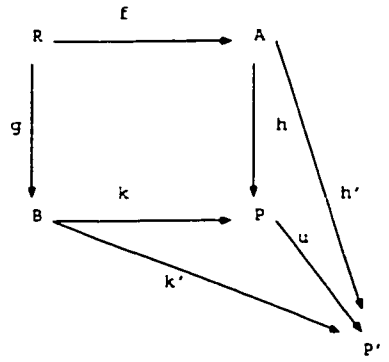

Figure 3.1: Push-out and Module Interfaces

Formally, given specification morphism f: $R \rightarrow A$ and g:$R \rightarrow B$, a specification P together with specification morphisms h:$A \rightarrow P$ and k:$B \rightarrow P$ is called push-out (of f and g), if we have $h \circ f = k \circ g$, where $\circ$ denotes composition, furthermore, for all specification P' and morphisms h' and k' such that $h' \circ f = k' \circ g$. There exists an unique morphism u:$P \rightarrow P'$ such that $u \circ h = h'$ and $u \circ k = k'$. The specification P is the complete description of the module.

In general, protocols utilize services rendered by other protocols, and extend their services to be used in conjunction with other protocols to achieve the overall desired objective. In this respect, specifications A and B can constitute interfaces of the module. Specification B could declare attributes/operations that must be imported from other modules, and similarly, specification A could declare attributes/operations that can be exported to other modules. It is to be emphasized that interaction or relationship between the modules are expressed by means of morphisms, and categorical operations assist constructing larger modules resulting from these interactions.

In order to capture the protocol or module interactions, we propose a composition schema. Under the scheme, two modules are interconnected via export and import interfaces. the push-out of two modules is the resulting specification of the composed module. Figure depicts the composition operation. Here, module 1 imports via specification B_1 whatever module 2 exports via specification A_2. The compatibility of the parameters (or semantic constraints) is governed by the morphism s. Furthermore, the following property must be respected: $t \circ g_1 = f_2 \circ s$. In this case, the resulting module is $(R_1, B_2, A_1, P_{1,2})$, where $P_{1,2}$ is the push-out of P_1 and P_2 over B_1.

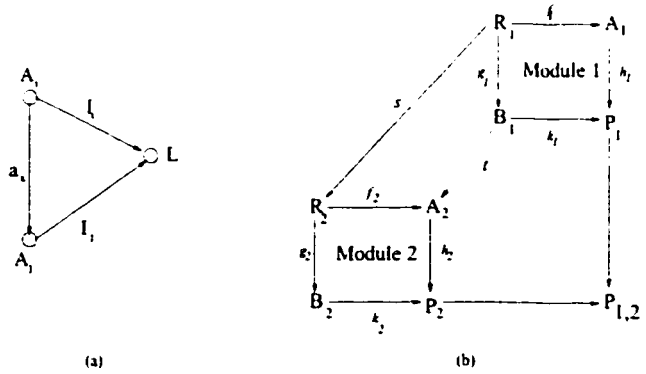

Figure 3.2: Colimit Function and Composition

Next, we highlight the salient features of basic building blocks mentioned on

the top most level in figure 2.1.

3.2 Modular Specification Framework

In this subsection, we present the modular framework of our approach for the formal specification composition. A system is composed by components, and each of the components is described by module. The modules can be interconnected to express the links between them. We utilize category theory as mentioned in last subsection to compose (compute) a module describing the whole system.

The starting point of our modular specification approach is to highlight and separate the global functionalities of the system. If we consider the Checkpointing application, four functionalities can be isolated. (1) Basic Primitives, (2) Clock Synchronization,(3) Atomic Broadcast and (4) Membership protocol. Each of these functionalities can be considered as a component and then specified in a module.

3.2.1 Component Modeling

We separate the internal context of each component specification into four parts. Moreover, for the interface we separate items constrained by the environment from items constrained by the internal behavior of the component [10]. These different parts are described in a module in different specifications.

To express the links between two specifications we define a morphism. A specification morphism $m : A \rightarrow B$ from a specification A to a specification B maps any element of the signature of A to an element of the signature of B that is compatible.

A module is composed of four specifications and four morphisms between them:

The body BOD is the complete description of the component. PAR, IMP and EXP constitute the interfaces of the module. The import IMP declares everything that is used by the module and defined by the environment. The module can impose

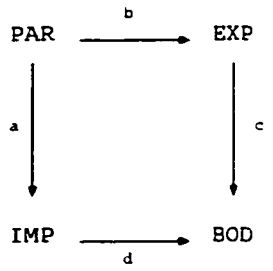

Figure 3.3: Module Interfaces

constraints on what is imported by including axioms in the IMP part. The export EXP contains elements that are defined by the module and made available to the environment. The parameter PAR contains the parameters if the module is generic. PAR also identifies the elements shared by IMP and EXP. Morphisms link the four specifications and allow renaming.

We specify four morphisms in the module to describe the links between the four parts. The items of PAR must have an image in IMP and in EXP to ensure correct definition of the morphisms. We need to show that the BOD part accepts all constraints on the imported items and does not add constraints on these imported items. The behaviors of BOD restricted to the imported items are exactly behaviors of IMP.

Our checkpointing system model can be built by constructing all the building blocks together according to the category theory.

3.2.2 Specware Supports

Specware supports the systematic construction of executable programs from axiomatic specifications via stepwise refinement. The most important aspect of Specware is the ability to represent explicitly the structure of specifications, refinements and program modules. The basis of Specware is a category of axiomatic specifications and specification morphisms. Specification structure is expressed via specification diagrams, directed multi-graphs whose nodes are labeled with specifications and arcs

with specification morphisms. Specification diagrams are useful both for composing specification from pieces and for designing structure of a system.

3.2.3 Specification-Construction Operations

The primary component of the Specware workspace is the category of specifications and specification morphisms. Diagrams in this category describe system structure. Specifications can be put together via colimits to obtain more complex specifications.

Specifications can either be directly given (as a set of sorts, operations, axioms, etc.) or constructed from other specifications via the following operations.

Translate <spec> by <renaming rules>

colimit of <diagram>

spec import<spec><spec-element> endspec

"Translate" creates a copy of a specification with some elements renamed according to the given renaming; an isomorphism is also created between the original and the translated specifications. "Colimit" is the standard operation from category theory; colimits are constructed using equivalence classes of sorts, operations, etc. "Import" place a copy of the imported specification in the importing specification; an inclusion morphism is also generated.

3.3 Basic Building Blocks

Based on the basic primitives and building-block protocols we identified in chapter 2, in this section we explain the procedures for specifying and combining the specifications of the components in the protocol stack.

3.3.1 Composing Time and Failure Specification

We show the composition of Time and Failure modules using Specware tool as an example to illustrate the concept of category-based compositional specification. The

following steps are being followed in our subsequent discussion.

(1)Specify Time;

(2)Specify the translation from Time to Failure;

(3)Specify Failure by importing data from Time;

(4)Specify the morphism and interpretation;

(5)Build the diagram which include two nodes:Time and Failure; one Morphism Time->Module;

(6)Perform the colimit calculation to yield the notion of a Time_Failure.

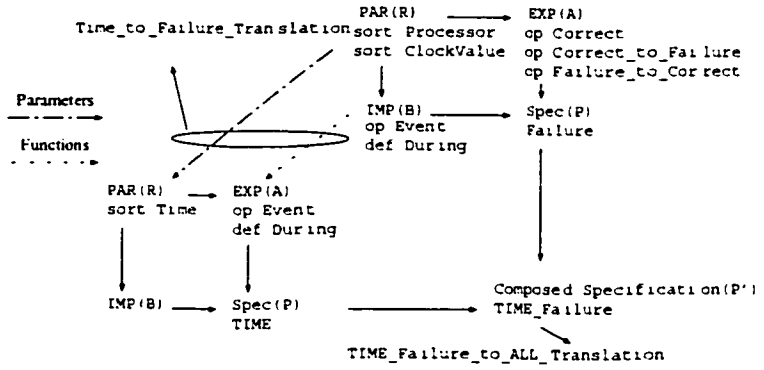

Figure 3.4: Composing Time and Failure Specifications

Refer to Figure 3.4 for illustration of these steps. The specification for module Time in Specware's MetaSlang language is shown below:

```
spec TIME is
sort Time = Nat
op   Event:   Time-> Boolean
op   during:  (Time->Boolean)*Time*Time->Boolean
def  during(Event, t0, t1):Boolean  = fa (t)( t0<t & t<t1)& (Event t)
end-spec
```

We next define *Time_to_Failure_TRANSLATION* to denote the translation from module *Time* to *Failure*. The translation is defined as a specification which maps the parameter part: Time to ClockValues, and the function part: during

to during. The morphism is as follows:

```
def TIME_to_ALL_TRANSLATION :
Spec = Specware.translate (TIME) by
["Time"|->"ClockValues","during"|->"during"]
```

In setting up the specification for the module Failure, we import the morphism specification TIME_to_Failure_TRANSLATION. The specification of module Failure is shown below:

```
spec Failure is
import TIME_to_ALL_TRANSLATION
sort  ClockValues = Nat
sort  Processors
op    Correct : Processors -> Boolean
op    Correct_to_Failure:Processors*ClockValues->Boolean
op    Failure_to_Correct:Processors*ClockValues->Boolean
end-spec
```

Following, we define the morphism and interpretation from module Time to Failure:

```
 def TIME_to_Failure : Morphism = TIME -> Failure
by  ["Time" -> "ClockValues","during"|->"during"]
def TIME_to_Failure_INTERPRETATION: Interpretation = TIME -> Failure
where  ["Time" |-> "ClockValues","during"|->"during"]
```

Finally, to construct the composite specification of these two modulars, we define the diagram with Time and Failure specification being the nodes , and the link between Time and Failure. The newly glued specification , TIME_Failure is defined as below:

```
def TIME_Failure : Diagram =
make Diagram ([TIME, Failure],
[TIME -> Failure  where  ["Time" |-> "ClockValues","during"|->"during"]])
def TIME_Failure_COLIMIT : Spec = colimit("TIME_Failure", TIME_Failure)
```

After having composed specification of TIME_Failure, we next define the

translation TIME_Failure_to_All which permits reuse of parameters and operations of TIME_Failure building block in subsequent components which are built utilizing these primitives.

```
def TIME_Failure_to_ALL_TRANSLATION :
Spec = Specware.translate (Failure)
by ["Correct" |-> "Correct",
"Correct_to_Failure" |-> "Correct_to_Failure",
"Failure_to_Correct" |-> "Failure_to_Correct"]
```

Essentially, our aim in this section, so far, has been to show how one would utilize operations of category theory to glue two specifications with the support of Specware environment. Next, we briefly mention how we construct the composite specification of the basic primitives, namely time,failure,module, and Communication.

3.3.2 Constructing Basic Building Blocks

As mentioned in Chapter 2, we require basic primitives of time, failure, communication and system model to formulate any dependable protocol operation. In order to construct a composite specification of basic building blocks, namely BaBuBl, which is composed by these four basic primitives, we follow the steps listed below:

(1) Compose specifications for time and failure primitives -(Time_Failure).

(2) Compose specifications Time_Failure and that of system model - (Time_Failure_Model).

(3) Compose specifications Time_Failure_Model and that of communication - (BaBuBl).

We show the steps in figure 3.5:

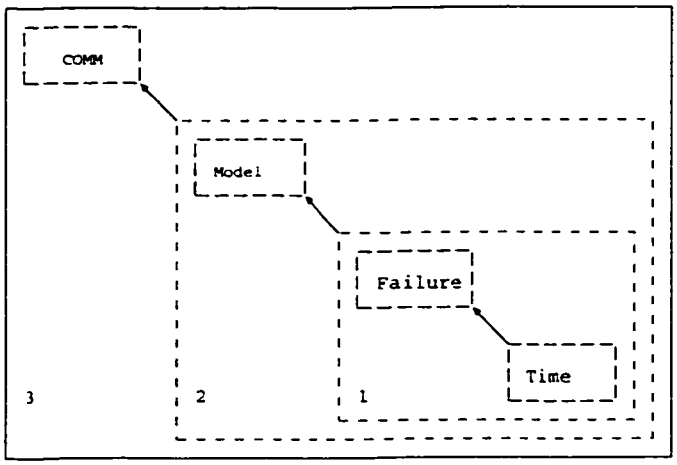

Figure 3.5: Steps of Composing Specification of Basic Building Blocks

3.3.2.1 Constructing System Model

We consider a distributed system framework comprised of a collection of processors (or node) connected via a communication network. Each processor has a clock and local memory of its own. Node in a system communicate with each other by passing messages over the defined communication network.

Basically, we declare variables and define functions that are going to be utilized later in setting up various formal specifications. For each processor p, the function C maps the clock value in real-time to the processor's local clock value. Another important notion is that of a message that is defined as a record type with fields representing sender's ID, clock values, sender's checkpointing round number and sequence number.

We construct the specification of component Model by importing from the morphisim Failure_to_Model_TRANSLATION.

```
spec Model is
import TIME_Failure_to_ALL_TRANSLATION
sort LocalClockVals = ClockValues
sort Index = Nat
sort Messages = {p:Processors, Tm:ClockValues, Km:Index, No:Nat}
```

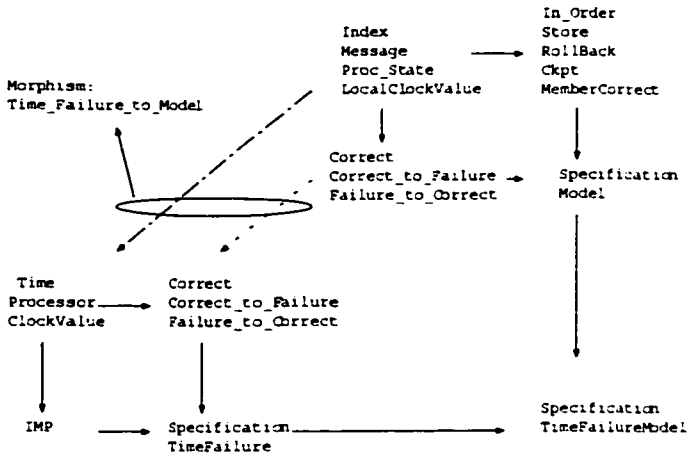

Figure 3.6: Composing Specification of Time_Failure_Model

sort Proc_state = {p:Processors, LC:LocalClockVals, n:Nat}

op { C, RC, Pi, PI, In_Order, Store, store, Rollback, Restore, Ckpt, ckpt, log, MemberCorrect, Member-Correct_Interval}

def MemberCorrect_Interval(T1,T2):Boolean = fa(p:Processors) during(MemberCorrect, T1,T2)

end-spec

Specify the morphism and interpretation

def TIME_Failure_to_Model : Morphism = Failure -> Model

by ["Time","during", "Correct","Correct_to_Failure" |->|

def TIME_Failure_to_Model_INTERPRETATION : Interpretation = Failure -> Model

where ["Time","during","Correct","Correct_to_Failure" |->|

Build the diagram:

def TIME_Failure_Model : Diagram =

make Diagram ([TIME_Failure_COLIMIT, Model],

[TIME_Failure_COLIMIT -> Model

where ["Time","during","Correct","Correct_to_Failure" |->||

Running the colimit calculation, yield the notion of TIME_Failure_Model:

def TIME_Failure_Model_COLIMIT : Spec = colimit ("TIME_Failure_Model", TIME_Failure_Model)

Define Translation for subsequent components:

```
def TIME_Failure_Model_to_All_TRANSLATION :
Spec = Specware.translate (Model)
by ["C", "RC", "Pi","PI","In_Order","Store", "store", "Rollback",
"Restore", "Ckpt", "ckpt", "log", "MemberCorrect", "MemberCorrect_Interval" ;->|
```

3.3.2.2 Constructing Basic Building Blocks

We construct the specification of Basic Building blocks by importing from the morphisim TIME_Failure_Model_to _All_TRANSLATION to the specification of COMM (Communications).

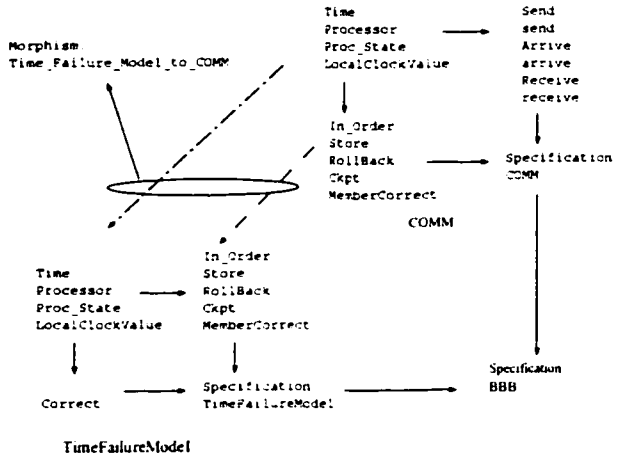

Figure 3.7: Composing Specification of Basic Building Blocks BBB

```
spec COMM is
import TIME_Failure_Model_to_All_TRANSLATION
op { Send, send, Arrive, arrive, Receive, receive }
axiom TimeStampComm is
fa(Pst:Proc_state, BroadcastDelay:ClockValues,n:Index)
fa(p,q:Processors,m:Messages,T:ClockValues)
MemberCorrect_Interval(T,T+BroadcastDelay)& send(p,m,q,T) =>arrive(q,m,p,T+BroadcastDelay)
end-spec
```

Specify the morphism and interpretation:

```
def TIME_Failure_Model_to_COMM: Morphism = Model -> COMM
by ["C", "RC", "Pi","PI","In_Order","Store", "store","Rollback", "Restore","Ckpt", "ckpt", "log",
"MemberCorrect","MemberCorrect_Interval" |->]
def TIME_Failure_Model_to_COMM_INTERPRETATION : Interpretation = Model —> COMM
where ["C", "RC", "Pi","PI","In_Order","Store", "store","Rollback", "Restore","Ckpt", "ckpt", "log",
"MemberCorrect","MemberCorrect_Interval" |->]
```

Build the diagram:

```
def BaBuBl : Diagram =
make Diagram ([TIME_Failure_Model_COLIMIT, COMM],
[TIME_Failure_Model_COLIMIT —> COMM where
["C", "RC", "Pi","PI","In_Order","Store", "store","Rollback", "Restore","Ckpt", "ckpt", "log",
"MemberCorrect","MemberCorrect_Interval" |->]
```

Running the colimit calculation, yield the notion of BaBuBl:

```
def BBB : Spec = colimit("Basic_Building_Blocks",BaBuBl)
```

After having composed specification of basic primitives, we next define the translation BBB_to_ALL_TRANSLATION which permits reuse of parameters and operations of these basic primitives in subsequent components which are built utilizing these primitives.

Define Translation for subsequent components:

```
def BBB_to_ALL_TRANSLATION:
Spec=Specware.translate(BBB) by
["C", "RC", "Pi","PI","In_Order","Store", "store","Rollback", "Restore","Ckpt", "ckpt", "log",
"MemberCorrect","MemberCorrect_Interval" |->]
```

We next describe the checkpointing protocol which utilizes synchronized clocks and use this as a case study to illustrate how we can construct specification for this protocol utilizing the formal theories/modules already been developed for building-block protocols.

3.4 Formal Specification of Checkpointing Protocol

In this section, we consider a checkpointing and recovery protocol that utilizes synchronized clocks, and which inherently depends on the building blocks of group communication services, namely synchronization, broadcast and membership.

In order to construct the specification for the checkpointing protocol, and that for the services provided by the protocol, we combine specifications of three building block protocols, namely synchronization, atomic broadcast, and membership.

(1) Utilizing the composite specification of basic primitives, we construct specification for synchronized clock, Sync_Clock, by combining BaBuBl and Sync,

(2) Combine Sync_Clock and Atomicbroadcast to yield composite specification Sync_Atomicbroadcast,

(3) Combine Sync_Atomicbroadcast and Membership to compose Sync_ABC_Membership,

(4) Combine Sync_ABC_Membership and CKPT_PTCL to get the specification for the checkpointing protocol: Checkpointing_Protocol,

(5) Combine Sync_ABC_Membership and CKPT_SERV to get the specification for services provided by the checkpointing protocol: Checkpointing-Service.

Utilizing the notion of category theory operations, such as colimit and push out, the construction of specifications for the checkpointing protocol and required services can be illustrated via diagram depicting composition of module specifications, we show that step by step in figures of following sub-sections.

3.4.1 Synchronization

Redundancy requires some form of synchronization among the independent data sources. The essential requirements is, as we described in chapter2, the properties of :

(1) Time Envelope:

Every process p has a local clock Cp with known bounded rate of drift $\rho \geq 0$ with respect to real-time. That is , for all ρ and all $t \geq t'$,

$$(1+\rho)^{-1} \leq \frac{C_p(t)-C_p(t')}{(t-t')} \leq (1+\rho)$$

Where $C_p(t)$ is reading of C_p at real-time t.

(2) Bounded Clock Skew:

There is such that for all t, and any two processes p and q,

$|C_p(t)-C_q(t)| \leq \rho$.

We construct the specification of synchronization by importing from the morphisim BBB_to _All_TRANSLATION to the specification .

```
spec Sync is
import BBB_to_ALL_TRANSLATION
sort Drift_rate = Nat
axiom TimeEnvelope is fa (p:Processors) fa (T:ClockValues) fa (Rho:Drift_rate)
(Correct p) => C(p,T)> T- Rho*T  &  C(p,T)<T + Rho*T
axiom BoundedSkew is fa (p,q :Processors) fa (T,e:ClockValues)
Correct(p) & Correct(q) => C(p,T)-C(q,T)<= e
end-spec
```

Specify the morphism and interpretation:

```
def BBB_to_Sync: Morphism =BBB->Sync by
by ["Time", "during", "Correct", "Correct_to_Failure", "Failure_to_Correct", "C", "RC", "Pi", "PI",
"In_Order", "Store", "store", "Rollback", "Restore", "Ckpt", "ckpt", "log", "MemberCorrect", "MemberCor-
rect_Interval", "Send", "send", "Arrive", "arrive", "Receive","receive"|->]
def BBB_to_Sync_INTERPRETATION:Interpretation=BBB—>Sync
where ["Time", "during", "Correct", "Correct_to_Failure", "Failure_to_Correct", "C", "RC", "Pi", "PI",
"In_Order", "Store", "store", "Rollback", "Restore", "Ckpt", "ckpt", "log", "MemberCorrect", "MemberCor-
rect_Interval", "Send", "send", "Arrive", "arrive", "Receive","receive"|->]
```

Build the diagram:

```
def Synch : Diagram =
makeDiagram ([BBB, Sync],
[BBB —> Sync where ["Time", "during", "Correct", "Correct_to_Failure", "Failure_to_Correct", "C",
"RC", "Pi", "PI", "In_Order", "Store", "store", "Rollback", "Restore", "Ckpt", "ckpt", "log", "MemberCorrect",
"MemberCorrect_Interval", "Send", "send", "Arrive", "arrive", "Receive","receive"|->]
```

Specify the morphism and interpretation, Run the colimit calculation, yield the notion of Sync_Clock:

```
def Sync_Clock : Spec = colimit("Synchronized_Clock",Synch)
```

Define Translation for subsequent components:

```
def Sync_Clock_to_All_Translation :
Spec=Specware.translate(Sync_Clock) by
[BBB -> Sync where ["Time", "during", "Correct", "Correct_to_Failure", "Failure_to_Correct", "C",
"RC", "Pi", "PI", "In_Order", "Store", "store", "Rollback", "Restore", "Ckpt", "ckpt", "log", "MemberCorrect",
"MemberCorrect_Interval", "Send", "send", "Arrive", "arrive", "Receive","receive","TimeEnvelope","BoundedSkew"]-
>||
```

3.4.2 Atomic Broadcast

Broadcast is the communication paradigm where the sender sends a message to all other nodes in the system. An atomic broadcast protocol guarantees delivery of message m entrusted by node p to node q satisfying termination, atomicity and order conditions. The following properties of atomic broadcasting are required:

Validity:If a correct process broadcast a message m , then all correct processes eventually deliver m.

Termination: Every message m whose broadcast was initiated by a correct processor at time T is delivered to all correct processors by the time (T + *BroadcastBound*)

```
Spec AtomicBroadcast is
import Sync_Clock_to_All_Translation
sort BroadcastBound= ClockValues
op  Broadcast: Processors * Messages * ClockValues ->Boolean
op  Deliver:   Processors * Messages * ClockValues ->Boolean
def validity is
ex(p, m, T) (Correct p)& Broadcast (p, m, T)=>
(fa (q,BroadcastDelay)(Correct q)& Deliver (p, m, (T+BroadcastDelay)))
def Termination is
fa(p) fa(m, T) (Correct p)& Broadcast (p, m, T)=>
    (fa (q, BroadcastDelay,BroadcastBound) (Correct q)& Deliver (p, m, (T + BroadcastDelay)) &Broad-
castDelay<BroadcastBound)
```

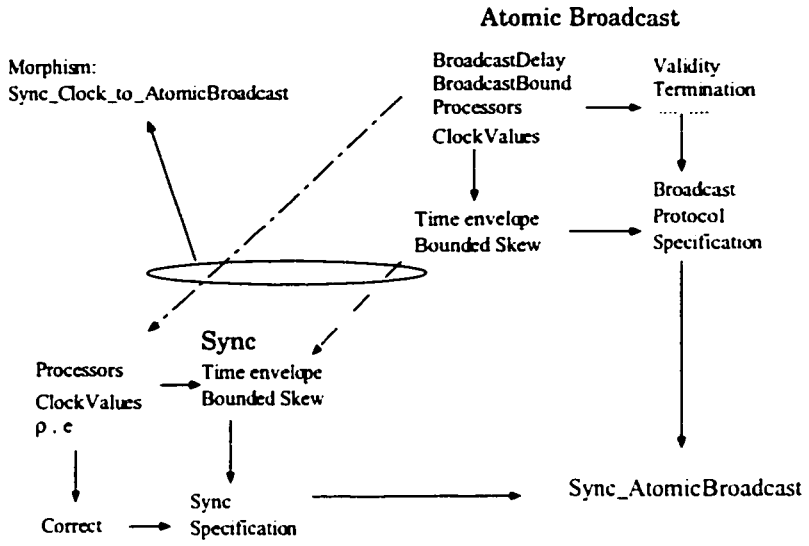

Figure 3.8: Composing Synchronization and Atomic Broadcast

end-spec

Specify the morphism and interpretation:

def Sync_Clock_to_AtomicBroadcast: Morphism = Sync_Clock->AtomicBroadcast by

["Time", "during", "Correct", "Correct_to_Failure", "Failure_to_Correct", "C", "RC", "Pi", "PI", "In_Order", "Store", "store", "Rollback", "Restore", "Ckpt", "ckpt", "log", "MemberCorrect", "MemberCorrect_Interval", "Send", "send", "Arrive", "arrive", "Receive", "receive", "TimeEnvelope", "BoundedSkew" |->|

def Sync_to_AtomicBroadcast_INTERPRETATION : Interpretation = Sync_Clock -> AtomicBroadcast where

["Time", "during", "Correct", "Correct_to_Failure", "Failure_to_Correct", "C", "RC", "Pi", "PI", "In_Order", "Store", "store", "Rollback", "Restore", "Ckpt", "ckpt", "log", "MemberCorrect", "MemberCorrect_Interval", "Send", "send", "Arrive", "arrive", "Receive", "receive", "TimeEnvelope", "BoundedSkew" |->|

Build the diagram:

def Sync_ABC: Diagram =

makeDiagram ([Sync_Clock, AtomicBroadcast|, [Sync_Clock -> AtomicBroadcast where

["Time","during", "Correct", "Correct_to_Failure", "Failure_to_Correct", "C", "RC", "Pi", "PI", "In_Order", "Store", "store", "Rollback", "Restore", "Ckpt", "ckpt", "log", "MemberCorrect", "MemberCorrect_Interval", "Send", "send", "Arrive", "arrive", "Receive", "receive", "TimeEnvelope", "BoundedSkew" |->|

Running the colimit calculation, yield the notion Sync_AtomicBroadcast

def Sync_AtomicBroadcast : Spec = colimit("Sync_AtomicBroadcast",Sync_ABC)

Define Translation for subsequent components:

def Sync_AtomicBroadcast_to_ALL_TRANSLATION:

Spec=Specware.translate(Sync_AtomicBroadcast) by

["Time", "during", "Correct", "Correct_to_Failure", "Failure_to_Correct", "C", "RC", "Pi", "PI", "In_Order", "Store", "store", "Rollback", "Restore", "Ckpt", "ckpt", "log", "MemberCorrect", "MemberCorrect_Interval", "Send", "send", "Arrive", "arrive", "Receive", "receive", "TimeEnvelope", "BoundedSkew", "Broadcast", "Deliver", "validity",
"Termination"|->|

3.4.3 Membership

Group Membership Protocols are used to ensure that the state information stored at each group member remains up-to-date and that at any time, all group members see the same state information - despite information propagation delays and failures. Essentials requirements include agreement on group membership, bounded failure detection delays, bound join delays etc.

We use following properties of Membership Protocol:

Boundedfailuredetectiondelay: there exists a time constant d_f such that if a processor belonging to group g fails at time t then, by time $t + d_f$, all members of g that stay correct in the interval $[t, t + d_j]$ will join a group G' that does not contain p.

Boundedjoindelay: There exists a time constant d_j such that if a processor starts at time t then, by time $t + d_j$, it will join a new group along with every other processor q that stays correct in the interval $[t, t + dj]$.

We construct the specification membership protocol by importing from the translation Sync_AtomicBroadcast_to _All_TRANSLATION:

```
spec Membership is
import Sync_AtomicBroadcast_to_ALL_TRANSLATION
op {joined , disjoined }
def Join is
fa (T,JoinDelay:ClockValues) fa (p,q:Processors) fa(C_P)
Failure_to_Correct(p, T) & during(C_P, T, T+JoinDelay)
=>joined(p,T+JoinDelay)
def Disjoin is
fa(T,DetectionDelay:ClockValues) fa (p:Processors) fa(NC_P)
Correct_to_Failure(p, T) & during(NC_P,T,T+DetectionDelay)
=>disjoined(p,T+DetectionDelay)
```

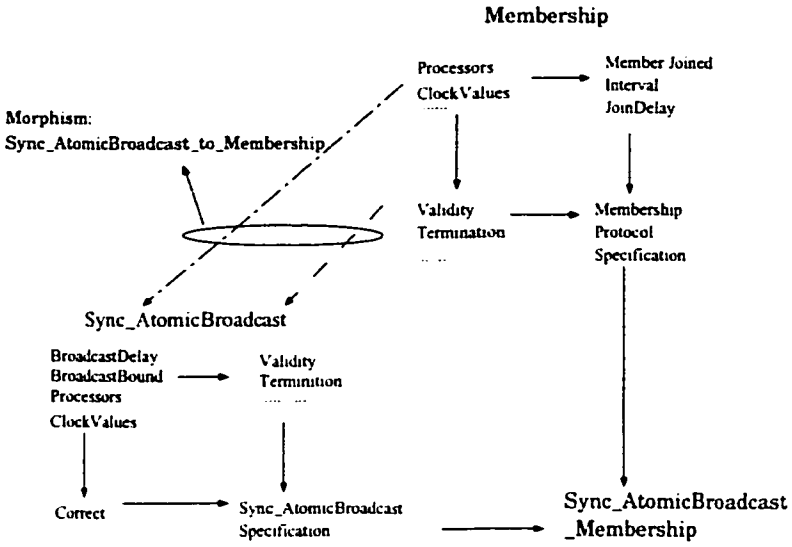

Figure 3.9: Composing Group Communication Protocols

end-spec

Specify the morphism and interpretation:

def Sync_AtomicBroadcast_to_Membership : Morphism = Sync_AtomicBroadcast -> Membership

by ["Time", "during", "Correct", "Correct_to_Failure", "Failure_to_Correct",...,"TimeEnvelope", "BoundedSkew", "Broadcast","Deliver", "validity", "Termination"|->|

def Sync_AtomicBroadcast_to_Membership_INTERPRETATION : Interpretation = Sync_AtomicBroadcast —> Membership

where ["Time","during", "Correct", "Correct_to_Failure", ..., "MemberCorrect_Interval", "Send", "send", "Arrive", "arrive", "Receive", "receive", "TimeEnvelope", "BoundedSkew", "Broadcast","Deliver", "validity", "Termination"|->|

Build the diagram:

def Sync_ABC_MEM: Diagram =

makeDiagram ([Sync_AtomicBroadcast,Membership|,

[Sync_AtomicBroadcast —> Membership where

["Time", "during", "Correct", "Correct_to_Failure", "Failure_to_Correct",...,"TimeEnvelope", "BoundedSkew", "Broadcast", "Deliver", "validity", "Termination"|->|])

Running the colimit calculation, yield the notion Sync_ABC_MEM:

def Sync_ABC_Membership : Spec = colimit("Sync_AtomicBroadcast_Membership", Sync_ABC_MEM)

Define Translation for subsequent components:

def Sync_ABC_MEM_to_ALL_TRANSLATION:

Spec=Specware.translate(Sync_ABC_Membership) by

["Time", "during", "Correct", "Correct_to_Failure", "Failure_to_Correct", ... ,"TimeEnvelope", "BoundedSkew", "Broadcast", "Deliver", "validity", "Termination", "Join", "Disjoin" |->||)

At this stage, we have formally composed specifications for building block protocols, we now formally specify required service of checkpointing protocol utilizing services extended by basic building block protocols.

3.4.4 Required Service of Checkpointing Protocol

In order to have a consistent set of checkpoints, records of orphan message and lost message must not be there in a valid checkpoint. We formalize the notion of "lost-messages" and "orphan-messages" in our specifications.

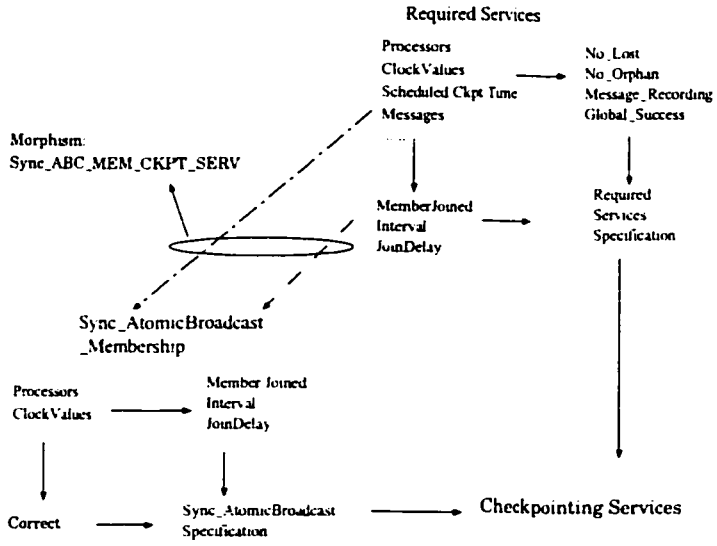

Figure 3.10: Composing Checkpointing Services

We construct the specification of the required service of checkpointing protocol by importing from the translation Sync_ABC_MEM_to _All_TRANSLATION:

```
spec CKPT_SERV is
import Sync_ABC_MEM_to_ALL_TRANSLATION
```

For any around n and any processor p, all the messages that were sent out before the p's scheduled local checkpoint, must be received by a target processor q

before the time $(C(q,T)+BroadcastBound+e)$. In the absence of failures, a message which was sent by p at time T, i.e., $(S\text{-}BroadcastBound\text{-}e) < C(p,T) \leq S$ will be delivered to the destination no later than the destination processor q's local time $(S+BroadcastBound+e)$, i.e., $(C(q,T)+BroadcastDelay) < (S+BroadcastBound+e)$, and no message is being lost. Note that for a specific round of checkpointing operation, S is the scheduled time instant.

```
def NO_LOST is
fa (m:Messages) fa(p,q:Processors)
 fa(e,S,T,BroadcastDelay,BroadcastBound:ClockValues) fa (rho:Nat)
   S-BroadcastBound-e<C(p,T) & C(p,T)<=S &
     MemberCorrect_Interval(S-BroadcastBound-rho*S-e, S+BroadcastBound+rho*S+e) =>
       (send(p,m,q,T) & arrive(q,m,p,T+BroadcastDelay) & In_Order(m) =>
          receive(q,m,p,T+BroadcastDelay)
            &(C(q,T)<=S+e =>C(q,T)+BroadcastDelay <S+BroadcastBound+e))
```

All the messages, that are received from other processors and were logged by the processor p, are those messages that were sent before the sender's current round of local checkpoint. This prevent the possibility of orphan messages.

```
def NO_ORPHAN is
fa (m:Messages) fa(p,q:Processors)
 fa(e,S,T,BroadcastDelay,BroadcastBound:ClockValues) fa (rho:Nat)
   S-BroadcastBound-e<C(p,T) & C(p,T)<S+BroadcastBound+e &
     MemberCorrect_Interval(S-BroadcastBound-rho*S-e, S+BroadcastBound+rho*S+e) =>
       receive(p,m,q,T) & log(p,m,T)
```

In each round, when a processor establishes its checkpoint at time T which is before or at its scheduled checkpoint time S, then it stores its local state and broadcasts a "mark" message to all other processors in the group indicating its local success in establishing checkpoint. Once all processors have received the "mark" message, then this round of checkpoint operation is successful. Furthermore, since "mark" message is also a type of message, as and when a "mark" message is received, we could know that all the previous messages were received successfully. Thus, broadcast of a "mark" message could be a sign indicating that the current round of checkpoint operation is successful. we formalize this notion as follows:

```
def CheckPoint_Proceed is
fa (Mark:Messages) fa(p,q:Processors)
```

```
fa(e,S,T,BroadcastDelay,BroadcastBound:ClockValues) fa (rho:Nat)
 S-BroadcastBound-e<C(p,T) & C(p,T)<S &
  MemberCorrect_Interval(S-BroadcastBound-rho*S-e, S+BroadcastBound+rho*S+e)=>
   (ckpt(p,T) & Broadcast(p, Mark,T) & Deliver(q, Mark, T+BroadcastDelay) &
    (C(q,T) <= S+e =>C(q,T) + BroadcastDelay<S+BroadcastBound+e))
```

We describe the description of the procedure of messages recording which covered the requirements of the message receiving and Logging in the processors point to point communications. For any processor p and q in the member group, during p's local clock interval $(S\text{-}BroadcastBound\text{-}e,S]$, in the absence of the omission failure, and no out of order happens, any message m which were send by p to q will be received by q before q's local clock $S+BroadcastBound+e$. and the message m will be logged by q if and only if the round number in the message m is less than the round number in the processor q's state. So that, all the messages which were logged were those messages which were sent before the sender's current round checkpointing. We formalize this notion as follows:

```
def Message_Recording is
fa (m:Messages) fa(p,q:Processors)
 fa(e,S,T,BroadcastDelay,BroadcastBound:ClockValues) fa (rho:Nat)
  S-BroadcastBound-e<C(p,T) & C(p,T)<=S &
   MemberCorrect_Interval(S-BroadcastBound-rho*S-e, S+BroadcastBound+rho*S+e)=>
    (send(p,m,q,T) & arrive(q,m,p,T+BroadcastDelay) & In_Order(m) =>
       receive(q,m,p,T+BroadcastDelay)
         &(C(q,T)<=S+e =>C(q,T)+BroadcastDelay <S+BroadcastBound+e))
    & (fa(m:Messages) receive(p,m,q,T) & log(p,m,T))
end-spec
```

After having specified the properties of the checkpointing protocol services, we compose the completed specification:

Specify the morphism and interpretation:

```
def Sync_ABC_MEM_to_CKPT_SERV: Morphism = Sync_ABC_Membership -> CKPT_SERV
by ["Time", "during", "Correct", "Correct_to_Failure", "Failure_to_Correct", "C", "RC", "Pi", "Pl",
"In_Order", "Store", "store", "Rollback", "Restore", "Ckpt", "ckpt", "log", "MemberCorrect", "MemberCor-
rect_Interval", "Send", "send", "Arrive", "arrive", "receive", "receive", "TimeEnvelope", "BoundedSkew", "Broad-
cast","Deliver", "validity", "Termination","Join","Disjoin"|->|
def Sync_ABC_MEM_to_CKPT_SERV_INTERPRETATION: Interpretation = Sync_ABC_Membership—
>CKPT_SERV
where
```

```
["Time", "during", "Correct", "Correct_to_Failure", "Failure_to_Correct", "C", "RC", "Pi", "PI", "In_Order",
"Store", "store", "Rollback", "Restore", "Ckpt", "ckpt", "log", "MemberCorrect", "MemberCorrect_Interval",
"Send", "send", "Arrive", "arrive", "receive", "receive", "TimeEnvelope", "BoundedSkew", "Broadcast","Deliver",
"validity", "Termination","Join", "Disjoin"|->]
```

Build the diagram:

```
def Sync_ABC_MEM_CKPT_SERV: Diagram =
makeDiagram ([Sync_ABC_Membership, CKPT_SERV],
[Sync_ABC_Membership —> CKPT_SERV where
["Time", "during", "Correct", "Correct_to_Failure", "Failure_to_Correct", "C", "RC", "Pi", "PI", "In_Order",
"Store", "store", "Rollback", "Restore", "Ckpt", "ckpt", "log", "MemberCorrect", "MemberCorrect_Interval",
"Send", "send", "Arrive", "arrive", "receive", "receive", "TimeEnvelope", "BoundedSkew", "Broadcast","Deliver",
"validity", "Termination","Join", "Disjoin"|->]])
```

Running the colimit calculation, yield the notion Sync_ABC_MEM_CKPT_SERV:

```
def CKPT_Service : Spec = colimit ("Checkpointing_Service", Sync_ABC_MEM_CKPT_SERV)
```

3.4.5 Checkpointing Protocol

We have formally specified the required service of the checkpointing protocol, and the protocol operations. We have also discussed formalization of services provided by basic group communication protocols. Based on these, the next objective is to establish the correctness of the checkpointing protocol operation. In this section, we first formally specify the protocol operation where all cooperating processes checkpoint periodically, each with the same period, Pe.

```
spec CKPT_PTCL is
import Sync_ABC_MEM_to_ALL_TRANSLATION
```

Arrive : A message is sent by processor p in the time T which $C(p,T)$ is inside the period of p's local clock (S-$BroadcastBound$-e,S]. The message will arrive the destination in the destination processor's local time $C(q,T)+BroadcastDelay$.

```
def Arrive is
fa (m:Messages) fa(p,q:Processors)
 fa(e,S,T,BroadcastDelay,BroadcastBound:ClockValues) fa (rho:Nat)
  S-BroadcastBound-e<C(p,T) & C(p,T)<=S &
   MemberCorrect_Interval(S-BroadcastBound-rho*S-e, S+BroadcastBound+rho*S+e)=>
    (send(p,m,q,T)=>(ex(T1:ClockValues) T1=T+BroadcastDelay & arrive(q,m,p,T1)))
```

Receive : A message m which was sent by processor p at T which $C(p,T)$ is

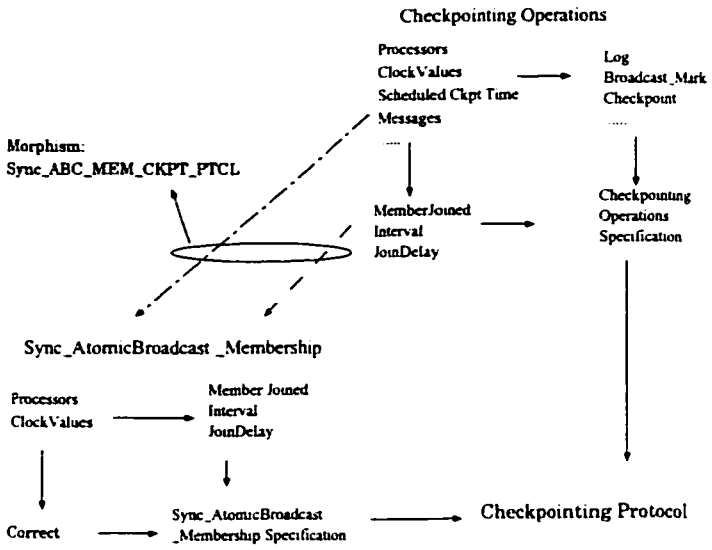

Figure 3.11: Composing Checkpointing Protocol

inside its local clock interval $(S\text{-}BroadcastBound\text{-}e, S]$. The message will arrive the destination before the destination processor's local time $S+BroadcastBound+e$. If no communication failure happens and the transmission delay ($BroadcastDelay$) is less than the bounded delay ($BroadcastBound$), and if the messages are in order, then m will be Received by q.

```
def Receive is
fa (m:Messages) fa(p,q:Processors)
 fa(e,S,T,BroadcastDelay,BroadcastBound:ClockValues) fa (rho:Nat)
  S-BroadcastBound-e<C(p,T) & C(p,T)<=S &
   MemberCorrect_Interval(S-BroadcastBound-rho*S-e, S+BroadcastBound+rho*S+e)=>
    (arrive(p,m,q,T) & BroadcastDelay<BroadcastBound & In_Order(m)
     =>receive(q,m,p,T))
```

Logging: If the round number in the massage m which was received by p at T is same with the local checkpoint round number of q, then this message will be logged by q.

```
def Logging is
fa (m:Messages) fa(p,q:Processors)
 fa(e,S,T,BroadcastDelay,BroadcastBound:ClockValues) fa (rho:Nat)
  S-BroadcastBound-e<C(p,T) & C(p,T)<S+BroadcastBound+e &
   MemberCorrect_Interval(S-BroadcastBound-rho*S-e, S+BroadcastBound+rho*S+e)=>
```

```
(receive(p,m,q,T)=>log(p,m,T))
```

Checkpoint: When processor p received message m before it made its current round checkpoint, and the m was sent by sender after the sender has done its checkpoint of the current round, the processor p will make checkpoint right at the time of the message received. Otherwise, the checkpoint time will be local clock value of p at S.

```
def CheckPoint is
fa (m:Messages) fa(p:Processors)
 fa(e,S,T,BroadcastDelay,BroadcastBound:ClockValues) fa (rho,n:Nat)
  S-BroadcastBound-e<C(p,T) & C(p,T)<=S &
  MemberCorrect_Interval(S-BroadcastBound-rho*S-e, S+BroadcastBound+rho*S+e)=>
    (if ( ex(m) log(p,m,T) & C(p,T)<S )
     then ckpt(p,T) & store(p,T) & Pi(p,T)=n+1
     else Ckpt(p,S) & Store(p,S) & PI(p,S)=n+1
    )
```

MARK: When a round of checkpoint operation is successfully done, each processor broadcasts mark message. And because mark message is also a type of message, it is required to be received in the order with those data messages. When mark message is received, we could know that all the previous data message was received successfully, this make sure that mark message could be a sign of checkpoint is successful.

```
def MARK is
fa (Mark:Messages) fa(p,q:Processors)
 fa(e,S,T,BroadcastDelay,BroadcastBound:ClockValues) fa (rho:Nat)
  S-BroadcastBound-e<C(p,T) & C(p,T)<=S &
   MemberCorrect_Interval(S-BroadcastBound-rho*S-e, S+BroadcastBound+rho*S+e)=>
    (ckpt(p,T)=>Broadcast(p,Mark,T))
end-spec
```

After having specified the properties of the checkpointing protocol, we compose the completed specification:

```
def Sync_ABC_MEM_to_CKPT_PTCL:Morphism=Sync_ABC_Membership->CKPT_PTCL
by ["Time", "during", "Correct", "Correct_to_Failure", "Failure_to_Correct", "C", "RC", "Pi", "PI",
"In_Order", "Store", "store", "Rollback", "Restore", "Ckpt", "ckpt", "log", "MemberCorrect", "MemberCor-
rect_Interval", "Send", "send", "Arrive", "arrive", "receive", "receive", "TimeEnvelope", "BoundedSkew", "Broad-
cast","Deliver", "validity", "Termination","Join","Disjoin"|->||]
```

```
def Sync_ABC_MEM_to_CKPT_PTCL_INTERPRETATION : Interpretation = Sync_ABC_Membership
—>CKPT_PTCL
    where ["Time", "during", "Correct", "Correct_to_Failure", "Failure_to_Correct", "C", "RC", "Pi", "PI",
"In_Order", "Store", "store", "Rollback", "Restore", "Ckpt", "ckpt", "log", "MemberCorrect", "MemberCor-
rect_Interval", "Send", "send", "Arrive", "arrive", "receive", "receive", "TimeEnvelope", "BoundedSkew", "Broad-
cast","Deliver", "validity", "Termination","Join","Disjoin"|->||]
```

Build the diagram:

```
def Sync_ABC_MEM_CKPT_PTCL: Diagram =
makeDiagram ([Sync_ABC_Membership, CKPT_PTCL|,
[Sync_ABC_Membership —> CKPT_PTCL where
["Time","during", "Correct", "Correct_to_Failure", "Failure_to_Correct", "C", "RC", "Pi", "PI", "In_Order",
"Store", "store", "Rollback", "Restore", "Ckpt", "ckpt", "log", "MemberCorrect", "MemberCorrect_Interval",
"Send", "send", "Arrive", "arrive", "receive", "receive", "TimeEnvelope", "BoundedSkew", "Broadcast","Deliver",
"validity", "Termination", "Join","Disjoin"|->||])
```

Running the colimit calculation, yield the notion $Sync\_ABC\_MEM\_CKPT\_PTCL$:

```
def CKPT_Protocol : Spec = colimit("Checkpointing_Protocol", Sync_ABC_MEM_CKPT_PTCL)
```

3.5 Discussion

In this chapter, we have seen how a complex protocol like the checkpointing protocol can be built from smaller sub-protocols or/and basic primitives using the compositional techniques of category theory, thereby illustrating that our proposed category-theory based formal framework serves as a design tool to systematically compose complex distributed dependable protocols. The key idea presented in this chapter is that if we identify the individual blocks that make up the global protocol (checkpointing protocol), then with the help of category theory concepts of morphism and colimits we can compose the identified sub-blocks into the global protocol. We envision that this compositional approach would facilitate easier design, specification and verification processes and also the overall treatment of protocols with stricter real time and dependable requirements. The building blocks that we had identified in order to build up the checkpointing protocol by composition will be useful to system designers because they will permit thoroughly and rigorously tested formal

theories of required system and component behavior, and will support system design decisions and modification.

Chapter 4

Formal Specification and Verification Using PVS

In this chapter we show how a complex protocol can be formalized and verified by the reuse of the individual functional primitives of the chosen protocols, with PVS tool. Section 1 describes our assertional-based composition approach to protocol verification. In Section 2, we formalize the basic functional primitives and basic group communication services. Section 3 presents our assertion-based formalization and verification of the checkpointing protocol. We conclude with discussions in Section 4.

4.1 A Compositional Approach to Protocol Verification

Utilizing our category-theory-based framework presented in Chapter 3, we have identified basic primitives and group communication protocols. Also we have shown the operation and presented interactions among building blocks and how the modules interact with each other via impost and export interfaces. Using Specware, we

have already finished the compositional specification for the checkpointing protocols and the required services which support the goal of satisfying global checkpointing protocol.

We start our verification from specifying a network of processors whose local clocks are approximately synchronized. Based on synchronized clocks, we implement a more powerful communication mechanism, Atomic Broadcast. Furthermore, we implement a membership protocol for monitoring the processors in the set. These can then be used for higher level services, such as checkpointing protocol.

In this thesis, we use an assertional method to verify distributed protocols. That is , logical formulas (assertions) characterize the required service, the protocols, and assumptions about the underlying communication mechanism. Besides primitives to reason about timing, there are also primitives to express assumptions about the of components and the failure hypothesis explicity.

In our context of protocol verification, protocols within the class of group communication protocols are disjoined in nature, i.e. , neither of them can change the parameter accessed by the other one. The verification step makes use of the proof rule for disjoint parallel programs:

$$\frac{\{p_i\}S_i\{q_i\},\ i \in \{1,\ldots,n\}}{\{\bigwedge^n_{i=1} p_i\}[S_1\|\ldots\|S_n]\{\bigwedge^n_{i=1} q_i\}}$$

Here, p and q are assertions, and essentially define the pre-operational conditions and the expected/required services of the protocol S_i after its complete execution, respectively. Note that S_i could be considered as $\bigwedge^n_{i=1} S_i(j)$ where $S_i(j)$ denotes the protocol S_i running on a processor j, $1 \leq j \leq k$.

Let ASMP denotes the required assumptions about time, failure, communication, and system models. ABC, GM, and $CKPT_{op}$ denote atomic broadcast protocol, group membership protocol, and checkpointing operations running on all correct processors. Let Correct and P_X denote the processors being correct, and the properties (or services) the protocol X must satisfy (or provide), respectively. Note that P_{CKPTop} represents properties that correspond to receiving a message, recording (or logging) a message and successfully saving the system state, whereas, P_{CKPT} represents properties corresponding to forming a consistent system state. Adapting the proof rule for disjoint parallel programs, we present the following refinement of the proof rule:

1 ASMP

2 {Correct}ABC{P_{ABC}}

3 {Correct}GM{P_{GM}}

4 {Correct}$CKPT_{op}${$P_{CKPT_{op}}$}

5 $P_{ABC} \bigwedge P_{GM} \bigwedge P_{CKPT_{op}} \rightarrow P_{CKPT}$

{ASMP$\bigwedge$Correct}{ABC||GM||$CKPT_{op}$}{P_{CKPT}}

In order for the *conclusion* of the above proof rule to be true, we must make sure that the *premises* are true. Essentially, we assume that premises 1,...,4 are true, and our objective then is to show that Premise-5 holds. We elaborate on this step in the subsequent sections. We begin with formally specifying the required or expected services of the checkpointing operation.

In subsequent sections, formalization of basic building blocks and checkpointing protocol is structured as follows:

- Specify basic primitives (Time, Failure and system models, and communication mechanisms) to express basic group communication services,
- Specify services provided by basic group communication protocols,

- Specify the required services that should be provided by the protocol under study,
- Specify the protocol operations performed by each processor in the system,
- Verify the protocol by proving that the protocol operations along with supporting services rendered by basic building block protocols lead to the required services to be provided by the protocol.

To obtain mechanical support, we use the interactive proof checker PVS (Prototype Verification System). The PVS specification language is a higher order typed logic and specifications can be structured into a hierarchy of (parameterized) theories. The tool contains a proof checker to construct proofs interactively and to rerun proofs automatically after small changes.

4.2 Formalization of Basic Functional Primitives

In this section, we present the development of PVS specifications of basic functional primitives being used in formally composing dependable protocols. Note that we provide discussion before each PVS code fragment to describe the concept that has been actually formalized.

The primitive theories formalized in the subsequent sections are needed to specify basic group communication services typically required in distributed operations.

4.2.1 Timing and Failure Model

The formal theory Time is defined to reason about timing aspects of distributed protocols. This parameterized theory takes three arguments, namely, a time domain Time and two orders $<$ and $\leq$ defined on Time. These predicates are defined in the PVS prelude.

```
Time primitive  Time[Time:type,
   <:(strict_order?[Time]),
   <=:(partial_order?[Time])]: theory
Begin
assuming
leq:assumption FORALL(t1,t2:Time):t1<=t2<=>t1<t2 OR t1=t2
end assuming
t,t0,t1:var Time
Events:Type=[Time->bool]
P:var Events
during?(P, t0, t1):bool= Forall (t| t0<t and t<t1) : P(t)
END Time
```

To specify the time when failure happened in the group processors, we define two booleans $Correct\_to\_Failure(r)(t)$ and $Failure\_to\_Correct(r)(t)$: For any processor r fails at any time t, is denoted by $Correct\_to\_Failure(r)(t)$, in opposite, $Failure\_to\_Correct(r)(t)$. A clock value is defined as discrete time using integer type. Two orders $<$ and $\leq$defined on it are imported via Time theory.

```
Failure Primitive
Failure [Processors:type]: theory
BEGIN
ClockValues:type=int
importing Time[ClockValues,<,<=]
t,t0,t1:var ClockValues
r:var Processors
correct(r)(t):bool
Correct_to_Failure(r)(t):bool
Failure_to_Correct(r)(t):bool
END Failure
```

4.2.2 System Model

The distributed system consists of a collection of processes which belongs to the set of processors, MemAll. Each process executes on a single processor. Information exchange only occurs through message exchange. Each process Pi has processor state $Proc\_state$, denoted by a record type which includes all application variables: *processorid*, $LocalClockValue$ and the *Indexnumber* for the checkpointing.

The state changes according to the operations the process performs. Each process executes a sequence of events/actions which is either a send, receive, Checkpointing, Message logging, State store, Rollback or operation that transforms its state like *Indexnumber* increases. Ordering of events in the distributed system is true that the sequence of events within a single process can be placed in a total order. Furthermore, to declare the system situation, two booleans are defined to describe that all the processors in the system are correct at a specific time and during a specific time interval.

The theory *Model* is parameterized by the type processors. Basically, in this theory we declare variables and define functions that are going to be utilized later in setting up various formal specifications.

For each processor p, the function C maps the clock value in real-time to the processor's local clock value. Another important notion is that of a message that is defined as a record type with fields representing sender's ID, clock values, sender's checkpointing round number and sequence number.

Also to present a processor's state, a type Proc_state is defined as a record type which contains the processor's ID, local Clock value and current checkpointing ground number. These parameters are being used in specifying checkpointing protocol operations.

```
System Model
Model[Processors:type]:theory
Begin
importing Failure[Processors]
T,T0,T1,T2:VAR ClockValues
LocalClockVals:type = ClockValues
LC:VAR LocalClockVals
NonNegClockValues.type={T|T>=0}
S,Pe,e:ClockValues
Index:type=int
n:var Index
C:[Processors,ClockValues->LocalClockVals]
RC:[Processors,LocalClockVals->ClockValues]
p,q:var Processors
```

```
Messages:type=[#P:Processors,Tm:ClockValues,Km:Index,No:nat#]
m:var Messages
Proc_state:type=[#P:Processors, LC:LocalClockVals, n:Index#]
Pst:var Proc_state
Pi:[Processors,ClockValues->Index]
PI:[Processors,LocalClockVals->Index]
In_Order(m):bool
MemAll:setof[Processors]
Store(p)(LC):bool
store(p)(T):bool=Store(p)(C(p,T))
RollBack_Memall(n)(T):bool
Restore_Memall(n)(T):bool
Ckpt(p)(LC):bool
ckpt(p)(T):bool=Ckpt(p)(C(p,T))
log(p,m)(T):bool
Checkpoint_done:bool
MemberCorrect(T):bool = forall (p|member(p,MemAll)): correct(p)(T)
MemberCorrect_Interval(T0,T): bool= during?(MemberCorrect,T0,T)
Correct_interval:theorem
forall (p|member(p,MemAll), q|member(q,MemAll)): MemberCorrect_Interval(T0,T)
=>during?(correct(p),T0,T) and during?(correct(q),T0,T
End Model
```

4.2.3 Communication Primitives

Whenever a message is sent, the event of sending a message occurs before the event of receiving the message. We define the communication relation, denoted by send and receive, as follows: for any message m, Send(m) $\rightarrow$Arrive(m) (where Send(m) is the event of sending the message and Arrive(m) the receiving event).

Each process Pi maintains a logical clock $C(Pi)$ The clock is a monotonically increasing counter which related to the real time. Each site timestamps events with the value of its logical clock. For each send and state transforming operation: timestamp equal with the current value of the local logical clock. Let m be the event of sending a message: then the message piggybacks the Timestamp. On receiving m in the destination, a process Pj computes max (C(Pj), $Timestamp$) to identify if the message m is over time. According to the checkpointing algorithm, a message

which arrived to the destination may not be received(Denoted by Receive(m)) due to the time bound.

Group communication protocols employ diagram to transmit messages between processors. We define primitives to express that a processor p sends (or receive) a message m to (or from) processor q at local clock value C(p,T), ie,. at real-time T.

```
Communication Primitive
Comm[Processors:type]: theory
begin
importing Model[Processors]
T0,T:var ClockValues
BroadcastDelay: var ClockValues
BroadcastBound: ClockValues
LC:var LocalClockVals
p,q:var Processors
n:var Index
m:var Messages
Pst:var Proc_state
Send(p,m,q)(LC):bool
send(p,m,q)(T):bool=Send(p,m,q)(C(p,T))
Arrive(q,m,p)(LC):bool
arrive(q,m,p)(T):bool=Arrive(q,m,p)(C(q,T))
Receive(q,m,p)(LC):bool
receive(q,m,p)(T):bool= Receive(q,m,p)(C(q,T))
```

Based on these basic communciation primitives, we now formally specify the communication mechanism used in timestamp-based checkpointing protocol. Essentially, when a message is transmitted, the sending processor configures the message by including its own ID, its current local clock value, the current round number of the checkpointing operation and the sequence number of the message. The underlying property of the communciation protocol is that a message m sent by processor p at time T will arrive at the destination q by the time $T+BroadcastDelay$, if no omission failure occurs. Note that T denotes the physical time.

```
TimestampComm:bool=
FORALL Pst,m,BroadcastDelay,T,n:
Forall (p|member(p,MemAll), q|member(q,MemAll)):
MemberCorrect_Interval(T,T+BroadcastBound) and
```

```
Pst'LC=C(p,T) and send(p,m,q)(T) and Pst'n=n
=>m'P=p and m'Tm=C(p,T) and m'Km=n
=>arrive(q,m,p)(T+BroadcastDelay)
End Comm
```

So far, we have formally specified basic primitives that are needed to express required services of basic group communication protocols. Building up on these theories, we next present formal representation of services provided by building-block protocols.

4.3 Formalization of Basic Group Communication Services

Application Servers replicate the system state and use group communication protocols to co-ordinate their activities in the presence of failures. The empirical formulation of a generic group communication protocol involves the following procedures: (a) synchronization - keeping processor clock approximately synchronized, (b) broadcast-providing consistent information to multiple processors in the distributed system, ensuring message ordering and guaranteed delivery, and (c) membership-ensuring up-to-date state information at each group member. Following this, we present PVS theories to formalize the basic assumptions.

4.3.1 Synchronization

In order to have a common time line to coordinate various operational activities among different basic building blocks being utilized in formulating the checkpointing protocol, we have assumed that processor clocks are externally synchronized, i.e., processor clocks are within some given maximum deviation from an external time reference, which keeps real-time. Note that externally synchronized clocks are also internally synchronized where clocks of different processors are kept within some

maximum relative deviation of each other.

We formalize the reasoning between a correct processor and prop[erties of clock synchronization primitive as follows.

```
Clock Synchronization
Sync[Processors:TYPE]:Theory
BEGIN
importing Comm[Processors]
Drift_Rate:type=ClockValues
T,D:VAR ClockValues
LC:VAR LocalClockVals
p,q:var Processors
Rho:Drift_Rate
TimeEnvelope:axiom
correct(p)(T) => C(p,T)>T-Rho*T and C(p,T)<T+Rho*T
BoundedSkew :axiom
correct(p)(T) and correct(q)(T)=>abs(C(p,T)-C(q,T))<e
End Sync
```

4.3.2 Atomic Broadcast

To support the checkpoint protocol, two important properties that Validity and Termination of Atomic Broadcast are employed for the system to constraint all the broadcasted messages can arrived the destination without failure. Atomic broadcast has properties that all correct processors deliver the same set of messages and this set include all the messages broadcast by correct processors, and the message delivered to the destination by the bounded time [26]. Here we define the atomic broadcasting as operations the process executes a sequence of events/actions which is *Broadcast*, *Deliver* and *Receive_Broadcasting*. For any message which is delivered to the destination within the *Broadcast_Bound*, it will be received by the destination processor by operation *Receive_Broadcast*(m), otherwise, communication failure happened.

An important objective in distributed system is to provide consistent information to multiple processors in the system. Since the basic communication primitive supported by a network is point to point communication, broadcast services must

make use of this basic primitive. The basic assumption made here is that in the absence of failures, a message broadcasted by p at time T will be delivered to the destination processor q no later than its local clock value $(C(q,T)+BroadcastBound+e)$. In terms of global time, for the Kth checkpoint scheduled at time S, message interactions among different processors are being considered over the time interval $(S - BroadcastBound - rho*S - e,\ S + BroadcastBound + rho*S + e)$. The required services of the broadcast function are analyzed/ considered within this time interval. The following services are essentially required from the broadcast function and are formally specified below:

- *Validity*: if a correct processor broadcasts a message m, then all correct processors eventually deliver m.
- *Termination*: every message whose broadcast was initiated by a correct processor at time T is delivered to all correct processors by the time $(T+BroadcastBound)$

```
Atomic Broadcasting
Broadcast[Processors:type]: THEORY
 BEGIN
 importing Sync[Processors]
 T:VAR ClockValues
 LC:var LocalClockVals
 p,q:var Processors
 m,mark:var Messages
 BroadcastDelay:var NonNegClockValues
 Broadcast(p,m,q)(LC): bool
 broadcast(p,m,q)(T): bool=Broadcast(p,m,q)(C(p,T))
 Deliver(p,m,q)(LC): bool
 deliver(p,m,q)(T): bool=Deliver(p,m,q)(C(p,T))
 Receive_broadcasting(p,m,q)(LC):bool
receive_broadcasting(p,m,q)(T):bool= Receive_broadcasting(p,m,q)(C(p,T))
Validity : bool=
FORALL mark,BroadcastDelay:
Forall (p|member(p,MemAll), q|member(q,MemAll)):
Forall (T|S-BroadcastBound-e<C(p,T) and C(p,T)<=S):
MemberCorrect_Interval(S-BroadcastBound-Rho*S-e,S+BroadcastBound+Rho*S+e)=>
(broadcast(p,mark,q)(T) => deliver(q,mark,p)(T+BroadcastDelay))
```

```
BroadcastTermination:bool =
FORALL mark,BroadcastDelay:
Forall (p|member(p,MemAll),q|member(q,MemAll)):Forall (T|S-BroadcastBound-e<C(p,T) and C(p,T)<=S):
MemberCorrect_Interval(S-BroadcastBound-Rho*S-e,S+BroadcastBound+Rho*S+e)=>
(deliver(q,mark,p)(T+BroadcastDelay) and BroadcastDelay<BroadcastBound and In_Order(mark)
=>receive_broadcasting(q,mark,p)(T+BroadcastDelay))
END Broadcast
```

4.3.3 Membership

Membership protocols basically ensure that the state information stored at each group member remains up-to-date and that at any time, all group members see the same state information despite information propagation delays and failures. We assume that a group membership service exists which provides for the detection of a process failure and orderly rejoining of a process after recovery. These are some of the required services from a membership block over the checkpointing protocol composition.

- *Agreement on group membership*: if p and q are joined to the same group and are both alive then their membership views are identical.
- *Recognition*:if p is alive joined and joined to a group then its id will be included in its membership view.
- *Bounded failure detection delay*: there exists a time constant d_f such that if a processor belonging to group g fails at time t then, by time $t+d_f$, all members of g that stay correct in the interval $[t.t+d_j]$ will join a group G' that does not contain p.
- *Bounded join delay*: There exists a time constant d_j such that if a processor starts at time t then, by time $t+d_j$, it will join a new group along with every other processor q that stays correct in the interval $[t,t+dj]$.

```
BEGIN
importing Broadcast[Processors]
T,T0:VAR ClockValues
T1,T2:ClockValues
LC:VAR LocalClockVals
p,q:var Processors
n:Index
DetectionDelay:ClockValues
JoinDelay:ClockValues
joined(p)(T):bool
View(p)(LC):bool
view(p)(T):bool=View(p)(C(p,T))
Agree:bool= FORALL (p,q:Processors),(T:ClockValues):
(correct(p)(T))(correct(q)(T))(joined(p)(T))(joined(q)(T))=>
view(p)(T)=view(q)(T)
Recognition:bool= FORALL (T:ClockValues),(p:Processors):
(correct(p)(T))(joined(p)(T))=>member(p,view(p)(T))
Disjoined(p)(LC):bool
disjoined(p)(T):bool=Disjoined(p)(C(p,T))
Join:bool = FORALL T: Forall (p|member(p,MemAll)): Failure_to_Correct(p)(T)=>joined(p)(T+JoinDelay)
Detect:bool= FORALL T: Forall (p|member(p,MemAll)): Correct_to_Failure(p)(T) => disjoined(p)(T+DetectionDelay)
MemberJoined(T):bool =forall (p|member(p,MemAll)): joined(p)(T)
MemberJoined_Interval(T0,T):bool=during?(MemberJoined,T0,T)
Joined_interval:theorem
forall (p|member(p,MemAll), q|member(q,MemAll)): MemberJoined_Interval(T0,T)
=>during?(joined(p),T0,T) and during?(joined(q),T0,T)
END Mem_Spec
```

At this stage, we have formally specified the required services of basic building-block protocols inherently used in dependable distributed protocols. With regard to reusability of formal theories of these protocol components of group communication services, we emphasize that formal theories of synchronization and broadcast could be utilized to modularly compose and verify membership protocol operations and its desired services. In this thesis, our objective is to show how group membership along with broadcast and synchronization components can be used to compose and verify checkpointing and recovery protocol.

In the next section, we give an overview of a timestamp-based checkpointing protocol, and identify requirements which are being supported by services provided

by basic protocols. Following this, we formally specify the required services of the checkpointing protocol, and its operations, and then, formally verify the correctness of the checkpointing protocol operation.

4.4 Formal Specification and Verification of Checkpointing Protocol

Checkpointing protocols utilizing synchronized clocks would essentially require the building blocks of group communication services, namely synchronization, broadcast and membership. In particular, this case study highlights the fact that, based on basic real-time and fault-tolerant services, distributed algorithms, such as checkpointing, can be implemented from a network of processors with local clocks and a simple communication mechanism, and the correctness of the protocol operations can be established utilizing proof constructs of these basic blocks.

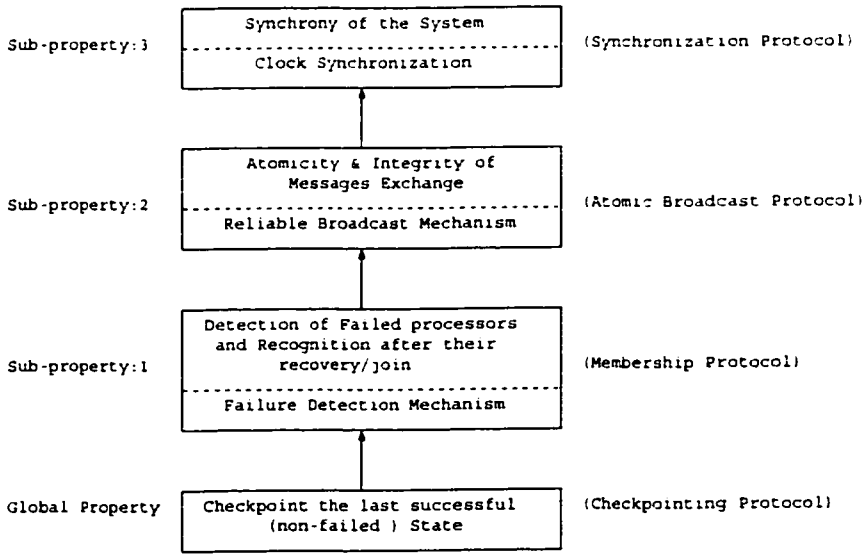

Figure 4.1: Dependencies on Sub-protocol Properties

In figure 4.1, we illustrate the decomposition of the global property into sub-properties supported by various building block. For the main global property of

establishing consistent set of checkpoints to be true, sub-property: 1 must be true, i.e., the checkpoint operation must have an up-to-date information of each group member provided by the group membership protocol, which is inherently being supported by failure detection and join mechanisms. Now in order for sub-property: 1 to be true, reliable broadcast mechanism must support atomicity and integrity of messages being exchanged(i.e., sub-property:2), which can be provided by atomic broadcast protocol. The correctness of sub-property:2 depends on bounded message transmission and synchrony assumptions of the system (sub-property:3) wherein clocks are approximately synchronized to each other within a specified skew.

4.4.1 Formal specifying required Services of Checkpointing Protocol

In order to have a consistent set of checkpoints, records of orphan message and lost message must not be there in a valid checkpoint. We begin with formalizing the notion of "lost-messages" and "orphan-messages."

For any around n and any processor p, all the messages that were sent out before the p's scheduled local checkpoint, must be received by a target processor q before the time $(C(q,T)+BroadcastBound+e)$. In the absence of failures, a message which was sent by p at time T,i.e.,$(S - BroadcastBound - e) < C(p,T) \leq S$ will be delivered to the destination no later than the destination processor q's local time $(S+BroadcastBound+e)$, i.e.,$(C(q,T) + BroadcastDelay) < (S + BroadcastBound + e)$, and no message is being lost. Note that for a specific round of checkpointing operation, S is the scheduled time instant.

```
Checkpointing Services
Ckpt_Serve[Processors:type]: THEORY
 BEGIN
 importing Mem_Spec[Processors]
 T:VAR ClockValues
 m,mark:Var Messages
 Pst:var Proc_state
```

```
  p,q:var Processors
  BroadcastDelay:var NonNegClockValues
No_lost:bool =
FORALL m:
Forall (BroadcastDelay| BroadcastDelay<BroadcastBound):
Forall (p|member(p,MemAll), q|member(q,MemAll)):
Forall (T|S-BroadcastBound-e<C(p,T) and C(p,T)<=S):
MemberCorrect_Interval(S-BroadcastBound-Rho*S-e,S+BroadcastBound+Rho*S+e)=>
(send(p,m,q)(T) and arrive(q,m,p)(T+BroadcastDelay) and In_Order(m) and (abs(C(p,T)-C(q,T))<=e))
=>receive(q,m,p)(T+BroadcastDelay) and C(q,T)+BroadcastDelay<S+BroadcastBound+e
```

All the message that are received from other processors and were logged by the processor p, are those messages that were sent before the sender's current round of local checkpoint. This prevents the possibility of orphan messages. This condition is formally specified as follows:

```
No_orphan:bool=
Forall m:
Forall (p|member(p,MemAll),q|member(q,MemAll)):
Forall (T|S-BroadcastBound-e<C(p,T) and C(p,T)<S+BroadcastBound+e):
MemberCorrect_Interval(S-BroadcastBound-Rho*S-e,S+BroadcastBound+Rho*S+e)=>
Forall (m|receive(p,m,q)(T) and log(p,m)(T)):
m'Km<Pi(p,T)
```

In each round, when a processor finishes its checkpointing at time T which is before or at it's scheduled checkpoint local clock S, it stores its local state and broadcast a "mark" message to all other processors in the group indicating its local success in establishing checkpoint. Once all processors have broadcasted the "mark" message to all other processor in the group, and that all processors have received the "mark" message, then this round of checkpoint operation is successful. And because mark message is also a type of message, it is required to be received in the order with those data messages. When mark message is received, we could know that all the previous data message was received successfully. This makes sure that mark message could be a sign that checkpoint operation is successful. We formalize this notion as follows:

```
CheckPoint_Proceed : bool=
Forall mark:
Forall (BroadcastDelay|BroadcastDelay<BroadcastBound):
```

Forall (p|member(p,MemAll),q|member(q,MemAll)):

Forall (T|S-BroadcastBound-e<C(p,T) and C(p,T)<=S):

MemberCorrect_Interval(S-BroadcastBound-Rho*S-e,S+BroadcastBound+Rho*S+e)=>

(ckpt(p)(T) and broadcast(p,mark,q)(T) and deliver(q,mark,p)(T+BroadcastDelay) and In_Order(mark) and abs(C(p,T)-C(q,T))<=e =>receive_broadcasting(q,mark,p)(T+BroadcastDelay) and C(q,T)+BroadcastDelay < S+BroadcastBound+e)

We describe the description of the procedure of messages recording which covered the requirements of the message receiving and logging in the processors point to point communications. For any processor p and q in the member group, during P's local clock interval $(S\text{-}BroadcastBound\text{-}e,S]$, in the absence of the omission failure, and no out of order happens, any message m which were send by p to q will be received by q before q's local clock $S + BroadcastBound + e$. and the message m will be logged by q if and only if the round number in the message m is less than the round number in the processor q's state. So that, all the messages which were logged were those messages which were sent before the sender's current round checkpointing.

Message Recording

Messages_Recording:bool=

Forall m:

Forall (BroadcastDelay|BroadcastDelay<BroadcastBound):

Forall (p|member(p,MemAll),q|member(q,MemAll)):

Forall (T|S-BroadcastBound-e<C(p,T) and C(p,T)<=S):

MemberCorrect_Interval(S-BroadcastBound-Rho*S-e,S+BroadcastBound+Rho*S+e)=>

(send(p,m,q)(T) and arrive(q,m,p)(T+BroadcastDelay) and In_Order(m) and abs(C(p,T)-C(q,T))<=e

=>receive(q,m,p)(T+BroadcastDelay) and C(q,T)+BroadcastDelay<S+BroadcastBound+e)

and

FORALL (T|S-BroadcastBound-e<C(q,T) and C(q,T)<S+BroadcastBound+e):

FORALL (m|log(q,m)(T) and receive(q,m,p)(T)):m'Km<Pi(q,T)

END Ckpt_Serv

4.4.2 Formally Specifying Protocol Operations

In this section, we first formally specify the protocol operation where all cooperating processes checkpoint periodically, each with the same period, Pe. Below, we list the definitions needed for the protocol specifications.

We begin with formalizing the initial state of the processor. When the system is started, all processors are "correct" and "joined"(From membership). At this time, for any processor p, the processor state Pst at time $T = 0$ has the checkpoint round number 0. The checkpointing operations starts at time 0.

```
Checkpointing Protocol
Ckpt_Ptcl[Processors:type]: THEORY
 BEGIN
 importing Mem_Spec[Processors]
 T:var ClockValues
 BroadcastDelay:var NonNegClockValues
 m,mark:var Messages
 p,q :var Processors
 Pst:var Proc_state
Initiation: bool=
FORALL Pst:
FORALL (p|member(p,MemAll)):
MemberCorrect(0) and MemberJoined(0) and Pst'LC=0 and Pst'n=0
% The checkpoint time S equal the period Pe times the round number n.
Checkpoint: bool= S=n*Pe
```

For any processor p, the Checkpointing interval for round # n is presented in terms of its local clock values by (S-BroadcastBound-e, S+BroadcastBound-e]. It is to note that forall the processors, their checkpointing interval is bounded at the real time interval (S-Rho*S-BroadcastBound-e,S+Rho*S+BroadcastBound-e]. To proof the protocol, we assume during the checkpoint interval, no processor failure happens. Communication failures are allowed if and only if the failure could be recovered by the communication protocol within the checkpoint period before the mark message was received by any of all the group members For those conditions of failure happens, we specify them in the specification of Recovery theory

Arrive: A message is sent by processor p in the time T which C(p,T) is inside the period of p's local clock (S-BroadcastBound-e,S]. The message will arrive the destination in the destination processor's local time C(q,T)+BroadcastDelay.

```
Arrive : bool=
FORALL m,BroadcastDelay:
Forall (p|member(p,MemAll), q|member(q,MemAll)):
Forall (T|S-BroadcastBound-e<C(p,T) and C(p,T)<=S):
```

```
MemberCorrect_Interval(S-BroadcastBound-Rho*S-e,S+BroadcastBound+Rho*S+e)=>
(send(p,m,q)(T) => arrive(q,m,p)(T+BroadcastDelay))
```

Receive: A message m which was sent by processor p at T which $C(p,T)$ is inside its local clock interval $(S-BroadcastBound-e, S]$. The message will arrive the destination before the destination processor's local time $S+BroadcastBound+e$. If no communication failure happens and the transmission delay ($BroadcastDelay$) is less than the bounded delay ($BroadcastBound$), and if the messages are in order, then m will be Received by q.

```
Receive : bool=
FORALL m,BroadcastDelay:
Forall (p|member(p,MemAll), q|member(q,MemAll)):
Forall (T|S-BroadcastBound-e<C(p,T) and C(p,T)<=S):
MemberCorrect_Interval(S-BroadcastBound-Rho*S-e,S+BroadcastBound+Rho*S+e)=>
(arrive(q,m,p)(T+BroadcastDelay) and BroadcastDelay<BroadcastBound and In_Order(m)
=>receive(q,m,p)(T+BroadcastDelay))
```

If the round # in the massage m which was received by p at T is same with the local checkpoint round number of q in $Pi(q,T)$, then this message will be logged by q.

```
Logging:bool=
Forall m:
Forall (p|member(p,MemAll), q|member(q,MemAll)):
Forall (T|S-BroadcastBound-e<C(p,T) and C(p,T)<S+BroadcastBound+e):
MemberCorrect_Interval(S-BroadcastBound-Rho*S-e,S+BroadcastBound+Rho*S+e)=>
(log(p,m)(T) iff receive(p,m,q)(T) and m'Km<Pi(p,T))
```

When processor p received message m before it made its current round checkpoint, and the m was sent by sender after the sender has done its checkpoint of the current round, the processor p will make checkpoint right at the time of the message received. Otherwise, the checkpoint time will be local clock value of p at S.

```
CheckPoint:bool=
Forall (p|member(p,MemAll)):
Forall (T|S-BroadcastBound-e<C(p,T) and C(p,T)<=S):
MemberCorrect_Interval(S-BroadcastBound-Rho*S-e,S+BroadcastBound+Rho*S+e)=>
IF
   (Exists m: log(p,m)(T) and C(p,T)<S and m'Km>n)
Then
   ckpt(p)(T) and store(p)(T) and Pi(p,T)=n+1
```

```
else
   Ckpt(p)(S) and Store(p)(S) and PI(p,S)=n+1
Endif
```

When a processor finished establishing its checkpoint, it will broadcast the "mark" message to all of the other processors in the member group.

```
MARK: bool=
FORALL mark,BroadcastDelay:
FORALL (p|member(p,MemAll),q|member(q,MemAll)):
FORALL (T|S-BroadcastBound-e<C(p,T) and C(p,T)<=S):
MemberCorrect_Interval(S-BroadcastBound-Rho*S-e,S+BroadcastBound+Rho*S+e)=>
(ckpt(p)(T)=> broadcast(p,mark,q)(T))
END Ckpt_Ptcl
```

At this stage, we have formally specified the required service of the checkpointing protocol, and the protocol operations. We have also discussed formalization of services provided by basic group communication protocols. Based on these, the next objective is to establish the correctness of the checkpointing protocol operation.

4.4.3 Checkpointing Protocol Verification

In this section, we verify the checkpointing protocol by proving that the protocol operations along with supporting services rendered by basic building block protocols lead to the required services to be provided by the checkpointing protocol.

```
Checkpointing Protocol
Ckpt_Prof[Processors:type]: THEORY
BEGIN
importing Ckpt_Serv[Processors]
importing Ckpt_Ptcl[Processors]
No_lost_prf: lemma
Arrive and Receive => No_lost
No_orphen_prf : lemma
Logging =>No_orphan
CheckPoint_Proceed_prf:lemma
CheckPoint and Validity and BroadcastTermination and MARK=>CheckPoint_Proceed
```

The global checkpoint is successful if during the checkpointing time zone there exists and only exists one time instant T, that is the checkpoint time. At this moment, the processor who just completed the local checkpointing will broadcast

mark message to all other processors in the group membership. If each processor finishes this step, then the current round checkpointing successed. Also during the checkpoint time zone, whenever a message comes,the processor would check its time stamp and determine if it should be received or logged. This procedure is running at all the time.

```
Global Success
Global_Success: Theorem
if
Checkpoint_done
then
CheckPoint and MARK and Validity and BroadcastTermination =>CheckPoint_Proceed
else
Logging and Arrive and Receive =>Messages_Recording
endif
END Ckpt_Prol
```

4.4.4 Dealing with the Failure Recovery

Following we are going to address recovery from a communication failure and from a process(or processor) crash detected during the checkpointing interval (critical interval). It is important to note that a group membership service exists which provides for the detection of a process failure and orderly rejoining of a process after recovery. We begin with describing the communication failure detected in the critical interval.

```
Recovery[Processors:type]: THEORY
 BEGIN
 importing Mem_Spec[Processors]
 T,T1,T2:VAR ClockValues
 n:var Index
 m,mark:var Messages
 p,q:var Processors
 Pst:var Proc_state
 BroadcastDelay:var NonNegClockValues
 Block_MemAll(T):bool
 Comm_Protocols_Memall(T):bool
```

When both sender and receiver are correct, the sender p send message m to the receiver q but q never get it, then the message m is missing and omission failure happens.

```
Omission_Failure : bool=
FORALL m,BroadcastDelay,T:
Forall (p|member(p,MemAll), q|member(q,MemAll)):
MemberCorrect_Interval(T,T+BroadcastBound) and send(p,m,q)(T)
=>not Arrive(q,m,p)(T+BroadcastDelay)
```

When both sender and receiver are correct, the sender p send message m to the receiver q, q get the message m but the time is over the maximum delay bound. then the time-out failure happens.

```
BoundedDelayComm_Failure : bool=
FORALL m,BroadcastDelay,T:
Forall (p|member(p,MemAll), q|member(q,MemAll)):
MemberCorrect_Interval(T,T+BroadcastBound) and send(p,m,q)(T)
=>Arrive(q,m,p)(T+BroadcastDelay) and BroadcastDelay>BroadcastBound
```

When both sender and receiver are correct, the sender p send message m to the receiver q, q get the message m within the maximum delay bound but m is out of order, then failure happens.

```
Out_of_Order_Failure:bool=
FORALL m,BroadcastDelay,T:
Forall (p|member(p,MemAll), q|member(q,MemAll)):
MemberCorrect_Interval(T,T+BroadcastBound) and send(p,m,q)(T)
=>Arrive(q,m,p)(T+BroadcastDelay) and BroadcastDelay<BroadcastBound and not In_Order(m)
```

If all above failures happens. the solution basically is to recall the communication protocol. and under the communication protocol to do the retransmission. This depends on the communication protocols. Things need to be mentioned here are when communication failure happens, the group processors will no block there. Instead of that, the communication protocol start to running for fixing the problem.

```
Communication_Failure_Recovery: bool=
Forall T: Omission_Failure or BoundedDelayComm_Failure or Out_of_Order_Failure => Comm_Protocols_Memall(T)
```

Basically a communication failure could cause a group checkpoint rollback when the mark message failure could not be fixed till the end of the local checkpoint interval. If this happens, the group processors will rollback and recovery from the

previous checkpoint round # n-1.

```
Communication_Failure_Caused_Rollback : bool=
FORALL BroadcastDelay:
Forall (p|member(p,MemAll), q|member(q,MemAll)):
Forall (T|S-BroadcastBound-e<C(p,T) and C(p,T)<=S):
Exists mark:
MemberCorrect_Interval(S-BroadcastBound-Rho*S-e,S+BroadcastBound+Rho*S+e)
=>((not receive_broadcasting(q,mark,p)(T+BroadcastDelay))
=>RollBack_Memall(n-1)(RC(q,S+BroadcastBound+e)) and Restore_Memall(n-1)(RC(q,S+BroadcastBound+e)))
```

Failure_Block : Whenever a processor failure at any time T, all the processors would be blocked at T by Block_MemAll(n)(T) And the failure processor p, would be treated by membership protocol as Disjoined at time T+DetectionDelay

```
Failure_Block: bool=
FORALL m,BroadcastDelay,T:
Exists (p|member(p,MemAll)):
Correct_to_Failure(p)(T) and Pi(p,T)=n
=>Block_MemAll(n)(T) and Disjoined(p)(T+DetectionDelay)
```

Processor_Correct: When a Processor come back to be correct at time T1, it would be recognized by membership protocol as Joined at time T1+JoinDelay. And if at time T1, all the processors are correct. then the membership protocol would recognize that all processors are Joined (MemberJoined) at T1+JoinDelay.

```
Processor_Correct: bool=
(Exists (p|member(p,MemAll)): Failure_to_Correct(p)(T1))
and MemberCorrect_Interval(T1,T2) and T2>T1+JoinDelay
=> MemberJoined_Interval(T1+JoinDelay, T2)
```

When a failure processor come back to be correct at time T, and at this time, all processors are correct, this implies all processors can be recognized by membership protocol as joined after a JoinDelay, then at time T1+JoinDelay, the system will rollback and recovery from the previous checkpoint round # n-1.

```
Failure_Rollback: bool=
(Exists (p|member(p,MemAll)): Failure_to_Correct(p)(T1))
and MemberCorrect_Interval(T1,T2)
and MemberJoined_Interval(T1+JoinDelay, T2) and T2>T1+JoinDelay
=>RollBack_Memall(n-1)(T1+JoinDelay) and Restore_Memall(n-1)(T1+JoinDelay)
```

The Processor failure recovery: all the processors in a group will restart from the previous checkpoint when all failed processors come back to be correct and are

recognized by a membership protocol as "joined. "

```
Processor_Failure_Recovery :bool=
(Exists (p|member(p,MemAll)):Failure_to_Correct(p)(T1)) and MemberCorrect_Interval(T1,T2) and
T2>T1+JoinDelay
=> RollBack_Memall(n-1)(T1+JoinDelay) and Restore_Memall(n-1)(T1+JoinDelay)
Failure_Recovery: Theorem
Processor_Correct and Failure_Rollback => Processor_Failure_Recovery
End Recovery
```

4.5 Disscussion

The proof of lemmas is straightforward. It is important to point out that the formalization of basic services and verification of the checkpointing protocol is performed within the same time interval, and thus, it ensures the overall correctness of the stack of protocols being used in formulating the checkpointing protocol. Below, we include the status report that is being generated by the PVS tool.

Proof summary for theory_Prof

No_lost_prf.....proved - complete [0](9.31 s)

No_orphan_prf.....proved - complete [0](4.20 s)

CheckPoint_Proceed_prf.....proved - complete [0](40.65 s)

Global_Success.....proved - complete [0](68.03 s)

Theory totals:4 formulars, 4 attempted, 4 succeeded (122.19 s)

Chapter 5

Discussion and Conclusions

By decomposing a complex protocol and reusing the formal specification and verification of individual functional building blocks, the formalization of the overall protocol becomes easier. Our research has been to utilize protocol building blocks for the construction of the compositional specification and verification of a variety of dependable distributed protocols. If a stack of basic components in dependable distributed system, such as the primitives and sub-protocols can be formulated, then these elements can support the systematic and hierarchical development of dependable distributed protocols. By defining priori validating building blocks for dependable distributed protocols, larger and more complex protocols can be easily specified and verified.

In this thesis we have identified and specified basic building blocks which are inherent to provide Real-time and Fault Tolerant services. We first specified the basic primitives, i.e. Timing, Failure, System model and Communication primitive, and subsequently highlighted the services of group communication sub-protocols including clock synchronization, atomic broadcasting and membership protocol. The properties of these sub-protocol building blocks act as the basis of hierarchical protocol stack. In order to obtain the checkpointing protocol (refer to the system model [22] [23]) out of these sub-protocol building blocks, we have identified the

inter-dependencies of these building blocks. It is important to determine what restrictions or requirements one particular block imposed on the other. This part was crucial in the sense that it determined whether there were any conflicts among the building blocks that are composed/configured together. In the checkpointing protocol composition example discussed in the thesis, all building blocks had the same failure model and synchrony assumptions. For this particular case, the inter-block interactions were relatively straightforward.

Our aim in compositional verification of dependable protocols is to illustrate the fact that a global property of the protocol operation can be decomposed into elementary sub-properties that are provable in more basic protocols , i.e., at the sub-protocol level, and then map these properties based on their functionalities to obtain the desired global property. The compositional verification allows the complex reasoning of how an overall system can be reduced to a much simpler reasoning of a collection of components. The building blocks with parameterized theories could be easily instantiated for developing the protocol under consideration, and for that reason we use the formal tool PVS which facilitates modular composition. The built-in typechecker in PVS can flag inconsistencies in type definitions appearing in different theories, or instantiations of imported theories of building blocks, and thus can detect simple errors at an early stage.

Two approaches are introduced in my thesis namely the assertional approach and category theory approach. Assertional approach allows a building block directly import specifications of other building blocks. The composite block can directly use parameters and properties included from the basic primitives and building blocks in lower level. To verify the composite protocol, required services and protocol operations are needed to be specified, and we have to prove that the protocol operations can lead to the required services by using proof rules. In this thesis, the disjoint parallel program proof rule has been used. The formal specification and verification was performed using PVS. The category theory approach allows via shared parameters

defined in import morphism to connect two building blocks. Note that only shared parameters can be included in the morphism for the consequent colimit calculation. Once the diagram is figured out, we can obtain the composite protocol specification by colimit calculation. Comparing the above two approaches, the complexity of assertional approach is to specify the services and operations of protocol as well selecting a proof rule, whereas the category theory approach is complex at figuring out the shared parameters and logical relations among those building blocks. The specification composition process was supported by Specware.

Obviously the advantage of compositional approach is that it allows the specification and verification of a complex system to be decomposed into the specification and verification of simpler components. Especially, the complex reasoning about an overall system can be reduced to simpler reasoning about a collection of components. Also based on specification composition, the reusable system components can be defined.

Through category theory framework, the properties/operations provided by each component can be specified in the standard manner, and the framework can be used to combine the individual properties/operations into a compositional specification. It is convenient to write specifications because the required composition can be obtained from the colimit calculation of the diagram which is formed from the sequence of component specifications. This supports the combination of basic building blocks (sub-protocols) together for the composition of complex systems (protocols) which is on the top of the hierarchical protocol stack. The benefit is that most of the reasoning about a composite system can be reduced to reasoning about individual components.

Nevertheless we note that the components must be proven to be appropriate for composition before the actual composition. In this thesis, we import the system model and identify the basic reusable building blocks from [22] [23]. Also the framework requires to clearly declare the relationships, the allowed translations and

morphisms between individual components.

On the other hand, the difficulty of using the modular specification approach rather than specifying the example as a monolithic entity is that it is necessary to specify how the individual components interact with each other and to specify the shared parameters and functions of the components. However, this is actually a trade-off between the individualistic specification and the compositional specification. By explicitly modeling the interaction between the components, the correspondence between the model and the actual system is more obvious. It is also important to note that the modular specification approach has advantages from a maintenance standpoint. Although more experience is required using the framework, we believe that the benefits resulting from the modular approach are worthy when compared to the increased complexities resulting from specifying the interaction between components.

As a future research work, it is suggested to have a generalized theory outlining procedures for compositional specification and verification covering a wide spectrum of dependable distributed protocols. This would enhance the effectiveness of our proposed compositional approach based on category theory concepts.

Bibliography

[1] LNCS 1427, pp. 521-525, Springer-Verlag, 1998.

[2] R. Alur, T.A. Henzinger, F.Y.C. Mang, S. Qadeer, S.K. Rajamani, S. Tasiran, "MOCHA: Modularity in Model Checking." Proceedings of the Tenth International Conference on Computer-aided Verification (CAV '98),

[3] [2] A. Arora, M.G. Gouda, "Closure and Convergence: A Foundation of Fault Tolerant Computing." IEEE Trans. on SoftwareEngineering, 19(10), pp. 1015-1027, Oct. 1993.

[4] M. Barborak, M. Malek, A. Dahbura, "The Consensus Problem in Fault-Tolerant Computing." ACM Computing Surveys, vol. 25, no. 2, pp. 171-220, June, 1993.

[5] M. Barr, C. Wells, *Category Theory for Computing Science*, Second Edition, Prentice Hall, 1990.

[6] W.C.Carter, "A time for Reflection." Proc. FTCS 12th Ann. Int'l Symp. Fault-Tolerant Computing, IEEE Computer Soc. Press, Los Alamitos, Calif., 1982, p.41

[7] F. Cristian, F. Jahanian, "A Timestamp-Based Checkpointing Protocol for Long-Lived Distributed Computations," *Proc. of Reliable Dist. Soft. & Database Systems*, pp. 12-20, 1991.

[8] F. Cristian, "Understanding Fault-Tolerant Distributed Systems." *Comm. of the ACM*, 34(2), pp. 57–78, Feb. 1991.

[9] R. De Prisco, et al., "Building Blocks for High Performance and Fault-Tolerant Distributed Systems." Details Available at http://www.lcs.mit.edu/research/projects, 1999.

[10] Marielle Doche, Christel Segun, Virginie Wiels, "A modular Approach to Specify and Test an Electrical Flight Control System " Fourth International Workshop on Formal Methods for Industrial Critical Systems, Trento Italy, July 1999.

[11] J.L. Fiadeiro, T. Maibaum, "Temporal Theories as Modularisation Units for Concurrent System Specification," *Formal Aspects of Computing*, 4(3), pp. 239–272, 1992.

[12] R. Guerraoui, A. Schiper, "Consensus Service: A Modular Approach for Building Agreement Protocols in Distributed Systems," *Proc. of Fault-Tolerant Computing Symposium*, pp. 168-177, June 1996.

[13] Y. Gurevich, R. Mani, "Group Membership Protocol: Specification and Verification." *Specification and Validation Methods*, pp. 295–328, Oxford University Press, 1995.

[14] T. Henzinger, S. Qadeer, S.K. Rajamani, "You Assume, We Guarantee: Methodology and Case Studies." Proc. of CAV'98 June/July 1998.

[15] M.A. Hiltunen, R.D. Schlichting, "An Approach to Constructing Modular Fault-Tolerant Protocols." Proc. of the 12th IEEE Symposium on Reliable Distributed System, pp. 105-114, Oct. 1993.

[16] X. Liu, C. Kreitz, R. van Renesse, J. Hickey, M. Hayden, K. Birman, R. Constable, "Building Reliable, High-Performance Communication Systems from Components." Operating Systems Review, 34(5), pp. 80-92, Dec. 1999.

[17] P.R. Lorczak, A.K. Caglaya, D.E. Eckhardt, "A Theoretical Investigation of Generalized Voters for Redundant Systems." Proc. of FTCS-19, pp. 444-451, 1989.

[18] P. Michel, V. Wiels, "A Framework for Modular Formal Specification and Verification." Proc. of FME'97, 1997.

[19] P. Ramanathan, K.G. Shin, "Use of Common Time Base for Checkpointing and Rollback Recovery in a Distributed System." IEEE Trans. on Software Engineering, 19(6), pp. 571-583, June 1993.

[20] J. Rushby, "Systematic Formal Verification for Fault-Tolerant Time-Triggered Algorithms," *IEEE Trans. on Software Engineering*, 25(5), pp. 651-660, Sept./Oct. 1999.

[21] G. Singh, I. Buricea, Z. Mao, "Composition of Service Specifications," *IEEE Int. Conf. on Network Protocols*, Oct 1998.

[22] P. Sinha, N. Suri, "Modular Composition of Redundancy Management Protocols in Distributed Systems: An Outlook on Simplifying Protocol Level Formal Specification and Verification." *Proc. of Int. Conf. on Distributed Computing Systems-21*, pp. 255-263, 2001.

[23] P. Sinha, N. Suri, "On Simplifying Modular Specification and Verification of Distributed Protocols." *Proc. of High Assurance Systems Engineering-6*, pp. 173-181, 2001.

[24] P. Sinha, N. Suri, "Identification of Test Cases Using a Formal Approach." Proc. of FTCS-29, pp. 314-321, 1999.

[25] P. Sinha, N. Suri, "On the Use of Formal Techniques for Analyzing Dependable Real-time Protocols." Proc. of RTSS-20, pp. 126-135, 1999.

[26] N. Suri, M. Hugue, C. Walter, "Synchronization Issues in Real-Time Systems," Proc. of IEEE, 82(1), pp. 41-54, 1994.

[27] N. Suri, P. Sinha, "On the Use of Formal Techniques for Validation." Proc. of FTCS-28, pp. 390-399, 1998.

[28] Y. V. Srinivas and Richard Jullig, "Specware(TM): Formal Support for Composing Software," *Proc. of the Conference on Mathematics of Program Construction*, Germany, July 1995.

[29] Amjad Umar " Distributed Computing A practical synthesis," Prentice Hall 1993

[30] R. van Renesse, K. Birman, S. Maffeis, "Horus: A Flexible Group Communication System." Communication of the ACM, 39(4), pp. 76-83, April 1996.

[31] P. Verissimo, C. Almeida, "Quasi-synchronism: A Step Away from the Traditional Fault-Tolerant Real-Time System Models." IEEE Bulletin of the TCOS, 7(4), pp. 35-39, Winter 1995.

[32] C. Walter, P. Lincoln, N. Suri, "Formally Verified On-Line Diagnosis." IEEE Trans. on Software Engineering, SE 23(11), pp. 684-721, Nov. 1997.

[33] P. Zhou, J. Hooman, "Formal Specification and Compositional Verification of an Atomic Broadcast Protocol." *Real-Time Systems*, 9(2), pp. 119–145, 1995.