MODULAR COMPOSITION AND VERIFICATION OF TRANSACTION PROCESSING PROTOCOLS USING CATEGORY THEORY

Vasudevan Janarthanan

A Thesis

In

The Department

Of

Electrical & Computer Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Applied Science
in Electrical & Computer Engineering
at Concordia University
Montréal, Québec, Canada

March 2003

0-612-77971-8

Abstract

Modular Composition and Verification of Transaction Processing Protocols Using Category Theory

Vasudevan Janarthanan

Establishing the correctness of reliable distributed protocols supporting dependable applications necessitates modular/compositional approaches to tackle the inherent complexity of these protocols. Efforts involved in the specification and verification of these reliable distributed protocols can be considerably reduced if the protocol is composed utilizing smaller components (building-blocks) possessing individual functionalities that are integral parts of the overall protocol operation. In this thesis, we introduce techniques utilizing the concepts of category theory for the modular composition of dependable distributed protocols. In particular, we show how by defining external interfaces of basic modules, and morphisms linking two different modules, a larger or more complex protocol can be formally composed and verified. To illustrate the effectiveness of the proposed methodology for compositional specification and verification, in this thesis, we present a modular composition and verification of a transaction processing protocol namely the non-blocking atomic three phase commit (3PC) protocol using category theoretic concepts. Specifically, we illustrate how the overall global properties of the protocol can be proved by utilizing constructs of local sub-properties of the inherent building blocks of the 3PC protocol. A key benefit of this modular approach is that these identified building blocks would be helpful to system designers for their capability of specifying and facilitating rigorously tested and pretested formal theory modules of required system and component behavior, and also supporting system design decisions and modifications.

Acknowledgments

It has been a great privilege for me to work with Dr.Purnendu Sinha. an exceptional researcher and teacher, who introduced me to real-time systems, distributed computing, modular composition techniques and component-based design. He has been extraordinarily patient and supportive, having been always available for discussion and responding speedily to research reports. I would like to take this opportunity to thank him for his continued encouragement and guidance throughout the course of my research.

My sincere thanks goes to Professor Michael Barr of McGill University, Montreal, Canada, for his valuable insights on Category Theory. Given his high profile stature in the field of Category Theory, I feel highly honoured to mention his name in my thesis.

I also thank Professor James McDonald and Mr.David Cyrluk of Kestrel Institute, California, for their timely inputs about the *SPECWARE* tool. Their email responses had been pretty helpful in the successful completion of this work.

This research has been a part of the NSERC, FCAR and FRDP grants awarded to Dr.Purnendu Sinha. The project provided an excellent environment for acquiring valuable academic and practical experience in distributed computing, modular composition techniques and component-based design.

I extend my whole-hearted thanks to my parents for their encouragement, help and constant drive throughout the entire period of the thesis. Finally I am grateful to all my fellow colleagues in the research group for their valuable discussions during this research.

Vasudevan Janarthanan, March 2003.

I dedicate this work to my loving Dad and Mom ....

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter addresses the issues related to dependable distributed protocols. We begin by highlighting the growing importance of such protocols in distributed computing and their inherent properties. We then look into the difficulties involved in the design and analysis of these protocols, followed by the issue of application of formal methods in solving some of these difficulties, and also highlight some of its inherent limitations. After briefly discussing these issues, we next propose the idea of modularization in order to simplify the specification and verification processes of these protocols and the concept of category theory as a means to compose the modularized components.

1.1 Dependable Distributed System and Protocols

In recent years, there has been a considerable interest in the design and analysis of distributed systems wherein users interact with processors that are networked together with many other processors. A system that consists of a collection of two or more independent computers which co-ordinate their processing through the exchange of messages is called as a distributed system. Distributed systems are now commonplace like electronic mail systems, distributed databases, communication networks, office systems, banking systems and real-time control systems. One of the main reasons for the increasing importance of distributed computing system is its ability to provide dependable services. Distributed systems [6] tend to be more tolerant to faults than traditional architectures: a distributed system may be constructed so that it continues functioning despite the failure of some of its components.

Distributed systems are built of many processing units with a combination of non-volatile and volatile memories, and are subject to various kinds of failures. Processors can crash, losing the contents of their volatile memories, can proceed at erratic speeds, or can simply stop without warning. The communication media in distributed systems are also subject to failures, resulting in loss, reordering and duplication of messages. Distributed systems are complex in the sense that they have a large number of components, many activities running concurrently, and components might even fail independently. To help cope with this complexity, a variety of protocols have been formalized, ranging from protocols for communication among the processors to their recovery in times of faults. Distributed systems offering fault-tolerant features usually utilize the services of a group of protocols each having its own inherent properties and operational activities. Protocols are the set of rules that software components on different machines use to realize a given abstraction. They work in tandem with different functionalities, ultimately helping in realizing the global functionality of the system as a whole [5, 7, 30, 33].

Distributed systems with dependability qualities have to basically satisfy the reliability and availability properties in order to assure the fault-tolerant features. Reliability is the probability of a system functioning correctly over a given period of time, whereas availability is the probability of a system functioning correctly at any given time. The most difficult part of any distributed system is to organize the computations so that a correct result is found. Achieving reliability in a distributed system is not an easy task. Communications between nodes and their operations must be properly related. The main concerns with reliability are the effective coordination of the various computational nodes and the message content, and proper organization of data with the degree of replication. This requires a reliable network, as well as reliable protocols for communication among the various nodes, diagnose and recover faults in the system, help in reaching an uniform agreement (decision) among the various nodes and also terminate the process when need arises. A major strength of the dependability concept is its integrative nature, which enables to put into perspective the more classical notions of reliability, availability, safety, security, maintainability, which are then seen as attributes of dependability. We define these terms in the next section.

1.1.1 Inherent Properties and Features of Dependable Distributed Protocols

Any dependable distributed protocol would have in it some inherent properties symbolizing its overall functionalities. These properties ultimately form the services offered by the protocol as a whole. Some of them are:

- *Reliability*: Ensures that the protocol is operational at a particular time instant, i.e., continuity of correct service.
- *Availability*: Ability of the protocol to deliver its service at any given instant of time.
- *Safety*: Safety property assures that something bad will not happen during the protocol execution.
- `Atomicity`: Atomicity property ensures that either all the operations are performed successfully or none of them are performed.
- *Integrity*: Prevention of unauthorized modification or the deletion or destruction of system assets.
- *Maintainability*: Ability of the protocol to undergo modifications and refinements.

- *Stability*: Stability implies that the protocol, in response to a transient fault, converges to a legitimate state in finite time.
- *Liveness*: This property assures that eventually something good must happen during the protocol execution.
- *Serializability*: This property ensures that the effect of executing a collection of atomic actions is equivalent to some serial schedule in which actions are executed one after another.
- *Recoverability*: The recoverability property makes sure that the external effect of an atomic action is all or nothing, meaning that either all the state modifications performed by the atomic action take place or none of them.
- *Synchrony*: This property helps the protocol to perform its intended function within a finite and known time bound.
- *Scalability*: A distributed system is described as scalable if it remains effective when there is a significant increase in the number of resources and the number of users.
- *Transparency*: It is defined as the concealment from the user and the application programmer of the separation of components in a distributed system, so that the system is perceived as a whole rather than as a collection of independent components.

Having defined the properties of dependable distributed protocols, we will next discuss some of the issues in the design and analysis of such protocols.

1.1.2 Issues in design and analysis of Dependable Distributed Protocols

Dependable distributed protocols are characterized by their fault tolerant capabilities (i.e.) their ability to function normally in spite of failures in the distributed system. This is a major issue during the design of dependable protocols like transaction processing, where the protocols must be designed to ensure consistent site information in spite of failures in the distributed system. Failures in a distributed environment could be of different kinds. When components are executing on several different machines, and are connected by computer networks, there could be network related problems like delays or link failures or site crashes or even message losses. Therefore determining the tolerance of dependable distributed protocols to failures in the component is crucial for maintaining the dependability feature of such protocols. Hence protocols designed for such dependable distributed systems, have to incorporate a methodology to assess the fault tolerance of the system to failures occurring in its components.

Transaction processing is the software technology that makes distributed computing reliable. Protocols utilized for transaction processing need to satisfy atomicity and non-blocking properties. The three phase commit protocol, which we have considered in this thesis as a case study is a kind of transaction processing protocol resilient to site failures, i.e., it does not block in the event of site failures. Modularity allows for the decomposition of a large protocol into smaller components or sub-blocks each having its own functionalities and properties. thereby helping in the reusability of these decomposed sub- blocks (modules). Applying modular composition techniques for decomposing the global 3PC protocol into sub-protocols (sub-blocks) results in the protocol utilizing these decomposed sub-protocols (sub-blocks) in order to satisfy the various properties of the protocol. henceforth making it simple to understand, program and prove. However, the design and analysis of most transaction processing protocols implicitly assume services rendered by these sub-protocols without much regard to their interactions and their ability to provide a cohesive and consistent service collectively. Conventional approaches to testing protocols are based on a static paradigm of specifying the precise set of inputs and outputs of each test. Due to unobvious subtleties in sub-protocol interactions and resulting large operational state space, conventional simulation and prototyping techniques are limited in assuring the correctness of the protocol level operations. Identifying dependencies among sub-protocols is just one facet of the correct composition of a given generic protocol. Another facet is to ensure that these decomposed sub-protocols are compatible, i.e, they are based on the same set of assumptions.

Another fundamental issue in distributed computing is the problem of synchronization among the various processors (sites) in the distributed system. Synchronization is the regulation of the evolution of concurrent processors, and subsequently of the occurrence of observable events, as a function of the history of events in the system of processors [4]. Synchronization is required in order to provide concurrency and co-operation among the various processors in the distributed system.

- *Concurrency*: Individual processors, even those that belong to independent applications, may need to enter a state of concurrency either to share access to system resources or to control access to shared data.
- *Co-operation*: Processes that belong to the same application can co-operate for proper execution of the application.

Hence while designing protocols for a fault-tolerant distributed environment, issues like synchronization, concurrency and co-operation should be taken into account. Now that we have illustrated the issues in the design and analysis of dependable distributed protocols, we will next enumerate the difficulties in realizing those issues.

1.1.3 General Difficulties

Dependable distributed protocols are extremely difficult to design and implement because of the unpredictability of message transfer delays, process speeds and of failures. Hence system designers have to cope with these difficulties intrinsically associated with the dependable protocols in the distributed system. They are also hard to test in systems, since the deterministic testing requires considerable extra communication among the distributed workload generation and fault-injection agents for coordination purposes. This extra communication and the additional demands on real-time schedulers result in perturbations to the natural behavior of the system, whose observation becomes infeasible.

Failure handling is a crucial aspect while designing protocols for distributed systems because of the impossibility of reaching an agreement among the various components in the presence of an arbitrary (Byzantine) failure. Concurrency, which helps in maintaining consistent scheduling of concurrent threads and also helps in preserving their dependencies, is a tough scenario, and so the protocols designed should take into consideration the issue of concurrency among the various components in the distributed system. There is also a possibility of occurrence of deadlock and livelock situations in distributed systems because of the large magnitude of components in such systems. The protocols should be designed in such a way that it avoids the occurrence of both these situations in the distributed system.

Another problem which is of fundamental importance in reliable distributed computing while designing dependable distributed protocols is the issue of consensus among the various components of the distributed systems. Arriving at a common decision among the nodes in a distributed system requires the assistance of a number of effective algorithms which at times could prove to be a costly and serious issue. In transaction processing protocols, a standard set of ACID (Atomicity, Consistency, Isolation and Durability) properties (See chapter 3) must be ensured by a combination of concurrency control and recovery protocols. It has been argued that in the existing literature, these protocols are often studied in isolation, making strong assumptions about each other. The problem of combining them in a formal way is largely ignored.

In the following section, we project the role of formal methods in solving some of the above mentioned difficulties.

1.1.4 Role of formal methods in alleviating these difficulties

Developing protocols that are targeted towards specific failure scenarios and their handling mechanisms requires precise and accurate descriptions of specifications, designs and implementations. Formal specification and verification have long been recognized as giving the highest degree of assurance [19, 20, 23]. It is widely acknowledged that the adoption of formal methods, facilitates the writing of clear, unambiguous, complete specifications, and makes it possible to provide an automatic support to a variety of validation and verification activities. Specifically, formal specifications, being unambiguous mathematical objects, can be employed for generating execution sequences, which constitute an useful starting point for validation and verification, in the form of testing. A formal semantic model can also help programmers to reason carefully about the correctness of implementations of the distributed protocols, because they increase the clarity of requirements, identify hidden assumptions that the system must operate on, and certify the consistency of requirements and the correctness of designs among other benefit [9, 24, 29, 44].

In order to check the correctness of formal specification, two kinds of verification techniques are employed in formal methodology:

1) *Theorem Provers*: The theorem provers are based on logic-based specification

languages and provide support to the proof of correctness properties, expressed as logical formulas. The proofs of correctness offered by theorem provers are an important capability in the formal analysis of dependable distributed protocols. These theorem provers help in finding the unobvious mistakes in the formal specification and thereby giving the developers and designers a quick feedback about the various problems in the design. There are two kinds of theorem provers, namely the one which requires user interactions (proof strategy) at each step of the verification, and second which carries out most of the basic proofs without much user interactions, sometimes requiring proof strategies for some unobvious conjectures.

2)*Model Checkers*: A model checking specification consists of two parts. One part is the model: a state machine defined in terms of variables, initial values for the variables, and a description of the conditions under which variables may change value. The state space is defined by the possible combinations of values for the variables. The other part is temporal logic constraints over states and execution paths. Conceptually, a model checker visits all reachable states and verifies that the temporal logic properties are satisfied over each possible path, that is, the model checker determines if the state machine is a model for the temporal logic formula.

We next illustrate the limitations of using formal methodologies in the specification and verification of dependable distributed protocols.

1.1.5 Limitations of formal methods

In literature, numerous dependable distributed protocols have been subjected to specification and verification utilizing formal techniques and tools. But over our studies in applying formal methods for specification and verification, we have observed that formal analysis typically requires intensive effort for both specifying and verifying each specific protocol. Further, the emphasis has been on the specification and verification of that specific protocol itself without giving much regard to its concurrent or future use in the overall system's operation. Hence, much of formal specifications and proof constructs cannot easily be reused 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 protocols. Model Checking is a formal verification technique in which the entire reachable state-space of the system is enumerated and checked for irregular behaviors. But this verification methodology cannot be applied for verifying the correctness of models in a distributed system because of the large system size and the non-deterministic nature of the state reachability analysis of the system. Also it is believed that software development using formal methods is a process of successive refinements from abstract specifications into concrete specifications. Refinement rules may be used to demonstrate that the concrete specifications satisfy the corresponding abstract ones. However, there are serious limitations of the refinement rules in both theory and in practical applications. The limitations include

- that the refinement rules are not sufficient to guarantee that a refined specification (concrete specification) satisfies the user's actual requirements if it satisfies the abstract specification.
- that the existing refinement rules are not always applicable in theory during the successive refinements.
- that the refinement rules are difficult to be applied effectively in practice due to uncertainties and resource constraints.

In the next section, we discuss the necessity of modularization techniques in simplifying the analysis of a complex protocol.

1.1.6 Need for Modularization

A good structure is important in large specifications for interpreting, testing and managing changes. Furthermore, a well-chosen structure greatly facilitates understanding, modification and validation of a specification. Ideally, a specification structure allows one to isolate changes within a small number of components of a specification, and to reason about the impacts of a change on interconnected components. In a large project, it is desirable to be able to mix specification and composition steps such that at any particular moment in the process, we may have established only some of the properties of the components, and some of the composition relations.

Modularization is a well-known technique for simplifying complex communication systems [31]. This technique allows the decomposition of a large block or protocol into smaller components or sub-blocks each having its own functionalities and properties [16]. A component or sub-block is a body of code or specification that is deliverable, independently deployable, and ready for integration in larger protocols. In an ideal component world, a pool of interchangeable items is created, and protocols are built by choosing components from several pools, adapting them, and connecting them as desired [43].

The important aspect of such component-based protocol design and analysis is the need to understand properties of the global protocol, their individual components, and how they interact [1, 8, 10, 13]. The key requirement for the development of robust, maintainable, and composable component modules is a mechanism for specifying modules' invariants and abstractions, and for ensuring that those invariants and abstractions are respected [18, 26]. Moreover, the stronger the invariants and abstractions that can be specified and enforced, the more robust the resulting protocol design would be. We will next see some of the work that is related to our topic of study.

1.1.7 Related Work

In [33], the author introduces the various problems in attaining an atomic non-blocking commit protocol and derives a solution through a building-block approach. Instantiations of the proposed $NBAC$ protocol [33] use timeout mechanisms, reliable multicast primitives and unreliable failure detectors as basic components, and follow the modular approach introduced in [2].

In [16], the authors have advocated the idea that consensus can be viewed as a basic building block for building fault-tolerant agreement protocols. They have illustrated the proposed modular approach utilizing the notion of *consensus service* to build agreement protocols, such as non-blocking atomic commitment and synchronous multicast protocols.

In [21], the authors show how to verify mechanically a transaction processing system. In such systems, a standard set of ACID (Atomicity, Consistency, Isolation and Durability) properties must be ensured by a combination of concurrency control and recovery protocols. It has been argued that in the existing literature, these protocols are often studied in isolation, making strong assumptions about each other. The problem of combining them in a formal way is largely ignored. The paper [21] illustrates how to formally specify and verify a transaction processing system, integrating strict two-phase locking, undo/redo recovery and two-phase commit.

In [36, 37], the authors had shown the initial approaches on developing a framework based on the concepts of category theory for modular composition of dependable distributed protocols. In [37], the hierarchical composition of the FDIR protocol is shown using the attributes of constituent building blocks of a particular class of protocols namely the Redundancy Management protocols. In [36], the authors first show the formulation of a group membership protocol, and then constructed a checkpointing protocol by utilizing the group membership function as one of its building block.

1.1.8 Motivation

The idea presented in [21] is similar to what we have presented in this thesis. However, we would like to emphasize that we provide a more rigorous formal framework utilizing Category Theory [36, 37] that provides for a better compositional specification, verification and traceability of different attributes and operations. We argue that category theory provides a precise and convenient conceptual language, and tool to model complex systems since it provides a rich body of theory for reasoning about objects and relations between them, namely specifications and their interconnections. It is also sufficiently abstract that it can be applied to a wide range of different specification languages. Furthermore, in order to relate to the actual operational behavior of the 3PC protocol [2, 4, 15, 22, 27, 38, 39, 40], we have incorporated an exhaustive set of functionally supporting protocols in analyzing the correctness of the overall 3PC operation. However, we would like to reiterate here that the sub-blocks we had identified were based on the basic understanding of the 3PC protocol operation, and would work fine for the class of protocol (Transaction Processing) considered in this thesis.

The extensions described in this thesis were motivated by the need to support the evolution of large specifications. We needed to consider how to adapt the categorical framework to support the following requirements:

- Support for traceability as a specification evolves, by explicitly representing the relationships between specification components, and between specification and validation properties. The framework needs to support an ability to trace these relationships to their rationales, and to support tracing of the impacts of change.
- Support for compositional verification, so that global system (protocol) properties can be decomposed across the structure of a specification, and such that we limit the number of proofs that have to be re-checked when a change is made.
- Support for the process of defining relationships (morphisms) between specification, with the ability to handle morphisms effectively across the various compositional specifications of the system (protocol).

We next explain how our category theoretic approach helps in the modular composition of complex dependable distributed protocols.

1.1.9 Category Theoretic Approaches for Modularization

Category theory [3, 11] has been used for a number of years as a framework for composing formal specification based on composition of algebraic specifications. But more recently [12] have developed an approach where each component of a system is described by a theory in temporal logic and theories are interconnected by specification morphisms. Category theory provides an excellent basis for providing structure in formal specifications [17, 25]. The basic principle to specify a system using this framework is to specify each component of a system separately and then use the pushout or colimit to compose the specifications. It provides a coherent and well-founded theoretical basis for representing structure in existing specification languages, thus avoiding the need to add structuring primitives within each language. Finally, category theory lends itself well to automation, so that, for example, the composition of two specifications can be derived automatically, provided that the category of specifications obeys certain properties.

In a large protocol, it is desirable to mix both specification and composition steps such that at any particular moment in the process, we may have established only some of the properties of the components, and some of the composition relations. This reflects the reality of large-scale specifications constructed by a team of people. But such specifications tend to be inconsistent for most of their life-cycle. As the specification evolves, each change may introduce many inconsistencies. Since category theory employs the "correct by construction" approach for the purpose of modular composition, wherein components are specified, proved correct and then composed together in such a way to preserve their properties [34, 42], the inconsistencies that might arise during the specification evolution would be eliminated before the change is applied to the specification.

In this thesis, we demonstrate how the categorical framework can be adapted for modularly composing the smaller components (sub-blocks or sub-protocols) into a larger global protocol. There are three elements to our approach: (1)decomposition of the complex protocol into sub-blocks based on the identified global properties. (2)the ability to define and interpret various morphisms over composition, (3)compositional specification of the sub-blocks, and (4)the integration of properties in the same framework as the specifications. Our approach is supported by a tool (*SPECWARE*) that implements all the categorical concepts and operations needed for the definition of modules. The tool can perform all the compositional operations defined on modules and thus automatically build the specification of the global protocol from the specifications of its sub-blocks. We summarize the contributions of this thesis in the next section.

1.1.10 Contribution

Our aim in this thesis has been to apply our proposed category-theoretical approach for protocol composition to a complex (and also a practical) transaction processing protocol integrating *all* sub-protocols which are instrumental in achieving the correct protocol level operations. Specifically, we

- identify building blocks of a *transaction processing protocol* namely the *centralized non-blocking three phase commit (3PC) protocol*,
- highlight their inter-dependencies and functionalities in order to achieve the overall global properties of the protocol,
- apply concepts of category theory to compose the 3PC protocol utilizing the sub-protocols and address issues involved in block interactions, and finally

- demonstrate how by breaking down complex protocol blocks into smaller sub-blocks, it becomes relatively easy to verify the global properties of the protocol. This is because, by verifying (and utilizing) the smaller sub-protocols, we can be rest assured the correctness of the overall complex protocol as it is now formed of inter-related sub-protocols.

1.1.11 Organization of the thesis

The rest of the thesis is structured as follows. In chapter 2, we introduce the concepts of Category Theory and explain its role in modular composition. In chapter 3, we discuss the non-blocking atomic three phase commit protocol as a case study, identify various *building blocks* in the 3PC protocol and discuss their functionalities and properties. Chapter 4 gives the formal analysis of the identified building blocks with respect to the global properties of 3PC with colimit diagrams. In chapter 5, we provide the specifications for the global properties of the 3PC using Specware [28, 32, 41], and also discuss the proofs for those properties. Chapter 6 concludes with discussions, and a note on our ongoing and future work in modular protocol composition and verification.

Chapter 2

Category Theory

In this chapter we explain a formal framework utilizing concepts of category theory to facilitate a rigorous and consistent composition out of system building-block protocols. At first, we provide an overview of the modularization technique to formulate general principles, and to give an informal introduction into our algebraic concept of modules (building-block protocols). Next we introduce the algebraic specification of modules using category theory along with their interconnections. We then discuss the mechanism of composition of these modules (sub-blocks or sub-protocols or components) to form the required global composed module (protocol). Lastly we provide details about the SPECWARE tool that we have used in this thesis for the purpose of specifying and composing the decomposed sub-protocols of a dependable distributed protocol.

2.1 Modularization

Modularization is one of the main principles in protocol development. The main problem is to divide the protocol to be built and the workload appropriately so that protocol development becomes rational and manageable. Modules (sub-blocks or sub-protocols) can be seen as the basic building blocks being used for modularization. A module mainly comprises of three components:

- `Interface`: The interface collects all resources (properties) and their inner relationships, which are provided by the module to its outside environment. It also includes the resources (properties) and relationships that are taken from the outside environment and used in the module.
- `Construction`: The construction of a module defines the individual functioning and denotation of resources (properties). A module does not abstract from this definition, but explicitly contains this definition as part of it. But in general the construction is not provided in the interface.
- `Behavior`: The behavior represents the overall functioning of the module based on its interface and construction. It also represents a particular semantic view of the module.

These components of a module are to be given syntactically as well as semantically, and form a conceptual unit. The components of a modular system are primarily the following:

- `Modules`: Modules form the building blocks of a modular system. They represent particular system components that should be seen as an unit.
- `Module Interconnections`: Module interconnections define the way modules interact with each other or how they are tied together. They form the architectural structure of a modular system.
- `Operations on Modules`: Operations on modules define modules out of given modules and module interconnections. In this way they change the view of the architectural structure of a modular system. Like the components of a module the components of a modular system, including module operation, are given syntactically and semantically.

2.2 Module Specification using Category Theory

In this section, we outline the category theory based modular specification framework. We first define some general terms of category theory which are used in the remainder of the thesis.

Category

A category is composed of two collections:

- the *objects* of the category.
- the *morphisms (arrows)* of the category.

These two collections must respect the following properties:

- each morphism f is associated with an object A that is its domain and an object B that is its codomain. Notation: $f\colon A \rightarrow B$.
- for all morphisms $f\colon A \rightarrow B$ and $g\colon B \rightarrow C$, there exists a composed morphism $g \circ f\colon A \rightarrow C$ and the composition law is associative, i.e., for all $h\colon C \rightarrow D$, $h \circ (g \circ f) = (h \circ g) \circ f$.
- for each object A of the category, there exists an identity morphism id such that:

$$\forall f\colon B \rightarrow A,\ id \circ f = f$$

$$\forall f\colon A \rightarrow B,\ f \circ id = f$$

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 $SPEC = (SIG,AX)$ consists 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 that (a) axioms are translated to theorems, and (b) source operations are translated compatibly to target operations.

Category theory provides a framework to describe links between objects, and to manipulate them by means of operations. Here we describe one such operation namely the pushout operation.

Pushout

A *pushout* of a pair of morphisms with same source $f\colon A \rightarrow B$ and $g\colon A \rightarrow C$ in a category is an object D and a pair of morphisms $p\colon B \rightarrow D$ and $q\colon C \rightarrow D$ such that the square commutes (figure 2.1):

$$f \circ p = g \circ q$$

and such that the following universal condition holds: for all objects D' and all morphisms $p'\colon B \rightarrow D'$ and $q'\colon C \rightarrow D'$ such that $p' \circ f = q' \circ g$, there exists an unique morphism $u\colon D \rightarrow D'$ such that $u \circ q = q'$ and $u \circ p = p'$.

The second part of the definition ensures that the D chosen to construct the pushout is the "minimal" such D amongst all the candidates D'. The generalization of this operation to several objects and morphisms is called a colimit. Module

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "g"'] & B \arrow[d, "p"] \arrow[ddr, "p'"] & \\
C \arrow[r, "q"'] \arrow[drr, "q'"'] & D \arrow[dr, "u"] & \\
 & & D'
\end{tikzcd}$$

Figure 2.1: Pushout of two morphisms f and g

specifications are defined by utilizing the notion of *push-out* operation.

Colimit

For the treatment of structuring mechanisms for categorical construction, we utilize the concept of *colimit* operation in category theory. In order to use the colimit operation to combine specifications, the morphisms between the specifications have to be first indicated. This is also called as the *diagram of specifications*. A diagram is a directed multigraph whose nodes are labeled with specifications and whose arcs are labeled with morphisms. The colimit operation is then applied to a diagram of specifications linked by morphisms. The colimit contains all the elements of the specifications in the diagram, but only elements that are linked by arcs in the diagram are identified in the colimit. In Figure 2.2, for objects A_i and A_j, and morphism a_x, the *colimit* is an object L and a family of morphisms I_i, I_j such that for each $I_i : A_i \rightarrow L$, $I_j : A_j \rightarrow L$, and $a_x : A_i \rightarrow A_j$, then $I_j \circ a_x = I_i$.

Conceptually, the colimit of a set of specifications is the "shared union" of those specifications based on the morphisms between the specifications. Moreover,

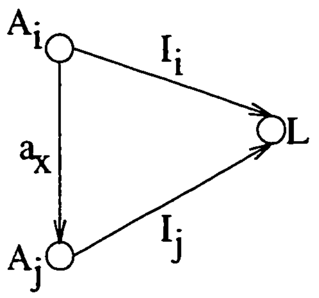


Figure 2.2: Colimit Function

these morphisms define equivalence classes of sorts and operations. For example, if a morphism for specification A to specification B maps sort α to sort β, then α and β are in the same equivalence class and thus in a single sort in the colimit specification of A, B, and the morphism between them. Therefore, the colimit operation creates a new specification, the colimit specification, and a cone morphism from each specification to the colimit specification. These cone morphisms satisfy the condition that the translation of any sort or operation along any of the morphisms in the diagram leading to the colimit specification is equivalent.

Constituent Parts of Algebraic Module Specifications

An algebraic module specification consists of components, called *import, export, parameter* and *body* as shown in Figure 2.3.

A module specification *MOD = (PAR, EXP, IMP, BOD, f, h, g, k)* consists of four specifications:

- *PAR*, called *parameter* specification

$$\begin{tikzcd}
PAR \arrow[r, "f"] \arrow[d, "g"'] & EXP \arrow[d, "h"] \\
IMP \arrow[r, "k"'] & BOD
\end{tikzcd}$$

Figure 2.3: Module Interfaces

- *EXP*, called *export* interface specification
- *IMP*, called *import* interface specification
- *BOD*, called *body* specification

and four mapping morphisms *f, h, g, k* such that the following diagram commutes (i.e. $f \circ h = g \circ k$).

- `Import Interface`: The import interface is used to specify those resources which are to be provided by other modules and used in the modules' body for construction of the resources to be exported. It is an algebraic specification consisting of a signature, which names and types the resources to be imported and eventually lists properties of these resources, which form restrictions for the import of actual resources and provide information for the use of these resources in the body of the module. The explicit formulation of an import interface is especially useful in the stepwise development of a modular system. It allows a top down way of construction where resources are named and used, but only later to be realized by other modules.

- `Export Interface`: The export interface contains those resources which are realized by the module at hand to be used by other modules or an application environment. In a module specification these resources are declared in the same way as the resources of the import interface. It restricts sorts and operations treated in a module to those which are visible for the user of the module. This realizes hiding of resources, which serves the purpose of protection of resources, abstraction from internal details, and independence from particular forms of construction in the body of the module.

- `Parameter Part`: The parameter part is a part common to import and export, and sometimes intersection of both. It can be seen as the parameter of the whole module as it appears to the outside by its interface. Its useful in declaring all those resources of the parameter of the full system, which concern this particular module.

- `Body Part`: The body part of a module contains the construction of the resources declared in the specification of the export interface. For this purpose, the body may contain auxiliary sorts and operations which do not belong to any other part of the module but depend on the particular choice of construction. The realization of sorts and operations declared in the specification of the export interface is encapsulated in the module, not accessible to the user of the module.

- `Component Interconnection`: The relationship between the various components of the specification module namely the parameter, import, export and body is established by mapping morphisms which consistently map the four components. During modular composition of two individual module specifications, specification morphisms are used to map the parameter parts of the two modules. In order to arrive at the final composed module specification based on the two individual modules, colimit morphisms are used to map the body parts of both the modules.

Interconnection of Module Specifications

Interconnections of algebraic module specifications are explicitly declared by specification morphisms which express how resources in the interfaces and the parameter part are matched. Though such interconnections may be simple in many cases, the expressiveness of interconnections based on specification morphisms allows renaming and identification of resources. Besides that, the concept of specification morphism guarantees that the matching of the corresponding components of the two modules being interconnected, is consistent with the declaration in these components. Since the result of the interconnection of module specifications is a new composed module specification, each interconnection mechanism can be considered as an operation on module specifications. These are operations on a higher level than those considered within abstract data types and abstract modules.

2.3 Composition

To capture module interactions, our proposed composition scheme allows two modules to be interconnected via export and import interfaces. The *push-out* of the two modules is the resulting specification of the composed module. Figure 2.4 depicts the composition operation. In the figure, *Module1* has four objects namely Parameter(R_1), Export(A_1), Import(B_1) and the pushout of these three objects giving the Body(P_1) which is the specification of Module1. Similarly *Module2* has R_2, A_2, B_2 and P_2 as its Parameter, Export, Import and Body respectively. In Figure 2.4(a), *Module1* imports via specification B_1 whatever *Module2* exports via specification A_2. The compatibility of the parameters (or semantic constraints) is governed by the morphism t. Furthermore, the following property must be respected: $g_1 \circ s = t \circ f_2$. Basically, in category-theoretic terms, modules 1 and 2 are diagrams which commute themselves individually by the specification morphism relationship of $f_1 \circ h_1 = g_1 \circ k_1$ and $f_2 \circ h_2 = g_2 \circ k_2$ and their colimit would produce the required composed module, which would now commute by the relationship $f_1 \circ h_1 \circ m_2 = t \circ g_2 \circ k_2 \circ m_1$ as in Figure 2.4(b). Since the composed module also commutes, its specification is proved correct thereby helping in the reusability of the module.

The module(P_{12}) got by composing the two sub-modules(1 and 2) is also a diagram with its Parameter as the parameter of module1, Export as the export of module1, Import as the import of module2 and the Body as the union of bodies P_1 and P_2 over the export of Module2. In this case, the resulting composed module P_{12} is

(R_1, A_1, B_2, P_{12}), where P_{12} is the *push-out* of P_1 and P_2 over B_1(See Figure 2.4(b)).

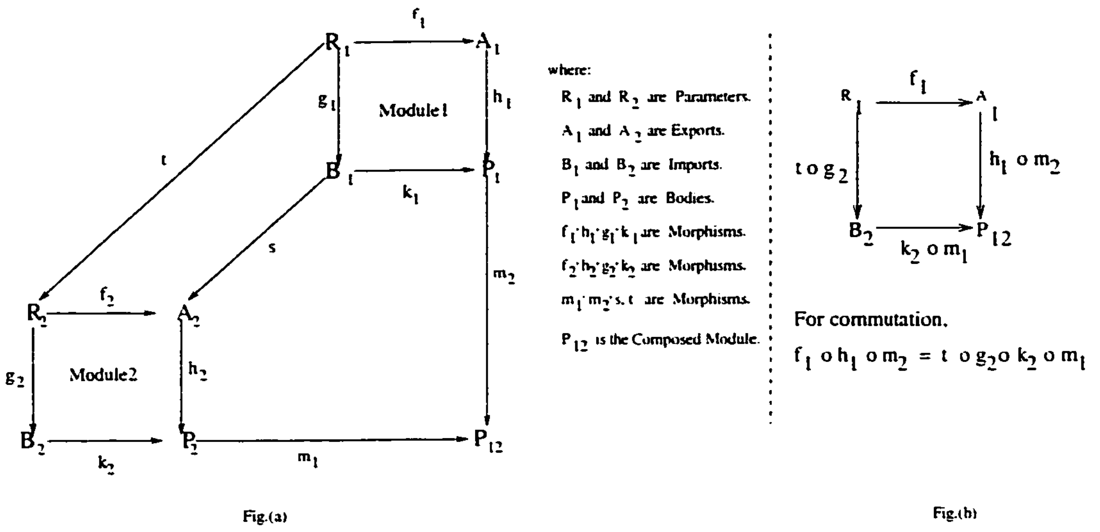


Figure 2.4: (a) Composition of Two Modules and (b) Composed Module

For complex protocols, a module may import parameters from several different modules, and also the specification consisting of syntactic and semantic requirements may compose of several different small specifications. 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 B. In other words, specifications A and B correspond to guarantees and assumptions, respectively. 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.

2.4 Specware Tool

Specware is an automated software development system that allows users to precisely specify the desired functionality of their applications and to generate provably correct code based on these requirements. It helps in the specification of large and complex systems by combining small and simple specifications, and those system specifications can be refined into a working system by the controlled stepwise introduction of implementation design decisions, in such a way that the refined specifications and ultimately the working code is a provably correct refinement of the original system specification.

Specware aids in expressing requirements as formal specifications without regard to the ultimate implementation or target language. Specifications describe the desired functionality of a program independently of such implementation concerns as architecture, algorithms, data structures, and efficiency. This makes it possible to focus on the correctness, which is crucial to the reliability of large software systems. Using Specware, the analysis of the problem can be kept separate from the implementation process, and implementation choices can be introduced piecemeal, making it easier to backtrack or explore alternatives.

Specware allows to articulate software requirements, make implementation choices, and generate provably correct code in a formally verifiable manner. The progression of specifications forms a record of the system design and development that is invaluable for system maintenance.

Chapter 3

Transaction Processing Protocols

In this chapter 3, we discuss the process of transaction in a distributed database. We then illustrate the importance of commit protocols for successful transaction processing in a distributed environment. Next we introduce the non-blocking atomic three phase commit protocol as a case study, identify various *building blocks* in the 3PC protocol by the modularity mechanism and discuss their functionalities and properties.

3.1 Transaction Processing

A transaction is the fundamental unit of processing in a database management system. Transfer of money from one account to another, reservation of train tickets, filing of tax returns, entering marks on a student's grade sheet, are all examples of transactions. The primary feature of a transaction is that it is an atomic unit of work that is either completed in its entirety or not done at all. The successful execution of the transaction results in the transaction being committed, which means that its effects on the database are permanent regardless of possible subsequent system failures. If, for any reason, it is not possible to commit the transaction, all the effects arising out of the partial execution of the transaction are removed from the database, and the transaction is said to have been aborted. In short, a transaction is an "all or nothing" unit of execution.

For a variety of reasons, many database applications store their data distributed across multiple sites that are connected by a communication network. In this environment, a single transaction may have to execute at many sites, based on the locations of the data that it needs to process. A potential problem associated with distributed transaction execution is that some sites could decide to commit the transaction while the others could decide to abort it, resulting in a violation of transaction atomicity. To address this problem, distributed database systems use a *transaction commit protocol.* A commit protocol ensures the uniform commitment of the distributed transaction, that is, it ensures that all the participating sites agree on the final outcome (commit or abort) of the transaction. Most importantly, this guarantee is valid even in the presence of site or network failures.

Over the last two decades, a variety of distributed transaction commit protocols have been proposed. To achieve their functionality, these commit protocols typically require exchange of multiple messages, in multiple phases, between the participating sites. A commit protocol is said to be non-blocking if, in the event of a site failure, it permits transactions that were being processed at the failed site to terminate at the operational sites without waiting for the failed site to recover. With blocking protocols, there is a possibility of transaction processing grinding to halt in the presence of failures. Non-blocking protocols, on the other hand, are designed to ensure that such major disruptions do not occur. To achieve their functionality, however, they usually incur additional messages than their blocking counterparts. The three phase commit protocol that we have considered as a case study in this thesis is a kind of non-blocking protocol.

A transaction is characterized by the following *ACIDS* properties namely:

- `Atomicity`: A transaction is an atomic unit of work, that is, effectively either all or none of the transaction's operations are performed. If all the operations have been performed successfully, the transaction commits. If some operation of the transaction fails, the partial results of the transaction are undone and the transaction aborts. Thus, it gives an illusion that the transaction either completed successfully or was not even started.
- `Consistency`: The consistency of a transaction means its correctness. An individual execution of a transaction must take the database from one consistent state to another consistent state. It is usually the responsibility of application programmer to ensure the correctness of the transaction.
- `Isolation`: An incomplete transaction cannot make its database modifications

visible to other transactions before its commitment. Violation of this property may lead to "cascading aborts" of the transactions.

- `Durability`: Once a transaction has committed, the system must guarantee that the effects of the transaction will never be lost despite subsequent failures of the system.
- `Serializability`: The concurrent execution of a set of transactions is equivalent to some serial execution of the same set of transactions. Guaranteeing serializability lets the transaction programmer write the transaction in its individuality without worrying about other transactions that may be executing concurrently.

3.2 Distributed Commit Protocols

A distributed transaction executes at multiple sites. To ensure the atomicity of the distributed transaction, all cohorts of the transaction must reach an uniform decision (commit or abort). This guarantee is provided by the *transaction commit protocols*. A common model of distributed transaction execution is shown in Figure 3.1.

At the site where the transaction is submitted, a master process is created to coordinate the execution of the transaction. At a site where the transaction needs to access data, a cohort process on behalf of the transaction is created. Usually, there

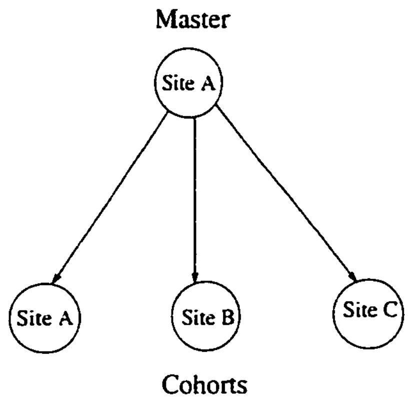


Figure 3.1: Distributed Transaction Execution

is only one cohort on behalf of the transaction at each such site. The master sends the *startwork* message to a cohort when some data at the site of the cohort is required to be accessed. Depending on the transaction architecture, the master may send the *startwork* messages to multiple cohorts without waiting for their responses, that is, the cohorts of the transaction will execute in a parallel fashion. Or the master may send the subsequent *startwork* message only after the previous cohort has completed the work assigned to it, that is, the cohorts of the transaction execute one after the other in a sequential fashion. A cohort, after successfully executing the master's request, sends a *workdone* message to the master. After receiving the *workdone* message, the master may send more work to the same cohort, or may decide to send the next piece of work to another cohort of the transaction.

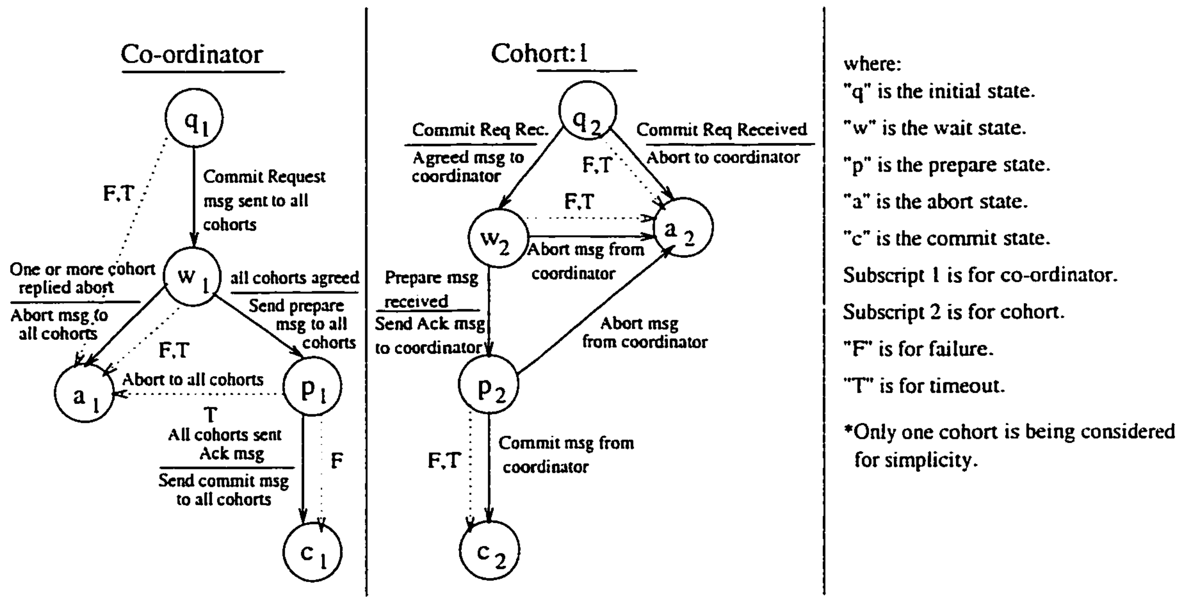
The master may decide to conclude the transaction when all the work assigned to the transaction is completed. Or the decision may be necessitated at the instance of the user submitting the transaction. In any case, the decision to commit or abort the transaction must be uniform - the master and all cohorts must agree on a common decision. It might seem that the fact that a cohort had sent a *workdone* message means that the cohort was willing to commit the transaction. The problems here are these: First, the sending of the *workdone* message does not enforce any binding on the cohort to agree on the decision. Due to reasons such as concurrency control, performance, etc., the cohort could still be aborted even after the *workdone* message was sent. Second, there can be failures - communication links may fail, or some of the sites hosting the cohorts of the transaction may fail. Therefore, a commit protocol is needed to ensure that all cohorts and the master reach an uniform decision - a decision that will be binding on all sites even if a failure occurs. A variety of transaction commit protocols have been devised, the latest being the `Three Phase Commit Protocol`, which we have considered as a case study in this thesis.

A Digression about Failures

Two points are worth mentioning before we start the description of the 3PC protocol: First, how does a site know that a communication link or a remote site has failed? And second, how are failures handled? Generally, a timeout mechanism is used to address the first question. If the response to a message is not received from the remote site within the timeout period, the local site assumes that either the communication link to the remote site, or the remote site itself has failed. For the second question, when a site recovers from a failure, it is handed over to the recovery manager of the site. The recovery manager scans the log, and if it finds a *commit* log record for a transaction, it knows that the transaction had committed before the failure occurred. In the same way, if it finds an *abort* log record, it knows that the transaction was aborted before the failure occurred. In these cases. the recovery manager ensures that the effects of the committed transactions are redone and the effects of the aborted transactions are undone. The only problem is for the transactions about which neither a *commit* nor an *abort* log record is found. The actions that recovery process need to take in this case depend on the commit protocol being used.

3.3 Three Phase Commit Protocol

The three phase commit protocol is a kind of transaction processing protocol which is resilient to site failures, i.e, it does not block in the event of site failures. For example, if the master fails after initiating the protocol but before conveying the decision to its cohorts, these cohorts will become blocked and remain so until the master recovers and informs them of the final decision. During the blocked period, the cohorts may continue to hold system resources such as locks on data items, making these unavailable to other transactions, which in turn become blocked waiting for the resources to be relinquished, i.e., "cascading blocking" results. It is easy to see that, if the blocked period is long, it may result in major disruption of transaction processing activity. To ensure that commit protocols are non-blocking in the event of site failures, operational sites should agree on the outcome of the transaction (while guaranteeing global atomicity) by examining their local states. In addition, the failed sites, upon recovery must all reach the same conclusion regarding the outcome (abort or commit) of the transaction. This decision must be consistent with the final outcome of the transaction based solely on their local state (without contacting the sites that were operational). We now explain the 3PC protocol by viewing the finite state automata that illustrates the various scenarios of the three phase commit protocol with timeout and failure transitions.



Co-ordinator
q1
F,T
Commit Request msg sent to all cohorts
One or more cohort replied abort
Abort msg to all cohorts
w1
all cohorts agreed
Send prepare msg to all cohorts
F,T
Abort to all cohorts
a1
p1
T
All cohorts sent Ack msg
Send commit msg to all cohorts
F
c1
Cohort: 1
q2
Commit Req Rec.
Agreed msg to coordinator
F,T
Commit Req Received
Abort to coordinator
F,T
w2
a2
Abort msg from coordinator
Prepare msg received
Send Ack msg to coordinator
Abort msg from coordinator
p2
F,T
Commit msg from coordinator
c2
where:
"q" is the initial state.
"w" is the wait state.
"p" is the prepare state.
"a" is the abort state.
"c" is the commit state.
Subscript 1 is for co-ordinator.
Subscript 2 is for cohort.
"F" is for failure.
"T" is for timeout.
*Only one cohort is being considered for simplicity.

Figure 3.2: Three Phase Commit Protocol with the Co-ordinator and a Cohort

Initially the co-ordinator of the transaction (which is currently in initial state *q1*) sends out a *Commit request message* to all cohorts and goes to the wait state *w1*. Then if all the cohorts (which are in the initial state *q2*) receive this *request* message from the co-ordinator, they either go to the *abort* state *a2* (in which case the site has already sent an abort message to the co-ordinator) or the *wait* state *w2* (in which case the site has sent an agreed message to the co-ordinator) or the *initial* state *q2* itself (in which case the site hasn't received the request message from the co-ordinator). If a cohort fails, the co-ordinator times out waiting for the *agreed* message from the failed cohort. In this case, the co-ordinator aborts the transaction and sends *abort* messages to all the cohorts. This is the first phase of operation of the three phase commit protocol.

In the second phase, the co-ordinator sends a *prepare* message to all the cohorts, if all the cohorts have sent *agreed* messages in the first phase. Otherwise, the co-ordinator will send an *abort* message to all the cohorts. On receiving a prepare message, a cohort sends an *acknowledge* message to the co-ordinator. If the co-ordinator fails before sending prepare messages, i.e, in state *w1*, it aborts the transaction upon recovery, according to the *failure transition*. The cohorts time out waiting for the prepare message, and also abort the transaction as per the *timeout transition*.

In the third phase, on receiving acknowledgments to the prepare messages from all the cohorts, the co-ordinator sends a *commit* message to all the cohorts. A cohort, on receiving a commit message, *commits* the transaction. If the co-ordinator fails before sending the commit message, i.e, in state *p1*, it commits the transaction upon recovery, according to the failure transition from state *p1*. The cohorts time out waiting for the commit message. They commit the transaction according to the timeout transition from state *p2*. However, if a cohort fails before sending an acknowledgment message to a prepare message, the co-ordinator times out in state *p1*.

The co-ordinator aborts the transaction and sends *abort* messages to all the cohorts. The failed cohort, upon recovery, will abort the transaction according to the failure transition from state *w2*. In the above explanation, the presence of state p (which is the prepare state), ensures that *no two concurrent states in the global state have both the commit and abort states*, which is the required condition for non-blocking in commit protocol.

3.4 Assumptions

The overall correctness of the 3PC protocol depends on the following assumptions that are also the guiding factors in determining the inherent building blocks for the three-phase commit protocol:

1. *FIFO* communication.
2. *Reliable network* without *partitioning.*
3. *Synchronous* state transition.
4. Logging always done on a *stable* storage medium.
5. No two local states which are concurrent have an *abort* and a *commit* state.
6. *Synchronous timer* mechanism on each site.
7. *Reliable termination* mechanism in times of failure.
8. *Independent* recovery.
9. *Global state* mechanism containing *local states* of all participants and co-ordinator.

10. *Centralized* commit protocol.
11. *No crash* when either co-ordinator or participants fail.
12. Site failure model: *Failure transition* sending a prefix message along with time out messages.
13. *Voting* mechanism to elect backup co-ordinator when a co- ordinator fails.
14. *Two way channel* communication between co-ordinator and participants.
15. *Sequential* ordering of messages between co-ordinator and the participants.
16. *Serializability* and *Recoverability.*
17. No *domino effect.*

3.5 Inherent Building Blocks of the 3PC Protocol

In this section, we reason about the identification of the various building blocks and the relationships between them in terms of the specific functionalities of each of the building blocks. By rigorous analysis of the various functionalities, properties and operations of the three phase commit protocol [7], we have identified different protocols or sub-blocks which are needed for the normal global operation of the 3PC. In order for formal techniques to be effective in validating the correctness of the overall operation of dependable protocols, one needs to incorporate a broader view of protocols' interactions and their effect on the correctness of global properties. Thus we feel that by decomposing a complex protocol like 3PC into smaller sub-blocks based on their individual applications, each contributing to the overall operation, would help the system designer in validating the correctness of the 3PC protocol.

Table 3.1: Various Building Blocks of 3PC

1. Controller Protocol 1.1 Broadcast Protocol 1.2 Consensus Protocol	6. Checkpointing Protocol
2. Snapshot Protocol	7. Recovery Protocol
3. Voting (or) Election Protocol	8. Decision Making Protocol
4. Undo/Redo Logging Protocol	9. Termination Protocol
5. Two-Phase Locking Protocol	10. Failure/Time-out Management Protocol

Based on the different building blocks that we have identified, an important step in composing a protocol is to describe the inter-dependencies of these building blocks in the overall three phase commit protocol operation (See Figure 3.3).

In Figure 3.3, we show how the different sites (nodes) in the network interact with each other for doing transactions using the three phase commit protocol. What the diagram really illustrates is that, each of the sites shown, has a three phase commit protocol running inside them and that the nodes use this protocol while taking decisions during the transactions with each other. In the following section, we reason about the identification of these building blocks and the relationships between them in terms of the specific functionalities of each of the building blocks.

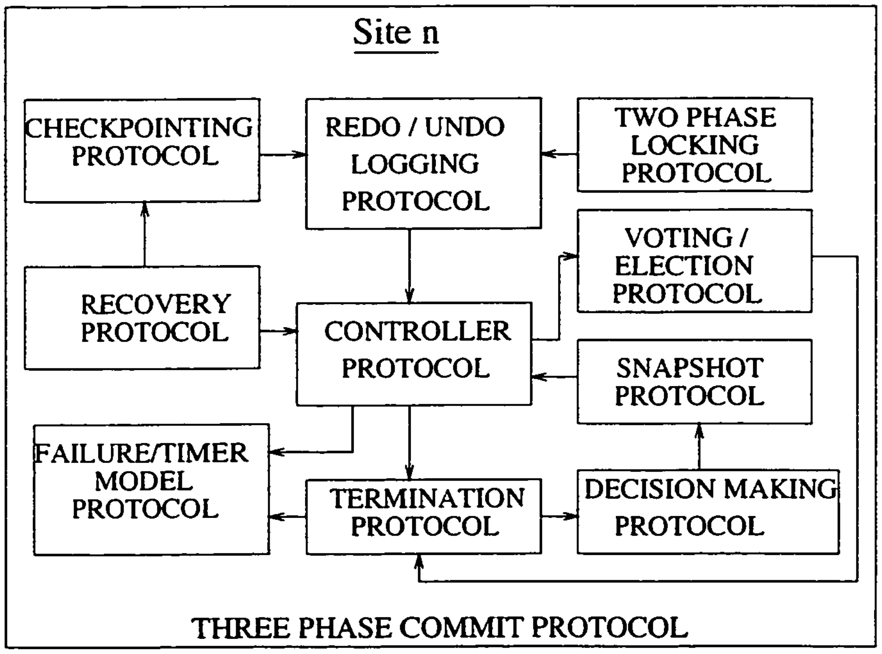

Site n
CHECKPOINTING PROTOCOL
REDO / UNDO LOGGING PROTOCOL
TWO PHASE LOCKING PROTOCOL
VOTING / ELECTION PROTOCOL
RECOVERY PROTOCOL
CONTROLLER PROTOCOL
SNAPSHOT PROTOCOL
FAILURE/TIMER MODEL PROTOCOL
TERMINATION PROTOCOL
DECISION MAKING PROTOCOL
THREE PHASE COMMIT PROTOCOL

Figure 3.3: Global View of Modulated 3PC with Inherent Building Blocks

3.5.1 Functionalities of the Various Building Blocks

Controller Protocol

This protocol co-ordinates all the activities of the entire three phase commit protocol. This protocol initially recognizes the various parties (participants or components or sites) in the distributed system who are willing to do a transaction. After all the participants involved in the transaction find out the leader (co-ordinator) and the sub-ordinates (cohorts or participants) using the voting protocol, the controller protocol in the co-ordinator sends out a multicast commit message to all the cohorts with the help of the broadcast protocol. Finally this protocol helps the different participants involved in the transaction to reach a common decision among them using the consensus agreement protocol. The controller protocol depicted in Figure 3.3 has both the *broadcast* and *consensus* protocols in it.

Requirements:

- If a participant fails, then rest of the participants must recognize the failure.
- Should allow recovery from mid-commitment failure.
- Should have reliable broadcasting mechanism between the participants or sites.
- Should have uniform agreement procedure among the various participants.
- Cause all actions to become permanent so that they cannot be undone in the future.
- Commitment must be executed at the end of a transaction.

- Should have sufficient provision for collecting the various local states of the sites in the network and to store them as global state vectors in the snapshot protocol.

Broadcast Protocol

This protocol is part of the controller sub-block in our design. The sending and receiving of messages between the co-ordinator and the participants are considered to be reliable and atomic. This means, to execute A-Broadcast m, a process simply Reliable-Broadcast m, i.e., it sends m to all its neighbors in the system. When a process R-delivers m, it schedules its A-delivery at local time $(T + \delta)$ where Δ is the time constant. If f is the maximum number of processes that may crash and δ is the upper bound delay, then $\Delta = (f + 1)\delta$.

Requirements:

- Termination: If a correct process multicasts a message m to P, then some correct process of P eventually delivers m or all processes of P are faulty.
- Validity: If a process p delivers a message m, then m has been multicast to a set P and p belongs to P.
- Integrity: A process p delivers a message m at most once and there is no duplication.
- Uniform Agreement: If any process of P delivers message m, then all correct processes of P deliver m.

- Timeliness: There is a time constant Δ such that if the multicast of m is initiated at real time T, then no process delivers m after $(T + \Delta)$.
- Simple Agreement: If a correct process of P delivers a message m. then all correct processes of P delivers m.

Consensus Protocol

This protocol is also part of the controller sub-block in our design. As the coordinator sends a request message to *commit* to all the participants, the non-faulty participants form an agreement among themselves based on various information they have about each other and finally decide to commit or abort. This way the non-faulty participants in the system would reach a consensus among themselves.

Requirements:

- Termination: Every correct process (site) eventually decides some value.
- Integrity: A process (site) decides at most once.
- Validity: If a process decides X, then X was proposed by some process.
- Agreement: No two correct processes decide differently.
- Uniform Agreement: No two processes decide differently.

Snapshot Protocol

The snapshot protocol gets activated as and when new local state transitions take place thereby updating the global state which is basically the combination of all the local states. This protocol helps the decision making protocol to check whether no two local states in its global state vector has both the *commit* and *abort* states. If a site crashes or fails after committing, its local states are still stored in the global state vector as long as it gets recovered. But if a site fails before committing, its local states are no longer present or kept in the global state vector.

Requirements:

- Global state is always consistent, i.e., its state vector doesn't have both a *commit* state and an *abort* state.
- Global state transition occurs whenever a local state transition occurs at a participating site.
- Local state transitions are instantaneous and mutually exclusive events.
- Exactly one local transition occurs during a global transition.

Voting (or) Election Protocol

At the start of a transaction, this protocol helps in assigning the co-ordinator for the network and subsequently the other sites in the network as participants. But when the assigned co-ordinator fails, then this protocol is used to elect a *back-up* co-ordinator which then takes care of the transitions in the network. This protocol uses the termination protocol to find out whether a co-ordinator has failed or not in the network.

Requirements:

- When the termination protocol is invoked due to a site failure. and if that site happened to be the co-ordinator, then the voting/election protocol must be invoked to elect the back-up co-ordinator.
- Once the back-up is chosen, it would base the commit decision only on its local state.
- If the concurrency set for the current state of the *back-up* co-ordinator contains a *commit* state, then the transaction is committed, otherwise *aborted.*
- The back-up must issue a message to all sites to make a transition to its local state. and then issues a *commit* or *abort* message to each site.
- Should have the *master* (co-ordinator) protocol and *slave* (participants or cohorts) protocol.

Undo/Redo Logging Protocol

This protocol is used in times of volatile loss of memory data units of the transaction or in times of failure recovery when the last committed state should be known for efficient recovery of the failed site. Initially an undo entry is done to undo all the earlier data. Then just before committing the transaction, the site's data (state units) are redone in the stable log. This protocol is closely associated with the checkpointing protocol which in turn helps in the recovery through the recovery protocol. Undo and redo must function even if there is a second crash during recovery.

Requirements:

- Log must be kept in stable storage.
- Undo entry in stable log before writing into it.
- Redo entry in stable log before committing the transaction.
- Must write actions to log before actually taking them.
- Keep minimum number of values in the log at a time.

Two Phase Locking Protocol

This protocol helps in the proper logging of data during the active transaction. The locking protocol allows a lock (shared resource log) to be acquired before the transaction and then asks for unlocking once the transaction is over. This way, serializability would be maintained while logging data onto the stable storage.

Requirements:

- Only one transaction at a time may write lock an object.
- Write lock is implemented by a simple 1 bit write lock flag.
- Write lock should enforce complete mutual exclusion on the object.
- Multiple transactions may be read locking an object at the same time.
- Read lock is implemented by using a read counter which holds the number of transactions currently holding a read lock.
- If an object is write locked, no read locks are allowed.

- Transaction must unlock all objects before finishing.

Checkpointing Protocol

When transient failures occur, rollback recovery takes place with the help of the checkpointing protocol wherein the last successful or committed states of the sites reside. The checkpoints are normally decided by the co-ordinating processes and they use the logging mechanism to get the data items. Two checkpoints need to be stored at any time, one called the *permanent checkpoint* which cannot be undone and other called the *tentative checkpoint* which can be changed to a permanent one later.

Requirements:

- No *domino effect.*
- A set of checkpoints of different processes should form a consistent system state (i.e) orphan and lost messages should not be present.
- Prior to establishing its K^{th} checkpoint, a process should not consume any message sent by a process after establishing its K^{th} checkpoint.
- All co-operating processes checkpoint periodically, each with the same period Π which is $> (\beta + \delta)$.

Recovery Protocol

When a site fails by either failure transition or timeout transition, this protocol gets invoked and uses the checkpointing protocol to get the last committed state of the failed site. It tries to recover the site from its transient fault by rolling back to its checkpointed state. The recovered site then gets into the transaction process of the active system.

Requirements:

- Must restore an earlier possible state of the failed process using a checkpoint from a stable storage medium and also replay the logged messages.
- recognize the set of processes whose states depend on lost states using the dependency information and roll them back.
- Commit messages to the outside when it is known that the states that generated the messages will never need to be undone.
- A site failure after committing, should after recovery through this protocol join back with other non-faulty sites in the transaction.

Decision Making Protocol

Whenever a new transaction occurs or when new state transitions take place in the network, the snapshot protocol updates its global state with new local states of the sites in the network. The decision making protocol helps to check constantly the global states for any inconsistency in the local states. This protocol also helps in the termination of a transaction if the two rules of the three phase commit protocol are not satisfied by the co-operating sites.

Requirements:

- Must check the global state to see if there exists any local state such that its concurrency set contains both an *abort* and a *commit* state.
- Must make sure that the global state doesn't have a non-committable local state whose concurrency set contains a *commit* state.
- Should terminate the transaction using the termination protocol if any one of the above two conditions fail.

Termination Protocol

When the failure protocol invokes the termination protocol with the failed site as co-ordinator, the termination protocol gets involved in the process of choosing a *back-up* co-ordinator using the voting protocol. This, the termination protocol would do provided the other correct or non-faulty sites satisfy the conditions of the non-blocking theorem. But if the decision protocol detects that the two conditions are not satisfied by the global state of the network, provokes the termination protocol to completely terminate the transaction.

Requirements:

- Must terminate the transaction temporarily if the current state of at least one operational site obeys the conditions of the non-blocking theorem.
- If the two rules of the non-blocking theorem are not satisfied by any operational site, then the entire transaction is permanently terminated by this protocol.

- Must aid in voting a *back-up* co-ordinator if the earlier assigned one fails.

Failure/Time-out Management Protocol

The model used in this protocol basically specifies the type of failure the network would have, and the timer mechanism in it helps to timeout actions when a site failure occurs. The timeout transition is then broadcasted to the entire network in order that other non-faulty and operational sites know about the failure of the timed-out site. This protocol is of utmost importance when a co-ordinator times out in the network. Now when this happens, this protocol invokes the termination protocol which in turn invokes the voting protocol to elect a back-up co-ordinator for the network.

Requirements:

- A site is operational if, in response to inputs, it behaves in a manner consistent with the specification; if it behaves otherwise, it has failed.
- Must specify the failure model for the network.
- If a site has a local clock whose drift rate with respect to real time is ρ, then δ has to be replaced by $(1 + \rho)\delta$ in timeout delays to compensate the worst drift rate.
- If a participant P does not receive from Q a response to a message 2δ time units after its sending, then the result is that Q has crashed.
- In the above case, before being notified of the failure of Q by the timeout mechanism, P will receive all the messages Q sent to it before crashing.

At this stage, we have identified the 3PC protocol building blocks and described the high-level framework for protocol composition. We now illustrate how these building blocks composed together via categorical operations help achieve in satisfying the global properties of the 3PC protocol.

3.6 Determining Dependencies among the Building Blocks of Three Phase Commit Protocol

After having given brief description of each of the sub-protocols, in this section we elaborate on their modular dependencies and in that respect, we have divided the building blocks into two sets of *sequential divisions*, with each set of sequence performing a part of the overall functionalities of the three phase commit protocol.

♣ *Sequential Division - 1*: The first set of sequential building blocks comprises of the *controller protocol*, *undo/redo logging protocol*, *two phase locking protocol*, *checkpointing protocol* and *recovery protocol*.

The overall functionality of the first sequence is to safely recover a failed site which is basically what the *recovery* block (protocol) does. But for the successful working of the recovery protocol, it needs the *checkpointing* protocol to inform about the last successful state so that the recovery protocol can start the recovery from that state. But the checkpointing protocol depends on a stable storage medium from where it can get the state information, which is what the *undo/redo logging* protocol does.

But before the checkpointing protocol can verify the log for last state information, a locking mechanism (*two phase locking* protocol) has to be applied on the log protocol to make sure that at a time no site does the operation of both reading and writing onto the log.

♣ *Sequential Division - 2*: The second set of sequential building blocks comprises of the *controller protocol, snapshot protocol, decision making protocol, termination protocol, voting/election protocol* and *failure/time-out management protocol.*

The global functionality of the second set of sequential building blocks is to elect or vote a backup coordinator when the initially assigned one fails. Now this is done by the *voting/election* protocol. But this protocol won't get activated as long as there is no termination call for the transaction. A call for termination is made when the coordinator fails by either a failure transition or a timeout transition. Also termination could take place due to non-satisfaction of the consistent state rule of three-phase commit protocol. The *decision making* protocol checks for these two rules and invokes the *termination* protocol if a site fails to satisfy the two rules. For all this to happen, the *snapshot* protocol is needed to give complete information about the local states of all the participating sites in the form of a global state.

In Figure 3.4, we show that by successively combining two sub-protocols using the colimit operation and associated morphisms, one can come up with a final (small) sub-protocol which would prove the global functionality of sequential division 1 of the three phase commit protocol.

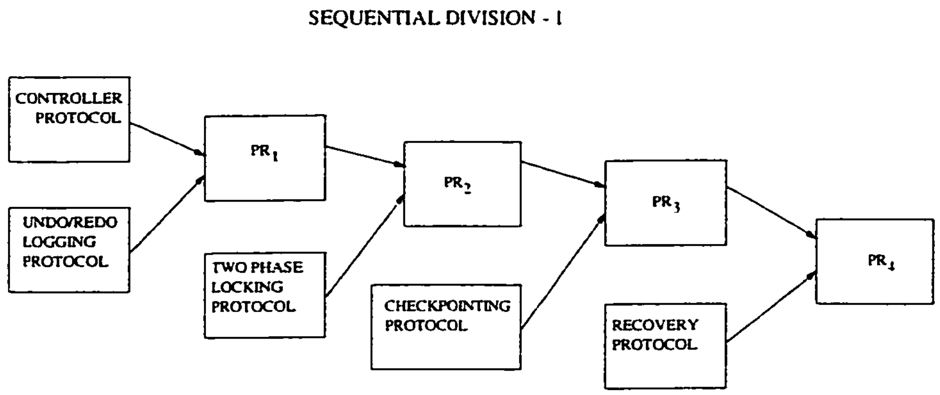


Figure 3.4: Modular Dependencies of Building Blocks based on Seq. Division–1

In Figure 3.4, initially we bring together the sub-protocols *controller* and *undo/redo log* with interactions between them being over the co-ordinator and participants information. By combining these two sub-protocols, we come up with a sub-protocol named PR_1 as depicted in Figure 3.4. This new sub-protocol, called the child sub-protocol, which is a conglomeration of two different sub-protocols (parent) would now satisfy the properties of both of its parents which means that PR_1 would both have information regarding the various components ready for transaction and also log their state information onto a stable storage medium.

Then this sub-protocol PR_1 is combined with another sub-protocol namely the *two phase locking* protocol. Now PR_1 and *two phase locking* protocol act as parents to produce a single child sub-protocol called PR_2 with interactions between them being over the transaction details of the various components involved. Next sub-protocol PR_2 gets composed with *checkpointing* protocol with their common morphism of site state data to arrive at sub-protocol PR_3. PR_3 gets combined with *recovery* protocol due their common relationship of stored state information of all the sites to form a new sub-protocol PR_4.

This way we composed two sub-protocols at a time based on their relationship for all the sub-protocols in the sequential division 1 and finally came up with a sub-protocol named PR_4. Sub-protocol PR_4 would satisfy the global functionality (which is the recovery of the failed site) of the sub-protocols in sequential division 1.

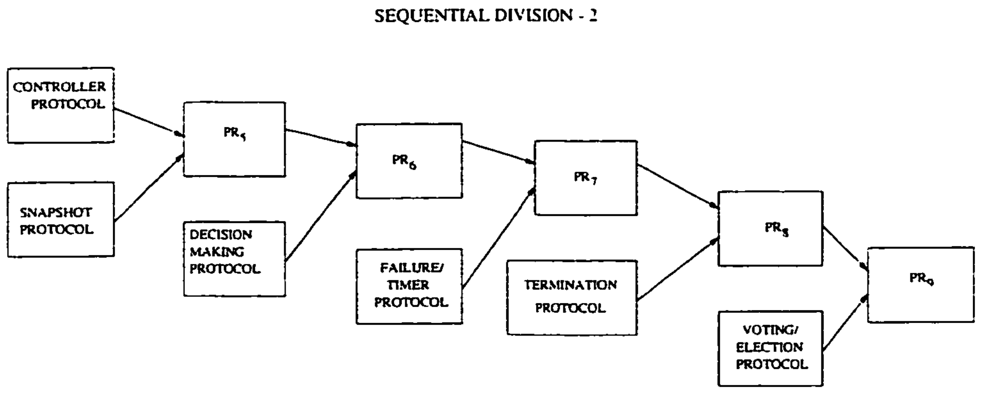


Figure 3.5: Modular Dependencies of Building Blocks based on Seq. Division-2

In Figure 3.5, we follow the similar steps as outlined in the case of sequential division 1 and came up with the final sub-protocol PR_9 which would satisfy the global functionality of the second set of sequential building blocks.

Chapter 4

Formal Analysis of Three Phase Commit Protocol

In this chapter, we do the formal analysis of the three phase commit protocol based on the building blocks identified in the previous chapter. In particular, we show how a global property of the three phase commit protocol could be decomposed into sub-properties and subsequently provable by one of the composed protocols (building blocks) for each of the sub-properties.

4.1 Analysis of the Global Properties of 3PC

In this section, we show how a global property of the three phase commit protocol could be decomposed into sub-properties and subsequently provable by one of the composed protocols (building blocks) for each of the sub-properties. It is important to note that all these identified sub-protocols do not come into picture at all time in the overall operation of the 3PC. Some of these sub-protocols are used in satisfying specific requirements of the 3PC protocol operation. Given this fact, we have identified three different modes in the overall operation of 3PC where the sub-protocols have been utilized for meeting specific requirements. The three global properties associated with these three modes are:

- *Serializability of Transactions*
- *Consistent State Maintenance*
- *Roll-Back Recovery*

In subsequent subsections, we show how a few specific sub-protocols can be composed together to achieve each one of the three identified global properties mentioned above, namely the Serializability of Transactions, Consistent State Maintenance and the Roll-Back Recovery.

4.1.1 Analysis of Global Property–1: Serializability of Transactions

In a distributed system several users may read and update information concurrently. If the operations of the various user transactions are not interleaved in a correct fashion, several undesirable situations may arise, such as creation of inconsistent data or users receiving inconsistent information. Serializability enables the coordination of concurrent accesses of data by the various transactions so that the effect is the same as if the transactions ran one at a time.

The serializability property states that the effect of executing a collection of atomic actions is equivalent to some serial schedule in which the actions are executed one after another. The technique we have employed for implementing serializability is by two phase locking mechanism. In this scheme, a lock is associated with each shared resource, with the requirement that a lock be acquired prior to any access of the associated resource. In order for the locking to be employed, we need to have a stable storage medium with undo and redo mechanisms, a procedure to achieve consensus and a reliable communication among the various processors in the distributed system.

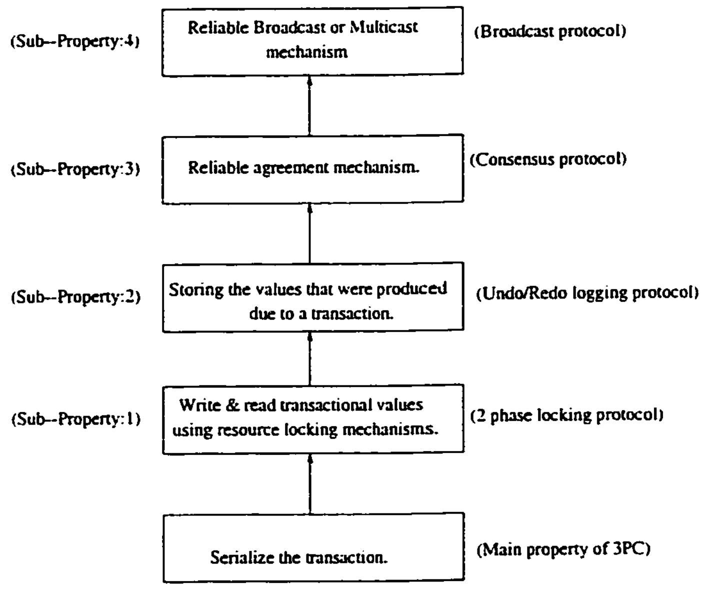

(Sub-Property:4)
Reliable Broadcast or Multicast mechanism
(Broadcast protocol)
(Sub-Property:3)
Reliable agreement mechanism.
(Consensus protocol)
(Sub-Property:2)
Storing the values that were produced due to a transaction.
(Undo/Redo logging protocol)
(Sub-Property:1)
Write & read transactional values using resource locking mechanisms.
(2 phase locking protocol)
Serialize the transaction.
(Main property of 3PC)

Figure 4.1: Serializability of Transactions: Dependencies on Sub-Protocol Properties

In Figure 4.1, for the main global property of *serializability of transactions* to be true, sub-property: 1 must be true, i.e, *write and read transactional values using resource locking mechanism*, which can be proved using the *two phase locking* sub-protocol. Now for sub-property: 1 to exist, sub-property: 2 must be valid, i.e, *store the values that were produced due to a transaction*, which can be verified using the *undo/redo logging* sub-protocol. Again, to realize sub-property: 2, sub-property: 3 must be true, i.e, *a reliable agreement mechanism among the sites or participants*, which is provable by the *consensus* sub-protocol. Finally for all the above-said sub-properties to be realized, sub-property: 4 must be absolutely possible, i.e, *a reliable broadcast or multicast mechanism*, provable by the *broadcast* sub-protocol.

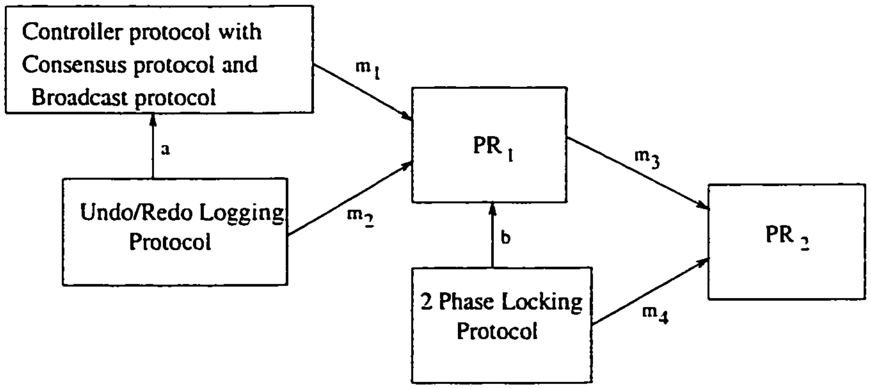

Controller protocol with Consensus protocol and Broadcast protocol
a
Undo/Redo Logging Protocol
m1
m2
PR1
b
2 Phase Locking Protocol
m3
m4
PR2

Figure 4.2: Composition of Sub-protocols to attain Serializability Property

Figure 4.2 illustrates the dependency of the global property of *serializability of transactions* on various sub-properties. In Figure 4.2, at first the *controller* and *undo/redo logging* sub-protocols are modularly composed to come up with a new sub-protocol PR_1. PR_1 now has the properties of both the controller and undo/redo logging sub-protocols. Next we modularly compose the PR_1 and *two phase locking* sub-protocols to come up with the sub-protocol PR_2. Now PR_2 would satisfy the global property of *serializability of transactions* of the 3PC protocol. Next we give the category-theoretic reasoning for this property using morphisms and colimits.

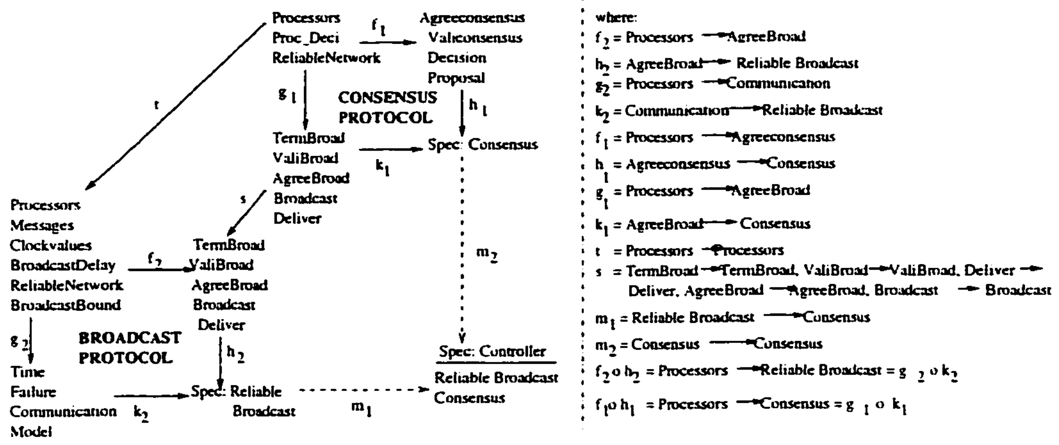

Processors
Proc_Deci
ReliableNetwork
f1
Agreeconsensus
Valiconsensus
Decision
Proposal
g1
CONSENSUS PROTOCOL
h1
TermBroad
ValiBroad
AgreeBroad
Broadcast
Deliver
k1
Spec: Consensus
t
s
Processors
Messages
Clockvalues
BroadcastDelay
ReliableNetwork
BroadcastBound
f2
TermBroad
ValiBroad
AgreeBroad
Broadcast
Deliver
m2
g2
BROADCAST PROTOCOL
h2
Time
Failure
Communication
Model
k2
Spec: Reliable Broadcast
m1
Spec: Controller
Reliable Broadcast
Consensus
where:
f2 = Processors → AgreeBroad
h2 = AgreeBroad → Reliable Broadcast
g2 = Processors → Communication
k2 = Communication → Reliable Broadcast
f1 = Processors → Agreeconsensus
h1 = Agreeconsensus → Consensus
g1 = Processors → AgreeBroad
k1 = AgreeBroad → Consensus
t = Processors → Processors
s = TermBroad → TermBroad, ValiBroad → ValiBroad, Deliver → Deliver, AgreeBroad → AgreeBroad, Broadcast → Broadcast
m1 = Reliable Broadcast → Consensus
m2 = Consensus → Consensus
f2 o h2 = Processors → Reliable Broadcast = g2 o k2
f1 o h1 = Processors → Consensus = g1 o k1

Figure 4.3: Composition of Broadcast and Consensus Protocols

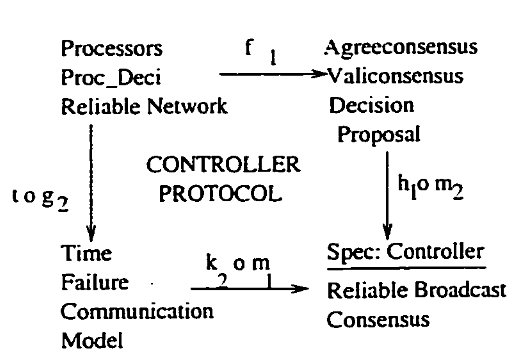
Modularly Composing the Sub-Protocols via Category Theoretic operations

In figure 4.3, each of the modules are depicted in the same way as figure 2.4 with the four sets of objects as parameters, export, import and body (spec) with their corresponding morphisms. The Broadcast protocol has the *processor information, type of messages, clock value, network reliability assumption, delay and bound features* of broadcast as its parameters. It imports the basic primitives like *Time, Failure, Communication and Model* from previous specifications. The specific properties of this protocol are shown as the export entities, namely *TermBroad* defining the termination property of the broadcast protocol, *ValiBroad* defining the validation property, *AgreeBroad* showing the common agreement property and *Broadcast and Deliver* illustrating the basic operations that are to be employed while defining the above mentioned properties.

There are three kinds of morphisms in the figure, namely the specification morphisms *t, s*, mapping morphisms f_1, g_1, h_1, k_1, f_2, g_2, h_2 *and* k_2 and colimit morphisms m_1, m_2. The specification morphism t relates the parameters of both the composing modules. For example, in the figure, t maps the parameters of consensus protocol with the parameters of broadcast protocol. The mapping morphisms help in the mapping of the four objects inside a module. For example, in the figure, f_1 maps Processors in the distributed system to the property `Agreeconsensus`, meaning that the property `Agreeconsensus` has processors in its property definition. This could be clearly appreciated from the Specware specification wherein `def Agreeconsensus` uses the information about the various processors in the distributed system for its definition. The colimit morphisms help in mapping the final specifications of both the modules. All other modules have also been specified in the same way as explained above.

In Figure 4.3, the *broadcast* protocol and *consensus* protocol are modularly composed to come up with the *controller* protocol. Composing two modules in the way shown above is possible only if each of these two modules commute individually through the specification morphism relationship.

Figure 4.4 shows the composed diagram (module) of the controller protocol. Now *controller* block has the properties of both broadcast and consensus sub-blocks, i.e, reliable in broadcasting and also uniform in decision-making (consensus). We can see from the figure that the final composed module also commutes by the specifi-



where:
f_1 = Processors $\longrightarrow$ Agreeconsensus
$h_1 \circ m_2$ = Agreeconsensus $\longrightarrow$ Consensus
$t \circ g_2$ = Processors $\longrightarrow$ Communication
$k_2 \circ m_1$ = Communication $\longrightarrow$ Consensus
$f_1 \circ h_1 \circ m_2$ = Processors $\longrightarrow$ Consensus
$t \circ g_2 \circ k_2 \circ m_1$ = Processors $\longrightarrow$ Consensus

Figure 4.4: Composed Diagram of Controller Protocol

cation morphism relationship which proves the correctness of the composition and also guarantees the proper working of the composed module if it is to be reused or inherited elsewhere.

Note that, when this composed module is reused elsewhere for the purpose of further composition with other modules, the morphism notations are renamed for the purpose of simplicity and consistency in naming. For example, the composed diagram of controller protocol in figure 4.4 has the morphisms f_1, $t \circ g_2$, $h_1 \circ m_2$ and $k_2 \circ m_1$, whereas, when it is reused for composing with another module namely the Undo/ Redo protocol (as shown in figure 4.5), the morphisms have been renamed as f_2, g_2, h_2 *and* k_2. This has been done just for the purpose of keeping monotony in the naming of morphisms in all the figures and that there is no change in the actual morphism mappings. For example, Morphism f_1 (as in figure 4.4) = Morphism f_2 (as in figure 4.5) = Processors $\rightarrow$ Agreeconsensus.

The *controller* protocol and *undo/redo log* protocol are modularly composed to come up with a new protocol called PR_1, as depicted in Figure 4.5. In the figure, the

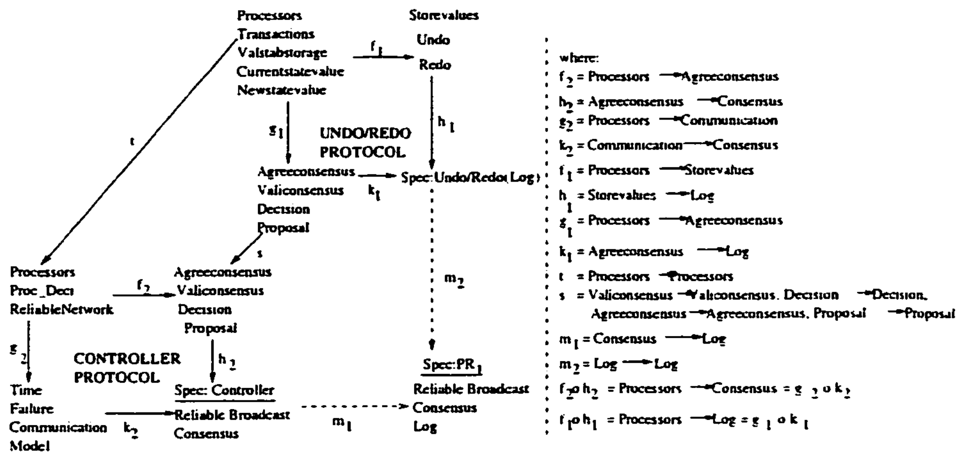

Processors
Transactions
Valstabstorage
Currentstatevalue
Newstatevalue
f_1
Storevalues
Undo
Redo
g_1
UNDO/REDO PROTOCOL
h_1
Agreeconsensus
Valiconsensus
Decision
Proposal
k_1
Spec:Undo/Redo(Log)
t
s
Processors
Proc_Deci
ReliableNetwork
f_2
Agreeconsensus
Valiconsensus
Decision
Proposal
m_2
g_2
CONTROLLER PROTOCOL
h_2
Time
Failure
Communication
Model
k_2
Spec: Controller
Reliable Broadcast
Consensus
m_1
Spec:PR$_1$
Reliable Broadcast
Consensus
Log
where:
f_2 = Processors → Agreeconsensus
h_2 = Agreeconsensus → Consensus
g_2 = Processors → Communication
k_2 = Communication → Consensus
f_1 = Processors → Storevalues
h_1 = Storevalues → Log
g_1 = Processors → Agreeconsensus
k_1 = Agreeconsensus → Log
t = Processors → Processors
s = Valiconsensus → Valiconsensus. Decision → Decision. Agreeconsensus → Agreeconsensus. Proposal → Proposal
m_1 = Consensus → Log
m_2 = Log → Log
$f_2 o h_2$ = Processors → Consensus = $g_2 o k_2$
$f_1 o h_1$ = Processors → Log = $g_1 o k_1$

Figure 4.5: Composition of Controller and Undo/Redo Protocols

parameter `Valstabstorage` in the Undo/Redo protocol is the stable storage medium for the processor state values.

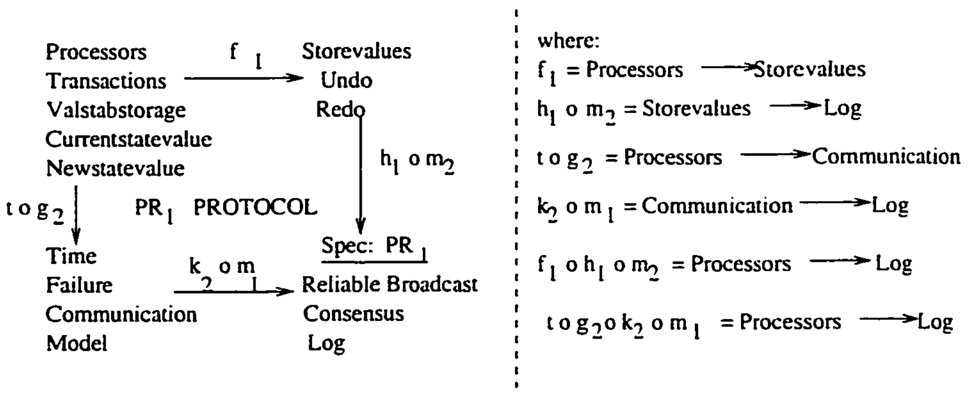

Processors
Transactions
Valstabstorage
Currentstatevalue
Newstatevalue
f_1
Storevalues
Undo
Redo
h_1 o m_2
t o g_2
PR$_1$ PROTOCOL
Spec: PR$_1$
Time
Failure
Communication
Model
k_2 o m_1
Reliable Broadcast
Consensus
Log
where:
f_1 = Processors → Storevalues
h_1 o m_2 = Storevalues → Log
t o g_2 = Processors → Communication
k_2 o m_1 = Communication → Log
f_1 o h_1 o m_2 = Processors → Log
t o g_2o k_2 o m_1 = Processors → Log

Figure 4.6: Composed Diagram of PR_1 Protocol

Figure 4.6 shows the composed diagram (module) of the PR_1 protocol. Now PR_1 block has the properties of both controller and undo/redo log sub-blocks, i.e, reliable in broadcasting, uniform in decision-making (consensus) and also store the values of the transaction onto a stable storage medium.

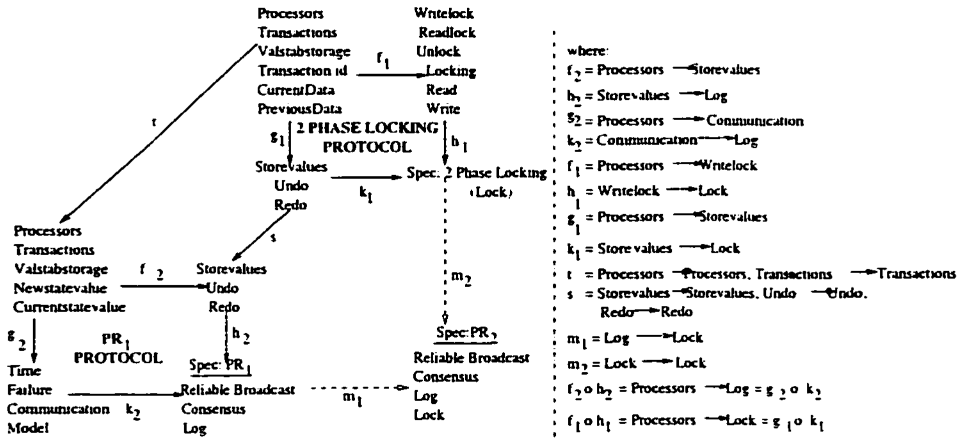


Figure 4.7: Composition of PR_1 and 2 Phase Locking Protocols

Next we modularly compose the PR_1 protocol and *two phase locking* protocol to arrive at PR_2 protocol (See figure 4.7).

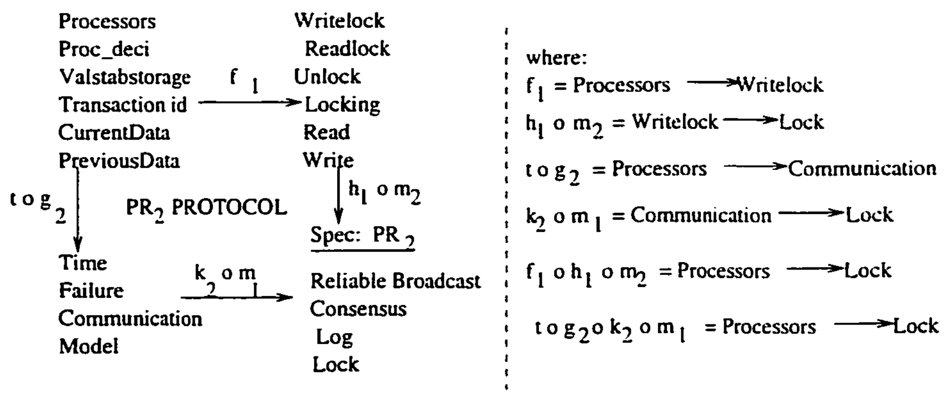


Figure 4.8: Composed Diagram of PR_2 Protocol

Figure 4.8 shows the final composed diagram of PR_2 protocol. Now the composed PR_2 block has the locking properties in its final specification along with the other properties which it is dependent on namely reliable broadcast, consensus and logging. Since it has the locking mechanism, the PR_2 protocol would satisfy the *serializability property* of the three phase commit protocol thereby ensuring the coordination of concurrent accesses of data by the various transactions so that the effect is the same as if the transactions ran one at a time.

By selectively identifying sorts and operations out of three sub-protocols, namely *controller protocol, undo/redo logging protocol,* and *two phase locking protocol* for different morphism definitions, we have essentially constructed a sub-protocol PR_2 which satisfies the *serializability of transactions* property.

4.1.2 Compositional Verification of Global Property–1

Serializability enables the coordination of concurrent accesses of data by the various transactions so that the effect is the same as if the transactions ran one at a time. The serializability property states that the effect of executing a collection of atomic actions is equivalent to some serial schedule in which the actions are executed one after another.

The formal representation of the serializability definition is given below:

$$\forall(p, q : Processors, m : Messages, t : Transactions, v, commit, abort : Proc.Decision)$$

$$\forall(T : Clockvalues, N : Transactionid, X : Transactionvalueinstablestorage)$$

$$\forall(y : Currentstatevalue, z : Newstatevalue, Y : Currentdata, Z : Previousdata) :$$

$$correct(p, T) \wedge correct(q, T) \wedge$$

$$if((Deliver(p, m, T) \rightarrow Deliver(q, m, (T + \gamma + \delta))) \wedge$$

$$(Decision(p, v, T) \rightarrow Decision(q, v, T)) \wedge$$

$$(Undo(t, abort, X, y) \wedge Redo(t, commit, X, z) \rightarrow Log(t, X, z))$$

$$then(\neg(Write(T, Y, X)) \wedge \neg(Locking(N, Y)) \wedge Unlock(N, Z))$$

$$\rightarrow (Read(T, Y, X) \wedge Locking(N, Y))$$

$$else(\neg(Read(T, Y, X)) \wedge \neg(Locking(N, Y)) \wedge Unlock(N, Z))$$

$$\rightarrow (Write(T, Y, X) \wedge Locking(N, Y)))$$

where,

- γ is the maximum broadcastdelay.
- δ is the broadcastbound.
- Network is assumed to be a reliable one without partitions.
- `correct(p,T)`: processor p is correct at clock value T.
- `correct(q,T)`: processor q is correct at clock value T.
- `Broadcast(p,m,T)`: processor p broadcasts message m at real-time T.
- `Deliver(p,m,T)`: processor p delivers message m at real-time T.

In order to prove the above mentioned global property, we need to first establish the sub properties which the global property is dependent on. So for the purpose of a proper locking mechanism, a stable storage medium with a standard undo and redo operation is needed which in turn depends on the properties of a consensus procedure which is not possible without the properties of a reliable communication among the various processors in the distributed system. Hence we need to first prove the properties of the reliable broadcast mechanism.

The properties provided by the reliable broadcast are the following:

•`Termination`:- Every correct process eventually delivers some message.

SP1: TermBroad

$$\exists(p, m, T) : correct(p) \wedge Broadcast(p, m, T) \rightarrow$$

$$\forall(q, \gamma) : correct(q) \wedge Deliver(q, m, (T + \gamma))$$

•`Validity`:- If a correct process broadcasts a message m, then all correct processes eventually deliver m.

SP2: ValiBroad

$$\exists(p, m, T) : correct(p) \wedge Broadcast(p, m, T) \rightarrow$$

$$\forall(q, \gamma, \delta) : correct(q) \wedge Deliver(q, m, (T + \gamma + \delta)) \wedge (\gamma < \delta)$$

•`Uniform Agreement`:- If a process delivers a message m, then all correct processes eventually deliver m.

SP3: AgreeBroad

$$\exists(p)\forall(m, T) : correct(p) \wedge Deliver(p, m, T) \rightarrow$$

$$\forall(q, \gamma, \delta) : correct(q) \wedge Deliver(q, m, (T + \gamma + \delta))$$

Utilizing the following operational definitions, we state the various properties of the consensus protocol:

• `Decision(p,v,T)`: processor p arrives at a final decision v which is either to commit the transaction or to abort it.

• `Proposal(p,v,T)`: processor p proposes a decision v at real time T to either commit or to abort the transaction.

The properties are:

•`Validity`:-If a process decides X, then X was proposed by some process.

SP4: Validconsensus

$$\forall(p, q, T)\exists(v) : Decision(p, v, T) \rightarrow Proposal(q, v, T)$$

This means that if a processor p makes a decision v which could be an abort or a commit, then that decision was initially made by another processor q.

•`Uniform Agreement`:-No two processes decide differently.

SP5: Agreeconsensus

$$\forall(p, q, v, T) : Decision(p, v, T) \rightarrow Decision(q, v, T)$$

Here both the processors p and q make the same decision whether to commit or abort.

Now that we have stated the properties of reliable communication medium and a consensus procedure, we will next state the properties of the stable storage medium with the undo/redo operation.

The various operational definitions are:

• `Undo(t,v,X,y)`: Undo the current state value y taken during the transaction t based on the processor decision v (abort) from the stable storage X before writing into it.

• `Redo(t,v,X,z)`: Redo the new state value z taken during the transaction t based on the processor decision v (commit) into the stable storage X before committing the transaction.

• `Log(t,X,z)`: Stores the new state value z of the processor got during the transaction t onto the stable storage X.

The property to be satisfied is:

SP6: Storevalues

$$\forall(p, q, t)\forall(commit, abort)\forall(y, z) : Undo(t, abort. X, y)\wedge$$

$$Redo(t, commit, X, z) \rightarrow Log(t, X, z)$$

Log is represented as a sequence of entries of the form $[t,X,v]$, where z is the value that transaction t wrote into data item X. Log also contains sets of committed and aborted transactions. Each change to the database causes a log entry. Log has initial and final states (or) initial and differential state transitions (or) initial and operations performed. When volatile memory is lost in a system failure, the protocol examines the log, finds the last committed values of all data items that have been updated and restores them as new volatile values of these data items. Entry is undone in the stable log before writing into it and redone before the transaction is committed.

Having stated the individual properties of reliable communication medium, consensus and logging procedures, we can now establish the global property of "serializability of transactions". This property allows a transaction to access a data item only if it is currently holding a lock on that item. Hence we need a method to hold a lock on an item (shared resource log). Note that the type of lock depends on whether the transaction will read or write to the item.

The various operational primitives of a locking protocol are:

- `Read(t,Y,X)`: Reads the current data Y of the transaction t from the stable storage medium X.
- `Write(t,Y,X)`: Writes the current data Y of the transaction t onto the stable storage medium X.
- `Locking(N,Y)`: Locks the current state value Y with N as the transaction id.
- `Unlock(N,Z)`: Unlocks the previous state value Z with N as the transaction id.

The various properties are:

SP7: Writelock

$$\forall(p,q,t)\forall(N,X,Y,Z) : \neg(Read(t,Y,X)) \wedge \neg(Locking(N,Y)) \wedge Unlock(N,Z)$$

$$\rightarrow Write(t,Y,X) \wedge Locking(N,Y)$$

Only one transaction at a time may write lock an item and it is implemented by a simple 1 bit write lock flag. Write lock should enforce complete mutual exclusion on the item. An item is write locked if the item is not currently read locked (resource) and that the item had been already unlocked and free to be locked again.

Multiple transactions may be read locking an item at the same time. Read lock is implemented by using a read counter which holds the number of transactions currently holding a read lock.

SP8: Readlock

$$\forall(p,q,t)\forall(N,X,Y,Z) : \neg(Write(t,Y,X)) \wedge \neg(Locking(N,Y)) \wedge Unlock(N,Z)$$

$$\rightarrow Read(t, Y, X) \wedge Locking(N, Y)$$

An item is read locked if there are no current write locks on the item and that the item is free to be locked again. This property helps in the proper logging of data during the active transaction. The locking protocol allows a lock (shared resource log) to be acquired before the transaction and then asks for unlocking once the transaction is over. This way, serializability would be maintained while logging data onto the stable storage.

As we have composed modules to get a composite protocol which satisfies the Serializability property, it was of utmost importance that all the morphisms we have defined along the way essentially ensure that all the necessary attributes and operations from each individual building block protocols are carried on/included in the composite specification. The fact that we have successfully composed the specification which satisfies the specified requirements in turn guarantees that the morphisms have been correctly defined. Through this strong mathematical framework, we have a capability to perform a backward propagation, i.e., essentially, we have the traceability capability. We elaborate on this in our subsequent discussions.

Nuances of Morphisms in Compositional Verification

In the formal representation of `Serializability` definition, the conditional statement $Deliver(p, m, T) \rightarrow Deliver(q, m, (T+\gamma+\delta))$ is satisfied by the `AgreeBroad` property of the `Reliable Broadcast` protocol. The operation which extends this

`AgreeBroad` property gets mapped from the original `Reliable Broadcast` protocol block to the composite specification namely the PR_2 protocol via the morphism s (figure 4.3) which is $AgreeBroad \rightarrow AgreeBroad$ meaning that the `AgreeBroad` property of `Reliable Broadcast` protocol gets exported to the `Consensus` protocol as its imported property and the morphism h_2 (figure 4.3) which is $AgreeBroad \rightarrow ReliableBroadcast$ meaning that the `AgreeBroad` property ultimately maps it to the final property in the specification. From here on, this final property namely the ReliableBroadcast in the specification gets carried on till the global composite protocol PR_2 through the morphisms m_1 as labeled in figures 4.3, 4.5 and 4.7.

The next conditional statement in the definition namely $Decision(p, v, T) \rightarrow Decision(q, v, T)$ is satisfied by the `Agreeconsensus` property of the `Consensus` protocol. The operation which extends this `Agreeconsensus` property gets mapped from the original `Consensus` protocol block to the composite specification namely the PR_2 protocol via the morphism s (figure 4.5) which is $Agreeconsensus \rightarrow Agreeconsensus$ meaning that the `Agreeconsensus` property of `Consensus` protocol gets exported to the `Undo/Redo` protocol as its imported property and the morphism h_2 (figure 4.5) which is $Agreeconsensus \rightarrow Consensus$ meaning that the `Agreeconsensus` property ultimately maps it to the final property in the specification. From here on, this final property namely the Consensus in the specification gets carried on till the global composite protocol PR_2 through the morphisms m_1 as labeled in figures 4.5 and 4.7.

The third condition in the definition namely $Undo(t, abort, X, y) \wedge Redo(t,$

commit, $X, z) \rightarrow Log(t, X, z)$ is satisfied by Storevalues property of the Undo/Redo protocol. The operation which extends this Storevalues property gets mapped from the original Undo/Redo protocol block to the composite specification namely the PR_2 protocol via the morphism s (figure 4.7) which is $Storevalues \rightarrow Storevalues$ meaning that the Storevalues property of Undo/Redo protocol gets exported to Two Phase Locking protocol as its imported property and the morphism h_2 (figure 4.7) which is $Storevalues \rightarrow Log$ meaning that the Storevalues property ultimately maps it to the final property in the specification. From here on, this final property namely the Log in the specification gets carried on till the global composite protocol PR_2 through the morphism m_1 as labeled in figure 4.7. Now if all these conditional statements are satisfied, then the final property statement in the serializability definition would be also satisfied by the fact that the Two Phase Locking protocol which helps in concurrency of data transaction depends on the attributes of the Undo/Redo protocol by importing its properties as shown in figure 4.7.

4.1.3 Analysis of Global Property–2: Consistent State Maintenance

Let the system state at a particular time be defined as a history of events constituting the set of all components (site) states at that time. Then, a system state is said to be consistent if for every event corresponding to the receipt of a message in

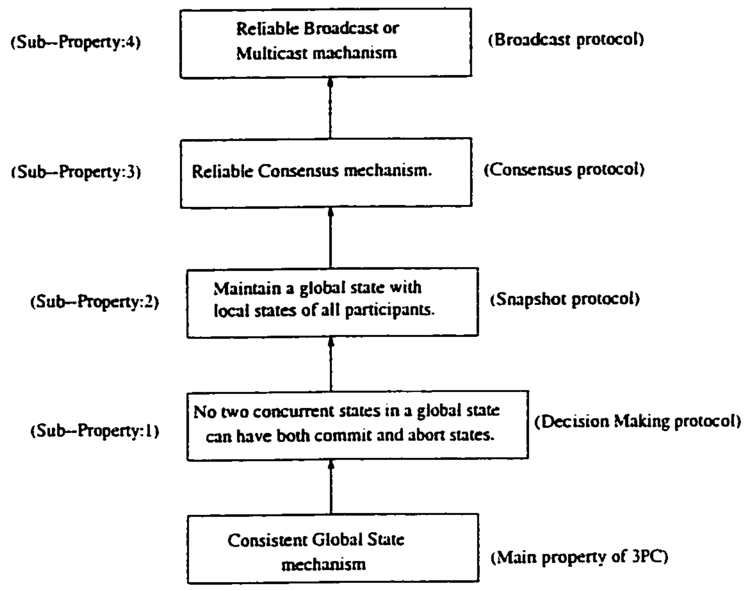

(Sub–Property:4)
Reliable Broadcast or Multicast machanism
(Broadcast protocol)
(Sub–Property:3)
Reliable Consensus mechanism.
(Consensus protocol)
(Sub–Property:2)
Maintain a global state with local states of all participants.
(Snapshot protocol)
(Sub–Property:1)
No two concurrent states in a global state can have both commit and abort states.
(Decision Making protocol)
Consistent Global State mechanism
(Main property of 3PC)

Figure 4.9: Consistent State Maintenance: Dependencies on Sub-Protocol Properties

the state, the event corresponding to the sending of that message is also included.

As depicted in Figure 4.9, for the main global property of *consistent state maintenance* to be true, sub-property: 1 must be true, i.e, *no two concurrent states in a global state should have both commit and abort states*, which can be proved using the *decision making* sub-protocol. Now for sub-property: 1 to exist, sub-property: 2 must be valid, i.e, *a global state should have local states of all participants*, which can be verified using the *snapshot* sub-protocol. Again, to realize sub-property: 2, sub-property: 3 must be true, i.e, *a reliable agreement mechanism among the sites or participants*, which is provable by the *consensus* sub-protocol. Finally for all the above-said sub-properties to be realized, sub-property: 4 must be absolutely possible, i.e, *a reliable broadcast or multicast mechanism*, provable by the *broadcast* sub-protocol.

Figure 4.10 illustrates the dependency of the global property of *consistent*

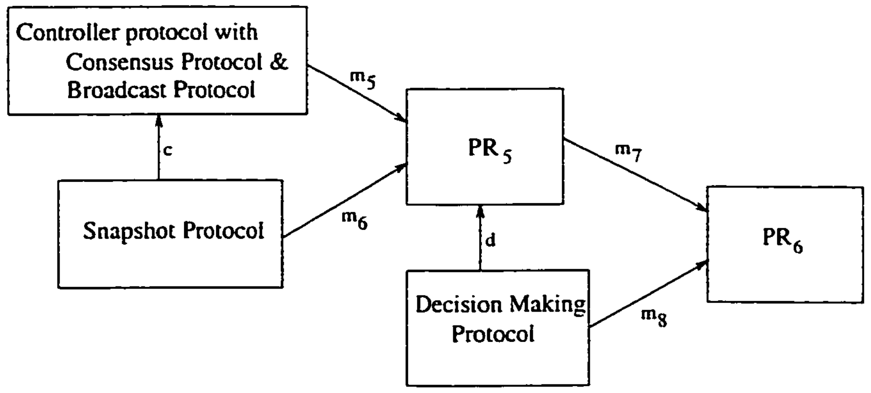

Controller protocol with Consensus Protocol & Broadcast Protocol
c
Snapshot Protocol
m5
m6
PR5
d
Decision Making Protocol
m7
m8
PR6

Figure 4.10: Composition of Sub-protocols to attain Consistent State Property

state maintenance on various sub-properties. Now in Figure 4.10, at first the *controller* sub-protocol and *snapshot* sub-protocol are modularly composed to come up with a new sub-protocol PR_5. PR_5 now has the properties of both the controller and snapshot sub-protocol. Next we modularly compose the PR_5 sub-protocol and *decision making* sub-protocol to come up with the sub-protocol PR_6. Now the new sub-protocol PR_6 would satisfy the global property of *consistent global state maintenance* of the three phase commit protocol. Next we give the category-theoretic reasoning for this property using morphisms and colimits.

Modularly Composing the Sub-Protocols via Category Theoretic operations

For this property, the composition of broadcast and consensus sub-protocols to form the controller sub-protocol is the same as that of the first global property. However for the sake of continuous readability, we have again shown it here.

In Figure 4.11, the *broadcast* protocol and *consensus* protocol are modularly composed to come up with the *controller* protocol. Composing two modules in the

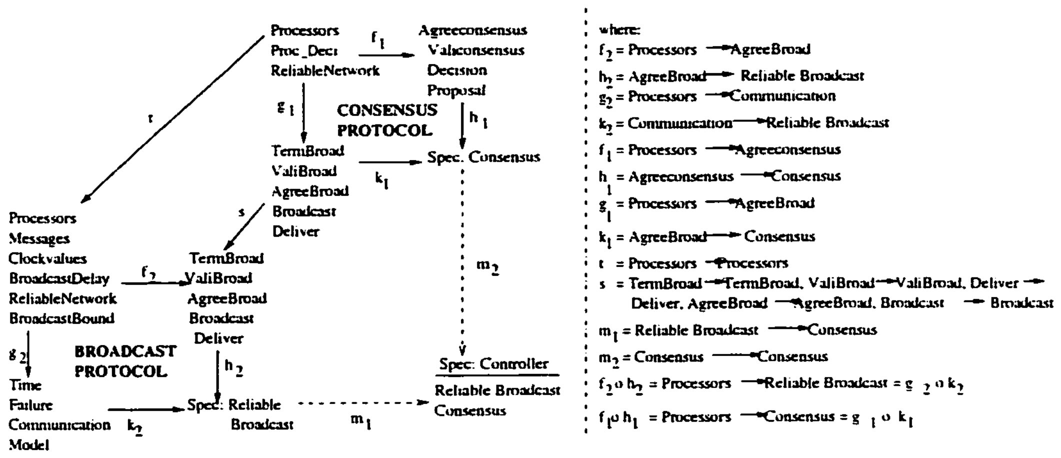

Processors
Proc_Deci
ReliableNetwork
f1
Agreeconsensus
Valiconsensus
Decision
Proposal
g1
CONSENSUS PROTOCOL
h1
TermBroad
ValiBroad
AgreeBroad
Broadcast
Deliver
k1
Spec. Consensus
t
s
m2
Processors
Messages
Clockvalues
BroadcastDelay
ReliableNetwork
BroadcastBound
f2
TermBroad
ValiBroad
AgreeBroad
Broadcast
Deliver
g2
BROADCAST PROTOCOL
h2
Time
Failure
Communication
Model
k2
Spec: Reliable Broadcast
m1
Spec: Controller
Reliable Broadcast
Consensus
where:
f2 = Processors → AgreeBroad
h2 = AgreeBroad → Reliable Broadcast
g2 = Processors → Communication
k2 = Communication → Reliable Broadcast
f1 = Processors → Agreeconsensus
h1 = Agreeconsensus → Consensus
g1 = Processors → AgreeBroad
k1 = AgreeBroad → Consensus
t = Processors → Processors
s = TermBroad → TermBroad, ValiBroad → ValiBroad, Deliver → Deliver, AgreeBroad → AgreeBroad, Broadcast → Broadcast
m1 = Reliable Broadcast → Consensus
m2 = Consensus → Consensus
f2 o h2 = Processors → Reliable Broadcast = g2 o k2
f1 o h1 = Processors → Consensus = g1 o k1

Figure 4.11: Composition of Broadcast and Consensus Protocols

way shown above is possible only if each of these two modules commute individually through the specification morphism relationship.

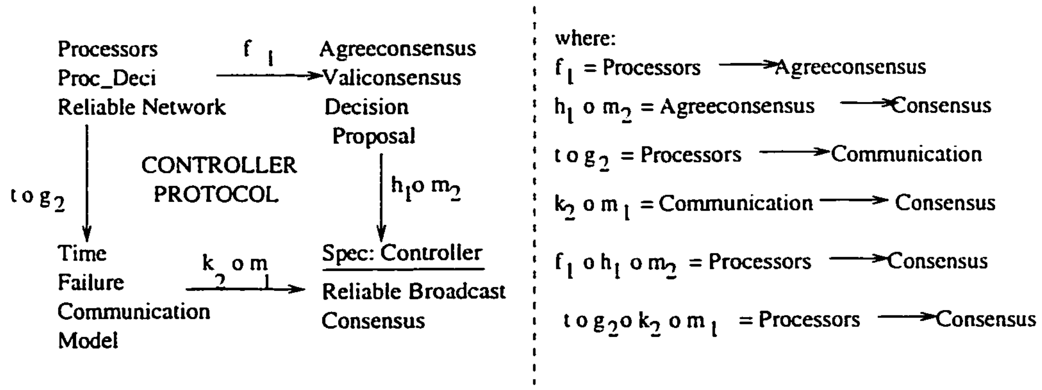

Processors
Proc_Deci
Reliable Network
f1
Agreeconsensus
Valiconsensus
Decision
Proposal
CONTROLLER PROTOCOL
t o g2
h1 o m2
Time
Failure
Communication
Model
k2 o m1
Spec: Controller
Reliable Broadcast
Consensus
where:
f1 = Processors → Agreeconsensus
h1 o m2 = Agreeconsensus → Consensus
t o g2 = Processors → Communication
k2 o m1 = Communication → Consensus
f1 o h1 o m2 = Processors → Consensus
t o g2 o k2 o m1 = Processors → Consensus

Figure 4.12: Composed Diagram of Controller Protocol

Figure 4.12 shows the composed diagram (module) of the controller protocol. Now *controller* block has the properties of both broadcast and consensus sub-blocks, i.e, reliable in broadcasting and also uniform in decision-making (consensus). We can see from the figure that the final composed module also commutes by the specification morphism relationship which proves the correctness of the composition and also guarantees the proper working of the composed module if it is to be reused or inherited elsewhere.

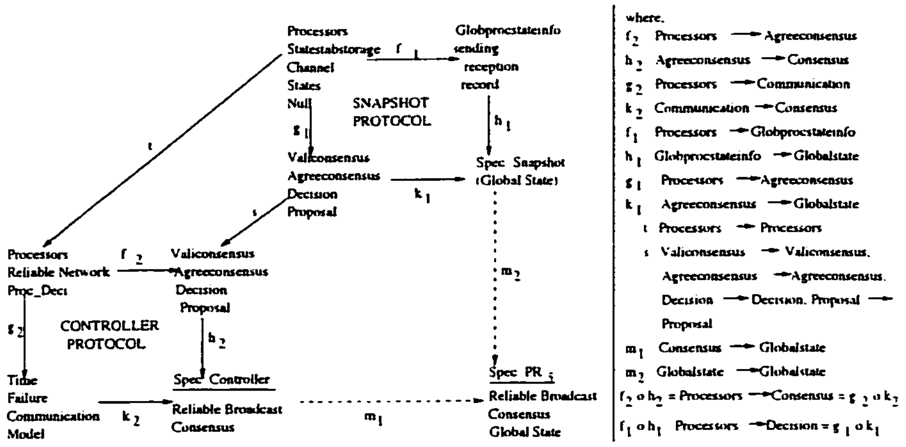

Processors
Statestabstorage
Channel
States
Null
Globprocstateinfo
sending
reception
record
f_1
SNAPSHOT PROTOCOL
g_1
h_1
Valiconsensus
Agreeconsensus
Decision
Proposal
k_1
Spec Snapshot (Global State)
t
s
Processors
Reliable Network
Proc_Deci
f_2
Valiconsensus
Agreeconsensus
Decision
Proposal
m_2
g_2
CONTROLLER PROTOCOL
h_2
Time
Failure
Communication
Model
k_2
Spec Controller
Reliable Broadcast
Consensus
m_1
Spec PR 5
Reliable Broadcast
Consensus
Global State
where,
f_2 Processors → Agreeconsensus
h_2 Agreeconsensus → Consensus
g_2 Processors → Communication
k_2 Communication → Consensus
f_1 Processors → Globprocstateinfo
h_1 Globprocstateinfo → Globalstate
g_1 Processors → Agreeconsensus
k_1 Agreeconsensus → Globalstate
t Processors → Processors
s Valiconsensus → Valiconsensus, Agreeconsensus → Agreeconsensus, Decision → Decision, Proposal → Proposal
m_1 Consensus → Globalstate
m_2 Globalstate → Globalstate
$f_2 \circ h_2$ = Processors → Consensus = $g_2 \circ k_2$
$f_1 \circ h_1$ Processors → Decision = $g_1 \circ k_1$

Figure 4.13: Composition of Controller and Snapshot Protocols

The *controller* protocol and *snapshot* protocol are modularly composed to come up with a new protocol called PR_5, as depicted in Figure 4.13.

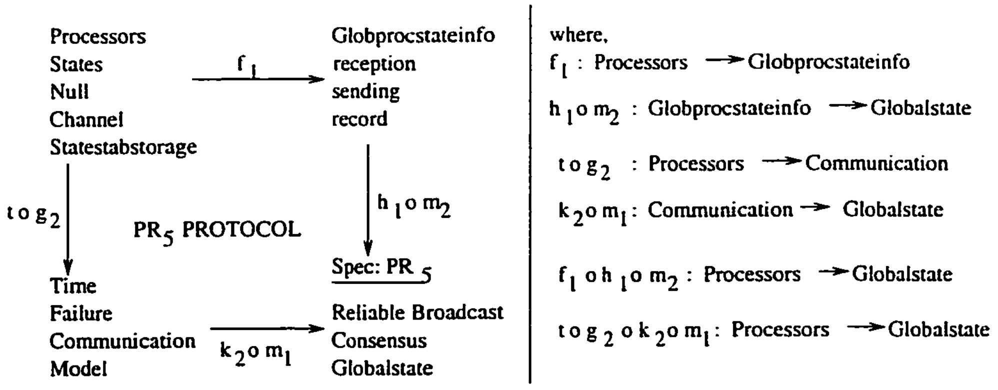

Processors
States
Null
Channel
Statestabstorage
f_1
Globprocstateinfo
reception
sending
record
$t \circ g_2$
PR_5 PROTOCOL
$h_1 \circ m_2$
Spec: PR 5
Time
Failure
Communication
Model
$k_2 \circ m_1$
Reliable Broadcast
Consensus
Globalstate
where,
f_1 : Processors → Globprocstateinfo
$h_1 \circ m_2$: Globprocstateinfo → Globalstate
$t \circ g_2$: Processors → Communication
$k_2 \circ m_1$: Communication → Globalstate
$f_1 \circ h_1 \circ m_2$: Processors → Globalstate
$t \circ g_2 \circ k_2 \circ m_1$: Processors → Globalstate

Figure 4.14: Composed Diagram of PR_5 Protocol

Figure 4.14 shows the composed diagram (module) of the PR_5 protocol. Now PR_5 block has the properties of both controller and snapshot sub-blocks, i.e, reliable in broadcasting, uniform in decision-making (consensus) and information about the local states of all participants.

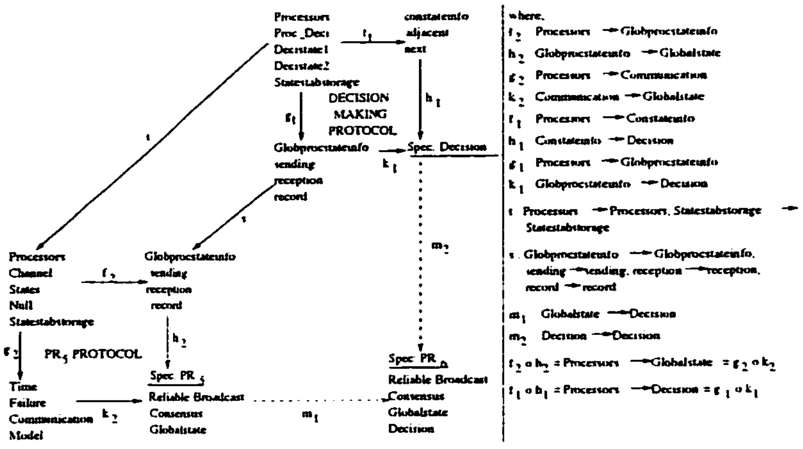
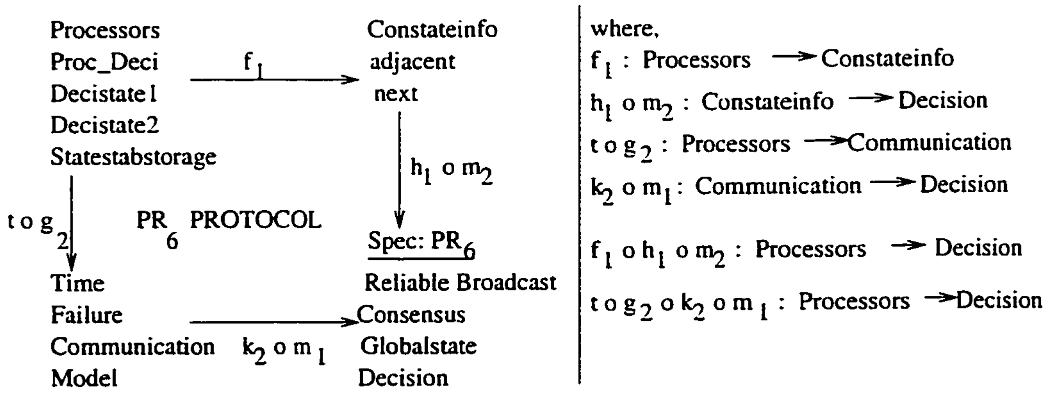


Figure 4.15: Composition of PR_5 and Decision Making Protocols

Next we modularly compose the PR_5 protocol and *decision making* protocol to arrive at PR_6 protocol (See figure 4.15).



Processors
Proc_Deci
Decistate1
Decistate2
Statestabstorage
f1
Constateinfo
adjacent
next
h1 o m2
t o g2
PR6 PROTOCOL
Spec: PR6
Time
Failure
Communication
Model
k2 o m1
Reliable Broadcast
Consensus
Globalstate
Decision
where,
f1 : Processors → Constateinfo
h1 o m2 : Constateinfo → Decision
t o g2 : Processors → Communication
k2 o m1 : Communication → Decision
f1 o h1 o m2 : Processors → Decision
t o g2 o k2 o m1 : Processors → Decision

Figure 4.16: Composed Diagram of PR_6 Protocol

Figure 4.16 shows the final composed diagram of PR_6 protocol. Now the composed PR_6 block has the decision making mechanism in its final specification along with the other properties which it is dependent on namely reliable broadcast, consensus and participants' local state information. Since it has the decision making mechanism, the PR_6 protocol would satisfy the *Consistent State Maintenance* property of the three phase commit protocol.

By selectively identifying sorts and operations out of three sub-protocols, namely *controller protocol, snapshot protocol,* and *decision making protocol* for different morphism definitions, we have essentially constructed a sub-protocol PR_6 which satisfies the *Consistent State Maintenance* property.

4.1.4 Compositional Verification of Global Property–2

This property ensures that the global state comprising of all the local states of the processors in the distributed system is always consistent (i.e.) its state vector doesn't have both a *commit* state and an *abort* state.

The formal representation of the consistent state definition is given below:

$$\forall(p, q : Proc., m, M, N : Messages, v, commit, abort : Proc.Decision, c : Channel)$$

$$\forall(T : Clockvalues, Null : Messages, X : Statestablestorage, s, S : States)$$

$$correct(p, T) \wedge correct(q, T) \wedge$$

$$if((Deliver(p, m, T) \rightarrow Deliver(q, m, (T + \gamma + \delta))) \wedge$$

$$(Decision(p, v, T) \rightarrow Decision(q, v, T)) \wedge$$

$$((record(q, s, M, X) \wedge record(q, s, Null, X)) or (record(q, s, m, X) \wedge$$

$$record(q, s, N, X) \wedge (\neg(reception(q, M, c, p, T)))))$$

$$then((\neg(next(commit, abort))) \wedge adjacent(\neg(commit), commit)))$$

where,

- γ is the maximum broadcastdelay.
- δ is the broadcastbound.
- Network is assumed to be a reliable one without partitions.
- `correct(p,T)`: processor p is correct at clock value T.
- `correct(q,T)`: processor q is correct at clock value T.
- `Broadcast(p,m,T)`: processor p broadcasts message m at real-time T.
- `Deliver(p,m,T)`: processor p delivers message m at real-time T.

In order to prove the above mentioned global property, we need to first establish the sub properties which the global property is dependent on. For the purpose of maintaining a consistent global state, a decision making protocol is needed which contains the rules needed for satisfying the global property, which in turn depends on the properties of a snapshot protocol containing the local states of all the processors in the distributed system. Now this is not possible without the properties of a consensus procedure and a reliable communication among the various processors in the distributed system. Hence we need to first prove the properties of the reliable broadcast mechanism.

The properties provided by the reliable broadcast are the following:

•`Termination`:- Every correct process eventually delivers some message.

SP1: TermBroad

$$\exists(p, m, T) : \mathit{correct}(p) \wedge \mathit{Broadcast}(p, m, T) \rightarrow$$

$$\forall(q, \gamma) : \mathit{correct}(q) \wedge \mathit{Deliver}(q, m, (T + \gamma))$$

•`Validity`:- If a correct process broadcasts a message m, then all correct processes eventually deliver m.

SP2: ValiBroad

$$\exists(p, m, T) : \mathit{correct}(p) \land \mathit{Broadcast}(p, m, T) \rightarrow$$

$$\forall(q, \gamma, \delta) : \mathit{correct}(q) \land \mathit{Deliver}(q, m, (T + \gamma + \delta)) \land (\gamma < \delta)$$

•`Uniform Agreement`:- If a process delivers a message m, then all correct processes eventually deliver m.

SP3: AgreeBroad

$$\exists(p)\forall(m, T) : \mathit{correct}(p) \land \mathit{Deliver}(p, m, T) \rightarrow$$

$$\forall(q, \gamma, \delta) : \mathit{correct}(q) \land \mathit{Deliver}(q, m, (T + \gamma + \delta))$$

Utilizing the following operational definitions, we state the various properties of the consensus protocol:

• `Decision(p,v,T)`: processor p arrives at a final decision v which is either to commit the transaction or to abort it.

• `Proposal(p,v,T)`: processor p proposes a decision v at real time T to either commit or to abort the transaction.

The properties are:

•`Validity`:-If a process decides X, then X was proposed by some process.

SP4: Valiconsensus

$$\forall(p, q, T)\exists(v) : \mathit{Decision}(p, v, T) \rightarrow \mathit{Proposal}(q, v, T)$$

This means that if a processor p makes a decision v which could be an abort or a commit, then that decision was initially made by another processor q.

•`Uniform Agreement`:–No two processes decide differently.

SP5: Agreeconsensus

$$\forall(p, q, v, T) : Decision(p, v, T) \rightarrow Decision(q, v, T)$$

Here both the processors p and q make the same decision whether to commit or abort.

Now that we have stated the properties of reliable communication medium and a consensus procedure, we will next state the properties of the snapshot protocol having the local states of all the processors in the distributed system.

The various operational definitions are:

• `sending(p,m,c,q,T)`: This operator helps in sending a message m from processor p to processor q through channel c at real time T.

• `reception(q,m,c,p,T)`: Processor q receives the message m from processor p through channel c at real time T.

• `record(q,s,m,X)`: Processor q records state s and message m onto the stable storage X.

The property to be satisfied is:

SP6: Globprocstateinfo

$$\forall(p, q, T)\forall(m, M, N, Null)\forall(c, s, S, X) : sending(p, \dot{M}, c, q, T) \wedge record(p, s, N, X)$$

$$\wedge(\neg(sending(p,m,c,q,T+1))) \rightarrow reception(q,M,c,p,T) \rightarrow (if(\neg(record(q,s,M,X)))$$

$$then record(q,s,M,X) \wedge record(q,s,Null,X) else record(q,s,m,X) \wedge record(q,s,N,X)$$

$$\wedge(\neg(reception(q,M,c,p,T))))$$

This property ensures the recording of all the local states of the processors in the distributed system based on the transactions among them.

Having stated the individual properties of reliable communication medium, consensus and state recording procedures, we can now establish the global property of "Consistent State Maintenance". This property ensures that the global state comprising of all the local states of the processors in the distributed system is always consistent (i.e.) its state vector doesn't have both a *commit* state and an *abort* state. In order to check this condition, a procedure containing the set of rules is needed, which is what the decision making protocol has in our approach.

The various operational primitives of a decision making protocol are:

• `next(commit,abort)`: This operation checks whether the decision next to the first one(commit) is an abort or a commit.

• `adjacent(commit,commit)`: This operation checks whether the decision adjacent to the first one(commit) is a commit or an abort.

The property to be satisfied is:

SP7: Constateinfo

$$\forall(p,q)\forall(commit, abort) : \neg(next(commit, abort)) \wedge adjacent(\neg(commit), commit)$$

By this property, its clear that the global state can never have two consecutive local state decisions has a commit and an abort.

As we have composed modules to get a composite protocol which satisfies the consistent state maintenance property, it was of utmost importance that all the morphisms we have defined along the way essentially ensure that all the necessary attributes and operations from each individual building block protocols are carried on/included in the composite specification. The fact that we have successfully composed the specification which satisfies the specified requirements in turn guarantees that the morphisms have been correctly defined. Through this strong mathematical framework, we have a capability to perform a backward propagation, i.e., essentially, we have the traceability capability. We elaborate on this in our subsequent discussions.

Nuances of Morphisms in Compositional Verification

In the formal representation of `Consistent state` definition, the conditional statement $Deliver(p, m, T) \rightarrow Deliver(q, m, (T{+}\gamma{+}\delta))$ is satisfied by the `AgreeBroad` property of the `Reliable Broadcast` protocol. The operation which extends this `AgreeBroad` property gets mapped from the original `Reliable Broadcast` protocol block to the composite specification namely the PR_6 protocol via the morphism s (figure 4.11) which is $AgreeBroad \rightarrow AgreeBroad$ meaning that the `AgreeBroad` property of `Reliable Broadcast` protocol gets exported to the `Consensus` protocol as its imported property and the morphism h_2 (figure 4.11) which is $AgreeBroad \rightarrow$

ReliableBroadcast meaning that the AgreeBroad property ultimately maps it to the final property in the specification. From here on, this final property namely the ReliableBroadcast in the specification gets carried on till the global composite protocol PR_6 through the morphisms m_l as labeled in figures 4.11, 4.13 and 4.15.

The next conditional statement in the definition namely $Decision(p, v, T) \rightarrow Decision(q, v, T)$ is satisfied by the Agreeconsensus property of the Consensus protocol. The operation which extends this Agreeconsensus property gets mapped from the original Consensus protocol block to the composite specification namely the PR_6 protocol via the morphism s (figure 4.13) which is $Agreeconsensus \rightarrow Agreeconsensus$ meaning that the Agreeconsensus property of Consensus protocol gets exported to the Snapshot protocol as its imported property and the morphism h_2 (figure 4.13) which is $Agreeconsensus \rightarrow Consensus$ meaning that the Agreeconsensus property ultimately maps it to the final property in the specification. From here on, this final property namely the Consensus in the specification gets carried on till the global composite protocol PR_6 through the morphisms m_l as labeled in figures 4.13 and 4.15.

The third conditional statement in the consistent state definition is satisfied by the Globprocstateinfo property of the Snapshot protocol. The operation which extends this Globprocstateinfo property gets mapped from the original Snapshot protocol block to the composite specification namely the PR_6 protocol via the morphism s (figure 4.15) which is $Globprocstateinfo \rightarrow Globprocstateinfo$ meaning that the Globprocstateinfo property of Snapshot protocol gets exported to Decision Making protocol as its imported property and the morphism h_2 (figure 4.15) which is $Globprocstateinfo \to Globalstate$ meaning that the Globprocstateinfo property ultimately maps it to the final property in the specification. From here on, this final property namely the Globalstate in the specification gets carried on till the global composite protocol PR_6 through the morphism m_1 as labeled in figure 4.15.

Now if all these conditional statements are satisfied, then the final property statement in the Consistent state maintenance definition would be also satisfied by the fact that the Decision Making protocol which helps in maintaining a consistent state among the local states of all the processors depends on the attributes of the Snapshot protocol by importing its properties as shown in figure 4.15.

4.1.5 Analysis of Global Property–3: Roll-Back Recovery

Roll-Back Recovery is used to maintain the atomicity of object operation execution, to ensure that the states of all components (sites) remain consistent following failure. This is possible by checkpointing the entire state of a component (site) periodically and to start from the most recent checkpoint when a failure occurs.

As depicted in Figure 4.17, for the main global property of *roll-back recovery* to be true, sub-property: 1 must be true, i.e, *checkpoint its last successful (non-failed) state*, which can be proved using the *checkpointing* sub-protocol. Now for sub-property: 1 to exist, sub-property: 2 must be valid, i.e, *systematically log the values in a sta-*

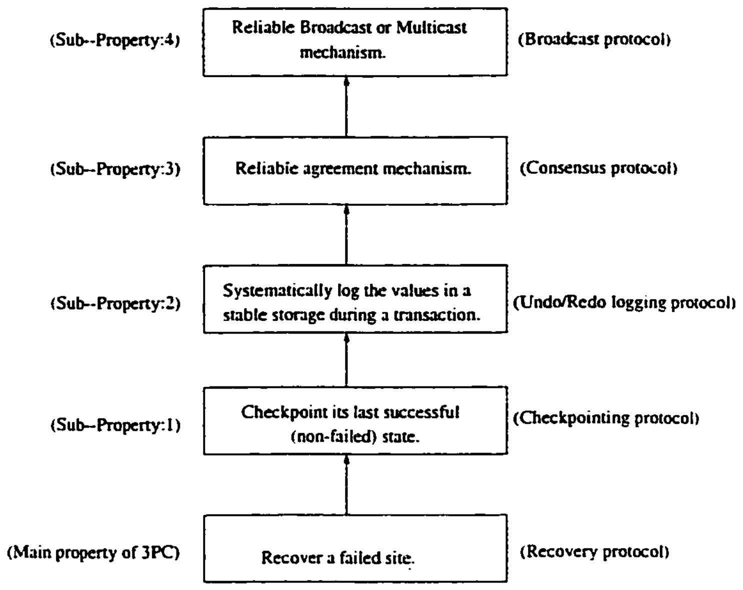

(Sub--Property:4)
Reliable Broadcast or Multicast mechanism.
(Broadcast protocol)
(Sub--Property:3)
Reliable agreement mechanism.
(Consensus protocol)
(Sub--Property:2)
Systematically log the values in a stable storage during a transaction.
(Undo/Redo logging protocol)
(Sub--Property:1)
Checkpoint its last successful (non-failed) state.
(Checkpointing protocol)
(Main property of 3PC)
Recover a failed site.
(Recovery protocol)

Figure 4.17: Roll-Back Recovery: Dependencies on Sub-Protocol Properties

ble storage during a transaction, which can be verified using the *undo/redo logging* sub-protocol. Again, to realize sub-property: 2, sub-property: 3 must be true, i.e, *a reliable agreement mechanism among the sites or participants*, which is provable by the *consensus* sub-protocol. Finally for all the above-said sub-properties to be realized, sub-property: 4 must be absolutely possible, i.e, *a reliable broadcast or multicast mechanism*, provable by the *broadcast* sub-protocol.

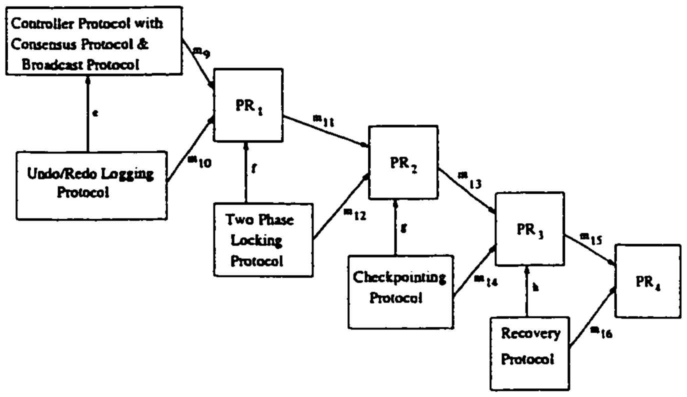


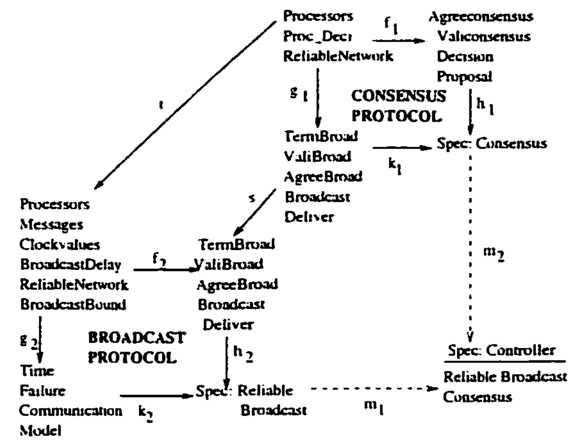
Figure 4.18: Composition of Sub-protocols to attain Recovery Property

Figure 4.18 illustrates the dependency of the global property of *roll-back recovery* on various sub-properties. Now in Figure 4.18, at first the *controller* sub-protocol and *undo/redo log* sub-protocol are modularly composed to come up with a new sub-protocol PR_1. PR_1 has the properties of both controller sub-protocol and undo/redo log sub-protocol. Next we modularly compose the PR_1 sub-protocol and *two-phase locking* sub-protocol to have a new sub-protocol PR_2. PR_2 has the combined properties of PR_1 and two-phase locking sub-protocol. This sub-protocol PR_2 combines with *checkpointing* sub-protocol to come up with the sub-protocol PR_3. PR_3 has the properties of PR_2 and checkpointing sub-protocol. The sub-protocol PR_3 is next composed with the *recovery* sub-protocol to produce a new sub-protocol PR_4. Now the final sub-protocol PR_4 would satisfy the global property of *roll-back recovery* of the three phase commit protocol. Next we give the category-theoretic reasoning for this property using morphisms and colimits.

Modularly Composing the Sub-Protocols via Category Theoretic operations

For this property, the compositional steps till the formation of sub-protocol PR_1 are the same as that of the first global property. However for the sake of continuous readability, the compositional steps to arrive at sub-protocol PR_1 are again shown in the following steps.

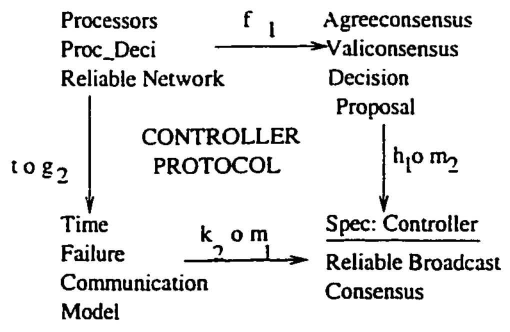
In Figure 4.19, the *broadcast* protocol and *consensus* protocol are modularly



where:
f_2 = Processors $\longrightarrow$ AgreeBroad
h_2 = AgreeBroad $\longrightarrow$ Reliable Broadcast
g_2 = Processors $\longrightarrow$ Communication
k_2 = Communication $\longrightarrow$ Reliable Broadcast
f_1 = Processors $\longrightarrow$ Agreeconsensus
h_1 = Agreeconsensus $\longrightarrow$ Consensus
g_1 = Processors $\longrightarrow$ AgreeBroad
k_1 = AgreeBroad $\longrightarrow$ Consensus
t = Processors $\longrightarrow$ Processors
s = TermBroad $\longrightarrow$ TermBroad, ValiBroad $\longrightarrow$ ValiBroad, Deliver $\longrightarrow$ Deliver, AgreeBroad $\longrightarrow$ AgreeBroad, Broadcast $\longrightarrow$ Broadcast
m_1 = Reliable Broadcast $\longrightarrow$ Consensus
m_2 = Consensus $\longrightarrow$ Consensus
$f_2 \circ h_2$ = Processors $\longrightarrow$ Reliable Broadcast = $g_2 \circ k_2$
$f_1 \circ h_1$ = Processors $\longrightarrow$ Consensus = $g_1 \circ k_1$

Figure 4.19: Composition of Broadcast and Consensus Protocols

composed to come up with the *controller* protocol. Composing two modules in the way shown above is possible only if each of these two modules commute individually through the specification morphism relationship.



where:
f_1 = Processors $\longrightarrow$ Agreeconsensus
$h_1 \circ m_2$ = Agreeconsensus $\longrightarrow$ Consensus
$t \circ g_2$ = Processors $\longrightarrow$ Communication
$k_2 \circ m_1$ = Communication $\longrightarrow$ Consensus
$f_1 \circ h_1 \circ m_2$ = Processors $\longrightarrow$ Consensus
$t \circ g_2 \circ k_2 \circ m_1$ = Processors $\longrightarrow$ Consensus

Figure 4.20: Composed Diagram of Controller Protocol

Figure 4.20 shows the composed diagram (module) of the controller protocol. Now *controller* block has the properties of both broadcast and consensus sub-blocks, i.e, reliable in broadcasting and also uniform in decision-making (consensus). We can see from the figure that the final composed module also commutes by the specification morphism relationship which proves the correctness of the composition and also guarantees the proper working of the composed module if it is to be reused or inherited elsewhere.

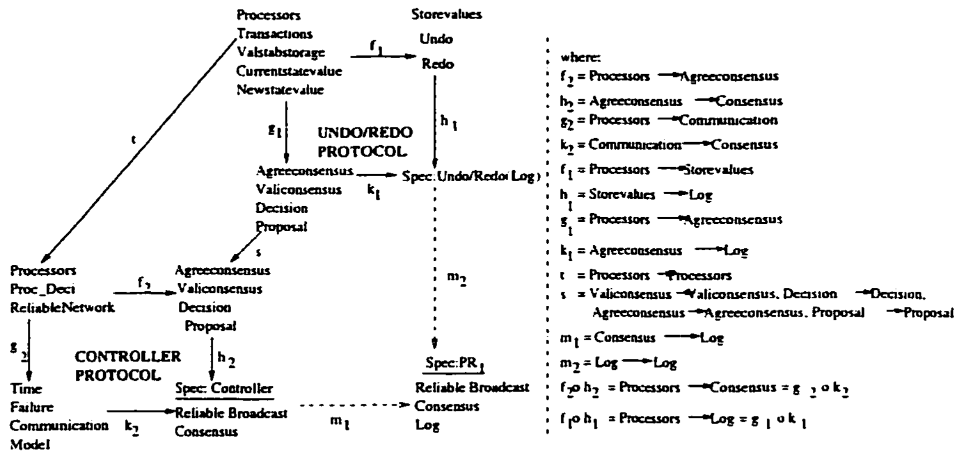


Figure 4.21: Composition of Controller and Undo/Redo Logging Protocols

The *controller* protocol and *undo/redo log* protocol are modularly composed to come up with a new protocol called PR_1, as depicted in Figure 4.21. In the figure, the parameter `Valstabstorage` in the Undo/Redo protocol is the stable storage medium for the processor state values.

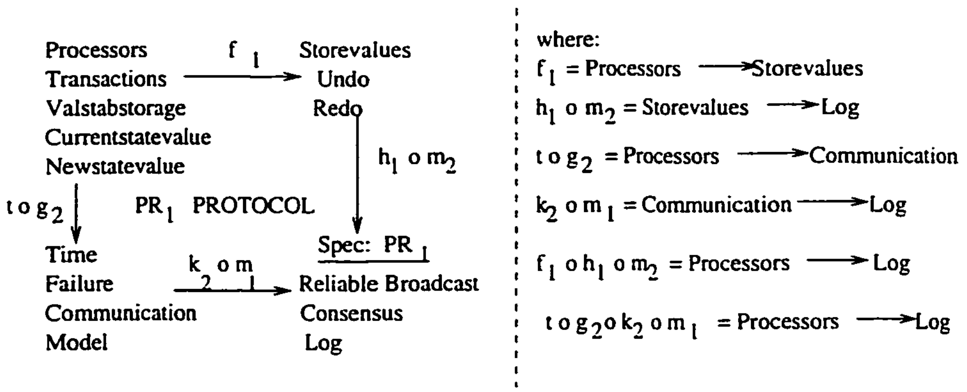


Figure 4.22: Composed Diagram of PR_1 Protocol

Figure 4.22 shows the composed diagram (module) of the PR_1 protocol. Now PR_1 block has the properties of both controller and undo/redo log sub-blocks. i.e, reliable in broadcasting, uniform in decision-making (consensus) and also store the values of the transaction onto a stable storage medium.

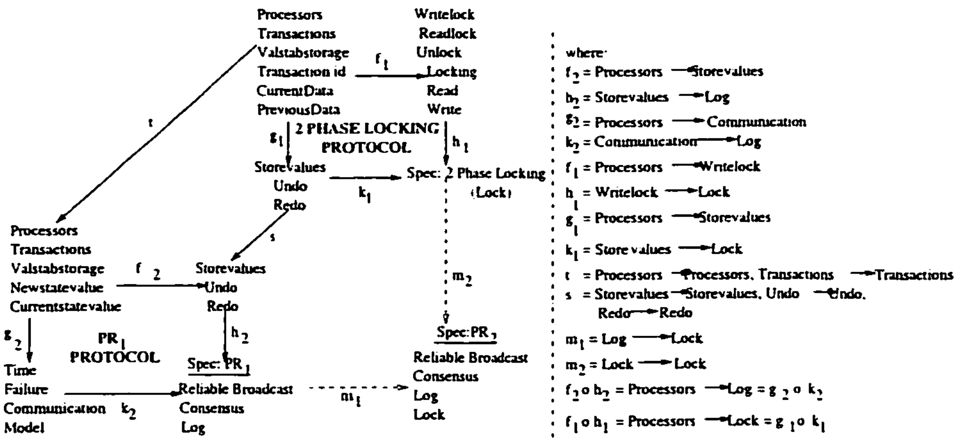


Figure 4.23: Composition of PR_1 and Two Phase Locking Protocols

Next we modularly compose the PR_1 protocol and *two phase locking* protocol to arrive at PR_2 protocol (See figure 4.23).

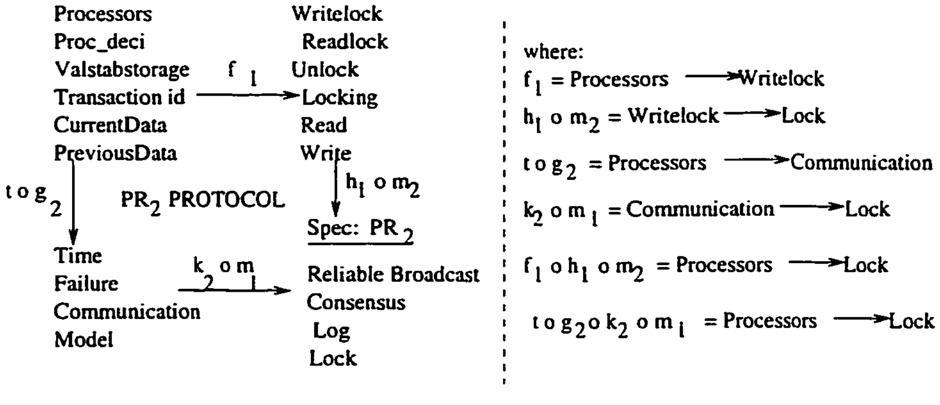
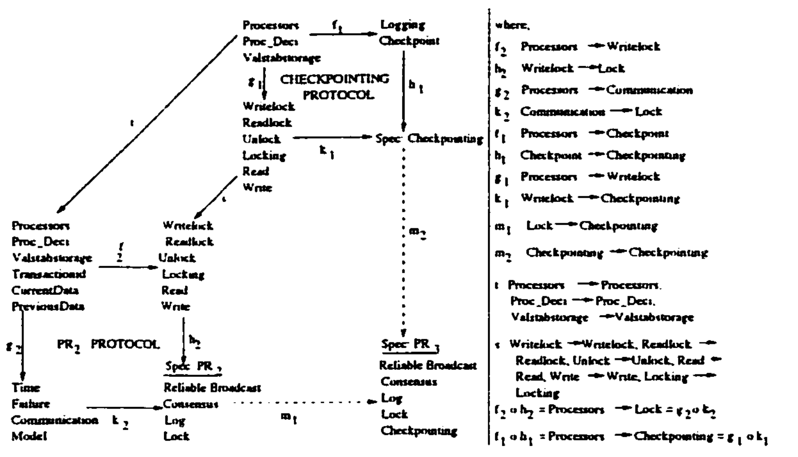


Figure 4.24: Composed Diagram of PR_2 Protocol

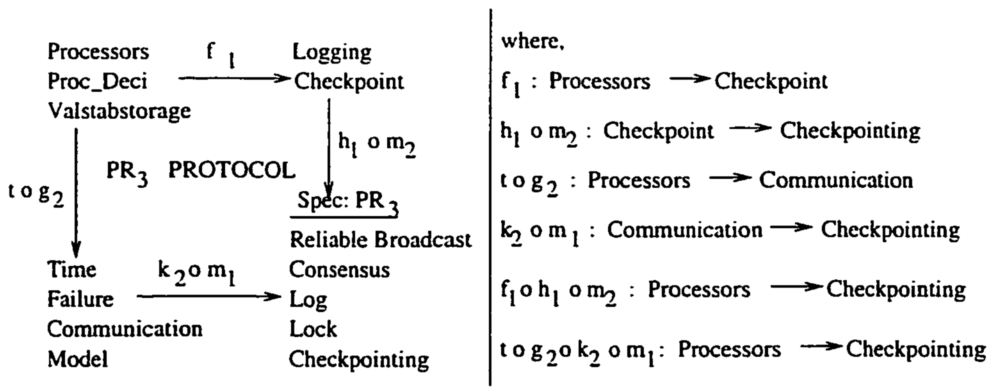
Figure 4.24 shows the composed diagram of PR_2 protocol. Now the composed PR_2 block has the locking properties in its final specification along with the other properties which it is dependent on namely reliable broadcast, consensus, and logging.



Processors
Proc_Deci
Valstabstorage
f_1
Logging
Checkpoint
g_1
CHECKPOINTING PROTOCOL
h_1
Writelock
Readlock
Unlock
Locking
Read
Write
k_1
Spec: Checkpointing
t
Processors
Proc_Deci
Valstabstorage
Transactionid
CurrentData
PreviousData
f_2
Writelock
Readlock
Unlock
Locking
Read
Write
s
m_2
g_2
PR_2 PROTOCOL
h_2
Spec: PR_2
Reliable Broadcast
Consensus
Log
Lock
Time
Failure
Communication
Model
k_2
m_1
Spec: PR_3
Reliable Broadcast
Consensus
Log
Lock
Checkpointing
where,
f_2 Processors → Writelock
h_2 Writelock → Lock
g_2 Processors → Communication
k_2 Communication → Lock
f_1 Processors → Checkpoint
h_1 Checkpoint → Checkpointing
g_1 Processors → Writelock
k_1 Writelock → Checkpointing
m_1 Lock → Checkpointing
m_2 Checkpointing → Checkpointing
t Processors → Processors, Proc_Deci → Proc_Deci, Valstabstorage → Valstabstorage
s Writelock → Writelock, Readlock → Readlock, Unlock → Unlock, Read → Read, Write → Write, Locking → Locking
$f_2 \circ h_2$ = Processors → Lock = $g_2 \circ k_2$
$f_1 \circ h_1$ = Processors → Checkpointing = $g_1 \circ k_1$

Figure 4.25: Composition of PR_2 and Checkpointing Protocols

In Figure 4.25, the PR_2 protocol and *checkpointing* protocol are modularly composed to come up with the PR_3 protocol.



Processors
Proc_Deci
Valstabstorage
f_1
Logging
Checkpoint
$h_1 \circ m_2$
PR_3 PROTOCOL
$t \circ g_2$
Spec: PR_3
Reliable Broadcast
Consensus
Log
Lock
Checkpointing
Time
Failure
Communication
Model
$k_2 \circ m_1$
where,
f_1 : Processors → Checkpoint
$h_1 \circ m_2$: Checkpoint → Checkpointing
$t \circ g_2$: Processors → Communication
$k_2 \circ m_1$: Communication → Checkpointing
$f_1 \circ h_1 \circ m_2$: Processors → Checkpointing
$t \circ g_2 \circ k_2 \circ m_1$: Processors → Checkpointing

Figure 4.26: Composed Diagram of PR_3 Protocol

Figure 4.26 shows the composed diagram of PR_3 protocol. Now the composed PR_3 block has the properties of reliable broadcast, consensus, logging, locking and checkpointing.

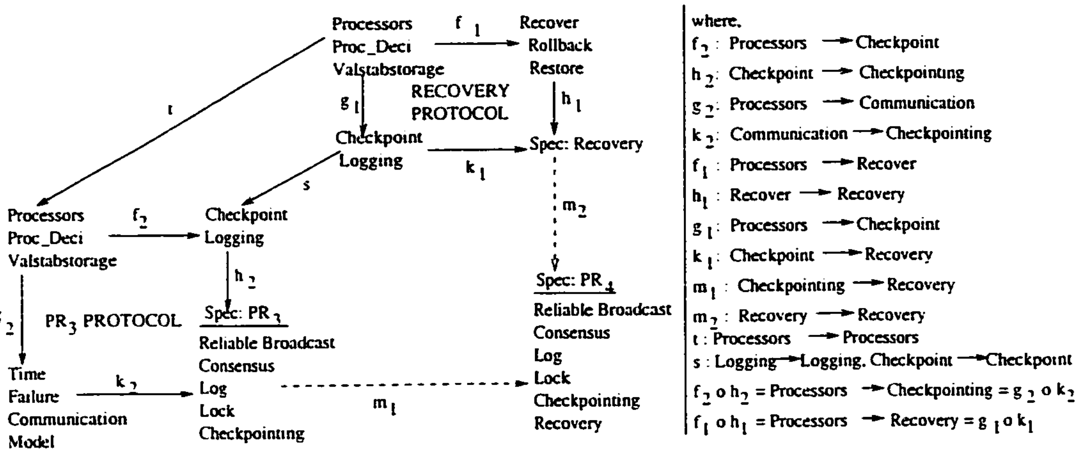

Processors
Proc_Deci
Valstabstorage
f 1
Recover
Rollback
Restore
t
g 1
RECOVERY
PROTOCOL
h 1
Checkpoint
Logging
k 1
Spec: Recovery
s
m 2
Processors
Proc_Deci
Valstabstorage
f 2
Checkpoint
Logging
h 2
g 2
PR3 PROTOCOL
Spec: PR 3
Reliable Broadcast
Consensus
Log
Lock
Checkpointing
Time
Failure
Communication
Model
k 2
m 1
Spec: PR 4
Reliable Broadcast
Consensus
Log
Lock
Checkpointing
Recovery
where,
f 2 : Processors → Checkpoint
h 2 : Checkpoint → Checkpointing
g 2 : Processors → Communication
k 2 : Communication → Checkpointing
f 1 : Processors → Recover
h 1 : Recover → Recovery
g 1 : Processors → Checkpoint
k 1 : Checkpoint → Recovery
m 1 : Checkpointing → Recovery
m 2 : Recovery → Recovery
t : Processors → Processors
s : Logging → Logging. Checkpoint → Checkpoint
f 2 o h 2 = Processors → Checkpointing = g 2 o k 2
f 1 o h 1 = Processors → Recovery = g 1 o k 1

Figure 4.27: Composition of PR_3 and Recovery Protocols

In Figure 4.27, the PR_3 protocol and *recovery* protocol are modularly composed to come up with the PR_4 protocol.

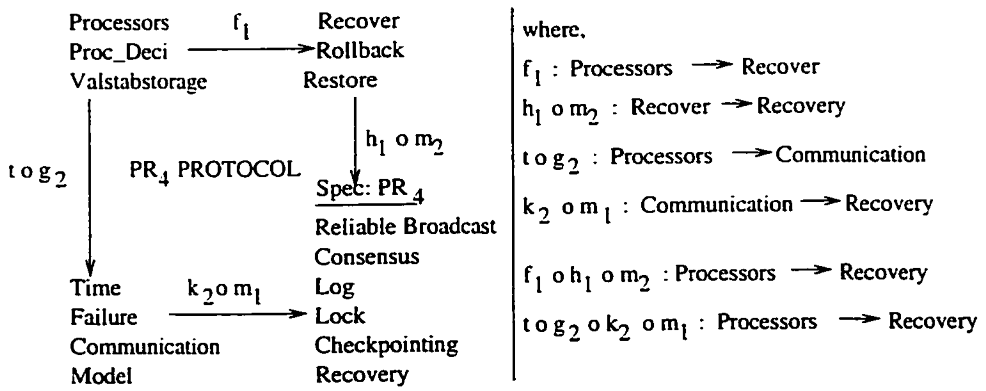

Processors
Proc_Deci
Valstabstorage
f 1
Recover
Rollback
Restore
h 1 o m 2
t o g 2
PR4 PROTOCOL
Spec: PR 4
Reliable Broadcast
Consensus
Log
Lock
Checkpointing
Recovery
Time
Failure
Communication
Model
k 2 o m 1
where,
f 1 : Processors → Recover
h 1 o m 2 : Recover → Recovery
t o g 2 : Processors → Communication
k 2 o m 1 : Communication → Recovery
f 1 o h 1 o m 2 : Processors → Recovery
t o g 2 o k 2 o m 1 : Processors → Recovery

Figure 4.28: Composed Diagram of PR_4 Protocol

Figure 4.28 shows the final composed diagram of PR_4 protocol. Now the composed PR_4 block has the properties of recovery in its final specification along with the other properties which it is dependent on namely reliable broadcast, consensus, logging, locking and checkpointing. Since it has the recovery mechanism, the PR_4

protocol would satisfy the *roll-back recovery property* of the three phase commit protocol thereby helping in maintaining the atomicity of object operation execution, in order to ensure that the states of all components (sites) remain consistent following failure.

By selectively identifying sorts and operations out of the sub-protocols: *controller* protocol, *undo/redo logging* protocol, *two phase locking* protocol, *checkpointing* protocol and *recovery* protocol for different morphism definitions, we have essentially constructed a sub-protocol PR_4 which satisfies the *Roll-Back Recovery* property.

4.1.6 Compositional Verification of Global Property–3

By roll-back recovery property, processor in error and other dependent processors would be rolled back to a consistent state and then restarted.

The formal representation of the Roll-Back Recovery definition is given below:

$$\forall(p, q : Proc., m : Messages, t : Transactions, v, commit, abort : Proc.Decision)$$

$$\forall(T : Clockvalues, N : Transactionid, X : Transactionvalueinstablestorage)$$

$$\forall(y : Currentstatevalue, z : Newstatevalue, Y : Currentdata, Z : Previousdata)$$

$$\forall(e : MaximumClockSkew, S : Checkpointperiod, n : Index) :$$

$$correct(p, T) \wedge correct(q, T) \wedge$$

$$if((Deliver(p, m, T) \rightarrow Deliver(q, m, (T + \gamma + \delta))) \wedge$$

$$(Decision(p, v, T) \rightarrow Decision(q, v, T))\wedge$$

$$(Undo(t, abort, X, y) \wedge Redo(t, commit, X, z) \rightarrow Log(t, X, z))\wedge$$

$$((\neg(Write(T, Y, X)) \wedge \neg(Locking(N, Y)) \wedge Unlock(N, Z)) \rightarrow$$

$$(Read(T, Y, X) \wedge Locking(N, Y))or$$

$$(\neg(Read(T, Y, X)) \wedge \neg(Locking(N, Y)) \wedge Unlock(N, Z)) \rightarrow$$

$$(Write(T, Y, X) \wedge Locking(N, Y))) \wedge ((ckpt(p, T) \wedge store(p, T)\wedge$$

$$Pi(p, T) = n + 1)or(Ckpt(p, S) \wedge Store(p, S) \wedge PI(p, S) = n + 1)))$$

$$then((ckpt(p, T)orCkpt(p, S)) \rightarrow Rollback(n, T) \rightarrow Restore(n, T))$$

where,

- γ is the maximum broadcastdelay.
- δ is the broadcastbound.
- Network is assumed to be a reliable one without partitions.
- `correct(p,T)`: processor p is correct at clock value T.
- `correct(q,T)`: processor q is correct at clock value T.
- `Broadcast(p,m,T)`: processor p broadcasts message m at real-time T.
- `Deliver(p,m,T)`: processor p delivers message m at real-time T.

In order to prove the above mentioned global property, we need to first establish the sub properties which the global property is dependent on. For the roll-back recovery property to hold true, a well defined checkpointing procedure has to be first set forth which then depends on a resource locking mechanism for exhibiting its functionalities. And for the purpose of a proper locking mechanism, a stable storage medium with a standard undo/redo operation is needed which in turn depends on the properties of a consensus procedure which is not possible without the properties of a reliable communication among the various processors in the distributed system. Hence we need to first prove the properties of the reliable broadcast mechanism.

The properties provided by the reliable broadcast are the following:

•`Termination`:- Every correct process eventually delivers some message.

SP1: TermBroad

$$\exists(p, m, T) : correct(p) \wedge Broadcast(p, m, T) \rightarrow$$

$$\forall(q, \gamma) : correct(q) \wedge Deliver(q, m, (T + \gamma))$$

•`Validity`:- If a correct process broadcasts a message m, then all correct processes eventually deliver m.

SP2: ValiBroad

$$\exists(p, m, T) : correct(p) \wedge Broadcast(p, m, T) \rightarrow$$

$$\forall(q, \gamma, \delta) : correct(q) \wedge Deliver(q, m, (T + \gamma + \delta)) \wedge (\gamma < \delta)$$

•`Uniform Agreement`:- If a process delivers a message m, then all correct processes eventually deliver m.

SP3: AgreeBroad

$$\exists(p)\forall(m, T) : correct(p) \wedge Deliver(p, m, T) \rightarrow$$

$$\forall(q,\gamma,\delta) : correct(q) \wedge Deliver(q,m,(T+\gamma+\delta))$$

Utilizing the following operational definitions, we state the various properties of the consensus protocol:

• `Decision(p,v,T)`: processor p arrives at a final decision v which is either to commit the transaction or to abort it.

• `Proposal(p,v,T)`: processor p proposes a decision v at real time T to either commit or to abort the transaction.

The properties are:

•`Validity`:-If a process decides X, then X was proposed by some process.

SP4: Valiconsensus

$$\forall(p,q,T)\exists(v) : Decision(p,v,T) \rightarrow Proposal(q,v,T)$$

This means that if a processor p makes a decision v which could be an abort or a commit, then that decision was initially made by another processor q.

•**`Uniform Agreement`**:-No two processes decide differently.

SP5: Agreeconsensus

$$\forall(p,q,v,T) : Decision(p,v,T) \rightarrow Decision(q,v,T)$$

Here both the processors p and q make the same decision whether to commit or abort.

Now that we have stated the properties of reliable communication medium and a consensus procedure, we will next state the properties of the stable storage medium with the undo/redo operation.

The various operational definitions are:

• `Undo(t,v,X,y)`: Undo the current state value y taken during the transaction t based on the processor decision v (abort) from the stable storage X before writing into it.

• `Redo(t,v,X,z)`: Redo the new state value z taken during the transaction t based on the processor decision v (commit) into the stable storage X before committing the transaction.

• `Log(t,X,z)`: Stores the new state value z of the processor got during the transaction t onto the stable storage X.

The property to be satisfied is:

SP6: Storevalues

$$\forall(p, q, t)\forall(commit, abort)\forall(y, z) : Undo(t, abort, X, y)\wedge$$

$$Redo(t, commit, X, z) \rightarrow Log(t, X, z)$$

Log is represented as a sequence of entries of the form [*t*,*X*,*v*], where z is the value that transaction t wrote into data item X. Log also contains sets of committed and aborted transactions. Each change to the database causes a log entry. Log has initial and final states (or) initial and differential state transitions (or) initial and operations performed. When volatile memory is lost in a system failure, the protocol examines the log, finds the last committed values of all data items that have been updated and restores them as new volatile values of these data items. Entry is undone in the stable log before writing into it and redone before the transaction is committed.

Once the transaction values are stored in the stable storage medium, we need a method to hold a lock on an item (shared resource log). Note that the type of lock depends on whether the transaction will read or write to the item.

The various operational primitives of a locking protocol are:

- `Read(t,Y,X)`: Reads the current data Y of the transaction t from the stable storage medium X.
- `Write(t,Y,X)`: Writes the current data Y of the transaction t onto the stable storage medium X.
- `Locking(N,Y)`: Locks the current state value Y with N as the transaction id.
- `Unlock(N,Z)`: Unlocks the previous state value Z with N as the transaction id.

The various properties are:

SP7: Writelock

$$\forall(p,q,t)\forall(N,X,Y,Z) : \neg(Read(t,Y,X)) \wedge \neg(Locking(N,Y)) \wedge Unlock(N,Z)$$
$$\rightarrow Write(t,Y,X) \wedge Locking(N,Y)$$

Only one transaction at a time may write lock an item and it is implemented by a simple 1 bit write lock flag. Write lock should enforce complete mutual exclusion on the item. An item is write locked if the item is not currently read locked (resource) and that the item had been already unlocked and free to be locked again.

Multiple transactions may be read locking an item at the same time. Read lock is implemented by using a read counter which holds the number of transactions currently holding a read lock.

SP8: Readlock

$$\forall(p,q,t)\forall(N,X,Y,Z) : \neg(Write(t,Y,X)) \wedge \neg(Locking(N,Y)) \wedge Unlock(N.Z)$$
$$\rightarrow Read(t,Y,X) \wedge Locking(N,Y)$$

An item is read locked if there are no current write locks on the item and that the item is free to be locked again. This property helps in the proper logging of data during the active transaction. The locking protocol allows a lock (shared resource log) to be acquired before the transaction and then asks for unlocking once the transaction is over. This way, serializability would be maintained while logging data onto the stable storage.

Once serializability is ensured. the processors in the distributed system could then start checkpointing the values they obtained during the transactions, thereby helping in the recovery process.

The various operational definitions are:

- `log(p,m,T)`: Processor p logs message m at clock time T.
- `ckpt(p,T)`: Processor p checkpoints at clock time T.
- `Ckpt(p,S)`: Processor p checkpoints at checkpoint period S.
- `store(p,T)`: Processor p stores at clock time T.
- `Store(p,S)`: Processor p stores at checkpoint period S.

The property to be satisfied is:

SP9: Checkpoint

$$\forall(p,m,n)\forall(e,S,T,\gamma,\delta : Clockvalues) : S-\delta-e < C(p,T) \wedge C(p,T) <= S \rightarrow$$

$$(if(\exists(m) : log(p,m,T) \wedge C(p,T) < S)$$

$$thenckpt(p,T) \wedge store(p,T) \wedge Pi(p,T) = n+1$$

$$elseCkpt(p,S) \wedge Store(p,S) \wedge PI(p,S) = n+1)$$

After having stated the individual properties of reliable communication medium, consensus, logging, locking and checkpointing procedures, we can now establish the global property of "Roll-Back Recovery". This property ensures that all faulty processors and the processors dependent on these faulty ones roll back to the non-faulty state for the purpose of recovery.

The various operational definitions are:

- `CorrecttoFailure(p,T)`: Processor p becomes faulty at clock time T.
- `Rollback(n,T)`: Rolls back as many number of times as the index n at clock time T.
- `Restore(n,T)`: Restores to the previous non-faulty state of the processor based on the index value n at clock time T.

The property to be satisfied is:

SP10: Recover

$$\forall(p,m,n)\forall(e,S,T,\gamma,\delta : Clockvalues) : S-\delta-e < C(p,T) \wedge C(p,T) <= S\wedge$$

$$CorrecttoFailure(p,T)\wedge(ckpt(p,T)orCkpt(p,S)) \rightarrow Rollback(n,T) \rightarrow Restore(n,T)$$

As we have composed modules to get a composite protocol which satisfies the Roll-Back Recovery property, it was of utmost importance that all the morphisms we have defined along the way essentially ensure that all the necessary attributes and operations from each individual building block protocols are carried on/included in the composite specification. The fact that we have successfully composed the specification which satisfies the specified requirements in turn guarantees that the morphisms have been correctly defined. Through this strong mathematical framework, we have a capability to perform a backward propagation, i.e., essentially, we have the traceability capability. We elaborate on this in our subsequent discussions.

Nuances of Morphisms in Compositional Verification

In the formal representation of `Roll-Back Recovery` definition, the conditional statement $Deliver(p, m, T) \rightarrow Deliver(q, m, (T+\gamma+\delta))$ is satisfied by the `AgreeBroad` property of the `Reliable Broadcast` protocol. The operation which extends this `AgreeBroad` property gets mapped from the original `Reliable Broadcast` protocol block to the composite specification namely the PR_4 protocol via the morphism s (figure 4.19) which is $AgreeBroad \rightarrow AgreeBroad$ meaning that the `AgreeBroad` property of `Reliable Broadcast` protocol gets exported to the `Consensus` protocol as its imported property and the morphism h_2 (figure 4.19) which is $AgreeBroad \rightarrow ReliableBroadcast$ meaning that the `AgreeBroad` property ultimately maps it to the final property in the specification. From here on, this final property namely the ReliableBroadcast in the specification gets carried on till the global composite protocol PR_4 through the morphisms m_l as labeled in figures 4.19, 4.21, 4.23, 4.25 and 4.27.

The next conditional statement in the definition namely $Decision(p, v, T) \rightarrow Decision(q, v, T)$ is satisfied by the `Agreeconsensus` property of the `Consensus` protocol. The operation which extends this `Agreeconsensus` property gets mapped from the original `Consensus` protocol block to the composite specification namely the PR_4 protocol via the morphism s (figure 4.21) which is $Agreeconsensus \rightarrow Agreeconsensus$ meaning that the `Agreeconsensus` property of `Consensus` protocol gets exported to the `Undo/Redo` protocol as its imported property and the morphism h_2 (figure 4.21) which is $Agreeconsensus \rightarrow Consensus$ meaning that the `Agreeconsensus` property ultimately maps it to the final property in the specification. From here on, this final property namely the Consensus in the specification gets carried on till the global composite protocol PR_4 through the morphisms m_l as labeled in figures 4.21, 4.23, 4.25 and 4.27.

The third condition in the definition namely $Undo(t, abort, X, y) \wedge Redo(t, commit, X, z) \rightarrow Log(t, X, z)$ is satisfied by the `Storevalues` property of the `Undo/Redo` protocol. The operation which extends this `Storevalues` property gets mapped from the original `Undo/Redo` protocol block to the composite specification namely the PR_4 protocol via the morphism s (figure 4.23) which is $Storevalues \rightarrow Storevalues$ meaning that the `Storevalues` property of `Undo/Redo` protocol gets exported to `Two Phase Locking` protocol as its imported property and the morphism h_2 (figure 4.23) which is *Storevalues* $\rightarrow$ *Log* meaning that the Storevalues property ultimately maps it to the final property in the specification. From here on, this final property namely the Log in the specification gets carried on till the global composite protocol PR_4 through the morphism m_1 as labeled in figures 4.23, 4.25 and 4.27.

The first part of the next conditional statement in the definition namely $((\neg(Write(T, Y, X)) \wedge \neg(Locking(N, Y)) \wedge Unlock(N, Z)) \rightarrow (Read(T, Y, X) \wedge Locking(N, Y))$ is satisfied by the Readlock property of the Two Phase Locking protocol and the second part of the conditional statement namely $(\neg(Read(T, Y, X)) \wedge \neg(Locking(N, Y)) \wedge Unlock(N, Z)) \rightarrow (Write(T, Y, X) \wedge Locking(N, Y))$ is satisfied by the Writelock property of the Two Phase Locking protocol. The operation which extends this Writelock property gets mapped from the original Two Phase Locking protocol block to the composite specification namely the PR_4 protocol via the morphism s (figure 4.25) which is *Writelock* $\rightarrow$ *Writelock* meaning that the Writelock property of Two Phase Locking protocol gets exported to Checkpointing protocol as its imported property and the morphism h_2 (figure 4.25) which is *Writelock* $\rightarrow$ *Lock* meaning that the Writelock property ultimately maps it to the final property in the specification. From here on, this final property namely the Lock in the specification gets carried on till the global composite protocol PR_4 through the morphism m_1 as labeled in figures 4.25 and 4.27.

The last conditional statement in definition namely $((ckpt(p, T) \wedge store(p, T) \wedge Pi(p, T) = n + 1) or (Ckpt(p, S) \wedge Store(p, S) \wedge PI(p, S) = n + 1))$ is satisfied by the

Checkpoint property of the Checkpointing protocol. The operation which extends this Checkpoint property gets mapped from the original Checkpointing protocol block to the composite specification namely the PR_4 protocol via the morphism s (figure 4.27) which is $Checkpoint \rightarrow Checkpoint$ meaning that the Checkpoint property of Checkpointing protocol gets exported to Recovery protocol as its imported property and the morphism h_2 (figure 4.27) which is $Checkpoint \rightarrow Checkpointing$ meaning that the Checkpoint property ultimately maps it to the final property in the specification. From here on, this final property namely the Checkpointing in the specification gets carried on till the global composite protocol PR_4 through the morphism m_1 as labeled in figure 4.27.

Now if all these conditional statements are satisfied, then the final property statement in the Roll-Back Recovery definition would be also satisfied by the fact that the Recovery protocol which helps in rolling back processors in error and other dependent processors to a consistent state and then restarting them depends on the attributes of the Checkpointing protocol by importing its properties as shown in figure 4.27.

Chapter 5

Compositional Specification and Verification of the Global Properties of 3PC Protocol using Specware

In this chapter, we provide the specification and verification for the three global properties of the three phase commit protocol discussed in Chapter 4 in terms of their individual building block specifications using the Specware [28, 32, 41] tool.

5.1 Specification of the Global Properties of 3PC

5.1.1 Specification of Serializability of Transactions Property

The serializability property states that the effect of executing a collection of atomic actions is equivalent to some serial schedule in which the actions are executed one after another. The technique we have employed for implementing serializability is by two phase locking mechanism. In order for the locking (two phase lock protocol) to be employed, we need to have a stable storage medium with undo and redo mechanisms (undo/redo protocol), a procedure to achieve consensus (consensus protocol) and a reliable communication (reliable broadcast protocol) among the various processors in the distributed system. This property could be realized by formalizing each of the individual protocols with the help of Specware tool and then composing them using category theory as illustrated in Section 4.1 in Chapter 4.

To begin with, some basic primitives like Time, Failure, Communication and Configurational Model are needed to be specified for the specification of the reliable broadcast protocol. We call these primitives as Basic Building Block primitives and denote them as *BBB*. *BBB* is specified in Specware as:

```
BBB = spec
sort Clockvalues = Nat
sort LocalClockvals = Clockvalues
sort Processors
sort Index = Nat
sort Messages = {p:Processors, Tm:Clockvalues, Km:Index, No:Nat}
sort Procstate = {p:Processors, LC:Clockvalues, n:Nat}
op Correct : Processors -> Boolean
op InOrder : Messages->Boolean
op Broadcast : Processors*Messages*Clockvalues->Boolean
op Deliver : Processors*Messages*Clockvalues->Boolean
endspec
```

The various parameters of the *BBB* are formalized as sorts and operations using specware and are translated so that these parameters could be used by other protocols for their functional specifications.

```
BBBtoALLTRANSLATION = translate(BBB) by
{Clockvalues +-> Clockvalues, LocalClockvals +-> LocalClockvals,
Processors +-> Processors, Index +-> Index, Messages +-> Messages,
Procstate +-> Procstate, Correct +-> Correct, InOrder +-> InOrder,
Broadcast +-> Broadcast, Deliver +-> Deliver}
```

By importing this translation. all the parameters of *BBB* in terms of its sorts and operations would be made available for the formalization of the *RELIABLEBROADCAST* protocol.

```
RELIABLEBROADCAST = spec
import BBBtoALLTRANSLATION
sort ReliableNetwork = Boolean
sort BroadcastDelay = Clockvalues
sort BroadcastBound = Clockvalues
op Clockdelay : Clockvalues*BroadcastDelay->Clockvalues
op Clockbound : Clockvalues*BroadcastDelay*BroadcastBound->Clockvalues
op TermBroad : Processors*Messages*Clockvalues->Boolean
op ValiBroad :  Processors*Messages*Clockvalues->Boolean
op AgreeBroad : Processors*Messages*Clockvalues->Boolean
```

Having specified the parameters needed for formalizing the RELIABLEBROADCAST protocol, we now provide the various attributes (properties) of the protocol in terms of its axioms. It is important to note that we are formalizing properties of RELIABLEBROADCAST protocol as axioms, as our goal is to utilize these properties *as is* in establishing the correctness of the Serializability property. This holds for other sub-protocols subsequently being used. For rigor, one would attempt to specify and verify these individual building block protocols' properties. That aspect is beyond the scope of this thesis.

```
axiom Broadcast is
```

```
fa(p:Processors, m:Messages, T:Clockvalues)
~(Deliver(p, m, T)) & Broadcast(p, m, T)

axiom Deliver is
fa(p:Processors, m:Messages, T:Clockvalues)
~(Broadcast(p, m, T)) & Deliver(p, m, T)

axiom Termbroad is
ex(p, m, T) Correct(p) & Broadcast(p, m, T) =>
(fa (q, i:BroadcastDelay) Correct(q) & Deliver(q,m,(Clockdelay(T,i))))

axiom Valibroad is
ex(p, m, T) Correct(p) & Broadcast(p, m, T) =>
(fa (q, i:BroadcastDelay, j:BroadcastBound) Correct(q) &
Deliver(q, m, (Clockbound(T, i, j))) & i<j)

axiom Agreebroad is
ex(p) fa(m:Messages, T:Clockvalues) Deliver(p, m, T) =>
(fa (q, i:BroadcastDelay, j:BroadcastBound)
Deliver(q, m, Clockbound(T, i, j)))

endspec
```

The translation is done in the same way as before, but this time it would have what *BBB* had translated, and the sorts, operations and properties of RELIABLEBROADCAST protocol also.

```
RELBROADtoALLTRANSLATION = translate(RELIABLEBROADCAST) by
{Clockvalues +-> Clockvalues, LocalClockvals +-> LocalClockvals,
Processors +-> Processors, Index +-> Index, Messages +-> Messages,
Procstate +-> Procstate, Correct +-> Correct, InOrder +-> InOrder,
Broadcast +-> Broadcast, Deliver +-> Deliver,
ReliableNetwork+->ReliableNetwork,BroadcastDelay+->BroadcastDelay,
BroadcastBound +-> BroadcastBound, TermBroad +-> TermBroad,
ValiBroad +-> ValiBroad, AgreeBroad +-> AgreeBroad}
```

Next we formalize the consensus protocol by first importing what the previous translation could offer, and then adding the parameteis needed for defining its various properties.

```
CONSENSUS = spec
import RELBROADtoALLTRANSLATION
sort ProcDeci = Boolean
op Decision : Processors*ProcDeci*Clockvalues->Boolean
op Proposal : Processors*ProcDeci*Clockvalues->Boolean
op Valiconsensus : Processors*ProcDeci*Clockvalues->Boolean
op Agreeconsensus : Processors*ProcDeci*Clockvalues->Boolean
```

The various properties of *Consensus* protocol are given below as axioms:

```
axiom Proposal is
fa(p:Processors, v:ProcDeci, T:Clockvalues)
~(Decision(p, v, T)) & Proposal(p, v, T)

axiom Decision is
fa(p:Processors, v:ProcDeci, T:Clockvalues)
~(Proposal(p, v, T)) & Decision(p, v, T)

axiom Valiconsensus is
fa(p,q:Processors, T,i,j:Clockvalues, m:Messages) ex(v:ProcDeci)
ValiBroad(p, m, T) & Decision(p, v, T) => Proposal(q, v, T)

axiom Agreeconsensus is
fa(p,q:Processors, v:ProcDeci, T,i,j:Clockvalues, m:Messages)
AgreeBroad(p, m, T) & Decision(p, v, T) => Decision(q, v, T)

endspec
```

Now in order to compose the specifications of *RELIABLEBROADCAST* and *CONSENSUS*, we first need to specify the various morphisms that link them. We formalize the morphism between these two specifications in the following way:

```
RELBROADtoCONSENSUS = morphism RELIABLEBROADCAST->CONSENSUS
{Broadcast+->Broadcast,Deliver+->Deliver,TermBroad +->TermBroad,
ValiBroad +-> ValiBroad, AgreeBroad +-> AgreeBroad}
```

We then define the diagram with RELIABLEBROADCAST and CONSENSUS specifications as the nodes, and the morphism as the link between them. Finally to construct the composite specification of these two modules, we take the colimit of the diagram as shown below:

```
CONSEN = diagram {
a +-> RELIABLEBROADCAST,
b +-> CONSENSUS,
i: a->b +-> morphism RELIABLEBROADCAST->CONSENSUS
{Broadcast+->Broadcast,Deliver+->Deliver,TermBroad+->TermBroad,
ValiBroad +-> ValiBroad, AgreeBroad +-> AgreeBroad}}

CONSENT = colimit CONSEN
```

The name CONSENT in the specification denotes the controller protocol as depicted in Figures 4.3 and 4.4 in Chapter4. Now CONSENT has the properties of both RELIABLEBROADCAST and CONSENSUS specifications, i.e, reliable in broadcasting and also uniform in decision-making (consensus). The translation is then done in the same way as before:

```
CONSENTtoALLTRANSLATION = translate(CONSENSUS) by
{Clockvalues +-> Clockvalues, LocalClockvals +-> LocalClockvals,
Processors +-> Processors, Index +-> Index, Messages +-> Messages,
Procstate +-> Procstate, Correct +-> Correct, InOrder +-> InOrder,
Broadcast +-> Broadcast, Deliver +-> Deliver,
ReliableNetwork+->ReliableNetwork,BroadcastDelay+->BroadcastDelay,
BroadcastBound +-> BroadcastBound, TermBroad +-> TermBroad,
ValiBroad +-> ValiBroad, AgreeBroad +-> AgreeBroad,
Valiconsensus+->Valiconsensus,Agreeconsensus+->Agreeconsensus,
ProcDeci+->ProcDeci,Decision+->Decision,Proposal+->Proposal}
```

The next protocol that is needed to be specified in order to verify the serializability property is the undo/redo protocol. In Figure 4.5, we illustrated the composition of the *controller* protocol and *undo/redo* protocol to come up with a new protocol called PR_1. But before that, we need to formalize the undo/redo protocol which is done in the following manner:

```
UNDOREDO = spec
import CONSENTtoALLTRANSLATION
sort Transactions = Boolean
sort Valstabstorage = Boolean
sort Currentstatevalue = Nat
sort Newstatevalue = Nat
op Log : Transactions*Valstabstorage*Newstatevalue->Boolean
op Undo:Transactions*ProcDeci*Valstabstorage*Currentstatevalue->Boolean
op Redo : Transactions*ProcDeci*Valstabstorage*Newstatevalue->Boolean
op Storevalues : Transactions*Valstabstorage*ProcDeci->Boolean

axiom Undo is
fa(t:Transactions, a:ProcDeci, X:Valstabstorage, y:Currentstatevalue)
~(Redo(t, a, X, y)) & Undo(t, a, X, y)

axiom Redo is
fa(t:Transactions, a:ProcDeci, X:Valstabstorage, y:Currentstatevalue)
~(Undo(t, a, X, y)) & Redo(t, a, X, y)

axiom Log is
fa(t:Transactions, a:ProcDeci, X:Valstabstorage)
fa(y:Currentstatevalue, z:Newstatevalue)
~(Undo(t, a, X, y)) & ~(Redo(t, a, X, y)) => Log(t, X, z)

axiom Storevalues is
```

```
fa(p,q:Processors) fa(T:Clockvalues,t:Transactions)
fa(commit,abort:ProcDeci)
fa(y:Currentstatevalue, z:Newstatevalue, X:Valstabstorage)
Agreeconsensus(p, commit, T) & Undo(t, abort, X, y) &
Redo(t, commit, X, z) => Log(t, X, z)

endspec
```

The morphism definition, diagram and colimit formations of the controller and undo/redo protocols are done in the same way as before:

```
CONSENTtoUNDOREDO = morphism CONSENSUS->UNDOREDO
{Valiconsensus +-> Valiconsensus, Agreeconsensus +-> Agreeconsensus,
Decision +-> Decision, Proposal +-> Proposal}

UNRE = diagram {
a +-> CONSENSUS,
b +-> UNDOREDO,
i: a->b +-> morphism CONSENSUS->UNDOREDO
{Valiconsensus +-> Valiconsensus, Agreeconsensus +-> Agreeconsensus,
Decision +-> Decision, Proposal +-> Proposal}}

UNREDO = colimit UNRE
```

The name UNREDO in the specification denotes the PR_1 protocol as depicted in Figures 4.5 and 4.6 in Chapter4. Now UNREDO has the properties of both Controller and undo/redo specifications, i.e, reliable in broadcasting, uniform in decision-making (consensus) and also store the values of the transaction onto a stable storage medium. The translation is then done in the same way as before:

```
UNREDOtoALLTRANSLATION = translate(UNDOREDO) by
{Clockvalues +-> Clockvalues, LocalClockvals +-> LocalClockvals,
Processors +-> Processors, Index +-> Index, Messages +-> Messages,
Procstate +-> Procstate, Correct +-> Correct, InOrder +-> InOrder,
Broadcast +-> Broadcast, Deliver +-> Deliver,
ReliableNetwork +-> ReliableNetwork, BroadcastDelay+->BroadcastDelay,
BroadcastBound +-> BroadcastBound, TermBroad +-> TermBroad,
ValiBroad+->ValiBroad,AgreeBroad+->AgreeBroad,ProcDeci+->ProcDeci,
Valiconsensus +-> Valiconsensus, Agreeconsensus +-> Agreeconsensus,
Decision+->Decision,Proposal+->Proposal,Transactions +->Transactions,
Valstabstorage+->Valstabstorage,Currentstatevalue+->Currentstatevalue,
Newstatevalue +-> Newstatevalue, Log +-> Log, Undo +-> Undo,
Redo +-> Redo, Storevalues +-> Storevalues}
```

The last component (protocol) that is needed for proving the global property is the two phase locking protocol which can be formally specified using Specware in the following way:

```
TWOPHASELOCK = spec
import UNREDOtoALLTRANSLATION
sort Transactionid
sort CurrentData
sort PreviousData
op Read : Transactions*CurrentData*Valstabstorage->Boolean
op Write : Transactions*CurrentData*Valstabstorage->Boolean
op Locking : Transactionid*CurrentData->Boolean
op Unlock : Transactionid*PreviousData->Boolean
op Readlock : Transactions*CurrentData*Valstabstorage->Boolean
op Writelock : Transactions*CurrentData*Valstabstorage->Boolean

axiom Read is
fa(t:Transactions, Y:CurrentData, X:Valstabstorage)
~(Write(t, Y, X)) & Read(t, Y, X)

axiom Write is
fa(t:Transactions, Y:CurrentData, X:Valstabstorage)
~(Read(t, Y, X)) & Write(t, Y, X)

axiom Locking is
fa(N:Transactionid, Y:CurrentData, Z:PreviousData)
```

```
(Unlock(N, Z)) & Locking(N, Y)

axiom Unlock is
fa(N:Transactionid, Y:CurrentData, Z:PreviousData)
~(Locking(N, Y)) & Unlock(N, Z)

axiom Readlock is
fa(p,q:Processors) fa(t:Transactions, N:Transactionid, X:Valstabstorage)
fa(Y:CurrentData, Z:PreviousData, z:Newstatevalue) Log(t, X, z) &
~(Write(t, Y, X)) & ~(Locking(N, Y)) & Unlock(N, Z) => Read(t, Y, X) &
Locking(N, Y)

axiom Writelock is
fa(p,q:Processors) fa(t:Transactions, N:Transactionid, X:Valstabstorage)
fa(Y:CurrentData, Z:PreviousData, z:Newstatevalue) Log(t, X, z) &
~(Read(t, Y, X)) & ~(Locking(N, Y)) & Unlock(N, Z) => Write(t, Y, X) &
Locking(N, Y)
```

By utilizing the various axioms provided by the individual specifications of *RELIABLEBROADCAST*, *CONSENSUS*, *UNDO/REDO* and *TWOPHASELOCKING* protocols, we formulate the theorem *Serialize* required for ultimately proving the global property of *Serializability of Transactions*.

```
theorem Serialize is
fa(p,q:Processors, T:Clockvalues, m:Messages, t:Transactions)
fa(i:BroadcastDelay, j:BroadcastBound)
fa(v,commit,abort:ProcDeci, N:Transactionid, X:Valstabstorage)
fa(y:Currentstatevalue, z:Newstatevalue, Y:CurrentData, Z:PreviousData)
(
if((Deliver(p, m, T) => Deliver(q, m, (Clockbound(T, i, j)))) &
(AgreeBroad(p,m,T) & Decision(p,v,T)=>AgreeBroad(q,m,(Clockbound(T,i,j)))
& Decision(q, v, T)) & (Agreeconsensus(p, commit, T) & Undo(t,abort,X,y)
& Redo(t, commit, X, z) => Log(t, X, z)))
then(Log(t, X, z) & (~(Write(t, Y, X))) & (~(Locking(N, Y))) &
Unlock(N, Z) => Read(t, Y, X) & Locking(N, Y))
else(Log(t, X, z) & (~(Read(t, Y, X))) & (~(Locking(N, Y))) &
Unlock(N, Z) => Write(t, Y, X) & Locking(N, Y)))
```

```
endspec

UNREDOtoTWOPHASELOCK = morphism UNDOREDO->TWOPHASELOCK
{Undo +-> Undo, Redo +-> Redo, Storevalues +-> Storevalues}

TPLOCK = diagram {
a +-> UNDOREDO,
b +-> TWOPHASELOCK,
i: a->b +-> morphism UNDOREDO->TWOPHASELOCK
{Undo +-> Undo, Redo +-> Redo, Storevalues +-> Storevalues}}

TPL = colimit TPLOCK


foo = print TPL


TPLtoALLTRANSLATION = translate(TWOPHASELOCK) by
{Clockvalues +-> Clockvalues, LocalClockvals +-> LocalClockvals,
Processors +-> Processors, Index +-> Index, Messages +-> Messages,
Procstate +-> Procstate, Correct +-> Correct, InOrder +-> InOrder,
Broadcast +-> Broadcast, Deliver +-> Deliver,
ReliableNetwork +-> ReliableNetwork, BroadcastDelay+->BroadcastDelay,
BroadcastBound +-> BroadcastBound, TermBroad +-> TermBroad,
ValiBroad +-> ValiBroad,AgreeBroad +-> AgreeBroad,ProcDeci+->ProcDeci,
Valiconsensus +-> Valiconsensus, Agreeconsensus +-> Agreeconsensus,
Decision +-> Decision,Proposal+->Proposal,Transactions+->Transactions,
Valstabstorage+->Valstabstorage,Currentstatevalue+->Currentstatevalue,
Newstatevalue +-> Newstatevalue,Log +-> Log,Undo +-> Undo,Redo+->Redo,
Storevalues +-> Storevalues, Read +-> Read, Write +-> Write,
Locking +-> Locking, Unlock +-> Unlock, Readlock +-> Readlock,
Writelock +-> Writelock}
```

We finally verify the global property by processing the above specification along with the theorem in Specware with a built-in interface to Snark theorem prover. The statement for the proof of the global property is as given below:

```
p1 = prove Serialize in TWOPHASELOCK using Agreebroad Agreeconsensus
Storevalues Readlock Writelock
```

The other two global properties are specified and verified in a similar manner utilizing the modular compositional techniques illustrated in Sections 4.1.2 and 4.1.4 in Chapter 4, and are discussed in subsequent sections.

5.1.2 Specification of Consistent State Maintenance Property

```
%
%Specification of the basic building block primitives
%
BBB = spec
sort Clockvalues = Nat
sort LocalClockvals = Clockvalues
sort Processors
sort Index = Nat
sort Messages = {p:Processors, Tm:Clockvalues, Km:Index, No:Nat}
sort Procstate = {p:Processors, LC:Clockvalues, n:Nat}
op Correct : Processors -> Boolean
op InOrder : Messages->Boolean
op Broadcast : Processors*Messages*Clockvalues->Boolean
op Deliver : Processors*Messages*Clockvalues->Boolean
endspec

%
%Translation of parameters to other blocks
%
BBBtoALLTRANSLATION = translate(BBB) by
{Clockvalues +-> Clockvalues, LocalClockvals +-> LocalClockvals,
Processors +-> Processors, Index +-> Index, Messages +-> Messages,
Procstate +-> Procstate, Correct +-> Correct, InOrder +-> InOrder,
Broadcast +-> Broadcast, Deliver +-> Deliver}

%
%Specification of the RELIABLEBROADCAST protocol
```

```
%
RELIABLEBROADCAST = spec
import BBBtoALLTRANSLATION
sort ReliableNetwork = Boolean
sort BroadcastDelay = Clockvalues
sort BroadcastBound = Clockvalues
op Clockdelay : Clockvalues*BroadcastDelay->Clockvalues
op Clockbound : Clockvalues*BroadcastDelay*BroadcastBound->Clockvalues
op TermBroad : Processors*Messages*Clockvalues->Boolean
op ValiBroad :  Processors*Messages*Clockvalues->Boolean
op AgreeBroad : Processors*Messages*Clockvalues->Boolean

%
%Various properties of the RELIABLEBROADCAST protocol
%
axiom Broadcast is
fa(p:Processors, m:Messages, T:Clockvalues)
~(Deliver(p, m, T)) & Broadcast(p, m, T)

axiom Deliver is
fa(p:Processors, m:Messages, T:Clockvalues)
~(Broadcast(p, m, T)) & Deliver(p, m, T)

axiom Termbroad is
ex(p, m, T) Correct(p) & Broadcast(p, m, T) =>
(fa (q, i:BroadcastDelay) Correct(q) & Deliver(q,m,(Clockdelay(T,i))))

axiom Valibroad is
ex(p, m, T) Correct(p) & Broadcast(p, m, T) =>
(fa (q, i:BroadcastDelay, j:BroadcastBound) Correct(q) &
Deliver(q, m, (Clockbound(T, i, j))) & i<j)

axiom Agreebroad is
ex(p) fa(m:Messages, T:Clockvalues) Deliver(p, m, T) =>
(fa (q, i:BroadcastDelay, j:BroadcastBound)
Deliver(q, m, Clockbound(T, i, j)))

endspec

%
%Translation of parameters and properties to other blocks
%
```

```
RELBROADtoALLTRANSLATION = translate(RELIABLEBROADCAST) by
{Clockvalues +-> Clockvalues, LocalClockvals +-> LocalClockvals,
Processors +-> Processors, Index +-> Index, Messages +-> Messages,
Procstate +-> Procstate, Correct +-> Correct, InOrder +-> InOrder,
Broadcast +-> Broadcast, Deliver +-> Deliver,
ReliableNetwork+->ReliableNetwork,BroadcastDelay+->BroadcastDelay,
BroadcastBound +-> BroadcastBound, TermBroad +-> TermBroad,
ValiBroad +-> ValiBroad, AgreeBroad +-> AgreeBroad}

%
%Specification of the CONSENSUS protocol
%
CONSENSUS = spec
import RELBROADtoALLTRANSLATION
sort ProcDeci = Boolean
op Decision : Processors*ProcDeci*Clockvalues->Boolean
op Proposal : Processors*ProcDeci*Clockvalues->Boolean
op Valiconsensus : Processors*ProcDeci*Clockvalues->Boolean
op Agreeconsensus : Processors*ProcDeci*Clockvalues->Boolean

%
%Various properties of the CONSENSUS protocol
%
axiom Proposal is
fa(p:Processors, v:ProcDeci, T:Clockvalues)
~(Decision(p, v, T)) & Proposal(p, v, T)

axiom Decision is
fa(p:Processors, v:ProcDeci, T:Clockvalues)
~(Proposal(p, v, T)) & Decision(p, v, T)

axiom Valiconsensus is
fa(p,q:Processors, T,i,j:Clockvalues, m:Messages) ex(v:ProcDeci)
ValiBroad(p, m, T) & Decision(p, v, T) => Proposal(q, v, T)

axiom Agreeconsensus is
fa(p,q:Processors, v:ProcDeci, T,i,j:Clockvalues, m:Messages)
AgreeBroad(p, m, T) & Decision(p, v, T) => Decision(q, v, T)

endspec

%
```

```
%Morphisms linking RELIABLEBROADCAST and CONSENSUS protocols
%
RELBROADtoCONSENSUS = morphism RELIABLEBROADCAST->CONSENSUS
{Broadcast+->Broadcast,Deliver+->Deliver,TermBroad+->TermBroad,
ValiBroad +-> ValiBroad, AgreeBroad +-> AgreeBroad}

%
%Colimit diagram between RELIABLEBROADCAST and CONSENSUS protocols
%
CONSEN = diagram {
a +-> RELIABLEBROADCAST,
b +-> CONSENSUS,
i: a->b +-> morphism RELIABLEBROADCAST->CONSENSUS
{Broadcast+->Broadcast, Deliver+->Deliver,TermBroad+->TermBroad,
ValiBroad +-> ValiBroad, AgreeBroad +-> AgreeBroad}}

CONSENT = colimit CONSEN

%
%Translation of parameters and properties to other blocks
%
CONSENTtoALLTRANSLATION = translate(CONSENSUS) by
{Clockvalues +-> Clockvalues, LocalClockvals +-> LocalClockvals,
Processors +-> Processors, Index +-> Index, Messages +-> Messages,
Procstate +-> Procstate, Correct +-> Correct, InOrder +-> InOrder,
Broadcast +-> Broadcast, Deliver +-> Deliver,
ReliableNetwork+->ReliableNetwork,BroadcastDelay+->BroadcastDelay,
BroadcastBound +-> BroadcastBound, TermBroad +-> TermBroad,
ValiBroad +-> ValiBroad, AgreeBroad +-> AgreeBroad,
Valiconsensus +-> Valiconsensus, Agreeconsensus+->Agreeconsensus,
ProcDeci +-> ProcDeci, Decision +-> Decision, Proposal+->Proposal}

%
%Specification of the SNAPSHOT protocol
%
SNAPSHOT = spec
import CONSENTtoALLTRANSLATION
sort States
sort Channel
sort Null = Messages
sort Statestabstorage = Boolean
op sending : Processors*Messages*Channel*Processors*Clockvalues->Boolean
op reception : Processors*Messages*Channel*Processors*Clockvalues->
Boolean
op record : Processors*States*Messages*Statestabstorage->Boolean

%
%Various properties of the SNAPSHOT protocol
%
axiom sending is
fa(p,q:Processors, M:Messages, c:Channel, T:Clockvalues)
~(reception(p, M, c, q, T)) & sending(p, M, c, q, T)

axiom reception is
fa(p,q:Processors, M:Messages, c:Channel, T:Clockvalues)
~(sending(p, M, c, q, T)) & reception(p, M, c, q, T)

axiom record is
fa(p,q:Processors, M:Messages, c:Channel, T:Clockvalues)
fa(s:States, X:Statestabstorage) record(p, s, M, X)

axiom Globprocstateinfo is
fa(p,q:Processors) fa(m,M,N,Null:Messages) fa(c:Channel,T,T':Clockvalues)
fa(s,S:States, commit:ProcDeci) fa(X:Statestabstorage)
Agreeconsensus(p,commit,T) & sending(p, M,c,q,T) & record(p,s,N,X)
& ~(sending(p, m, c, q, T')) => reception(q, M, c, p, T) =>
(if(~(record(q, s, M, X)))
then (record(q, s, M, X) & record(q, S, Null, X))
else (record(q,S,m,X) & record(q,s,N,X) & ~(reception(q,M,c,p,T))))

endspec

%
%Morphisms linking CONSENSUS and SNAPSHOT protocols
%
CONSENTtoSNAPSHOT = morphism CONSENSUS->SNAPSHOT
{Decision +-> Decision, Proposal +-> Proposal,
Valiconsensus +-> Valiconsensus, Agreeconsensus +-> Agreeconsensus}

%
%Colimit diagram between CONSENSUS and SNAPSHOT protocols
%
SNAPS = diagram {
a +-> CONSENSUS,
b +-> SNAPSHOT,
i: a->b +-> morphism CONSENSUS->SNAPSHOT
{Decision +-> Decision, Proposal +-> Proposal,
Valiconsensus +-> Valiconsensus, Agreeconsensus +-> Agreeconsensus}}

SNAP = colimit SNAPS

%
%Translation of parameters and properties to other blocks
%
SNAPtoALLTRANSLATION = translate (SNAPSHOT) by
{Clockvalues +-> Clockvalues, LocalClockvals +-> LocalClockvals,
Processors +-> Processors, Index +-> Index, Messages +-> Messages,
Procstate +-> Procstate, Correct +-> Correct, InOrder +-> InOrder,
Broadcast +-> Broadcast, Deliver +-> Deliver,
ReliableNetwork +-> ReliableNetwork, BroadcastDelay+->BroadcastDelay,
BroadcastBound +-> BroadcastBound, TermBroad +-> TermBroad,
ValiBroad +-> ValiBroad, AgreeBroad +-> AgreeBroad,
Valiconsensus +-> Valiconsensus, Agreeconsensus +-> Agreeconsensus,
ProcDeci +-> ProcDeci, Decision +-> Decision, Proposal+->Proposal,
sending +-> sending, reception +-> reception, record+->record}

%
%Specification of the DECISIONMAKING protocol
%
DECISIONMAKING = spec
import SNAPtoALLTRANSLATION
op next : ProcDeci*ProcDeci->Boolean
op adjacent : ProcDeci*ProcDeci->Boolean
op inconsistent : ProcDeci*ProcDeci->Boolean

%
%Various properties of the DECISIONMAKING protocol
%
axiom next is
fa(commit,abort:ProcDeci) ~(adjacent(~(commit),commit)) &
next(commit,abort)

axiom adjacent is
fa(commit,abort:ProcDeci) ~(next(commit,abort)) &
adjacent(~(commit),commit)
```

```
axiom inconsistent is
fa(commit,abort:ProcDeci) adjacent(commit,commit) &
next(commit,abort)

axiom Constateinfo is
fa(p,q:Processors) fa(commit,abort:ProcDeci, s:States, M:Messages)
fa(X:Statestabstorage) record(q,s,M,X) & (~(next(commit,abort))) &
adjacent(~(commit),commit)

%
%Theorem to be proved
%
theorem CSM is
fa(p,q:Processors, T:Clockvalues, m,M,N,Null:Messages, c:Channel)
fa(i:BroadcastDelay, j:BroadcastBound, s,S:States)
fa(v,commit,abort:ProcDeci, X:Statestabstorage)
(
if((Deliver(p, m, T) => Deliver(q, m, (Clockbound(T, i, j)))) &
(AgreeBroad(p,m,T) & Decision(p,v,T)=>AgreeBroad(q,m,(Clockbound(T,i,j)))
& Decision(q, v, T)) & ((Agreeconsensus(p, commit, T) & record(q,s,M,X)
& record(q, s, Null, X)) or (record(q, s, m, X) & record(q, s, N, X) &
(~(reception(q, M, c, p, T))))))
then(record(q,s,M,X) & (~(next(commit,abort))) &
adjacent(~(commit),commit))
else(inconsistent(commit,abort)))

endspec

%
%Morphisms linking SNAPSHOT and DECISIONMAKING protocols
%
SNAPtoDECISIONMAKING = morphism SNAPSHOT->DECISIONMAKING
{sending +-> sending, reception +-> reception, record +-> record}

%
%Colimit diagram between SNAPSHOT and DECISIONMAKING protocols
%
DECMAK = diagram {
a +-> SNAPSHOT,
b +-> DECISIONMAKING,
i: a->b +-> morphism SNAPSHOT->DECISIONMAKING
{sending +-> sending, reception +-> reception, record +-> record}}
```

```
DECISION = colimit DECMAK

foo = print DECISION

%
%Translation of parameters and properties to other blocks
%
DECISIONtoALLTRANSLATION = translate(DECISIONMAKING) by
{Clockvalues +-> Clockvalues, LocalClockvals +-> LocalClockvals,
Processors +-> Processors, Index +-> Index, Messages +-> Messages,
Procstate +-> Procstate, Correct +-> Correct, InOrder +-> InOrder,
Broadcast +-> Broadcast, Deliver +-> Deliver,
ReliableNetwork +-> ReliableNetwork, BroadcastDelay +-> BroadcastDelay,
BroadcastBound +-> BroadcastBound, TermBroad +-> TermBroad,
ValiBroad +-> ValiBroad, AgreeBroad +-> AgreeBroad,
Valiconsensus +-> Valiconsensus, Agreeconsensus +-> Agreeconsensus,
ProcDeci +-> ProcDeci, Decision +-> Decision, Proposal +-> Proposal,
sending +-> sending, reception +-> reception, record +-> record,
next +-> next, adjacent +-> adjacent}

%
%Proof of the consistent state maintenance property
%
p2 = prove CSM in DECISIONMAKING using Agreebroad Agreeconsensus
Globprocstateinfo Constateinfo inconsistent
```

5.1.3 Specification of Roll-Back Recovery Property

```
%
%Specification of the basic building primitives
%
BBB = spec
sort Clockvalues = Nat
sort LocalClockvals = Clockvalues
sort Processors
sort Index
sort Messages = {p:Processors, Tm:Clockvalues, Km:Index, No:Nat}
sort Procstate = {p:Processors, LC:Clockvalues, n:Nat}
op Correct : Processors->Boolean
op InOrder : Messages->Boolean
op Broadcast : Processors*Messages*Clockvalues->Boolean
op Deliver : Processors*Messages*Clockvalues->Boolean
endspec

%
%Translation of parameters to other blocks
%
BBBtoALLTRANSLATION = translate(BBB) by
{Clockvalues +-> Clockvalues, LocalClockvals +-> LocalClockvals,
Processors +-> Processors, Index +-> Index, Messages +-> Messages,
Procstate +-> Procstate, Correct +-> Correct, InOrder +-> InOrder,
Broadcast +-> Broadcast, Deliver +-> Deliver}

%
%Specification of the RELIABLEBROADCAST protocol
%
RELIABLEBROADCAST = spec
import BBBtoALLTRANSLATION
sort ReliableNetwork = Boolean
sort BroadcastDelay = Clockvalues
sort BroadcastBound = Clockvalues
op Clockdelay : Clockvalues*BroadcastDelay->Clockvalues
op Clockbound : Clockvalues*BroadcastDelay*BroadcastBound->Clockvalues
op TermBroad : Processors*Messages*Clockvalues->Boolean
op ValiBroad :  Processors*Messages*Clockvalues->Boolean
op AgreeBroad : Processors*Messages*Clockvalues->Boolean

%
%Various properties of the RELIABLEBROADCAST protocol
%
axiom Broadcast is
fa(p:Processors, m:Messages, T:Clockvalues)
~(Deliver(p, m, T)) & Broadcast(p, m, T)

axiom Deliver is
fa(p:Processors, m:Messages, T:Clockvalues)
~(Broadcast(p, m, T)) & Deliver(p, m, T)

axiom Termbroad is
ex(p, m, T) Correct(p) & Broadcast(p, m, T) =>
(fa (q, i:BroadcastDelay) Correct(q) & Deliver(q,m,(Clockdelay(T,i))))

axiom Valibroad is
```

```
ex(p, m, T) Correct(p) & Broadcast(p, m, T) =>
(fa (q, i:BroadcastDelay, j:BroadcastBound) Correct(q) &
Deliver(q, m, (Clockbound(T, i, j))) & (i<j))

axiom Agreebroad is
ex(p) fa(m:Messages, T:Clockvalues) Deliver(p, m, T) =>
(fa (q, i:BroadcastDelay, j:BroadcastBound)
Deliver(q, m, Clockbound(T, i, j)))

endspec

%
%Translation of parameters and properties to other blocks
%
RELBROADtoALLTRANSLATION = translate(RELIABLEBROADCAST) by
{Clockvalues +-> Clockvalues, LocalClockvals +-> LocalClockvals,
Processors +-> Processors, Index +-> Index, Messages +-> Messages,
Procstate +-> Procstate, Correct +-> Correct, InOrder +-> InOrder,
Broadcast +-> Broadcast, Deliver +-> Deliver,
ReliableNetwork+->ReliableNetwork,BroadcastDelay+->BroadcastDelay,
BroadcastBound +-> BroadcastBound, TermBroad +-> TermBroad,
ValiBroad +-> ValiBroad, AgreeBroad +-> AgreeBroad}

%
%Specification of the CONSENSUS protocol
%
CONSENSUS = spec
import RELBROADtoALLTRANSLATION
sort ProcDeci = Boolean
op Decision : Processors*ProcDeci*Clockvalues->Boolean
op Proposal : Processors*ProcDeci*Clockvalues->Boolean
op Valiconsensus : Processors*ProcDeci*Clockvalues->Boolean
op Agreeconsensus : Processors*ProcDeci*Clockvalues->Boolean

%
%Various properties of the CONSENSUS protocol
%
axiom Proposal is
fa(p:Processors, v:ProcDeci, T:Clockvalues)
~(Decision(p, v, T)) & Proposal(p, v, T)

axiom Decision is
```

```
fa(p:Processors, v:ProcDeci, T:Clockvalues)
~(Proposal(p, v, T)) & Decision(p, v, T)

axiom Valiconsensus is
fa(p,q:Processors, T,i,j:Clockvalues, m:Messages) ex(v:ProcDeci)
ValiBroad(p, m, T) & Decision(p, v, T) => Proposal(q, v, T)

axiom Agreeconsensus is
fa(p,q:Processors, v:ProcDeci, T,i,j:Clockvalues, m:Messages)
AgreeBroad(p, m, T) & Decision(p, v, T) => Decision(q, v, T)

endspec

%
%Morphisms linking RELIABLEBROADCAST and CONSENSUS protocols
%
RELBROADtoCONSENSUS = morphism RELIABLEBROADCAST->CONSENSUS
{Broadcast +-> Broadcast, Deliver +-> Deliver,TermBroad+->TermBroad,
ValiBroad +-> ValiBroad, AgreeBroad +-> AgreeBroad}

%
%Colimit diagram between RELIABLEBROADCAST and CONSENSUS protocols
%
CONSEN = diagram {
a +-> RELIABLEBROADCAST,
b +-> CONSENSUS,
i: a->b +-> morphism RELIABLEBROADCAST->CONSENSUS
{Broadcast +-> Broadcast, Deliver +-> Deliver,TermBroad+->TermBroad,
ValiBroad +-> ValiBroad, AgreeBroad +-> AgreeBroad}}

CONSENT = colimit CONSEN

%
%Translation of parameters and properties to other blocks
%
CONSENTtoALLTRANSLATION = translate(CONSENSUS) by
{Clockvalues +-> Clockvalues, LocalClockvals +-> LocalClockvals,
Processors +-> Processors, Index +-> Index, Messages +-> Messages,
Procstate +-> Procstate, Correct +-> Correct, InOrder +-> InOrder,
Broadcast +-> Broadcast, Deliver +-> Deliver,
ReliableNetwork+->ReliableNetwork,BroadcastDelay+->BroadcastDelay,
BroadcastBound +-> BroadcastBound, TermBroad +-> TermBroad,
ValiBroad +-> ValiBroad, AgreeBroad +-> AgreeBroad,
Valiconsensus +-> Valiconsensus, Agreeconsensus+->Agreeconsensus,
ProcDeci +-> ProcDeci, Decision +-> Decision,Proposal+->Proposal}

%
%Specification of the UNDOREDO protocol
%
UNDOREDO = spec
import CONSENTtoALLTRANSLATION
sort Transactions = Boolean
sort Valstabstorage = Boolean
sort Currentstatevalue = Nat
sort Newstatevalue = Nat
op Log : Transactions*Valstabstorage*Newstatevalue->Boolean
op Undo : Transactions*ProcDeci*Valstabstorage*Currentstatevalue->
Boolean
op Redo : Transactions*ProcDeci*Valstabstorage*Newstatevalue->Boolean
op Storevalues : Transactions*Valstabstorage*ProcDeci->Boolean

%
%Various properties of UNDOREDO protocol
%
axiom Undo is
fa(t:Transactions, a:ProcDeci, X:Valstabstorage, y:Currentstatevalue)
~(Redo(t, a, X, y)) & Undo(t, a, X, y)

axiom Redo is
fa(t:Transactions, a:ProcDeci, X:Valstabstorage, y:Currentstatevalue)
~(Undo(t, a, X, y)) & Redo(t, a, X, y)

axiom Log is
fa(t:Transactions, a:ProcDeci, X:Valstabstorage)
fa(y:Currentstatevalue, z:Newstatevalue)
~(Undo(t, a, X, y)) & ~(Redo(t, a, X, y)) => Log(t, X, z)

axiom Storevalues is
fa(p,q:Processors) fa(T:Clockvalues,t:Transactions)
fa(commit,abort:ProcDeci)
fa(y:Currentstatevalue, z:Newstatevalue, X:Valstabstorage)
Agreeconsensus(p,commit,T) & Undo(t,abort,X,y) & Redo(t,commit,X,z) =>
Log(t, X, z)
```

```
endspec

%
%Morphisms linking CONSENSUS and UNDOREDO protocols
%
CONSENTtoUNDOREDO = morphism CONSENSUS->UNDOREDO
{Valiconsensus +-> Valiconsensus, Agreeconsensus +-> Agreeconsensus,
Decision +-> Decision, Proposal +-> Proposal}

%
%Colimit diagram between CONSENSUS and UNDOREDO protocols
%
UNRE = diagram {
a +-> CONSENSUS,
b +-> UNDOREDO,
i: a->b +-> morphism CONSENSUS->UNDOREDO
{Valiconsensus +-> Valiconsensus, Agreeconsensus +-> Agreeconsensus,
Decision +-> Decision, Proposal +-> Proposal}}

UNREDO = colimit UNRE

%
%Translation of parameters and properties to other blocks
%
UNREDOtoALLTRANSLATION = translate(UNDOREDO) by
{Clockvalues +-> Clockvalues, LocalClockvals +-> LocalClockvals,
Processors +-> Processors, Index +-> Index, Messages +-> Messages,
Procstate +-> Procstate, Correct +-> Correct, InOrder +-> InOrder,
Broadcast +-> Broadcast, Deliver +-> Deliver,
ReliableNetwork +-> ReliableNetwork, BroadcastDelay +-> BroadcastDelay,
BroadcastBound +-> BroadcastBound, TermBroad +-> TermBroad,
ValiBroad +-> ValiBroad, AgreeBroad +-> AgreeBroad, ProcDeci+->ProcDeci,
Valiconsensus +-> Valiconsensus, Agreeconsensus +-> Agreeconsensus,
Decision +-> Decision,Proposal +-> Proposal,Transactions+->Transactions,
Valstabstorage +-> Valstabstorage,Currentstatevalue+->Currentstatevalue,
Newstatevalue +-> Newstatevalue, Log +-> Log, Undo +-> Undo,
Redo +-> Redo, Storevalues +-> Storevalues}

%
%Specification of the TWOPHASELOCK protocol
%
TWOPHASELOCK = spec
```

```
import UNREDOtoALLTRANSLATION
sort Transactionid
sort CurrentData
sort PreviousData
op Read : Transactions*CurrentData*Valstabstorage->Boolean
op Write : Transactions*CurrentData*Valstabstorage->Boolean
op Locking : Transactionid*CurrentData->Boolean
op Unlock : Transactionid*PreviousData->Boolean
op Readlock : Transactions*CurrentData*Valstabstorage->Boolean
op Writelock : Transactions*CurrentData*Valstabstorage->Boolean

%
%Various properties of TWOPHASELOCK protocol
%
axiom Read is
fa(t:Transactions, Y:CurrentData, X:Valstabstorage)
~(Write(t, Y, X)) & Read(t, Y, X)

axiom Write is
fa(t:Transactions, Y:CurrentData, X:Valstabstorage)
~(Read(t, Y, X)) & Write(t, Y, X)

axiom Locking is
fa(N:Transactionid, Y:CurrentData, Z:PreviousData)
(Unlock(N, Z)) & Locking(N, Y)

axiom Unlock is
fa(N:Transactionid, Y:CurrentData, Z:PreviousData)
~(Locking(N, Y)) & Unlock(N, Z)

axiom Readlock is
fa(p,q:Processors) fa(t:Transactions, N:Transactionid, X:Valstabstorage)
fa(Y:CurrentData, Z:PreviousData, z:Newstatevalue) Log(t, X, z) &
~(Write(t, Y, X)) & ~(Locking(N, Y)) & Unlock(N, Z) => Read(t, Y, X) &
Locking(N, Y)

axiom Writelock is
fa(p,q:Processors) fa(t:Transactions, N:Transactionid, X:Valstabstorage)
fa(Y:CurrentData, Z:PreviousData, z:Newstatevalue) Log(t, X, z) &
~(Read(t, Y, X)) & ~(Locking(N, Y)) & Unlock(N, Z) => Write(t, Y, X) &
Locking(N, Y)
```

```
endspec

%
%Morphisms linking UNDOREDO and TWOPHASELOCK protocols
%
UNREDOtoTWOPHASELOCK = morphism UNDOREDO->TWOPHASELOCK
{Undo +-> Undo, Redo +-> Redo, Storevalues +-> Storevalues}

%
%Colimit diagram between UNDOREDO and TWOPHASELOCK protocols
%
TPLOCK = diagram {
a +-> UNDOREDO,
b +-> TWOPHASELOCK,
i: a->b +-> morphism UNDOREDO->TWOPHASELOCK
{Undo +-> Undo, Redo +-> Redo, Storevalues +-> Storevalues}}

TPL = colimit TPLOCK

%
%Translation of parameters and properties to other blocks
%
TPLtoALLTRANSLATION = translate(TWOPHASELOCK) by
{Clockvalues +-> Clockvalues, LocalClockvals +-> LocalClockvals,
Processors +-> Processors, Index +-> Index, Messages +-> Messages,
Procstate +-> Procstate, Correct +-> Correct, InOrder +-> InOrder,
Broadcast +-> Broadcast, Deliver +-> Deliver,
ReliableNetwork +-> ReliableNetwork, BroadcastDelay +-> BroadcastDelay,
BroadcastBound +-> BroadcastBound, TermBroad +-> TermBroad,
ValiBroad +-> ValiBroad, AgreeBroad +-> AgreeBroad,ProcDeci+->ProcDeci,
Valiconsensus +-> Valiconsensus, Agreeconsensus +-> Agreeconsensus,
Decision+->Decision,Proposal+->Proposal,Transactions+->Transactions,
Valstabstorage+->Valstabstorage,Currentstatevalue+->Currentstatevalue,
Newstatevalue +-> Newstatevalue, Log +-> Log, Undo+->Undo,Redo+->Redo,
Storevalues +-> Storevalues, Read +-> Read, Write +-> Write,
Locking +-> Locking, Unlock +-> Unlock, Readlock +-> Readlock,
Writelock +-> Writelock}

%
%Specification of the CHECKPOINTING protocol
%
CHECKPOINTING = spec
```

```
import TPLtoALLTRANSLATION
op C : Processors*Clockvalues->LocalClockvals
op receive : Processors*Messages*Processors*Clockvalues->Boolean
op send : Processors*Messages*Processors*Clockvalues->Boolean
op log : Processors*Messages*Clockvalues->Boolean
op Ckpt : Processors*LocalClockvals->Boolean
op ckpt : Processors*Clockvalues->Boolean
op Store : Processors*LocalClockvals->Boolean
op store : Processors*Clockvalues->Boolean
op Pi : Processors*Clockvalues->Boolean
op PI : Processors*LocalClockvals->Boolean
op Checkpoint : Processors*Clockvalues->Boolean

%
%Various properties of CHECKPOINTING protocol
%
axiom receive is
fa(p,q:Processors, m:Messages, T:Clockvalues)
~(send(p, m, q, T)) & receive(p, m, q, T)

axiom send is
fa(p,q:Processors, m:Messages, T:Clockvalues)
~(receive(p, m, q, T)) & send(p, m, q, T)

axiom log is
fa(p,q:Processors, m:Messages, T:Clockvalues)
receive(p, m, q, T) & log(p, m ,T)

axiom Ckpt is
fa(p:Processors, T:Clockvalues, S:LocalClockvals)
~(ckpt(p,T)) & Ckpt(p,S)

axiom ckpt is
fa(p:Processors, T:Clockvalues, S:LocalClockvals)
~(Ckpt(p,S)) & ckpt(p,T)

axiom Store is
fa(p:Processors, T:Clockvalues, S:LocalClockvals)
~(store(p,T)) & Store(p,S)

axiom store is
fa(p:Processors, T:Clockvalues, S:LocalClockvals)
```

```
~(Store(p,S)) & store(p,T)

axiom Pi is
fa(p:Processors, T:Clockvalues, S:LocalClockvals)
~(PI(p,S)) & Pi(p,T)

axiom PI is
fa(p:Processors, T:Clockvalues, S:LocalClockvals)
~(Pi(p,T)) & PI(p,S)

axiom Logging is
fa(m:Messages) fa(p,q:Processors)
fa(e,T:Clockvalues, S:LocalClockvals, i:BroadcastDelay, j:BroadcastBound)
fa(t:Transactions, Y:CurrentData, X:Valstabstorage)
Readlock(t, Y, X) & ~(Writelock(t, Y, X)) &
((S-i-e)<C(p,T)) & (C(p,T)<=(S+j+e)) =>
(receive(p, m, q, T) => log(p, m, T))

axiom Checkpoint is
fa(m:Messages) fa(p:Processors) fa(n:Index)
fa(e,T:Clockvalues, S:LocalClockvals, i:BroadcastDelay, j:BroadcastBound)
fa(t:Transactions, Y:CurrentData, X:Valstabstorage)
~(Readlock(t, Y, X)) & Writelock(t, Y, X) &
(S-i-e)<(C(p,T)) & (C(p,T)<=S) =>
(if (ex(m) log(p,m,T) & (C(p,T)<S))
 then (ckpt(p,T) & store(p,T) & Pi(p,T))
 else (Ckpt(p,S) & Store(p,S) & PI(p,S)))

endspec

%
%Morphisms linking TWOPHASELOCK and CHECKPOINTING protocols
%
TPLtoCHECKPOINTING = morphism TWOPHASELOCK->CHECKPOINTING
{Read +-> Read, Write +-> Write, Locking +-> Locking,
Unlock +-> Unlock, Readlock +-> Readlock, Writelock +-> Writelock}

%
%Colimit diagram between TWOPHASELOCK and CHECKPOINTING protocols
%
CKPOINTING = diagram {
a +-> TWOPHASELOCK,
b +-> CHECKPOINTING,
i: a->b +-> morphism TWOPHASELOCK->CHECKPOINTING
{Read +-> Read, Write +-> Write, Locking +-> Locking,
Unlock +-> Unlock, Readlock +-> Readlock, Writelock +-> Writelock}}

CKPT = colimit CKPOINTING

%
%Translation of parameters and properties to other blocks
%
CKPTtoALLTRANSLATION = translate(CHECKPOINTING) by
{Clockvalues +-> Clockvalues, LocalClockvals +-> LocalClockvals,
Processors +-> Processors, Index +-> Index, Messages +-> Messages,
Procstate +-> Procstate, Correct +-> Correct, InOrder +-> InOrder,
Broadcast +-> Broadcast, Deliver +-> Deliver,
ReliableNetwork +-> ReliableNetwork, BroadcastDelay +-> BroadcastDelay,
BroadcastBound +-> BroadcastBound, TermBroad +-> TermBroad,
ValiBroad +-> ValiBroad,AgreeBroad +-> AgreeBroad,ProcDeci +-> ProcDeci,
Valiconsensus +-> Valiconsensus, Agreeconsensus +-> Agreeconsensus,
Decision +-> Decision,Proposal +-> Proposal,Transactions+->Transactions,
Valstabstorage +-> Valstabstorage,Currentstatevalue+->Currentstatevalue,
Newstatevalue +-> Newstatevalue,Log +-> Log,Undo +-> Undo,Redo +-> Redo,
Storevalues +-> Storevalues, Read +-> Read, Write +-> Write,
Locking +-> Locking, Unlock +-> Unlock, Readlock +-> Readlock,
Writelock +-> Writelock,receive +-> receive,log +-> log,Ckpt +-> Ckpt,
ckpt +-> ckpt, Store +-> Store, store +-> store, Pi +-> Pi, PI +-> PI,
Checkpoint +-> Checkpoint}

%
%Specification of the ROLLBACKRECOVERY protocol
%
ROLLBACKRECOVERY = spec
import CKPTtoALLTRANSLATION
op CorrecttoFailure : Processors*Clockvalues->Boolean
op Rollback : Index*Clockvalues->Boolean
op Restore : Index*Clockvalues->Boolean
op Recover : Index*Clockvalues->Boolean
op rollback : Index*LocalClockvals->Boolean
op restore : Index*LocalClockvals->Boolean
op recover : Index*LocalClockvals->Boolean

%
```

```
%Various properties of ROLLBACKRECOVERY protocol
%
axiom CorrecttoFailure is
fa(p:Processors, T:Clockvalues)
Correct(p) & CorrecttoFailure(p,T)

axiom Rollback is
fa(n:Index, T:Clockvalues)
~(Restore(n,T)) & Rollback(n,T)

axiom Restore is
fa(n:Index, T:Clockvalues)
~(Rollback(n,T)) & Restore(n,T)

axiom rollback is
fa(n:Index, S:LocalClockvals)
~(restore(n,S)) & rollback(n,S)

axiom restore is
fa(n:Index, S:LocalClockvals)
~(rollback(n,S)) & restore(n,S)

axiom Recover is
fa(p:Processors, n:Index) fa(e,T:Clockvalues)
fa(i:BroadcastDelay, j:BroadcastBound, S:LocalClockvals) Checkpoint(p,T)
& ((S-i-e)<C(p,T)) & (C(p,T)<=S) & CorrecttoFailure(p,T) &
(ckpt(p,T) => Rollback(n,T) => Restore(n,T))

axiom recover is
fa(p:Processors, n:Index) fa(e,T:Clockvalues)
fa(i:BroadcastDelay, j:BroadcastBound, S:LocalClockvals) Checkpoint(p,T)
& ((S-i-e)<C(p,T)) & (C(p,T)<=S) & CorrecttoFailure(p,T) &
(Ckpt(p,S) => rollback(n,S) => restore(n,S))

%
%Theorem to be proved
%
theorem RBR is
fa(p,q:Processors, T:Clockvalues, m:Messages, t:Transactions, n:Index)
fa(i:BroadcastDelay, j:BroadcastBound, S:LocalClockvals)
fa(v,commit,abort:ProcDeci, N:Transactionid, X:Valstabstorage)
fa(y:Currentstatevalue, z:Newstatevalue, Y:CurrentData, Z:PreviousData)
```

```
(
if( (Deliver(p, m, T) => Deliver(q, m, (Clockbound(T, i, j)))) &
(AgreeBroad(p,m,T) & Decision(p,v,T)=>AgreeBroad(q,m,(Clockbound(T,i,j)))
& Decision(q, v, T)) & (Agreeconsensus(p,commit,T) & Undo(t,abort,X,y)
& Redo(t,commit,X,z) => Log(t,X,z)) & ((Log(t,X,z) & (~(Write(t,Y,X))) &
(~(Locking(N, Y))) & Unlock(N, Z) => Read(t, Y, X) & Locking(N, Y)) or
(Log(t, X, z) & (~(Read(t,Y,X))) & (~(Locking(N,Y))) & Unlock(N, Z) =>
Write(t,Y,X) & Locking(N,Y))) & ((~(Readlock(t,Y,X)) & Writelock(t,Y,X) &
ckpt(p,T) & store(p,T) & Pi(p,T)) or (Ckpt(p,S) & Store(p,S) & PI(p,S))))
then(ckpt(p,T) => Rollback(n,T) => Restore(n,T))
else(Ckpt(p,S) => rollback(n,S) => restore(n,S)))

endspec

%
%Morphisms linking CHECKPOINTING and ROLLBACKRECOVERY protocols
%
CKPTtoROLLBACKRECOVERY = morphism CHECKPOINTING->ROLLBACKRECOVERY
{receive +-> receive, log +-> log, Ckpt +-> Ckpt, ckpt +-> ckpt,
Store +-> Store, store +-> store, Pi +-> Pi, PI +-> PI,
Checkpoint +-> Checkpoint}

%
%Colimit diagram between CHECKPOINTING and ROLLBACKRECOVERY protocols
%
RCOV = diagram {
a +-> CHECKPOINTING,
b +-> ROLLBACKRECOVERY,
i: a->b +-> morphism CHECKPOINTING->ROLLBACKRECOVERY
{receive +-> receive, log +-> log, Ckpt +-> Ckpt, ckpt +-> ckpt,
Store +-> Store, store +-> store, Pi +-> Pi, PI +-> PI,
Checkpoint +-> Checkpoint}}

RECO = colimit RCOV

foo = print RECO

%
%Translation of parameters and properties to other blocks
%
RECOtoALLTRANSLATION = translate(ROLLBACKRECOVERY) by
{Clockvalues +-> Clockvalues, LocalClockvals +-> LocalClockvals,
Processors +-> Processors, Index +-> Index, Messages +-> Messages,
Procstate +-> Procstate, Correct +-> Correct, InOrder +-> InOrder,
Broadcast +-> Broadcast, Deliver +-> Deliver,
ReliableNetwork +-> ReliableNetwork, BroadcastDelay +-> BroadcastDelay,
BroadcastBound +-> BroadcastBound, TermBroad +-> TermBroad,
ValiBroad +-> ValiBroad,AgreeBroad +-> AgreeBroad,ProcDeci +-> ProcDeci,
Valiconsensus +-> Valiconsensus, Agreeconsensus +-> Agreeconsensus,
Decision +-> Decision,Proposal +-> Proposal,Transactions+->Transactions,
Valstabstorage +-> Valstabstorage,Currentstatevalue+->Currentstatevalue,
Newstatevalue +-> Newstatevalue,Log +-> Log,Undo +-> Undo,Redo +-> Redo,
Storevalues +-> Storevalues, Read +-> Read, Write +-> Write,
Locking +-> Locking, Unlock +-> Unlock, Readlock +-> Readlock,
Writelock +-> Writelock,receive +-> receive,log +-> log,Ckpt +-> Ckpt,
ckpt +-> ckpt, Store +-> Store, store +-> store, Pi +-> Pi, PI +-> PI,
Checkpoint +-> Checkpoint, Rollback +-> Rollback,
CorrecttoFailure +-> CorrecttoFailure, Restore +-> Restore,
rollback +-> rollback, restore +-> restore, Recover +-> Recover,
recover +-> recover}

%
%Proof of the Rollback Recovery property
%
p3 = prove RBR in ROLLBACKRECOVERY using Agreebroad Agreeconsensus
Storevalues Readlock Writelock Checkpoint Recover recover
```

Chapter 6
Conclusion

Our aim in this thesis had been to apply our proposed category-theoretical approach for protocol composition to a complex (and also a practical) transaction processing protocol integrating *all* sub-protocols which are instrumental in achieving the correct protocol level operations. In order to obtain the overall global properties of the protocol, we first identified the various building blocks of a *transaction processing protocol* namely the *centralized non-blocking three phase commit (3PC) protocol* highlighting their inter-dependencies and functionalities as shown in chapter 3. We then applied the concepts of category theory to compose the 3PC protocol by utilizing its various sub-protocols as shown in chapter 4. We also demonstrated in the same chapter that how by breaking down complex protocol blocks into smaller sub-blocks, it becomes relatively easy to verify the global properties of the protocol. This is because, by verifying (and utilizing) the smaller sub-protocols, we can be rest assured the correctness of the overall complex protocol as it is now formed of inter-related sub-protocols. In this chapter, we first discuss some of the insights we got from the modular composition and verification of three phase commit protocol in terms of experience and viewpoints. We then provide details on our ongoing and future work in modular protocol composition and verification.

Experience & Viewpoint

We highlight some of the observations we have made over our effort in modularly composing the 3PC protocol.

- A major obstacle which we overcame was related to the process of identifying the various sub-blocks for the three phase commit protocol and also their inter-dependencies based on the overall functionality of the protocol. The difficulty was due to the fact that for different functionalities of the 3PC, the relationships between the sub-blocks vary widely and there is always a possibility of an addition or deletion of an identified sub-block.
- Since we have followed the *bottom-up* approach in the proofs of the global properties of the three phase commit protocol, there is a high dependency of one property on the other. For the final global property to be proved, it's a must that the base properties are first validated, and as we go up the hierarchy, it is constrained that base properties are always maintained. This is a challenging task in itself.
- With respect to the previous point, there is an imminent need for a technique which would help us answer the following question: *Which kind of protocol verification properties can be decomposed, and which cannot?* We believe that an answer to this question necessitates development of a formal framework

which would facilitate mechanisms for managing and analyzing dependencies among sub-protocols. We are currently investigating this aspect of modular composition of protocols.

- The morphisms that we had considered while proving the global properties of the 3PC are different for different properties and also vary for different protocols. We acknowledge that the sub-protocols we have identified probably would not cover the entire class of transaction processing protocols. However, with this as an initial stage, we are currently looking into coming up with a generic model which could handle most protocols within this class of protocols.
- It is possible that two initially non-conflicting sub-blocks may end up being operationally conflicting at some arbitrary implementation details which are not apparent at the high level of specification. Determining how much and how often implementation details will be needed to capture all subtleties of sub-block interactions is a challenging problem, and demands further research in this direction.

6.1 Discussions and Conclusion

In this thesis, we have shown how a complex protocol like the three phase commit protocol can be broken down into smaller sub–protocols by modularization. Given our approach, it becomes easier for the protocol specification and design community, since their task of specification and design is now confined to only smaller pieces of a global protocol. Also by this methodology, the testing and validation of complex protocols becomes simpler.

Through the case study of transaction processing protocol. our specific aim in this thesis has been to illustrate the fact that our proposed category-theory based formal framework serves as a design tool to systematically compose distributed dependable protocols. As we have been emphasizing throughout this thesis, morphisms linking different modules are effective means of highlighting any conflicts arising over composition. They essentially pinpoint properties that must be observed across different modules. The key idea presented in this thesis is that we can decompose global properties of the protocol level operations into small lemmas (local properties) provable in different sub-protocols and translate these sub-properties along morphisms to establish the desired properties.

We also emphasize that category-based formalization of basic building block-protocols permit reusability of these basic formal modules. Moreover, for any configuration of building blocks over a protocol composition, morphisms also provide a direct capability of tracing desired feasible path of any variable, attribute, or action.

Regarding future work, we plan to expand the proposed category-theory based approach for modular composition, and develop a general theoretical framework where one can potentially plug-in formal theories of specific building blocks to compose and verify the complex protocol level operations. Also we are exploring the possibilities of coming up with a more generic methodology for the identification of building blocks in a protocol, since the ones we had identified in this thesis for the three phase commit protocol, have been done on an ad-hoc basis based on the overall functionalities of the 3PC given in literature.

APPENDIX-I

Specification of the composite protocol PR_2 along with processing steps of Specware and proof results from Snark

```
CL-USER(1): :sw /Progra~1/Specware4.0/Examples/speccode1
 Processing spec at C:/Progra~1/Specware4.0/Examples/speccode1#BBB
 Processing spec at C:/Program Files/Specware4.0/Library/Base
 Processing spec at C:/Program Files/Specware4.0/Library/Base/Boolean
 Processing spec at C:/Program Files/Specware4.0/Library/Base/PrimitiveSorts
 Processing spec at C:/Program Files/Specware4.0/Library/Base/Compare
 Processing spec at C:/Program Files/Specware4.0/Library/Base/Functions
 Processing spec at C:/Program Files/Specware4.0/Library/Base/Integer
 Processing spec at C:/Program Files/Specware4.0/Library/Base/Nat
 Processing spec at C:/Program Files/Specware4.0/Library/Base/Char
 Processing spec at C:/Program Files/Specware4.0/Library/Base/String
 Processing spec at C:/Program Files/Specware4.0/Library/Base/List
 Processing spec at C:/Program Files/Specware4.0/Library/Base/Option
 Processing spec at C:/Program Files/Specware4.0/Library/Base/System
 Processing spec at C:/Program Files/Specware4.0/Library/Base/Show
 Processing translation at C:/Progra~1/Specware4.0/Examples/speccode1
 #BBBtoALLTRANSLATION
 Processing spec at C:/Progra~1/Specware4.0/Examples/speccode1
 #RELIABLEBROADCAST
 Processing translation at C:/Progra~1/Specware4.0/Examples/speccode1
 #RELBROADtoALLTRANSLATION
 Processing spec at C:/Progra~1/Specware4.0/Examples/speccode1#CONSENSUS
 Processing spec morphism at C:/Progra~1/Specware4.0/Examples/speccode1
 #RELBROADtoCONSENSUS
 Processing spec diagram at C:/Progra~1/Specware4.0/Examples/speccode1
 #CONSEN
 Processing spec morphism at C:/Progra~1/Specware4.0/Examples/speccode1
 #CONSEN
 Processing colimit at C:/Progra~1/Specware4.0/Examples/speccode1#CONSENT
 Processing spec at C:/Program Files/Specware4.0/Library/Base
 Processing spec at C:/Program Files/Specware4.0/Library/Base/Boolean
 Processing spec at C:/Program Files/Specware4.0/Library/Base/PrimitiveSorts
 Processing spec at C:/Program Files/Specware4.0/Library/Base/Compare
 Processing spec at C:/Program Files/Specware4.0/Library/Base/Functions
 Processing spec at C:/Program Files/Specware4.0/Library/Base/Integer
 Processing spec at C:/Program Files/Specware4.0/Library/Base/Nat
 Processing spec at C:/Program Files/Specware4.0/Library/Base/Char
 Processing spec at C:/Program Files/Specware4.0/Library/Base/String
 Processing spec at C:/Program Files/Specware4.0/Library/Base/List
 Processing spec at C:/Program Files/Specware4.0/Library/Base/Option
 Processing spec at C:/Program Files/Specware4.0/Library/Base/System
 Processing spec at C:/Program Files/Specware4.0/Library/Base/Show
 Processing translation at C:/Progra~1/Specware4.0/Examples/speccode1
 #CONSENTtoALLTRANSLATION
 Processing spec at C:/Progra~1/Specware4.0/Examples/speccode1#UNDOREDO
 Processing spec morphism at C:/Progra~1/Specware4.0/Examples/speccode1
 #CONSENTtoUNDOREDO
 Processing spec diagram at C:/Progra~1/Specware4.0/Examples/speccode1
 #UNRE
 Processing spec morphism at C:/Progra~1/Specware4.0/Examples/speccode1
 #UNRE
 Processing colimit at C:/Progra~1/Specware4.0/Examples/speccode1#UNREDO
 Processing translation at C:/Progra~1/Specware4.0/Examples/speccode1
 #UNREDOtoALLTRANSLATION
 Processing spec at C:/Progra~1/Specware4.0/Examples/speccode1#TWOPHASELOCK
 Processing spec morphism at C:/Progra~1/Specware4.0/Examples/speccode1
 #UNREDOtoTWOPHASELOCK
 Processing spec diagram at C:/Progra~1/Specware4.0/Examples/speccode1
 #TPLOCK
 Processing spec morphism at C:/Progra~1/Specware4.0/Examples/speccode1
 #TPLOCK
 Processing colimit at C:/Progra~1/Specware4.0/Examples/speccode1#TPL

spec
 sort BroadcastBound = Clockvalues
 sort BroadcastDelay = Clockvalues
 sort Clockvalues = Nat
 sort CurrentData
 sort Currentstatevalue = Nat
sort Index = Nat
sort LocalClockvals = Clockvalues
sort Messages = {Km:Index, No:Nat, Tm:Clockvalues, p:Processors}
sort Newstatevalue = Nat
sort PreviousData
sort ProcDeci = Boolean
sort Procstate = {LC:Clockvalues, n:Nat, p:Processors}
sort Processors
sort ReliableNetwork = Boolean
sort Transactionid
sort Transactions = Boolean
sort Valstabstorage = Boolean
op AgreeBroad : Processors * Messages * Clockvalues -> Boolean
op Agreeconsensus : Processors * ProcDeci * Clockvalues -> Boolean
op Broadcast : Processors * Messages * Clockvalues -> Boolean
op Clockbound : Clockvalues * BroadcastDelay * BroadcastBound->Clockvalues
op Clockdelay : Clockvalues * BroadcastDelay -> Clockvalues
op Correct : Processors -> Boolean
op Decision : Processors * ProcDeci * Clockvalues -> Boolean
op Deliver : Processors * Messages * Clockvalues -> Boolean
op InOrder : Messages -> Boolean
op Locking : Transactionid * CurrentData -> Boolean
op Log : Transactions * Valstabstorage * Newstatevalue -> Boolean
op Proposal : Processors * ProcDeci * Clockvalues -> Boolean
op Read : Transactions * CurrentData * Valstabstorage -> Boolean
op Readlock : Transactions * CurrentData * Valstabstorage -> Boolean
op Redo :
   Transactions * ProcDeci * Valstabstorage * Newstatevalue -> Boolean
op Storevalues : Transactions * Valstabstorage * ProcDeci -> Boolean
op TermBroad : Processors * Messages * Clockvalues -> Boolean
op Undo :
   Transactions * ProcDeci * Valstabstorage * Currentstatevalue->Boolean
op Unlock : Transactionid * PreviousData -> Boolean
op ValiBroad : Processors * Messages * Clockvalues -> Boolean
op Valiconsensus : Processors * ProcDeci * Clockvalues -> Boolean
op Write : Transactions * CurrentData * Valstabstorage -> Boolean
op Writelock : Transactions * CurrentData * Valstabstorage -> Boolean
axiom Read is
   fa(t : Transactions, Y : CurrentData, X : Valstabstorage)
    ~(Write(t, Y, X)) & Read(t, Y, X)
axiom Write is
   fa(t : Transactions, Y : CurrentData, X : Valstabstorage)
```

```
    ~(Read(t, Y, X)) & Write(t, Y, X)
axiom Locking is
   fa(N : Transactionid, Y : CurrentData, Z : PreviousData)
    Unlock(N, Z) & Locking(N, Y)
axiom Unlock is
   fa(N : Transactionid, Y : CurrentData, Z : PreviousData)
    ~(Locking(N, Y)) & Unlock(N, Z)
axiom Readlock is
   fa(p : a, q : Processors)
    fa(t : Transactions, N : Transactionid, X : Valstabstorage)
     fa(Y : CurrentData, Z : PreviousData, z : Newstatevalue)

      (Log(t, X, z) &
      (~(Write(t, Y, X)) & (~(Locking(N, Y)) & Unlock(N, Z)))) =>
      (Read(t, Y, X) & Locking(N, Y))
axiom Writelock is
   fa(p : a, q : Processors)
    fa(t : Transactions, N : Transactionid, X : Valstabstorage)
     fa(Y : CurrentData, Z : PreviousData, z : Newstatevalue)

      (Log(t, X, z) &
      (~(Read(t, Y, X)) & (~(Locking(N, Y)) & Unlock(N, Z)))) =>
      (Write(t, Y, X) & Locking(N, Y))
theorem Serialize is
   fa(p : Processors, q : Processors, T : Clockvalues, m : Messages,
      t : Transactions)
    fa(i : BroadcastDelay, j : BroadcastBound)
     fa(v : ProcDeci, commit : ProcDeci, abort : ProcDeci,
        N : Transactionid, X : Valstabstorage)
      fa(y : Currentstatevalue, z : Newstatevalue, Y : CurrentData,
         Z : PreviousData)
       if (Deliver(p, m, T) => Deliver(q, m, Clockbound(T, i, j))) &
          (
          ((AgreeBroad(p, m, T) & Decision(p, v, T)) =>
          (AgreeBroad(q, m, Clockbound(T, i, j)) & Decision(q, v, T))) &
          (
          (Agreeconsensus(p, commit, T) &
          (Undo(t, abort, X, y) & Redo(t, commit, X, z))) => Log(t, X, z)))
           then

          (Log(t, X, z) &
          (~(Write(t, Y, X)) & (~(Locking(N, Y)) & Unlock(N, Z)))) =>
```

```
            (Read(t, Y, X) & Locking(N, Y))

          else

          (Log(t, X, z) &
          (~(Read(t, Y, X)) & (~(Locking(N, Y)) & Unlock(N, Z)))) =>
          (Write(t, Y, X) & Locking(N, Y))
axiom Storevalues is
   fa(p : Processors, q : Processors)
    fa(T : Clockvalues, t : Transactions)
     fa(commit : ProcDeci, abort : ProcDeci)
      fa(y : Currentstatevalue, z : Newstatevalue, X : Valstabstorage)

       (Agreeconsensus(p, commit, T) &
       (Undo(t, abort, X, y) & Redo(t, commit, X, z))) => Log(t, X, z)
axiom Log is
   fa(t : Transactions, a : ProcDeci, X : Valstabstorage)
    fa(y : Currentstatevalue, z : Newstatevalue)
     (~(Undo(t, a, X, y)) & ~(Redo(t, a, X, y))) => Log(t, X, z)
axiom Redo is
   fa(t : Transactions, a : ProcDeci, X : Valstabstorage,
      y : Currentstatevalue) ~(Undo(t, a, X, y)) & Redo(t, a, X, y)
axiom Undo is
   fa(t : Transactions, a : ProcDeci, X : Valstabstorage,
      y : Currentstatevalue) ~(Redo(t, a, X, y)) & Undo(t, a, X, y)
axiom Agreebroad is
   ex(p : Processors)
    fa(m : Messages, T : Clockvalues)
     Deliver(p, m, T) =>
     fa(q : Processors, i : BroadcastDelay, j : BroadcastBound)
      Deliver(q, m, Clockbound(T, i, j))
axiom Valibroad is
   ex(p : Processors, m : Messages, T : Clockvalues)
    (Correct p & Broadcast(p, m, T)) =>
    fa(q : Processors, i : BroadcastDelay, j : BroadcastBound)
     (Correct q & (Deliver(q, m, Clockbound(T, i, j)) & (i < j)))
axiom Termbroad is
   ex(p : Processors, m : Messages, T : Clockvalues)
    (Correct p & Broadcast(p, m, T)) =>
    fa(q : Processors, i : BroadcastDelay)
     (Correct q & Deliver(q, m, Clockdelay(T, i)))
axiom Deliver is
```

```
    fa(p : Processors, m : Messages, T : Clockvalues)
     ~(Broadcast(p, m, T)) & Deliver(p, m, T)
 axiom Broadcast is
    fa(p : Processors, m : Messages, T : Clockvalues)
     ~(Deliver(p, m, T)) & Broadcast(p, m, T)
 axiom Proposal is
    fa(p : Processors, v : ProcDeci, T : Clockvalues)
     ~(Decision(p, v, T)) & Proposal(p, v, T)
 axiom Decision is
    fa(p : Processors, v : ProcDeci, T : Clockvalues)
     ~(Proposal(p, v, T)) & Decision(p, v, T)
 axiom Valiconsensus is
    fa(p : Processors, q : Processors, T : Clockvalues, i : a,
    j: Clockvalues, m : Messages)
     ex(v : ProcDeci)
      (ValiBroad(p, m, T) & Decision(p, v, T)) => Proposal(q, v, T)
 axiom Agreeconsensus is
    fa(p : Processors, q : Processors, v : ProcDeci, T : Clockvalues,
    i : a, j : Clockvalues, m : Messages)
     (AgreeBroad(p, m, T) & Decision(p, v, T)) => Decision(q, v, T)
endspec

 Processing translation at C:/Progra~1/Specware4.0/Examples/speccode1
 #TPLtoALLTRANSLATION
 Processing prove at C:/Progra~1/Specware4.0/Examples/speccode1#p1
 Processing spec at C:/Program Files/Specware4.0/Library/Base/ProverBase
p1: Theorem Serialize in TWOPHASELOCK is Proved!
    Snark Log file: C:/Progra~1/Specware4.0/Examples/snark/speccode1/p1.log
```

APPENDIX-II

Specification of the composite protocol PR_6 along with processing steps of Specware and proof results from Snark

```
CL-USER(1): :sw /Progra~1/Specware4.0/Examples/speccode2
Processing spec at C:/Progra~1/Specware4.0/Examples/speccode2#BBB
Processing spec at C:/Program Files/Specware4.0/Library/Base
Processing spec at C:/Program Files/Specware4.0/Library/Base/Boolean
Processing spec at C:/Program Files/Specware4.0/Library/Base/PrimitiveSorts
Processing spec at C:/Program Files/Specware4.0/Library/Base/Compare
Processing spec at C:/Program Files/Specware4.0/Library/Base/Functions
Processing spec at C:/Program Files/Specware4.0/Library/Base/Integer
Processing spec at C:/Program Files/Specware4.0/Library/Base/Nat
Processing spec at C:/Program Files/Specware4.0/Library/Base/Char
Processing spec at C:/Program Files/Specware4.0/Library/Base/String
Processing spec at C:/Program Files/Specware4.0/Library/Base/List
Processing spec at C:/Program Files/Specware4.0/Library/Base/Option
Processing spec at C:/Program Files/Specware4.0/Library/Base/System
Processing spec at C:/Program Files/Specware4.0/Library/Base/Show
Processing translation at C:/Progra~1/Specware4.0/Examples/speccode2
#BBBtoALLTRANSLATION
Processing spec at C:/Progra~1/Specware4.0/Examples/speccode2
#RELIABLEBROADCAST
Processing translation at C:/Progra~1/Specware4.0/Examples/speccode2
#RELBROADtoALLTRANSLATION
Processing spec at C:/Progra~1/Specware4.0/Examples/speccode2#CONSENSUS
Processing spec morphism at C:/Progra~1/Specware4.0/Examples/speccode2
#RELBROADtoCONSENSUS
Processing spec diagram at C:/Progra~1/Specware4.0/Examples/speccode2
#CONSEN
Processing spec morphism at C:/Progra~1/Specware4.0/Examples/speccode2
```

```
#CONSEN
Processing colimit at C:/Progra~1/Specware4.0/Examples/speccode2
#CONSENT
Processing spec at C:/Program Files/Specware4.0/Library/Base
Processing spec at C:/Program Files/Specware4.0/Library/Base/Boolean
Processing spec at C:/Program Files/Specware4.0/Library/Base/PrimitiveSorts
Processing spec at C:/Program Files/Specware4.0/Library/Base/Compare
Processing spec at C:/Program Files/Specware4.0/Library/Base/Functions
Processing spec at C:/Program Files/Specware4.0/Library/Base/Integer
Processing spec at C:/Program Files/Specware4.0/Library/Base/Nat
Processing spec at C:/Program Files/Specware4.0/Library/Base/Char
Processing spec at C:/Program Files/Specware4.0/Library/Base/String
Processing spec at C:/Program Files/Specware4.0/Library/Base/List
Processing spec at C:/Program Files/Specware4.0/Library/Base/Option
Processing spec at C:/Program Files/Specware4.0/Library/Base/System
Processing spec at C:/Program Files/Specware4.0/Library/Base/Show
Processing translation at C:/Progra~1/Specware4.0/Examples/speccode2
#CONSENTtoALLTRANSLATION
Processing spec at C:/Progra~1/Specware4.0/Examples/speccode2#SNAPSHOT
Processing spec morphism at C:/Progra~1/Specware4.0/Examples/speccode2
#CONSENTtoSNAPSHOT
Processing spec diagram at C:/Progra~1/Specware4.0/Examples/speccode2
#SNAPS
Processing spec morphism at C:/Progra~1/Specware4.0/Examples/speccode2
#SNAPS
Processing colimit at C:/Progra~1/Specware4.0/Examples/speccode2#SNAP
Processing translation at C:/Progra~1/Specware4.0/Examples/speccode2
#SNAPtoALLTRANSLATION
Processing spec at C:/Progra~1/Specware4.0/Examples/speccode2
#DECISIONMAKING
Processing spec morphism at C:/Progra~1/Specware4.0/Examples/speccode2
#SNAPtoDECISIONMAKING
Processing spec diagram at C:/Progra~1/Specware4.0/Examples/speccode2
#DECMAK
Processing spec morphism at C:/Progra~1/Specware4.0/Examples/speccode2
#DECMAK
Processing colimit at C:/Progra~1/Specware4.0/Examples/speccode2#DECISION

spec
 sort BroadcastBound = Clockvalues
 sort BroadcastDelay = Clockvalues
 sort Channel
sort Clockvalues = Nat
sort Index = Nat
sort LocalClockvals = Clockvalues
sort Messages = {Km:Index, No:Nat, Tm:Clockvalues, p:Processors}
sort Null = Messages
sort ProcDeci = Boolean
sort Procstate = {LC:Clockvalues, n:Nat, p:Processors}
sort Processors
sort ReliableNetwork = Boolean
sort States
sort Statestabstorage = Boolean
op AgreeBroad : Processors * Messages * Clockvalues -> Boolean
op Agreeconsensus : Processors * ProcDeci * Clockvalues -> Boolean
op Broadcast : Processors * Messages * Clockvalues -> Boolean
op Clockbound : Clockvalues * BroadcastDelay * BroadcastBound->Clockvalues
op Clockdelay : Clockvalues * BroadcastDelay -> Clockvalues
op Correct : Processors -> Boolean
op Decision : Processors * ProcDeci * Clockvalues -> Boolean
op Deliver : Processors * Messages * Clockvalues -> Boolean
op InOrder : Messages -> Boolean
op Proposal : Processors * ProcDeci * Clockvalues -> Boolean
op TermBroad : Processors * Messages * Clockvalues -> Boolean
op ValiBroad : Processors * Messages * Clockvalues -> Boolean
op Valiconsensus : Processors * ProcDeci * Clockvalues -> Boolean
op adjacent : ProcDeci * ProcDeci -> Boolean
op inconsistent : ProcDeci * ProcDeci -> Boolean
op next : ProcDeci * ProcDeci -> Boolean
op reception :
   Processors * Messages * Channel * Processors * Clockvalues->Boolean
op record : Processors * States * Messages * Statestabstorage->Boolean
op sending :
   Processors * Messages * Channel * Processors * Clockvalues->Boolean
axiom next is
   fa(commit : Boolean, abort : ProcDeci)
    ~(adjacent(~ commit, commit)) & next(commit, abort)
axiom adjacent is
   fa(commit : ProcDeci, abort : ProcDeci)
    ~(next(commit, abort)) & adjacent(~ commit, commit)
axiom inconsistent is
   fa(commit : ProcDeci, abort : ProcDeci)
    adjacent(commit, commit) & next(commit, abort)
axiom Constateinfo is
```

```
   fa(p : a, q : Processors)
    fa(commit : ProcDeci, abort : ProcDeci, s : States, M : Messages,
       X : Statestabstorage)
     record(q, s, M, X) &
     (~(next(commit, abort)) & adjacent(~ commit, commit))
theorem CSM is
   fa(p : Processors, q : Processors, T : Clockvalues, m : Messages,
      M : Messages, N : Messages, Null : Messages, c : Channel)
    fa(i : BroadcastDelay, j : BroadcastBound, s : States, S : States)
     fa(v : ProcDeci, commit : ProcDeci, abort : ProcDeci,
        X : Statestabstorage)
      if (Deliver(p, m, T) => Deliver(q, m, Clockbound(T, i, j))) &
         (
         ((AgreeBroad(p, m, T) & Decision(p, v, T)) =>
         (AgreeBroad(q, m, Clockbound(T, i, j)) & Decision(q, v, T))) &
         (
         (Agreeconsensus(p, commit, T) &
         (record(q, s, M, X) & record(q, s, Null, X))) or
         (record(q, s, m, X) &
         (record(q, s, N, X) & ~(reception(q, M, c, p, T))))))
          then
         record(q, s, M, X) &
         (~(next(commit, abort)) & adjacent(~ commit, commit))

      else inconsistent(commit, abort)
axiom Globprocstateinfo is
   fa(p : Processors, q : Processors)
    fa(m : Messages, M : Messages, N : Messages, Null : Messages)
     fa(c : Channel, T : Clockvalues, T' : Clockvalues)
      fa(s : States, S : States, commit : ProcDeci)
       fa(X : Statestabstorage)

        (Agreeconsensus(p, commit, T) &
        (sending(p, M, c, q, T) &
        (record(p, s, N, X) & ~(sending(p, m, c, q, T'))))) =>
        (reception(q, M, c, p, T) =>
        if ~(record(q, s, M, X))
            then record(q, s, M, X) & record(q, S, Null, X)
        else
        record(q, S, m, X) &
        (record(q, s, N, X) & ~(reception(q, M, c, p, T))))
axiom record is
```

```
   fa(p : Processors, q : Processors, M : Messages, c : Channel,
      T : Clockvalues)
    fa(s : States, X : Statestabstorage) record(p, s, M, X)
axiom reception is
   fa(p : Processors, q : Processors, M : Messages, c : Channel,
      T : Clockvalues) ~(sending(p,M,c,q,T)) & reception(p,M,c,q,T)
axiom sending is
   fa(p : Processors, q : Processors, M : Messages, c : Channel,
      T : Clockvalues) ~(reception(p,M,c,q,T)) & sending(p,M,c,q,T)
axiom Agreebroad is
   ex(p : Processors)
    fa(m : Messages, T : Clockvalues)
     Deliver(p, m, T) =>
     fa(q : Processors, i : BroadcastDelay, j : BroadcastBound)
      Deliver(q, m, Clockbound(T, i, j))
axiom Valibroad is
   ex(p : Processors, m : Messages, T : Clockvalues)
    (Correct p & Broadcast(p, m, T)) =>
    fa(q : Processors, i : BroadcastDelay, j : BroadcastBound)
     (Correct q & (Deliver(q, m, Clockbound(T, i, j)) & (i < j)))
axiom Termbroad is
   ex(p : Processors, m : Messages, T : Clockvalues)
    (Correct p & Broadcast(p, m, T)) =>
    fa(q : Processors, i : BroadcastDelay)
     (Correct q & Deliver(q, m, Clockdelay(T, i)))
axiom Deliver is
   fa(p : Processors, m : Messages, T : Clockvalues)
    ~(Broadcast(p, m, T)) & Deliver(p, m, T)
axiom Broadcast is
   fa(p : Processors, m : Messages, T : Clockvalues)
    ~(Deliver(p, m, T)) & Broadcast(p, m, T)
axiom Proposal is
   fa(p : Processors, v : ProcDeci, T : Clockvalues)
    ~(Decision(p, v, T)) & Proposal(p, v, T)
axiom Decision is
   fa(p : Processors, v : ProcDeci, T : Clockvalues)
    ~(Proposal(p, v, T)) & Decision(p, v, T)
axiom Valiconsensus is
   fa(p : Processors, q : Processors, T : Clockvalues, i : a,
   j : Clockvalues, m : Messages)
    ex(v : ProcDeci)
     (ValiBroad(p, m, T) & Decision(p, v, T)) => Proposal(q, v, T)
```

```
 axiom Agreeconsensus is
    fa(p : Processors, q : Processors, v : ProcDeci, T : Clockvalues,
    i : a, j : Clockvalues, m : Messages)
     (AgreeBroad(p, m, T) & Decision(p, v, T)) => Decision(q, v, T)
endspec

 Processing translation at C:/Progra~1/Specware4.0/Examples/speccode2
 #DECISIONtoALLTRANSLATION
 Processing prove at C:/Progra~1/Specware4.0/Examples/speccode2#p2
 Processing spec at C:/Program Files/Specware4.0/Library/Base/ProverBase
p2: Theorem CSM in DECISIONMAKING is Proved!
    Snark Log file: C:/Progra~1/Specware4.0/Examples/snark/speccode2/p2.log
```

APPENDIX-III

Specification of the composite protocol PR_4 along with processing steps of Specware and proof results from Snark

```
CL-USER(1): :sw /Progra~1/Specware4.0/Examples/speccode3
Processing spec at C:/Progra~1/Specware4.0/Examples/speccode3#BBB
Processing spec at C:/Program Files/Specware4.0/Library/Base
Processing spec at C:/Program Files/Specware4.0/Library/Base/Boolean
Processing spec at C:/Program Files/Specware4.0/Library/Base/PrimitiveSorts
Processing spec at C:/Program Files/Specware4.0/Library/Base/Compare
Processing spec at C:/Program Files/Specware4.0/Library/Base/Functions
Processing spec at C:/Program Files/Specware4.0/Library/Base/Integer
Processing spec at C:/Program Files/Specware4.0/Library/Base/Nat
Processing spec at C:/Program Files/Specware4.0/Library/Base/Char
Processing spec at C:/Program Files/Specware4.0/Library/Base/String
Processing spec at C:/Program Files/Specware4.0/Library/Base/List
Processing spec at C:/Program Files/Specware4.0/Library/Base/Option
Processing spec at C:/Program Files/Specware4.0/Library/Base/System
Processing spec at C:/Program Files/Specware4.0/Library/Base/Show
Processing translation at C:/Progra~1/Specware4.0/Examples/speccode3
#BBBtoALLTRANSLATION
Processing spec at C:/Progra~1/Specware4.0/Examples/speccode3
#RELIABLEBROADCAST
Processing translation at C:/Progra~1/Specware4.0/Examples/speccode3
#RELBROADtoALLTRANSLATION
Processing spec at C:/Progra~1/Specware4.0/Examples/speccode3#CONSENSUS
Processing spec morphism at C:/Progra~1/Specware4.0/Examples/speccode3
#RELBROADtoCONSENSUS
Processing spec diagram at C:/Progra~1/Specware4.0/Examples/speccode3
#CONSEN
Processing spec morphism at C:/Progra~1/Specware4.0/Examples/speccode3
```

```
#CONSEN
Processing colimit at C:/Progra~1/Specware4.0/Examples/speccode3
#CONSENT
Processing spec at C:/Program Files/Specware4.0/Library/Base
Processing spec at C:/Program Files/Specware4.0/Library/Base/Boolean
Processing spec at C:/Program Files/Specware4.0/Library/Base/PrimitiveSorts
Processing spec at C:/Program Files/Specware4.0/Library/Base/Compare
Processing spec at C:/Program Files/Specware4.0/Library/Base/Functions
Processing spec at C:/Program Files/Specware4.0/Library/Base/Integer
Processing spec at C:/Program Files/Specware4.0/Library/Base/Nat
Processing spec at C:/Program Files/Specware4.0/Library/Base/Char
Processing spec at C:/Program Files/Specware4.0/Library/Base/String
Processing spec at C:/Program Files/Specware4.0/Library/Base/List
Processing spec at C:/Program Files/Specware4.0/Library/Base/Option
Processing spec at C:/Program Files/Specware4.0/Library/Base/System
Processing spec at C:/Program Files/Specware4.0/Library/Base/Show
Processing translation at C:/Progra~1/Specware4.0/Examples/speccode3
#CONSENTtoALLTRANSLATION
Processing spec at C:/Progra~1/Specware4.0/Examples/speccode3#UNDOREDO
Processing spec morphism at C:/Progra~1/Specware4.0/Examples/speccode3
#CONSENTtoUNDOREDO
Processing spec diagram at C:/Progra~1/Specware4.0/Examples/speccode3
#UNRE
Processing spec morphism at C:/Progra~1/Specware4.0/Examples/speccode3
#UNRE
Processing colimit at C:/Progra~1/Specware4.0/Examples/speccode3
#UNREDO
Processing translation at C:/Progra~1/Specware4.0/Examples/speccode3
#UNREDOtoALLTRANSLATION
Processing spec at C:/Progra~1/Specware4.0/Examples/speccode3
#TWOPHASELOCK
Processing spec morphism at C:/Progra~1/Specware4.0/Examples/speccode3
#UNREDOtoTWOPHASELOCK
Processing spec diagram at C:/Progra~1/Specware4.0/Examples/speccode3
#TPLOCK
Processing spec morphism at C:/Progra~1/Specware4.0/Examples/speccode3
#TPLOCK
Processing colimit at C:/Progra~1/Specware4.0/Examples/speccode3#TPL
Processing translation at C:/Progra~1/Specware4.0/Examples/speccode3
#TPLtoALLTRANSLATION
Processing spec at C:/Progra~1/Specware4.0/Examples/speccode3
#CHECKPOINTING
```

```
 Processing spec morphism at C:/Progra~1/Specware4.0/Examples/speccode3
 #TPLtoCHECKPOINTING
 Processing spec diagram at C:/Progra~1/Specware4.0/Examples/speccode3
 #CKPOINTING
 Processing spec morphism at C:/Progra~1/Specware4.0/Examples/speccode3
 #CKPOINTING
 Processing colimit at C:/Progra~1/Specware4.0/Examples/speccode3#CKPT
 Processing translation at C:/Progra~1/Specware4.0/Examples/speccode3
 #CKPTtoALLTRANSLATION
 Processing spec at C:/Progra~1/Specware4.0/Examples/speccode3
 #ROLLBACKRECOVERY
 Processing spec morphism at C:/Progra~1/Specware4.0/Examples/speccode3
 #CKPTtoROLLBACKRECOVERY
 Processing spec diagram at C:/Progra~1/Specware4.0/Examples/speccode3
 #RCOV
 Processing spec morphism at C:/Progra~1/Specware4.0/Examples/speccode3
 #RCOV
 Processing colimit at C:/Progra~1/Specware4.0/Examples/speccode3#RECO

spec
 sort BroadcastBound = Clockvalues
 sort BroadcastDelay = Clockvalues
 sort Clockvalues = Nat
 sort CurrentData
 sort Currentstatevalue = Nat
 sort Index
 sort LocalClockvals = Clockvalues
 sort Messages = {Km:Index, No:Nat, Tm:Clockvalues, p:Processors}
 sort Newstatevalue = Nat
 sort PreviousData
 sort ProcDeci = Boolean
 sort Procstate = {LC:Clockvalues, n:Nat, p:Processors}
 sort Processors
 sort ReliableNetwork = Boolean
 sort Transactionid
 sort Transactions = Boolean
 sort Valstabstorage = Boolean
 op AgreeBroad : Processors * Messages * Clockvalues -> Boolean
 op Agreeconsensus : Processors * ProcDeci * Clockvalues -> Boolean
 op Broadcast : Processors * Messages * Clockvalues -> Boolean
 op C : Processors * Clockvalues -> LocalClockvals
 op Checkpoint : Processors * Clockvalues -> Boolean
op Ckpt : Processors * LocalClockvals -> Boolean
op Clockbound : Clockvalues * BroadcastDelay * BroadcastBound->Clockvalues
op Clockdelay : Clockvalues * BroadcastDelay -> Clockvalues
op Correct : Processors -> Boolean
op CorrecttoFailure : Processors * Clockvalues -> Boolean
op Decision : Processors * ProcDeci * Clockvalues -> Boolean
op Deliver : Processors * Messages * Clockvalues -> Boolean
op InOrder : Messages -> Boolean
op Locking : Transactionid * CurrentData -> Boolean
op Log : Transactions * Valstabstorage * Newstatevalue -> Boolean
op PI : Processors * LocalClockvals -> Boolean
op Pi : Processors * Clockvalues -> Boolean
op Proposal : Processors * ProcDeci * Clockvalues -> Boolean
op Read : Transactions * CurrentData * Valstabstorage -> Boolean
op Readlock : Transactions * CurrentData * Valstabstorage -> Boolean
op Recover : Index * Clockvalues -> Boolean
op Redo :
   Transactions * ProcDeci * Valstabstorage * Newstatevalue->Boolean
op Restore : Index * Clockvalues -> Boolean
op Rollback : Index * Clockvalues -> Boolean
op Store : Processors * LocalClockvals -> Boolean
op Storevalues : Transactions * Valstabstorage * ProcDeci -> Boolean
op TermBroad : Processors * Messages * Clockvalues -> Boolean
op Undo :
   Transactions * ProcDeci * Valstabstorage * Currentstatevalue->Boolean
op Unlock : Transactionid * PreviousData -> Boolean
op ValiBroad : Processors * Messages * Clockvalues -> Boolean
op Valiconsensus : Processors * ProcDeci * Clockvalues -> Boolean
op Write : Transactions * CurrentData * Valstabstorage -> Boolean
op Writelock : Transactions * CurrentData * Valstabstorage -> Boolean
op ckpt : Processors * Clockvalues -> Boolean
op log : Processors * Messages * Clockvalues -> Boolean
op receive : Processors * Messages * Processors * Clockvalues->Boolean
op recover : Index * LocalClockvals -> Boolean
op restore : Index * LocalClockvals -> Boolean
op rollback : Index * LocalClockvals -> Boolean
op send : Processors * Messages * Processors * Clockvalues -> Boolean
op store : Processors * Clockvalues -> Boolean
axiom CorrecttoFailure is
   fa(p : Processors, T : Clockvalues) Correct p & CorrecttoFailure(p,T)
axiom Rollback is
   fa(n : Index, T : Clockvalues) ~(Restore(n, T)) & Rollback(n, T)
```

```
axiom Restore is
   fa(n : Index, T : Clockvalues) ~(Rollback(n, T)) & Restore(n, T)
axiom rollback is
   fa(n : Index, S : LocalClockvals) ~(restore(n, S)) & rollback(n, S)
axiom restore is
   fa(n : Index, S : LocalClockvals) ~(rollback(n, S)) & restore(n, S)
axiom Recover is
   fa(p : Processors, n : Index)
    fa(e : Integer, T : Clockvalues)
     fa(i : BroadcastDelay, j : BroadcastBound, S : LocalClockvals)
      Checkpoint(p, T) &
      ((((S - i) - e) < C(p, T)) &
      ((C(p, T) <= S) &
      (CorrecttoFailure(p, T) &
      (ckpt(p, T) => (Rollback(n, T) => Restore(n, T))))))
axiom recover is
   fa(p : Processors, n : Index)
    fa(e : Integer, T : Clockvalues)
     fa(i : BroadcastDelay, j : BroadcastBound, S : LocalClockvals)
      Checkpoint(p, T) &
      ((((S - i) - e) < C(p, T)) &
      ((C(p, T) <= S) &
      (CorrecttoFailure(p, T) &
      (Ckpt(p, S) => (rollback(n, S) => restore(n, S))))))
theorem RBR is
   fa(p : Processors, q : Processors, T : Clockvalues, m : Messages,
      t : Transactions, n : Index)
    fa(i : BroadcastDelay, j : BroadcastBound, S : LocalClockvals)
     fa(v : ProcDeci, commit : ProcDeci, abort : ProcDeci,
        N : Transactionid, X : Valstabstorage)
      fa(y : Currentstatevalue, z : Newstatevalue, Y : CurrentData,
         Z : PreviousData)
       if (Deliver(p, m, T) => Deliver(q, m, Clockbound(T, i, j))) &
          (
          ((AgreeBroad(p, m, T) & Decision(p, v, T)) =>
          (AgreeBroad(q, m, Clockbound(T,i,j)) & Decision(q,v,T))) &
          (
          (
          (Agreeconsensus(p, commit, T) &
          (Undo(t,abort,X,y) & Redo(t, commit,X,z))) => Log(t,X,z)) &
          (
          (
          (
          (Log(t, X, z) &
          (~(Write(t, Y, X)) & (~(Locking(N, Y)) & Unlock(N, Z)))) =>
          (Read(t, Y, X) & Locking(N, Y))) or
          (
          (Log(t, X, z) &
          (~(Read(t, Y, X)) & (~(Locking(N, Y)) & Unlock(N, Z)))) =>
          (Write(t, Y, X) & Locking(N, Y)))) &
          (
          (~(Readlock(t, Y, X)) &
          (Writelock(t,Y,X) & (ckpt(p,T) & (store(p,T) & Pi(p,T))))) or
          (Ckpt(p, S) & (Store(p, S) & PI(p, S))))))
           then ckpt(p, T) => (Rollback(n, T) => Restore(n, T))
       else Ckpt(p, S) => (rollback(n, S) => restore(n, S))
axiom Checkpoint is
   fa(m : Messages)
    fa(p : Processors)
     fa(n : Index)
      fa(e : Integer, T : Clockvalues, S : LocalClockvals,
      i : BroadcastDelay, j : BroadcastBound)
       fa(t : Transactions, Y : CurrentData, X : Valstabstorage)

        (~(Readlock(t, Y, X)) &
        (Writelock(t, Y, X) & ((((S - i) - e) < C(p, T)) &
(C(p, T) <= S)))) =>

        if ex(m : Messages) log(p, m, T) & (C(p, T) < S)
            then ckpt(p, T) & (store(p, T) & Pi(p, T))
        else Ckpt(p, S) & (Store(p, S) & PI(p, S))
axiom Logging is
   fa(m : Messages)
    fa(p : Processors, q : Processors)
     fa(e : Integer, T : Clockvalues, S:LocalClockvals, i:BroadcastDelay,
        j : BroadcastBound)
      fa(t : Transactions, Y : CurrentData, X : Valstabstorage)

       (Readlock(t, Y, X) &
       (~(Writelock(t, Y, X)) &
       ((((S - i) - e) < C(p, T)) & (C(p, T) <= ((S + j) + e))))) =>
       (receive(p, m, q, T) => log(p, m, T))
axiom PI is
   fa(p : Processors, T : Clockvalues, S : LocalClockvals)
```

```
    ~(Pi(p, T)) & PI(p, S)
axiom Pi is
   fa(p : Processors, T : Clockvalues, S : LocalClockvals)
    ~(PI(p, S)) & Pi(p, T)
axiom store is
   fa(p : Processors, T : Clockvalues, S : LocalClockvals)
    ~(Store(p, S)) & store(p, T)
axiom Store is
   fa(p : Processors, T : Clockvalues, S : LocalClockvals)
    ~(store(p, T)) & Store(p, S)
axiom ckpt is
   fa(p : Processors, T : Clockvalues, S : LocalClockvals)
    ~(Ckpt(p, S)) & ckpt(p, T)
axiom Ckpt is
   fa(p : Processors, T : Clockvalues, S : LocalClockvals)
    ~(ckpt(p, T)) & Ckpt(p, S)
axiom log is
   fa(p : Processors, q : Processors, m : Messages, T : Clockvalues)
    receive(p, m, q, T) & log(p, m, T)
axiom send is
   fa(p : Processors, q : Processors, m : Messages, T : Clockvalues)
    ~(receive(p, m, q, T)) & send(p, m, q, T)
axiom receive is
   fa(p : Processors, q : Processors, m : Messages, T : Clockvalues)
    ~(send(p, m, q, T)) & receive(p, m, q, T)
axiom Storevalues is
   fa(p : Processors, q : Processors)
    fa(T : Clockvalues, t : Transactions)
     fa(commit : ProcDeci, abort : ProcDeci)
      fa(y : Currentstatevalue, z : Newstatevalue, X : Valstabstorage)

       (Agreeconsensus(p, commit, T) &
       (Undo(t, abort, X, y) & Redo(t, commit, X, z))) => Log(t, X, z)
axiom Log is
   fa(t : Transactions, a : ProcDeci, X : Valstabstorage)
    fa(y : Currentstatevalue, z : Newstatevalue)
     (~(Undo(t, a, X, y)) & ~(Redo(t, a, X, y))) => Log(t, X, z)
axiom Redo is
   fa(t : Transactions, a : ProcDeci, X : Valstabstorage,
      y : Currentstatevalue) ~(Undo(t, a, X, y)) & Redo(t, a, X, y)
axiom Undo is
   fa(t : Transactions, a : ProcDeci, X : Valstabstorage,
      y : Currentstatevalue) ~(Redo(t, a, X, y)) & Undo(t, a, X, y)
axiom Agreebroad is
   ex(p : Processors)
    fa(m : Messages, T : Clockvalues)
     Deliver(p, m, T) =>
     fa(q : Processors, i : BroadcastDelay, j : BroadcastBound)
      Deliver(q, m, Clockbound(T, i, j))
axiom Valibroad is
   ex(p : Processors, m : Messages, T : Clockvalues)
    (Correct p & Broadcast(p, m, T)) =>
    fa(q : Processors, i : BroadcastDelay, j : BroadcastBound)
     (Correct q & (Deliver(q, m, Clockbound(T, i, j)) & (i < j)))
axiom Termbroad is
   ex(p : Processors, m : Messages, T : Clockvalues)
    (Correct p & Broadcast(p, m, T)) =>
    fa(q : Processors, i : BroadcastDelay)
     (Correct q & Deliver(q, m, Clockdelay(T, i)))
axiom Deliver is
   fa(p : Processors, m : Messages, T : Clockvalues)
    ~(Broadcast(p, m, T)) & Deliver(p, m, T)
axiom Broadcast is
   fa(p : Processors, m : Messages, T : Clockvalues)
    ~(Deliver(p, m, T)) & Broadcast(p, m, T)
axiom Proposal is
   fa(p : Processors, v : ProcDeci, T : Clockvalues)
    ~(Decision(p, v, T)) & Proposal(p, v, T)
axiom Decision is
   fa(p : Processors, v : ProcDeci, T : Clockvalues)
    ~(Proposal(p, v, T)) & Decision(p, v, T)
axiom Valiconsensus is
   fa(p : Processors, q : Processors, T : Clockvalues, i : a,
   j : Clockvalues, m : Messages)
    ex(v : ProcDeci)
     (ValiBroad(p, m, T) & Decision(p, v, T)) => Proposal(q, v, T)
axiom Agreeconsensus is
   fa(p : Processors, q : Processors, v : ProcDeci, T : Clockvalues,
   i : a, j : Clockvalues, m : Messages)
    (AgreeBroad(p, m, T) & Decision(p, v, T)) => Decision(q, v, T)
axiom Read is
   fa(t : Transactions, Y : CurrentData, X : Valstabstorage)
    ~(Write(t, Y, X)) & Read(t, Y, X)
axiom Write is
```

```
   fa(t : Transactions, Y : CurrentData, X : Valstabstorage)
    ~(Read(t, Y, X)) & Write(t, Y, X)
 axiom Locking is
   fa(N : Transactionid, Y : CurrentData, Z : PreviousData)
    Unlock(N, Z) & Locking(N, Y)
 axiom Unlock is
   fa(N : Transactionid, Y : CurrentData, Z : PreviousData)
    ~(Locking(N, Y)) & Unlock(N, Z)
 axiom Readlock is
   fa(p : a, q : Processors)
    fa(t : Transactions, N : Transactionid, X : Valstabstorage)
     fa(Y : CurrentData, Z : PreviousData, z : Newstatevalue)

      (Log(t, X, z) &
      (~(Write(t, Y, X)) & (~(Locking(N, Y)) & Unlock(N, Z)))) =>
      (Read(t, Y, X) & Locking(N, Y))
 axiom Writelock is
   fa(p : a, q : Processors)
    fa(t : Transactions, N : Transactionid, X : Valstabstorage)
     fa(Y : CurrentData, Z : PreviousData, z : Newstatevalue)

      (Log(t, X, z) &
      (~(Read(t, Y, X)) & (~(Locking(N, Y)) & Unlock(N, Z)))) =>
      (Write(t, Y, X) & Locking(N, Y))
endspec

 Processing translation at C:/Progra~1/Specware4.0/Examples/speccode3
 #RECOtoALLTRANSLATION
 Processing prove at C:/Progra~1/Specware4.0/Examples/speccode3#p3
 Processing spec at C:/Program Files/Specware4.0/Library/Base/ProverBase
p3: Theorem RBR in ROLLBACKRECOVERY is Proved!
   Snark Log file: C:/Progra~1/Specware4.0/Examples/snark/speccode3/p3.log
```

Bibliography

[1] A. Arora, S. Kulkarni, "Component Based Design of Multitolerance," *IEEE Trans. on Soft. Engg.*, 24(1), pp.63–78,1998.

[2] O. Babaoglu, S. Toueg, "Non-blocking Atomic Commitment," *Distributed Systems*, Ed. S. Mullender,Adisson Wesley,1993.

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

[4] P.A. Bernstein, E. Newcomer, *Principles of Transaction Processing*, Morgan Kaufmann Publishers, 1997.

[5] N.T. Bhatti, R.D. Schlichting, "A System for Constructing Configurable High-Level Protocols," *Proceedings of SIGCOMM*, pp.138–150, August 1995.

[6] D. Cerutti, D. Pierson, '*Distributed Computing Environments*, McGraw-Hill Series on Computer Communications, 1993.

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

[8] 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.

[9] M. Doche, C. Seguin, V. Wiels, "A Modular Approach to Specify and Test an Electrical Flight Control System," *Proc. of FMICS (Formal Methods for Industrial Critical Systems) Trento, Italy, 1999.*

[10] N. Durgin, J. Mitchell, D. Pavlovic, "Protocol Composition and Correctness," Kestrel Institute Technical Report KES.U.00.1, January 2000.

[11] H. Ehrig, B. Mahr, *Fundamentals of Algebraic Specification 2, Module Specifications and Constraints*, Springer-Verlag, 1990.

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

[13] T. Fine, "A Framework for Composition," *Proc. of the Eleventh Annual Conference on Computer Assurance, Maryland, US*, pp.199–212, June 1996.

[14] B. Garbinato, R. Guerraoui, "Flexible Protocol Composition in BAST," *Proc. of Int. Conf. on Distributed Computing Systems-18*, pp.22-29, 1998.

[15] R. Guerraoui, A. Schiper, "The Decentralized Non-Blocking Atomic Commitment Protocol," *IEEE Symposium on Parallel and Distributed Systems, San Antonio, TX*, pp.2-9, 1995.

[16] R. Guerraoui, A. Schiper, "Consensus Service: A Modular Approach for Building Agreement Protocols in Distributed Systems," *Proc. of the 26th IEEE Symposium on Fault-Tolerant Computing Systems, Sendai*, pp.168-177, June 1996.

[17] J. Guo, "Using Category Theory to Model Software Component Dependencies," *Proc. of the Ninth Annual IEEE Int. Conference and Workshop on the Engineering of Computer-Based Systems (ECBS'02), Lund, Sweden*, April 2002.

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

[19] J. Hooman, *Specification and Compositional Verification of Real-Time Systems.* LNCS 558, Springer Verlag 1991.

[20] J. Hooman, "Verification of Distributed Real-Time and Fault-Tolerant Protocols," *Proc. of AMAST Conference*, (Springer-Verlag, Sydney, Australia), pp.261-275, 1997.

[21] J. Hooman, D. Chkliaev, P. Van Der Stok, "Mechanical Verification of Transaction Processing Systems," *3rd IEEE International Conference on Formal Engineering Methods*, pp.89-97, 2000.

[22] P. Jalote, *Fault Tolerance in Distributed Systems*, Prentice Hall, 1994.

[23] E. Juan, J.J.P. Tsai, *"Compositional Verification of High Assurance Systems*, Kluwer Academic Publisher, 2000.

[24] I. Keidar, R. Khazan, N. Lynch, A. Shvartsman, "An Inheritance-Based Technique for Building Simulation Proofs Incrementally," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, PP.1-29, January 2002.

[25] R. Li, L.M. Pereira, "Application of Category Theory in Model-Based Diagnostic Reasoning," *Proc. of 8th Florida Artificial Intelligence Research Symposium, Melbourne (FL,US)*, pp.123-127, 1995.

[26] X. Liu, et al., "Building Reliable, High-Performance Communication Systems from Components." *Operating Systems Review*, 34(5), pp.80-92, Dec.1999.

[27] N. Lynch, M. Merritt, W. Weihl, A. Fekete, *Atomic Transactions*, Morgan Kaufmann Publishers, 1994.

[28] J. McDonald, J. Anton, "SPECWARE - Producing Software Correct by Construction," Kestrel Institute Technical Report KES.U.01.3., March 2001.

[29] P. Michel, V. Wiels, "A Framework for Modular Formal Specification and Verification," *Proc. of Formal Methods Eng.*, 1997.

[30] S. Mishra, R.D. Schlichting, "Abstractions for Constructing Dependable Distributed Systems," *Technical Report TR 92-19, CS Department, University of Arizona*, 1992.

[31] S. Mishra, L.L. Peterson, R.D. Schlichting, "Modularity in the Design and Implementation of Consul," *Proceedings of the First IEEE Symposium on Autonomous Decentralized Systems*, pp.376-382, March 1993.

[32] D. Pavlovic, D.R. Smith, "Composition and Refinement of Behavioral Specifications," Kestrel Institute Technical Report KES.U.01.6, July 2001.

[33] M. Raynal, "Fault-Tolerant Distributed Systems: A Modular Approach to the Non-Blocking Atomic Commitment Problem," *INRIA, TR-2973*, Sept.1996.

[34] D.E. Rydeheard, R.M. Burstall, *Computational Category Theory*, Prentice Hall, 1988.

[35] M. Singhal, N.G. Shivratri, *Advance Concepts in Operating Systems*, McGraw-Hill, 1994.

[36] P. Sinha, N. Suri, "On Simplifying Modular Specification and Verification of Distributed Protocols," *Proc. of HASE-6*, pp.173-181, Oct.2001.

[37] 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 Intl. Conf. on Distributed Computing Systems-21*, pp.253-263, 2001.

[38] D. Skeen, "Non Blocking Commit Protocols," *Proc. of the ACM SIGMOD Intl. Conf. on the Management of Data*, pp.133-142, May 1981.

[39] D. Skeen, "A Quorum-Based Commit Protocol," *in Berkeley Workshop on Distributed Data Management and Computer Networks*, pp.69-80, Feb.1982.

[40] M. Stonebraker, D. Skeen, "A Formal Model of Crash Recovery in a Distributed System," *IEEE Transaction on Software Engineering*, May 1983.

[41] Y.V. Srinivas and R. Jullig, "Specware(TM): Formal Support for Composing Software," *Proc. of the Conference on Mathematics of Program Construction*, B. Moeller, Ed. LNCS 947, Springer-Verlag, pp.399-422, 1995.

[42] V. Wiels, S. Easterbrook, "Management of evolving specifications using category theory," *Proc. of Automated Software Engineering Conference, IEEE Press*, pp.12-21, 1998.

[43] P.T. Wojciechowski, S. Mena, A. Schiper, "Semantics of Protocol Modules Composition and Interaction," *Proc. of COORDINATION 2002*, pp.389-404.

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