Modelling, Analysing and Model Checking Commit Protocols

Tim Kempster

Doctor of Philosophy
University of Edinburgh
2000

To My Mother

Abstract

Distributed transactions are playing, and will continue to play, an increasingly important role in all forms of electronic business. A key ingredient of a distributed transaction is a commit protocol. We present a novel modelling technique for commit protocols and the environments in which they execute. We devise a new commit protocol X3PC using this modelling technique. We demonstrate that our modelling technique is flexible and formal enough to support automatic verification of behavioural properties of commit protocols, using tools such as model checking as well as more traditional proof techniques. It is possible to verify many different properties of commit protocols by expressing properties in temporal logics and then performing model checking to verify them. In order to carry out model checking a labeled transition system must first be generated from our models. We will describe different techniques that allow us to automatically generate transition systems.

The role of commit protocols in providing transaction isolation for distributed transactions is studied. We present novel definitions for the four different levels of transaction isolation first proposed by the ANSI community. By first modelling a system of multiple concurrent distributed transactions, using our new technique, we show how to verify that a particular level of isolation is attained within the system. This, once again, demonstrates the applicability and flexibility of our modelling technique.

Acknowledgements

I would like to express my gratitude towards my supervisors Peter Thanisch and Colin Stirling. Their continuous support, friendly encouragement, and general involvement over the years has made studying for this degree a pleasure.

Declaration

I declare that this doctoral thesis was composed by myself and the work contained therein is my own, except where explicitly stated otherwise in the text.

The following articles were published during my course of research.

- T. Kempster, C. Stirling, and P. Thanisch. A more committed quorum-based three phase commit protocol. In *LNCS: The Twelth International Symposium on Distributed Computing*, pages 246–257, 1998.
- T. Kempster, C. Stirling, and P. Thanisch. A critical analysis of the transaction internet protocol. In *Proceedings of the Second International Conference on Telecommunications and Electronic Commerce (ICTEC)*, pages 245–271, 1999. Longer version to Journal of Electronic Commerce Research.
- T. Kempster, C. Stirling, and P. Thanisch. Diluting ACID. *ACM SIGMOD Record*, 28(4), December 1999, pages 88–94.
- T. Kempster, C. Stirling, and P. Thanisch. Games–based model checking of protocols: counting doesn't count. In *Proceedings of the International Workshop on Distributed System Validation and Verification*, 2000, pages 111–117.
- T. Kempster, G. Brebner, and P. Thanisch. A transactional approach to network management. In *Proceedings of the 1999 Workshop on Databases in Telecommunications.* Springer-Verlag, 1999, pages 224–252.

Table of Contents

Chapter 1 Introduction **4**

1.1 Transactions 5

1.2 Message Passing Protocols 6

1.3 Model Checking 8

1.4 Commit Protocols 8

1.5 Thesis Contribution 9

1.6 Thesis Structure 10

Chapter 2 Commit Protocols **13**

2.1 Introduction 13

2.2 Core Atomic Commit Literature 15

2.3 Non-blocking Protocols 21

2.4 Two-Phase Commit Optimisations 23

2.5 Failure models 28

2.5.1 Site Failure 28

2.5.2 Communication Failure 29

2.6 Casting the Net Wider 32

2.7 Conclusions 32

Chapter 3 Modelling Atomic Commit Protocols **33**

3.1 Introduction 33

3.2 I/O Automata 33

3.3 A Knowledge Theoretic Approach 38

3.3.1 Modelling Distributed Systems 38

3.3.2 A Knowledge Logic 40

3.4 The Calculus for Communicating Systems 42

3.5 Comparing Techniques 44

3.6 The Views Model 45

3.6.1 Processes, local state and views 47

3.6.2 Protocol rules 48

3.6.3 Environment rules 48
3.6.4 Global state and executions 49
3.6.5 Modelling centralised two-phase commit 50
3.7 Summary 52

Chapter 4 A More Committed Three Phase Commit Protocol 54
4.1 Introduction 54
4.2 Adding failure to 2PC 56
4.2.1 2PC blocks 57
4.2.2 Help-Me messages 58
4.3 Modelling 3PC 60
4.4 Modelling quorum based commit protocols 60
4.4.1 Views and process state 60
4.4.2 Updating views 63
4.4.3 Protocol rules for E3PC 63
4.4.4 Q3PC: Skeen's Quorum-based 3PC 66
4.4.5 Configurations and executions 66
4.5 E3PC's advantage over Q3PC 66
4.6 Constructing X3PC from E3PC 67
4.7 X3PC Solves Atomic Commitment 69
4.8 Performance Comparison 73
4.9 Conclusions and Future Research Directions 74

Chapter 5 Model Checking Two Phase Commit 78
5.1 Introduction 78
5.2 Modelling Two-Phase Commit 79
5.3 Generating Transition Systems from Rules 80
5.3.1 A Concrete approach 81
5.3.2 Multi-set representation 83
5.4 Expressing Properties of Protocols 87
5.5 Games-Based Model Checking 89
5.6 Applying CTL^- to CON and MULTI 93
5.7 Checking CTL^- Properties of Two Phase Commit 95
5.8 Conclusions 96

Chapter 6 Diluting ACID 97
6.1 Introduction 97
6.2 Classical Recoverability and Serializability Theory 99

6.2.1 Recoverability . . . 99
6.2.2 Serializability . . . 101
6.3 Modelling simple schedules . . . 103
6.4 Extended Conflict Serializability . . . 104
6.5 Redefining Phenomena . . . 108
6.6 Disallowing Phenomena Provides Conflict Serializability . . . 110
6.7 Enriching Schedules with Predicate Accesses . . . 111
6.8 Conclusion . . . 113

Chapter 7 Verifying Isolation Levels in Distributed Transactions 115
7.1 Introduction . . . 115
7.2 Modelling Distributed Schedules . . . 116
7.3 Serializability of Distributed Schedules . . . 118
7.3.1 Local rules for distributed serializability . . . 120
7.4 Modelling Distributed Transactions . . . 122
7.4.1 Protocol Rules . . . 123
7.5 An example execution . . . 125
7.6 Verifying Isolation Levels . . . 127
7.7 Overlapping Prepare . . . 128
7.8 Conclusions . . . 130

Chapter 8 Conclusions and Future Research Directions 132
8.1 Introduction . . . 132
8.2 Commit Literature and Modelling Techniques . . . 132
8.2.1 Future Research Directions . . . 134
8.3 Putting our Model to Work . . . 135
8.3.1 Future Research Directions . . . 135
8.4 Automatic Verification Techniques . . . 136
8.4.1 Future Research Directions . . . 137
8.5 Commit and Isolation . . . 137
8.5.1 Future Research Directions . . . 138
8.6 Concluding Remarks . . . 139

Bibliography 140

Chapter 1

Introduction

Computers and computing systems are becoming connected in ways that will change the way people interact dramatically. Advances in computer networks [79] will provide huge bandwidth communication channels between every corner of the globe, connecting billions of autonomous computing devices into large distributed systems. In the future, for example, embedded processors in electrical appliances might communicate their usage patterns, via the Internet, to electricity providers allowing them to better plan production.

Important building blocks of these distributed computing environments are communication protocols that involve multiple (three or more) computing agents. In these protocols groups of autonomous computing agents exchange information to solve a common task. Examples are leadership election [29], atomic broadcast [49], and commit protocols [9]. In this thesis we will focus on commit protocols. Commit protocols are designed to execute in a wide variety of environments. For example, special variants of commit protocols exist for e-commerce when the agents involved do not trust one another [81].

Almost as soon as a new problem or novel environment arises a commit protocol is devised that is claimed to solve that problem. These solutions are providing the foundations of new and exciting applications in environments such as Intranets, wireless broadcast environments and high speed LANs.

It is important to fully understand a proposed protocol and to be confident that the behaviour of systems of processes executing the protocol is as intended. One way of achieving this is to formally verify properties of these systems. For example, one might want to ensure that a client purchasing a book over the Internet is only charged once the book is guaranteed to be delivered by some third party delivery company, even in the event of message loss.

As in many other areas of computer science the high demand and rewards for developing a rapid solution often leads to *ad hoc* implementations. Although these

solutions often do behave as intended (although many have reached production and later have failed), initially only informal arguments are proposed to support their correctness. In fact, proving the correctness of complex distributed systems is notoriously difficult. In this thesis we present a novel technique for modelling commit protocols. We demonstrate its applicability and show how it can be used to understand the role of a wide range of commit protocols as they interact with their environments and with each other.

1.1 Transactions

In almost all electronic business processes the unit of work can be thought of as a transaction [34]. A transaction is a group of actions that are executed (often by a transaction processing system) in a way that imparts certain guarantees on their behaviour. These guarantees are often referred to as the ACID properties of transactions. 'A'tomicity guarantees that either all the actions of a transaction take place or none take place. Transactions can also guarantee 'C'onsistency properties. For example in an e-commerce transaction electronic money is neither created nor destroyed by e-transactions. Consistency is slightly different from the other ACID properties in that it is the transaction itself not the transaction processing system that provides the property. A system can impart different 'I'solation guarantees on its transactions. This limits the extent to which interference is permitted between concurrently executing transactions. Finally 'D'urability guarantees are often imparted upon a transaction by the system in which it executes to ensure that if a transaction makes a change, then that change will survive a level of system failure [1].

Transactions perform actions on data objects. The data objects represent values of real world entities. For example, the wealth of a person is captured by the data object recording her bank account balance in a database. For the purposes of this thesis, we assume that the actions of a transaction fall into two categories, read and write. Although, traditionally these actions represented the actions carried out on data objects in a relational database, in this thesis, we will take a much broader view [80]. A write action changes the state of a system. This might be for example starting a robot welder in a car factory. Whereas a read access gathers a value from an external environment. For example reading a temperature gauge of a steel furnace.

In many transactions, it is often the case that the data objects of a transac-

[1] Obviously no system can protect against total destruction.

tion are distributed. For example, the transaction that represents the sale of an online book requires actions to be performed for billing, delivery and inventory. These actions are distributed across many different computers. For our purpose a transaction can be distributed even if all the processes in the transaction reside on the same computer system. We define a process as a thread of computing control that is autonomous and has the ability to fail without failing the system as a whole. We therefore include single processor machines running multiple processes or threads as distributed systems.

Although the theory of transactions was founded in early databases systems [28, 9], and extended to distributed systems with the advent of distributed file systems and distributed databases [25], the same theory has found applications in many different new areas. See [80] for a survey.

The birth and exponential growth of the Internet and wireless communication devices has meant that many millions of computing devices from vending machines to personal organisers have, or will soon have, the ability to communicate with one another. It will not be long before large amounts of electronic business is carried out by these connected autonomous processes[2], as they participate in distributed transactions.

There are three important properties that these new distributed computing environments exhibit. Firstly, the numbers of processes involved in a transaction can be very large. Secondly, the processes communicate by asynchronous message passing. In asynchronous message passing systems, the delay a message is subject to is not known by the processes in the system. Lastly, the reliability of the message passing medium that connects the processes, the network, may be very dubious. The processes themselves may also be unreliable. This means many different types of failure are possible in the system.

1.2 Message Passing Protocols

The distributed systems that we are interested in, in this thesis, are often referred to as asynchronous message passing systems [52]. In these systems processes communicate by sending messages to one another. Each process resides on some hardware platform and there is some kind of message passing medium which connects the hardware. In this way the processes exchange messages to achieve a common goal. In asynchronous systems the delay a message undergoes in order to propagate from one process to another is unknown by any process in the system. It

[2] We use the term process to mean an autonomous thread of execution.

is this property that makes the system asynchronous. In synchronous systems this delay is bounded and the bound is known to the processes in the system. Another category, partially synchronous also exists. Since asynchrony of message passing is the weakest assumption it provides the most applicable modelling framework. In some work, for example [27] the the asynchronous nature of message passing is exploited fully. In reality of course there is some bound in which a message will arrive if not lost. In the systems we model in this thesis we take this more practical approach and do not use derive results exploiting total asynchrony. Figure 1.1 sets the scene.

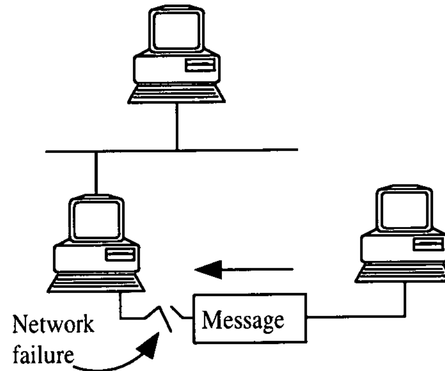

Figure 1.1: A simple message passing system in which three processes exchange messages in order to complete a common task.

We will also be interested in systems where messages are lost and systems where processes fail and then perhaps recover. Different assumptions have been made when modelling message and process failure. We will describe them in more detail in chapter 3. In summary, we are mostly interested in the case that messages are completely lost, sometimes referred to as omission failure, rather than corrupted or interfered with. In the case of process failure we will study the case that a process crashes and then perhaps recovers rather than models that capture process misbehaviour.

The actions that processes are permitted or required to take while exchanging messages and responding to failure constitute a protocol. Each process has some internal state. When it receives a message it performs an action based on this state and the content of the message it received. An action consists of changing state and sending messages. The aggregated behaviour of all the processes as they perform actions is a protocol execution.

1.3 Model Checking

Model checking [19] is an automatic technique for verifying properties of concurrent systems. The method has been very successful in verifying properties of complex sequential circuit designs and more recently it has been useful for verifying properties of communication protocols. Various methods have been employed to model check these systems, but in order to do so, one must first construct a formal model that describes the behaviour of the system. This often takes the form of a labelled transition system. A point in the labelled transition system describes the global state of the distributed system. A transition from one point to another represents an action occurring within the system, for example a message arriving at a process. The transitions are often labelled with the name of the action.

The need for model checking is clear. As distributed systems become more and more integrated in applications such as telephone switching networks, air traffic control systems etc. we become more and more reliant on them. In order to gain confidence in their correctness it is important to fully understand their behaviour. By providing formal modelling techniques, and methods whereby properties of these models can be formally verified, we make progress towards being confident that the systems we model behave as intended.

1.4 Commit Protocols

Processes carrying out distributed transactions make use of commit protocols to provide the atomicity and isolation guarantees of transactions. A commit protocol is executed by processes in a system in order to provide these guarantees. Commit protocols have been studied and developed for many years by academics and practitioners alike [50, 9, 68].

In order to study the behaviour of a commit protocol, the protocol and the environment in which it executes are first modelled. The behaviour of these models can be studied and, if the model is a true reflection of a potential implementation, the model will reflect the behaviour of the commit protocol when implemented. It is the modelling and verification of the behavioural properties of commit protocols that is central to this thesis.

The changing and diverse environments in which distributed transactions now execute give rise to a diverse set of modelling assumptions. In fact it is often more important to accurately describe the way a commit protocol interacts with its environment (for example how it reacts to failure within the system) than the

way the processes exchange messages while carrying out the protocol.

In this thesis we will provide a modelling technique that can be used to capture the behaviour of many different commit protocols and the environments in which they execute. We will demonstrate that this modelling technique is formal enough to support rigorous arguments about the behavioural properties of commit protocols. We will show it is flexible enough to model a wide variety of commit protocols and distributed computing environments. It is scalable in the sense that the arguments constructed using the model can be applied to arbitrary numbers of processes. This is important since the number of processes involved in transactions can be very large. We will also demonstrate that our modelling technique is highly amenable to automatic verification using model checking.

1.5 Thesis Contribution

In this thesis we derive a novel modelling technique which is particularly appropriate for modelling commit protocols and the environments in which they execute. The main novelty of this technique is its use of *views* to abstract details of message passing and communication failure. Our model uses rules to describe the actions processes take in a protocol. Each rule has a pre-condition and a post-action. If the pre-condition is satisfied at a process then the post-action may happen. Rather than modelling message passing explicitly we prefer to maintain, within the local state of a process, that process's view of other processes' local state. This approach affords an intuitive and highly scalable and flexible framework in which to describe commit protocols.

Using our technique we demonstrate its effectiveness by formally modelling many existing commit protocols and environments. Our models of these protocols are "modular" in the sense that a more complex protocol can be derived from a simple protocol by extending the model of the simple protocol. Using our technique we show first how to model simple protocols and environments and then show, by modifying and extending the rules of our models, how more complex protocols can be derived. We formally verify properties of commit protocols using these models. For example we show how they provide transaction atomicity.

A new protocol, X3PC is presented. This was devised by first modelling an existing protocol using our modelling technique and then extending this model in a very natural way. In this new model we derive interesting properties and formally verify them.

We show how to generate a labelled transition system from our views based

model that accurately captures all the possible execution behaviours of the protocol being modelled. In order to formally verify properties of commit protocols we must formally specify properties. We do this by describing properties using temporal logic in our case CTL [16]. Once a property has been described using CTL we show how it can be automatically verified by model checking the transition systems we previously generated.

Transactions can attain different levels of isolation depending on the type of concurrency control methods they use to restrict how one transaction might interfere with another. It is interesting to study how commit protocols are used to provide transaction isolation. In order to verify that a commit protocol provides a particular level of isolation we must first carefully define isolation levels. Recently, the ANSI community provided a specification [3] of different levels of isolation. This specification was criticised by Berenson *et al.* in [6] who gave a more complete and rigorous definition. Unfortunately, this criticism had some shortcomings. By building on the work of Berenson *et al.* we formally define the four isolation levels first proposed by the ANSI community in a way that allows more transaction concurrency than previous definitions.

By extending our definitions of isolation to distributed transactions we examine the role commit protocols play in providing transaction isolation. To do this we use our views based model to describe the behaviour of a very simple transaction processing environment. In this model multiple transactions perform read and write accesses on multiple data objects before carrying out a commit protocol. A subtle change to the model describes two different ways in which the commit protocol can be invoked to provide two different levels of isolation. We formally prove the level of isolation attained for each type of invocation.

1.6 Thesis Structure

In the next chapter we provide a survey of commit protocols and the environments in which they execute. This survey first looks at core literature on the subject. The topic of blocking, an undesirable feature of many commit protocols, is surveyed here. This leads to a discussion of protocols that provide resilience to blocking (defined and discussed later), so called non-blocking protocols. Commit protocols have been highly optimised over the years and we survey some of these optimisations. As previously mentioned we are just as interested in modelling the environments in which protocols execute as the protocols themselves. In the next chapter we survey many different types of failure models that have been used

when describing commit protocol behaviour. Finally in this chapter we discuss some of the newer application areas of commit protocols.

In chapter 3 we discuss existing modelling techniques that have been used to model the behaviour of commit protocols. We discuss three main general techniques that have been used; I/O automata, a knowledge theoretic approach and the calculus for communicating systems. We compare these techniques before introducing our views based model. We round off chapter 3 by using this views based model to model a very simple commit protocol, two-phase commit.

In chapter 4 we extend the simple two-phase commit model by enriching the environment in which it executes to include communication failure. We study some properties of the behaviour of the protocol in this enriched model and show how it can be made more resilient to failure. A model of three-phase commit is derived from two-phase commit that is used to introduce the main topic of this chapter, quorum based three phase commit protocols. Using our modelling technique we describe progressively more complex protocols leading to the development of a new protocol X3PC.

In chapter 5 we study the simple two-phase commit model introduced in chapter 3. We describe two ways to automatically generate a labelled transition system from views based models. These transition systems capture all possible behaviour of protocol executions. We define properties of protocols using CTL [16], a temporal logic, and verify these properties using the games based model checking technique proposed in [76]. This technique employs a game between two players, the verifier who tries to show that the CTL formula is true and the refuter who attempts to disprove it. Players take turns by making moves on the transition system or by performing operations on the formula to be proved. This approach is very intuitive. The runtime of this approach, as for most model checking algorithms, depends on the size of the transition system generated. It soon becomes apparent that the transition systems even for very simple commit protocols can be very large. This chapter examines a technique to reduce the size of the transition system while still capturing protocol behaviour.

In Chapter 6 we consider the isolation properties of transactions. In this chapter we produce novel definitions of isolation that formally describe the levels proposed by the ANSI community. The motivation for our definitions comes from the desire to study the role commit protocols play in providing transaction isolation.

In chapter 7 we generalise the definitions of the previous chapter to encompass distributed transactions. We then set about using our modelling technique

to model a very simple distributed transactions system. Using this model we formally show the role a commit protocol plays in providing different levels of transaction isolation. Chapter 8 provides a conclusion to the thesis and discusses possible future research directions.

Chapter 2

Commit Protocols

2.1 Introduction

Atomic commit protocols have been widely studied, both by academics and practitioners. Although the problem of atomic commit is fairly well defined, literature relating to the subject is diverse. This diversity, is in the main part, due to four interacting factors.

Firstly, modelling assumptions about the environment in which proposed atomic commit solutions execute vary. Different modelling parameters, for example asynchronous versus synchronous message passing assumptions, give rise to different distributed system models and thus different commit protocol solutions. Perhaps the most important modelling assumptions relate to the extent to which communication and site failure is allowed in the model. For example, the correctness of some proposed atomic commit protocols depends on the assumption that once a message is sent it is guaranteed to be delivered [4].

Secondly, different protocol solutions to the atomic commit problem address different design concerns or optimise the "cost" of execution of a protocol in different ways. Examples include protocols that minimise logging to non-volatile storage or reduce a protocol's vulnerability to certain types of failure. Although we are particularly interested in behavioural properties, rather than performance issues, many of the optimisations surveyed in this chapter were derived from observations about protocol behaviour.

Thirdly, the generality of the atomic commit problem has meant that, over the years, it has found applications in many different areas. Each area has posed different research questions which have been answered in different ways. These vary from the role of atomic commit in transaction processing monitors to interoperability issues and commit style transaction processing in e-commerce.

Lastly, literature tends to be divided between work that presents new atomic

commit protocols and work that produces negative results (e.g. non-existence proofs). In the latter case authors need to reason about "the set of all commit protocols". Their task is usually made simpler if they can define what constitutes a commit protocol in such a way as to exclude obviously useless protocols (e.g. a protocol that always aborts). This change of focus leads to different modelling assumptions. For instance, in the former case, a new protocol might be presented and then shown to be behaviourally correct assuming a known bound on message delay. By contrast, when a non-existence proof is derived, no such restriction needs to be placed on the model (in fact often the proof depends on restrictions like bounds on message delay not being in place). This unfortunately leads to sets of results that at first sight seem contradictory. However upon closer inspection of the models presented we see that the correctness of a protocol that seems to provide a counter example to a non-existence proof relies on assumptions that are not found in the environment of the model where the non-existence proof was derived. For example, it is well known that if sites can crash no two-phase commit protocol can be non-blocking [72][1]. This seems to contradict a non-blocking two-phase commit protocol presented in [4]. In reality, no such contradiction exists because in the latter text a communication service is assumed whereby a broadcast message is guaranteed to be delivered by all or no recipients. A method for implementing such a service is given, but in the worst case it makes the protocol non two-phased.

It is perhaps then not surprising that within this diverse field of literature there are few generally applicable modelling approaches in which the behavioural properties of commit protocols and their environments can be analysed. Furthermore, well founded precise models of atomic commit protocols and the environments in which they execute seem lacking for systems with more than a very low level of complexity. Descriptions of more complex commit protocols tend to be less formal and therefore only informal arguments are given to justify their correctness.

In this chapter we first focus on some fundamental work on the atomic commit problem. We then broaden our discussion in order to survey the more general field. Throughout the survey, the emphasis will be on the important behavioural aspects of commit protocols and the environments in which they execute. This will motivate the next chapter where a modelling technique for commit protocols, used throughout the dissertation, will be presented. A large part of the dissertation is concerned with modelling and model checking. This chapter does not contain a literature survey of general modelling techniques. This can be found at

[1]We will discuss blocking in much greater detail later in section 2.3.

the start of the next chapter before our model is presented. The focus of this thesis is on modelling and verification techniques of commit protocols and their role within transaction processing. For this reason an in depth discussion of practical transaction processing systems is inappropriate.

2.2 Core Atomic Commit Literature

Atomic commit protocols are best known in the field of distributed transaction processing (TP) [34]. Bernstein *et al.* [9] provide an excellent account of the role of commit protocols in this area. They provide a definition of the problem of atomic commit which has been adopted extensively. Their distributed system consists of a set of sites which communicate by sending messages to one another. At the outset the sites are required to vote either **yes** or **no** in order to start the process whereby they might terminate, that is, reach a "commit" or "abort" decision. A definition of the atomic commit problem is given by means of a set of axioms found in figure 2.1. A protocol that satisfies these criteria is said to solve the atomic commit problem. Although these axioms have often been referred to as a problem definition they do not provide a complete definition. For example, although they make reference to failure they say nothing about the nature of failure itself.

- **AC1** No two sites that decide, do so differently.
- **AC2** A site cannot reverse its decision once it has reached one.
- **AC3** If any site decides commit then all sites voted **yes**.
- **AC4** If all participants vote **yes** and no failures occur, then the decision will be commit.
- **AC5** At any point in the execution of the protocol, if all existing failures are repaired and no new failures occur for sufficiently long then all sites will reach a decision.

Figure 2.1: Axioms that capture the correctness of a commit protocol

The autonomous sites referred to model data managers (DMs) involved in a particular distributed transaction. Each site will vote **yes** if it is able to commit its part of a transaction or vote **no** if for some reason (for instance an integrity constraint is violated) it cannot. An ACP is deemed correct if it fulfills axioms **AC1-5**. Arguably these definitions seem to capture several different aspects of

correctness. **AC1** is also sometimes referred to as atomicity. **AC2** is often implicitly assumed. **AC3** is an important axiom of commit, without which commit reduces to the problem of consensus [27]. **AC4** and **AC5** are slightly different in that they put performance rather than behavioural restrictions on what constitutes a commit protocol. Another important difference is that their reference to failures suggests that both the protocol and the environment in which it executes are important when one wishes to determine correctness. We will return to this point later when we see that in order to give a complete account of commit protocol behaviour we often must include the behaviour of the environment in which it executes.

The above definition of atomic commit seems to capture the problem at hand although perhaps it is a little informal. It still leaves open questions like "What is the meaning of *sufficiently long* in **AC5**?", "What constitutes a failure (real or just suspected) and how it is detected in **AC4**?". Also under this definition if a permanent failure is experienced then the participating sites need not reach a decision. We will see that it is possible for some commit protocols to allow non-failed sites to reach a decision in many situations where only partial failure occurs.

As well as offering a definition of the atomic commit problem Bernstein *et al.* discuss solutions. In the *centralised two-phase commit* (2PC) protocol [32], a coordinator collects votes on whether or not participants can commit a transaction and broadcasts whether or not there is unanimity for commit.

Problems arise with 2PC when site and/or network failures occur. Some working sites may become *blocked*: they want to commit the transaction, but they are unable to proceed until one or more failures at other sites have been repaired. The blocked site must hold locks on resources on behalf of the stalled transaction, preventing other transactions from proceeding. In figure 2.2 a coordinator and two participants attempt to carry out centralised 2PC. The coordinator sends a `prepare` message to each participant who both respond with `yes` votes. Suppose the coordinator receives these votes and then crashes. This leaves both participants blocked[2]. They cannot commit or abort without first contacting other sites. After voting `yes` they are said to be in their *uncertainty phase.*

Despite its potential for blocking, the family of 2PC protocols, which include centralised 2PC, *decentralised* 2PC [70], *linear* or *nested* 2PC [32] and so on, form cornerstone solutions for the atomic commit problem. Centralised 2PC has been

[2]Of course the participants might be able to contact one another and resolve the outcome of the transaction, but in our simple example we insist that they rely on the coordinator to determine an outcome.

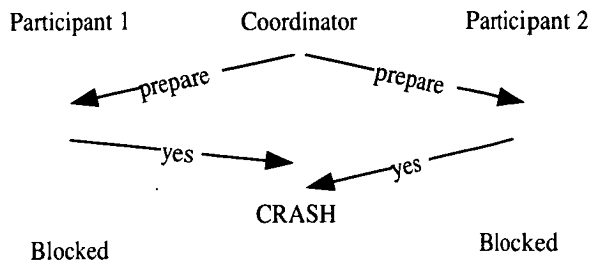

Figure 2.2: Particpants in two-phase commit become blocked after voting yes due to a failure at their coordinator.

the subject of a great deal of research and for reasons discussed later it has been adopted widely as a commercial solution.

Skeen and Stonebraker [69, 72] provide one of the first formal models of commit protocols. Within this model several results are derived. In their model finite state automata (FSA) are used to describe the behaviour of sites participating in a commit protocol. Sites communicate by passing messages and take atomic steps from one state to another based on incoming messages and their current state. A step may cause messages to be sent.

In this model messages are assumed to propagate within a known time, from one operational site to another. Because the message propagation time is bounded, site failure or message loss can be reliably detected using timeouts (if a site does not receive a message it is expecting, it can assume that the sender has crashed or the message is lost). Sites incorporate timeout transitions into their finite state control to model the behaviour of "timing out" when waiting for a message.

Skeen introduces the concept of a *termination protocol* which is executed by operational sites when a failure is detected in order to terminate a commit protocol (i.e. reach a commit or abort decision). He also introduces the concept of a *recovery protocol*. After a site fails it might recover at which point it then executes a recovery protocol. A protocol is said to have the *independent recovery* property if a recovering site can reach a commit or abort decision without requiring further communication. Clearly, independent recovery is potentially a desirable property of a protocol since it allows sites to terminate transactions autonomously. If a protocol does not have this property, upon recovery, a site will have to contact other sites to reach a decision. If, due to a failure, communication with other sites is not possible the site cannot proceed and is thus blocked. If however independent recovery is paid for (e.g. with extra messages) during normal failure free execution (as it is in the three-phase commit protocol discussed later), this

overhead may be unacceptable if failures are uncommon.

To reason about protocol executions, Skeen describes the global state of the distributed system he models as a vector of local site states together with a message buffer containing any undelivered messages still propagating through the network. A protocol step is then a transition in the global state brought about by one of the sites taking a step. A step may also change the outstanding messages in the message buffer, modelling the sending or delivery of messages.

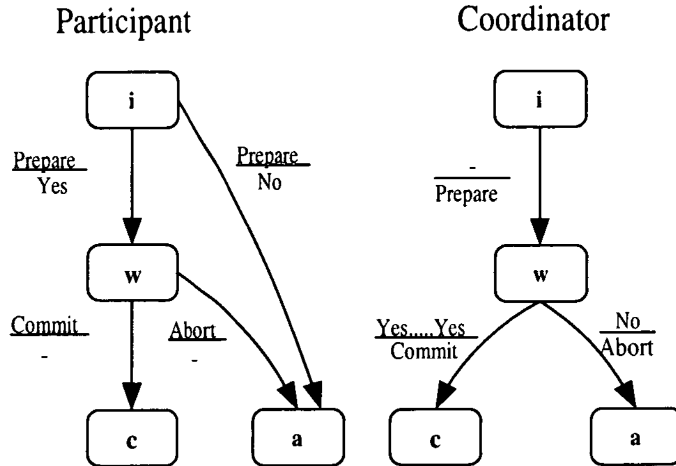

Figure 2.3: Skeen's FSA for 2PC. The local states are initial (**i**), wait (**w**), commit (**c**) and abort (**a**). Transitions are labeled $\frac{\text{rcv msg}}{\text{snd msg}}$ to represent the sending and receipt of messages.

Figure 2.3 shows the FSA for 2PC. In Skeen's model the labels on the transitions represent a request/response dialogue between the coordinator and its participants. For example if a participant receives a `prepare` message in state **i** it might then reply with a `no` vote and move to the **a** state.

Skeen developed the idea of concurrency sets. Two local states x and y are said to be potentially concurrent if there exists a reachable global state where two participants in the global state are in x and y respectively. Thus for at least one possible execution of the protocol, state x is occupied by one site at the same time that y is occupied by another site. In figure 2.3 the concurrency set for **w** is $\{\mathbf{w}, \mathbf{c}, \mathbf{a}\}$. Thus, for example, in some execution one site might be in state **w** at the same point at which a second site is in either **w**, **c** or **a**.

Skeen shows that if a protocol has a local state with both **c** and **a** in its concurrency set then in some execution, if failure occurs at a site, then that site cannot independently recover without potentially violating atomicity. This is because the recovering site can't safely move to **a** since some other site might be in **c** and similarly it can't move to **c** since another state might be in **a**. Since the

concurrency set of the local state **w** contains both **c** and **a** we can conclude that a process may not be able to recover independently if it were to fail, in this state.

By adding a buffer state called pre-commit (**pc**), to the 2PC FSA the three-phase commit (3PC) protocol is derived, see figure 2.4. 3PC does have the independent recovery property in the face of single site failure. In a committing run of 3PC after voting a site receives a `pre-commit` message, moves to the **pc** state and sends an acknowledgment to this message to its coordinator. Once the coordinator has collected acknowledgements from all sites it broadcasts the commit outcome. 3PC is non-blocking for single site failure[3], but it is not resilient to blocking if message loss is possible. In this restricted model where only single site failure is possible, once a failure has been detected a site can always make a safe transition to a final state, without having to contact other sites. This has the advantage that, provided only single site failures are possible, a recovery protocol exists such that any recovering site can terminate a transaction and any locks on resources held on behalf of that transaction can be released without having to wait for a failure to be repaired. For example, let participant p be in its **pc** state. If p times out, waiting for a message from its coordinator, it can safely move to abort. This is because the timeout signifies that the coordinator has crashed and that all the other participants, which are either in wait or pre-commit will also timeout and move to abort. Figure 2.4 shows all the timeout and failure transitions. If we allow double failure then we see this scheme has a major shortcoming. Suppose one participant is in state **pc** and its coordinator is also in state **pc**. Now suppose both sites crash. The crashes happen so close together that the participant (who is waiting for a commit message) does not get a chance to timeout. By studying figure 2.4 we can see that upon recovery the participant will move to state **c**, the failure transition. The coordinator upon recovery will move to state **a**, the failure transition, and so an inconsistent global state will be reached violating atomicity.

Although 3PC is non-blocking for single site failure, this extra resilience is of little practical use because in practice the type of failures that might occur (e.g. double failure) cannot be restricted. Furthermore, this extra resilience comes at the cost of an extra round of messages, even during failure-free execution of the protocol. These two facts have meant that 3PC has not been adopted in commercial systems.

[3]Single site failure means that when a site crashes it is guaranteed to have detected the possible crash failure of all other sites. Double failure occurs when one site fails before it detects the failure of another failed site. 3PC does not provide resilience to blocking in the latter case of double failure.

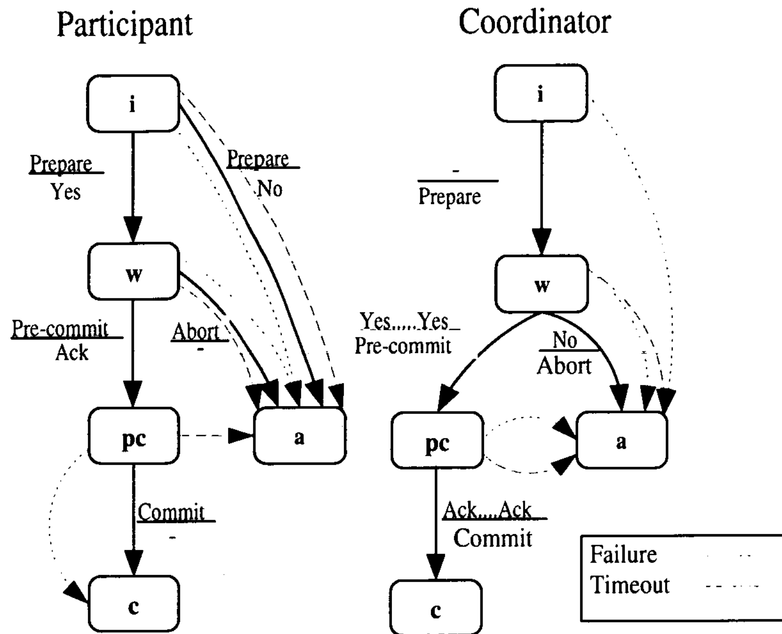

Figure 2.4: Skeen's FSA for 3PC. The local states are **i** (initial), **w** (wait), **pc** (pre-commit), **c** commit and **a** abort.

Skeen also models a type of communication failure called network partitioning. In this model sites can become partitioned into groups. When this happens messages in transit between sites are lost, and subsequently communication is only possible between sites in each component of the partition. It is shown that, in this richer model, no commit protocol can be non-blocking. We discuss an improvement to 3PC later which circumvents blocking for certain types of partitioning.

The formal model proposed by Skeen provides a structure in which to derive some fundamental results. The main focus of this work provides sufficient conditions which allow non-blocking solutions to the atomic commit problem. Perhaps quite pessimistically, it is shown that the simple 2PC protocol does not have the independent recovery property, thus upon recovery a site typically has to contact other sites in order to resolve the outcome of a transaction. By adding a buffer state the (3PC) protocol can protect against blocking in the presence of single site failure. If message loss, site partitioning or double failure are possible, Skeen shows no recovery protocol allows sites to independently recover, and so they might block.

2.3 Non-blocking Protocols

The fundamental observation [72] that no protocol can solve the non-blocking atomic commit problem in the presence of message loss motivated researchers to devise protocols that partially solve the non-blocking problem. The non-blocking result states that in the presence of message loss or network partitioning it is not possible to devise a protocol that is non-blocking for *all* sites. This does however admit the possibility of protocols that allow some of the sites to remain unblocked for certain types of failure.

In [71] Skeen extends his 3PC protocol to provide a protocol, called Q3PC, that is non-blocking for certain classes of communication failure. In this model communication failure isolates sites by partitioning them into connected groups or components. When a failure occurs the sites in each component execute a termination protocol. An election [29] takes place to elect a new coordinator within the group. This coordinator collects the states of the sites within that group. If any site has committed or aborted the decision is passed on to the non-decided sites in the component by the coordinator.

Some of the commit protocols we study in this thesis make use of quorums, which have the desirable property that a common site exists in any two quorate subsets of a set of sites. A quorum is a predicate Q over subsets of P which has the following property.

$$\forall X, Y \subseteq P, \text{ if } Q(X) \wedge Q(Y) \text{ then } X \cap Y \neq \emptyset$$

The simplest quorum scheme is "$Q(X)$ if X is a *majority* of sites" [63]. We assume a majority quorum scheme is used in our examples, although any quorum scheme with the above property is sufficient.

In a quorum based 3PC, if a quorum of sites exists in a component such that at least one of these sites is in state **pc** and the others have voted `yes` or are themselves in **pc** the coordinator of the new component can send a `pre-commit` message and after receiving a quorum of acknowledgements sends a `commit` message. Note, no other site outside of the connected component can abort or could have aborted.

If the coordinator determines that a quorum of sites exists within the component that have either voted `yes`, (if it determines that a site has voted `no` and aborted then it can immediately abort itself and inform the other sites to abort) or are in the pre-abort (**pa**) state[4], the coordinator sends a `pre-abort`

[4]The **pa** state is introduced in the quorum based 3PC. It is a symmetrical buffer state to **pc**.

message to the sites. On receipt of the `pre-abort` message the sites enter their **pa** state and acknowledge the message. Once a quorum of acknowledgements to the `pre-abort` message have been received an `abort` message is sent to the sites. If any quorate partition persists for sufficiently long, the termination protocol described will allow all sites in that quorum to terminate and thus these sites will not block.

Kiedar and Dolev [41] show that it is possible for Q3PC to block even when a quorum of sites form. In order to exhibit this pathological behaviour a communication failure must happen during the termination protocol of Q3PC. They call this repeated failure cascading failure. In this type of failure there are several successive partial network failures, and possibly some repairs too, but the network is not totally failure free at any time during the failure period. There may be times of calm where some progress is made but more disruption soon follows.

If a network undergoes cascading failure after which sites again become connected in a quorum, then the termination protocol of Q3PC can be insufficient to prevent these sites from becoming blocked. Kiedar and Dolev introduce two counters, namely last elected (le) and last attempt (la). The counters are updated to ensure that if a network event disrupts a quorum of sites A and later a quorum of sites B forms then the sites in A will have a strictly smaller la counter than the those in B. In the new protocol proposed, called E3PC, if two sites are in the **pa** or **pc** state, by using the la counter, E3PC can determine which site moved to that state most recently. This extra knowledge can be used to ensure a connected quorum does not block even in the event of cascading failure. In Kempster *et al.* [43] we model the Q3PC and E3PC protocols in order to provide a more precise account of their behaviour. Although E3PC does solve the problem of blocking under cascaded failure it will tend to terminate transactions by aborting them in many cases where it could have committed. By adding extra rules to our model we enhance the termination protocol of E3PC (to derive X3PC) so that a commit outcome is reached in many cases where an abort is reached in E3PC. We will return to this topic in greater deatail in Chapter 4.

Cheung and Kameda [12] analyse the set of different possible termination protocol executions of 3PC. They assign a probability to each possible different partitioning of sites that might result from a network failure. For example, if three sites are executing 3PC and failure takes place when one is in **c** and the other two are in **pc** respectively, a possible partition is that two sites are grouped in states **c** and **pc**, and another is separated in state **pc**.

Given a probability distribution over the set of possible partitions they define

the efficiency of a termination protocol as the expected number of sites which are not terminated by that protocol. Termination protocols are grouped into classes. A termination protocol is said to be *site optimal* within a class if it has the minimum expected number of blocked sites among all termination protocols of that class. Two classes of termination protocols are defined and site optimal termination protocols are derived within each class.

2.4 Two-Phase Commit Optimisations

So far our review has focused on literature that analysed the problem of commit in a rather isolated way. This analysis tended to focus on the subject of blocking. In many commercial environments where sites are highly reliable, and message loss only happens with a very low probability, and high transaction throughput is required, a different focus is then more appropriate.

It was shown in [74] that the commit part of transaction processing typically represents about one third of the total transaction duration. In a tightly coupled or centralised system the cost of logging dominates commit time, whereas in a geographically distributed system, the message latency becomes a more significant factor. This suggests that the commit time for such distributed systems will represent an even higher proportion of the total transaction time. This fact has motivated researchers to develop optimised versions of commit protocols. A faster commit protocol increases transaction throughput by not only reducing the time each transaction takes to execute but also reducing the time transactions hold locks, thereby increasing transaction concurrency in the system.

The simple models presented earlier do not take account of, amongst other things, the read and write operations transactions perform before entering their commit phase, how a group of concurrently executing transactions interact, and how commit protocols interact with transaction logs. In order to analyse the behaviour of these optimised commit protocols therefore, researchers developed richer models that included these features.

An excellent survey of a host of commercial two-phase commit optimisations was given by Chrysanthis *et al.* [14]. All the optimisations work in one of two ways. Either the number of messages (or message rounds) is reduced so that the overall commit process is faster or the number of forced log writes is reduced. A forced log write requires the commit process to wait until the write has been flushed to stable storage. This represents a significant overhead since writing to non-volatile storage is usually several orders of magnitude slower than writes

to volatile storage. Currently, minimum volatile memory cycle times are as low as 50 nanoseconds (DRAM) compared with minimum disk access of about 10 milliseconds.

To enable centralised 2PC to recover from site crashes it is necessary for both coordinator and participant sites to record, usually in the transaction log, their current state in the commit protocol as they proceed. If a site crashes, upon recovery it can then determine the correct course to take in order to terminate the transaction. In the most extreme case a site could force write a log record, whenever it changed state. Upon recovery it would be guaranteed to be in the state it was in before the crash and so could resume execution exactly at the point it crashed. Writing log records to stable storage before every state change is very expensive. For this reason researchers sought to minimise the number of log writes required.

In the *presumed abort* (PA) [57] 2PC variant, if a transaction coordinator or participant crashes, upon recovery, the absence of a log record for a transaction implies that a transaction should be aborted. If no trace of the transaction remains then no action need be taken, but in some cases partial effects of the transaction may need to be undone. Given this presumption sites must force write records when entering states where, upon recovery, the abort presumption might lead to incorrect behaviour (i.e. a violation of transaction atomicity **AC1**). Conversely, sites need not perform any writes to the log when the abort presumption leads to correct behaviour. For instance, when a coordinator decides commit it must force write this to its log. If it did not, and then subsequently crashed, upon recovery it would find no record of the transaction and if interrogated by a recovering participant as to the transaction outcome it would reply `abort`. The coordinator might have already sent `commit` messages to other participants who could have committed and thus this would violate atomicity.

The *presumed commit* (PC) 2PC variant [57] is similar to PA but the appropriate logging is designed to safeguard against violation of atomicity in the event of an erroneous commit presumption. For example, before sending a prepare message to participants a coordinator must force write a log record to logically erase the implicit commit presumption. Suppose prepare messages were sent before writing this record. One participant could receive the prepare message and reply `yes` and enter prepare, while another could reply `no` and enter its abort state. Meanwhile if the coordinator was to crash and recover it would interpret the lack of log entries for the transaction as commit. If the prepared participant times out waiting for a response from the coordinator and makes a further request it

will receive a `commit` message and move to c. Clearly, this violates atomicity. Figure 2.5 describes the logging activity required by a committing or aborting transaction using either PC or PA.

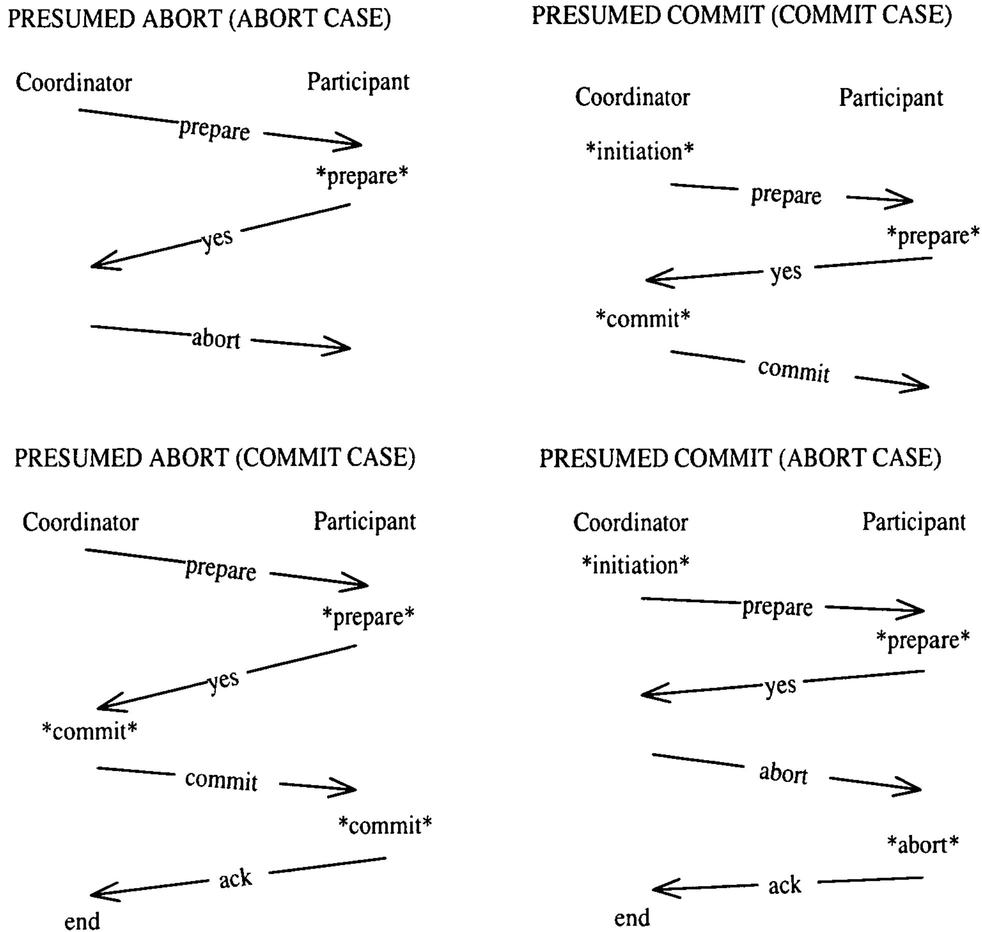

Figure 2.5: Logging required for PC and PA. We denote a forced log write, `*prepare*` and a non-forced write **end**.

At first sight it might be tempting to think that the PA protocol requires more messages and log writes than the PC protocol since most transactions commit and its presumption is to abort. Indeed for a committing update transaction PC requires one less forced log write at each participant and one less message to be sent from each participant. There are however other factors to consider.

Since many transactions involve sites that participate only as read-only sites [68], and furthermore many transactions are completely read-only at all sites, these sites need not be involved in the last round of the two-phase commit. The *read-only* optimisation allows sites to vote `read-only`, and then behave as if they voted

no and release their shared locks[5]. Coordinators interpret a `read-only` vote as a `yes` vote, but need not inform sites that voted `read-only` of the final decision.

Although the read-only optimisation can be used with PC, shared locks cannot be released as soon as they are if it is combined with PA. If a transaction is read-only at all sites, and presumed-abort with the read-only optimisation is used, no forced logging and only one message per participant is required. PC, on the other hand, requires the coordinator to force write an initiation record, to prevent erroneous commit presumptions, before asking sites to vote regardless of whether or not the transaction is read-only. Of course if the transaction coordinator knows before starting the transaction that all the actions of the transaction will be read-only then it need not force write a record.

If carefully analysed, for transaction mixes containing read-only operations, presumed-abort is usually more efficient in terms of messages and log write overheads. This observation has meant that PA with the read-only optimisation has been adopted in almost all distributed database and transaction processing products. These include Tandem's TMF [30], DEC's VMS [7], BEA systems' TUXEDO [40] and more recently Microsoft's DTC [20]. The 2PC protocol combined with PA is now part of the ISO/OSI and X/Open distributed transaction processing standards.

Attempts have been made to eliminate the requirement of the coordinator to force write an initiation record, when using PC. The *new presumed commit* 2PC variant [46] achieves this by maintaining two sets of transactions: the set of recent transactions, and the set of potentially initiated transaction. We omit the details here but by using these sets, after a crash, upon recovery a coordinator can safely presume a transaction has committed if it is *not* potentially initiated.

Yet another strategy, used this time to eliminate the voting phase, gives rise to the *unsolicited update vote* (UUV) 2PC variant [78]. In this protocol when a participant site acknowledges its last transaction operation it also votes. After receiving all the vote/acknowledgments the coordinator can go straight to the second phase and sends the sites the commit or abort decision. Unfortunately, this strategy has the shortcoming that a participant must know when it has finished all its operations for a particular transaction or as we will see in the early prepare protocol it must act as though each action was its last. The *early-prepare* protocol [75] works in a similar way. This time a site force writes a prepare record every time it sends an acknowledgement of a read or write operation. If this is the

[5]Transactions often lock data objects to provide different levels of isolation. A shared lock prevents other write accesses to a data object, while still allowing reads.

last action the coordinator can proceed to the last phase and send the outcome to the participant sites. If the coordinator sends another operation request to a site the transaction again becomes active at that site. Because sites must force write a record with every operation acknowledgement the early-prepare protocol requires an environment where the cost of writing to stable storage is low.

The *coordinator log* (CL) protocol and the *implicit yes vote* (IYV) protocols work in a similar fashion to early-prepare but improve upon early prepare by eliminating the need for participants to force write log records with each acknowledgment. In the coordinator log protocol this is accomplished by participants sending log records with their vote. In this way logging is centralised at the coordinator. The IYV protocol works in a similar way, while each participant still maintains a transaction log, it eliminates the requirement for each participant to log a prepare record by replicating the redo part of its log at the coordinator. Many of these variants (e.g. IYV and UUV) can be combined with PC to remove the requirement for PC to force write an initiation record when transactions are read-only. The argument against PC and in favour of PA when read-only transactions are present then swings more in favour of PC. There are many other two-phase commit optimisations for instance *last agent* [68] and *group-commit* [31].

We have discussed 2PC protocol optimisations where the structure of the commit protocol is flat. That is to say we analysed the situation where one coordinator manages several child participants. Hierarchical variants of 2PC are also common in transaction processing systems. In these variants coordinators, both coordinate, and are themselves participants. This gives rise to a transaction tree structure as found in IBM's LU 6.2 [23].

In this section we have surveyed several commit optimisations of the popular 2PC protocol. The emphasis has shifted away from analysing the behaviour of a single commit protocol and towards the interaction of many concurrent transactions and the way they interact in a distributed transaction processing environment. In so doing, the environment in which the protocols are studied has become much richer. Locking and logging are introduced. Different types of transactions such as read-only are considered and their impact on the optimisation of commit protocols was examined. This extra complexity required modelling, not only the commit protocol, but also the preceding read and write operations a transaction performs during its execution. Modelling this richer environment is much harder since not only are the components of the systems more complex but there may be arbitrary numbers of transactions executing concurrently.

The models that describe these richer systems are particularly useful for be-

havioural analysis. They tend to describe the different local states of processes in the system and also describe what kinds of messages are exchanged and when they are exchanged. If one is interested in measuring the performance of a protocol, often expressed as the number of messages sent or log writes performed, these types of models are often not appropriate. The reason for this is that in real distributed transaction processing systems many optimisations are employed. For example, messages are usually piggybacked to reduce the load on a network or log writes are batched and then performed together. When analysing the role of commit protocols from a performance perspective therefore other modelling techniques are often more appropriate. To date techniques for analysing performance have fallen into three broad categories. Prototype systems such as ARIES and R* [57] are built and different techniques are tested. Discrete event simulation models are constructed and performance is evaluated by executing these models with different modelling parameters [65]. Lastly, stochastic process algebra models such as PEPA [15] or GSPNs [67] can be used.

2.5 Failure models

Current literature relating to commit protocols makes different assumptions about failure. In general, failure is classified into two different categories *site failure* and *communication failure*. A model may assume that neither, one of, or both of these types of failure are possible.

2.5.1 Site Failure

Site failure is usually modelled by allowing sites in the system to crash. Some models allow them to recover from a crash failure and some do not. The concept of non-volatile state, often in the form of a transaction log, is often included in models that consider recovery. Upon recovery, sites lose their pre-crash state unless it was written to non-volatile storage before the crash. Lampson and Sturgis [47] discuss physical devices and their failure modes, and how to build stable storage and transactions on top of them. Many different models of crash failure exist.

Another model of failure is so called *Byzantine* failure [45] in which a faulty or rogue site may misbehave and produce spurious messages to disrupt a protocol. Some interesting bounds are derived on the number of rogue processes that are allowed in a protocol that solves the so called byzantine agreement problem before it is disrupted to such an extent that the protocol fails. A comparison of Byzantine

agreement and atomic commit is given in by Gray in [33].

2.5.2 Communication Failure

Communication failure is usually modelled in one of three different ways. The first assumes the sending and receipt of messages is unreliable. A message might be lost after sending and thus never received. This type of failure assumption can be found in the "best effort datagram" model on which the Internet Protocol (IP) is based [64]. Basu *et al.* [5] classify two types of unreliable link[6], known as *eventually reliable*, and *fair lossy* links. A fair lossy link guarantees that if an infinite number of messages are sent, then an infinite subset of these messages are received. Clearly such a link can lose an infinite number of messages. With an eventually reliable link, there is a time (not necessarily known) after which all messages sent are eventually received. Messages sent before that time may be lost. Any link that is eventually reliable is also fair lossy. From a practical perspective, the eventually reliable and fair lossy links seem to capture message passing behaviour in networks that might fail but at some point recover.

To see why the fair lossy assumption is useful consider two processes, a sender s and a receiver r connected by a bidirectional channel over which they pass messages. Process s wishes to send a message m to r. If we put no restriction on message loss it is obviously impossible to ensure that r receives m. If however the link is fair lossy one can adopt the following strategy. s can send copies of m forever, and r is guaranteed to eventually receive m. We still have the problem that s never stops sending messages. To fix this r can send an $ack(m)$ on every receipt of m and once s receives the $ack(m)$ it stops sending. This in turn means that r will stop sending $ack(m)$. Note the protocol is *quiescent*: eventually no process sends or receives messages.

If we now also allow processes to crash the situation changes. The simple protocol above still works but, if for instance r crashes before sending an $ack(m)$ s will send messages forever. The protocol is no longer quiescent. It turns out that there is no quiescent protocol that ensures that even if s and r do not crash then r eventually receives m. This would appear to suggest that quiescent reliable communication channels cannot be built on top of unreliable channels in asynchronous message passing systems. The problem centres around the inability to detect whether a site has failed or if it is just slow in responding. In fact this is the crux of the well known impossibility of consensus result of [27].

A failure detector [11] is an oracle that a process can query. It produces a list of

[6] An unreliable link contrasts with a reliable link which never loses a message.

processes that it suspects may have crashed. The list provided is unreliable in the sense that it might at any time make errors of omission and errors of commission in compiling its list of crashed processes. The *eventually perfect* failure detector $\diamond\mathcal{P}$ has the following properties: (1) if a process crashes there is a time after which it is permanently suspected, and (2) if a process does not crash for sufficiently long then there is a time after which it is never suspected. Using $\diamond\mathcal{P}$ we can modify our simple protocol to provide quiescence. If s has not received an $ack(m)$ from r it periodically consults $\diamond\mathcal{P}$ to see if r is suspected. If it is not suspected s sends a copy of m to r. Clearly, the protocol is now quiescent. It turns out $\diamond\mathcal{P}$ is the weakest failure detector that can be used to provide quiescent communication [2].

Unfortunately, $\diamond\mathcal{P}$ is not implementable in asynchronous systems so it would seem we are no nearer to solving the problem of quiescent communication over lossy links with process crashes. The goal posts have shifted but the problem still remains. However, Aguilera and Toueg [1] introduce a heartbeat failure detector $\mathcal{HB}$ that is not limited to just producing lists of suspects. Essentially each process sends a *keep alive* messages to all others. At p, $\mathcal{HB}(q)$ outputs the number of keep alive messages p has received from q. Using $\mathcal{HB}$, quiescent reliable communication is possible. Obviously, the $\mathcal{HB}$ mechanism itself is not quiescent (because of the keep alive messages) but it can be implemented as an operating system service.

Although of theoretical interest the theory of failure detectors does not seem to reflect message passing systems in a practical sense. For instance the TCP/IP protocol [22] has been used for many years to send billions of messages over the Internet (which does not provide reliable links) reliably. In essence[7] TCP implements a protocol where s sends m until it receives an $ack(m)$. If no $ack(m)$ is received it will retry but there is a bound on how long it will continue to retry, likewise r will not try to $ack(m)$ forever. Once this bound is reached TCP reports that the link or connection has been lost and gives up. In a practical sense this protocol provides reliable communication over links that may lose messages.

A second interpretation of communication failure is the partition model, already seen in our discussion of Skeen's quorum based protocol. In fact, the two models are related. If messages between one group of sites and another are lost for a period of time (perhaps because a router or bridge between those sites has failed) we could conclude that the sites are partitioned. Other relationships exist between site failure and communication failure. If a site is relied upon as a third party to forward messages (for instance using IP forwarding), and that site fails

[7] A complex sliding window is implemented with backoff and resend parameters for increased efficiency.

this causes communication failure. Furthermore, the distinction between site and communication failure is not always helpful since many protocols guarantee reliable communication between two sites as long as they both remain operational for "long enough". They do this by means of acknowledgements and re-sends. Ricciardi *et al.* [66] discuss the relationship between the partitioning model and the lossy link model. Once again there is a great diversity in the different types of failure assumptions made within commit protocol models. This often leads to a lack of clarity in their specification and analysis.

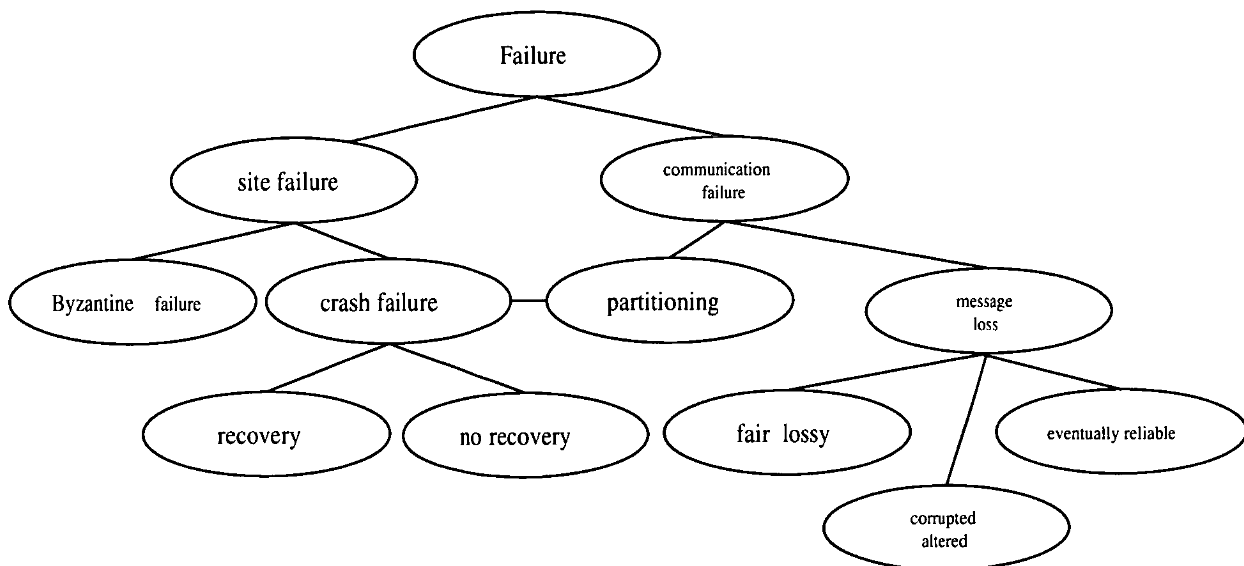

Figure 2.6: Relationship of different failure assumptions in protocol models.

Some message failure models allow for messages to be corrupted or maliciously altered while in transit. From a practical perspective corrupted messages are of little interest as algorithms exist which can detect, with very high accuracy, corrupted messages and either rectify the corruption in the case of Hamming Codes [39], or detect the corruption and discard the message, in the case of cyclic redundancy checks [26]. More interesting is the case where messages are deliberately altered. For instance, an attack on TIP, an Internet transaction commit protocol is possible by maliciously altering messages [44]. Again from a practical perspective, using crytography [79] messages can be securely signed and encrypted to prevent this type of malicious attack. Figure 2.6 summarises the domain of different failure assumptions that can be made in protocol models.

2.6 Casting the Net Wider

Atomic commit protocols have been adopted in many different areas. They are used for example in IBM's reliable message queue, MQSeries [56] to provide transactional semantics in messaging. Kempster *et al.* [42] discuss how to extend network management protocols with atomic commit protocols to provide transactional properties to network reconfiguration. Luan and Gligor [49] use them to implement a novel atomic broadcast and Li *et al.* [48] propose a commit style protocol to facilitate connection setup in telecom switches. Tygar [81] discusses commit style protocols and outlines some open research questions in the growing area of e-commerce. Thanisch [80] provides a survey of the role of atomic commit protocols in many of these less traditional environments.

The extent to which computing devices are becoming connected is staggering. The advent of wireless communication means that almost any device mobile or static can be connected to a network. Economies of scale are reducing the cost of sending messages over these networks dramatically. For example, it is now usually more expensive to produce and send a domestic customer's phone bill than it is to physically route their calls. As this trend continues more and more challenges will face protocol designers that want to provide new applications which exploit these new opportunities.

2.7 Conclusions

In this chapter we have reviewed the role of commit protocols in many different areas from transaction processing to e-commerce. Of particular interest was the way atomic commit protocols behave in isolation, particularly with respect to blocking, and also how they interact when they take part in more complex transaction processing systems. Although many different models have been proposed, no one model or modelling technique seems to be generally applicable. Furthermore, assumptions about the distributed environment in which the atomic commit protocols are studied varies widely. This is particularly true of assumptions made about site and communication failure. For these reasons it is difficult to mould the various models of commit into a hierarchy. Some models are non-comparable and thus a unified notation for all models is very difficult.

Chapter 3

Modelling Atomic Commit Protocols

3.1 Introduction

Although in the majority of literature on commit protocols pseudo-code and *ad hoc* arguments and notation are used, attempts have been made to use general modelling techniques. In this chapter we will examine three such modelling techniques for distributed systems: the I/O automata model [51, 52], a knowledge based model proposed by Hadzilacos [36] and the calculus for communicating systems [55]. We discuss the merits of each and examine how they have been used to provide a formal basis for analysing various aspects of commit protocol behaviour. After summarising the strengths and weaknesses of these approaches we introduce our views based modelling technique and compare it to the other modelling techniques presented.

3.2 I/O Automata

I/O automata provide a general modelling framework for describing a wide variety of distributed message passing systems. An I/O automaton models a component in a distributed system. Essentially it is a state machine with transitions that are associated with actions. These actions are one of three types: *input*, *output*, or *internal*. Input and output actions are used to communicate with other automata whereas internal actions are invisible outside of the automata. An example of a simple I/O automaton model of one bit latch is given in figure 3.1. A latch has an output action, *out* and an input action *in* whose purpose is to store a single bit of information. To describe distributed systems using I/O automata smaller automata are *composed* into larger systems by matching the input of one

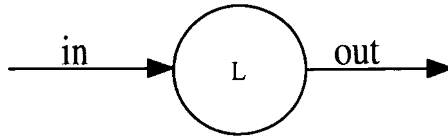

Figure 3.1: A Simple single bit latch I/O Automaton.

automaton with outputs of others with the same name.

Formally an I/O automaton, A, consists of five components:

- $sig(A)$, a signature consisting of three disjoint sets of actions. The internal actions $int(A)$, the output actions $out(A)$, and the input actions $in(A)$. We denote all the actions of A, $acts(A)$.
- $states(A)$ a (possibly infinite) set of states.
- $start(A)$ a non-empty set of initial states for A.
- $trans(A)$ a (possibly infinite) state transition relation. $trans(A) \subseteq states(A) \times acts(A) \times states(A)$, for every state $s \in states(A)$ and every input action $\pi \in in(A)$ there is a transition $(s, \pi, s') \in trans(A)$. The transition relation is often specified as a set of pre-conditions and post-actions see Table 3.1. The pre-condition restricts the set of states in which an action may take place and the post-action produces a new state from the state in which the action was applied. The set of pre-conditions, taken together, must ensure that for each input action and state there is a transition.
- $tasks(A)$, a partition of the set of external actions used to define what it means for the automaton to have *fair* executions.

The transition relation of I/O automata is usually given in a *pre-condition*, *post-action* or *effect* style. If a pre-condition is true then an action may happen. The action or effect is an indivisible event that takes a state s satisfying the pre-condition to a post state s'. For example the one bit latch I/O automaton is shown in Table 3.1.

An *execution fragment* of an I/O automaton is a finite sequence, $s_0 \xrightarrow{\pi_1} s_1 \ldots, s_{r-1} \xrightarrow{\pi_r} s_r$, or infinite sequence $s_0 \xrightarrow{\pi_1} s_1 \ldots, s_{r-1} \xrightarrow{\pi_r} s_r \xrightarrow{\pi_{r+1}} \ldots$ of alternating states and actions of A such that $(s_k, \pi_{k+1}, s_{k+1}) \in trans(A)$ for every $k \geq 1$. An execution fragment beginning with a start state is called an *execution*. The set of executions is denoted $exec(A)$. It is often useful to discuss just the external behaviour of A. We call this a *trace* of A, denoted $trace(A)$ which is the

Signature	
Input: *in*	Output: *out*
States	
boolean b; initially false	
Transitions	
in	*out*
precondition: $\neg b$	precondition: b
$b := \texttt{tt}$	$b := \texttt{ff}$
Tasks	
$\{\{out\}, \{in\}\}$	

Table 3.1: An I/O automaton for simple one bit latch

subsequence of an execution restricted to just the external actions. For example a trace of the one bit latch automaton is *in out in out* ....

The I/O automata model has an implicit notion of fairness. It is argued that the most interesting executions are those that are fair[1]. In other words if the task classes represent independent threads of execution within each automaton each thread should not be starved within any execution. More formally an execution fragment α of A is said to be *fair* if the following condition holds for each class C of $tasks(A)$:

- If α is finite, then C is not enabled[2] in the final state of α.
- If α is infinite, then α contains either infinitely many events from C or infinitely many occurrences of states in which C is not enabled. This is sometimes referred to as strong fairness [59]

A key property of I/O Automata is the ability to compose multiple automata into larger systems. When two[3] automata A and B are composed, denoted $A||B$, the composition operator identifies actions with the same name, say π, from A and B. In the composed automaton when a component takes a step involving π both A and B take a step involving π. Several restrictions are imposed: 1) A and B must have disjoint sets of internal actions because internal actions should not be seen outside of the automata, and so should not take part in communication. 2) A and B should have disjoint output actions, this ensures that only

[1] It is also interesting to see what are the consequences for a system which does not guarantee fairness.

[2] A class C is said to be enabled in state s when some action in C can happen from s.

[3] We consider the case for two, but the case for many is similar.

one output can control inputs of other automata. There is no restriction on the number of inputs a single output action may control. The transitions of the composed automata are obtained by allowing all the component automata that have a particular action π to participate simultaneously in steps involving π, while all other component automata do nothing. The task partition of the composition is formed by taking the union of the component's task partitions. A hiding operator $\backslash$ is also defined so that output actions can be internalised preventing them from being used for communication. For example the latch automaton, L, of figure 3.1 could be restricted so that it could only produce output using such an operator thus $L \backslash \{out\}$. A renaming operator is also introduced. $L\{out/synch\}$ is the I/O automata L with its *out* action renamed *synch*. Figure 3.2 shows how two single bit latch automata can be composed into a two bit buffer.

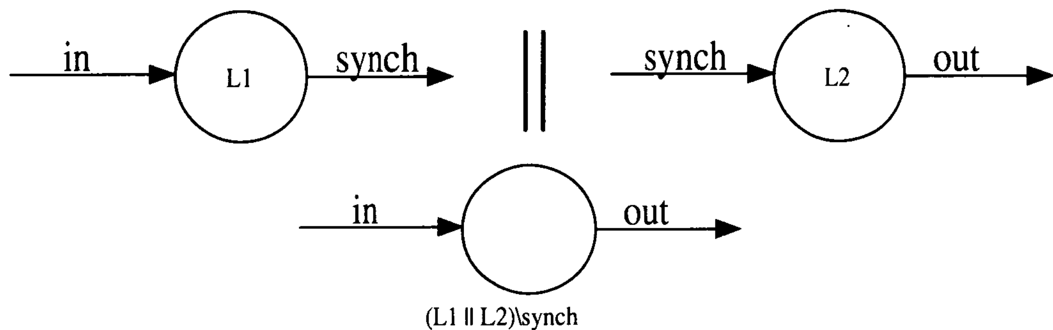

Figure 3.2: Composing two one bit latches.

We have discussed how I/O automata can be used to model distributed systems but it is useful to see how, once modelled, properties of these systems can be proved. Because the set of all execution traces captures all possible behaviour, safety properties over these traces can be expressed, by asserting that in every[4] trace (or fair trace) of an automaton some undesirable action never happens. Similarly, liveness properties can be expressed by stating that in every trace (or fair trace) a desirable action eventually happens.

The I/O automata model lends itself to compositional reasoning. Suppose automaton A is constructed by composing a set of automata $\{A_i : i \in I\}$. Suppose also that A_i satisfies some trace property P_i and each property P_i belongs to a special class of trace properties that are preserved under composition. We can then deduce that the composed I/O automaton A will satisfy the composed trace property P. We will return to this subject later in chapter 5 when we carefully define these types of properties for our views based model.

[4]This is sometimes referred to as a strong safety property as opposed to a weak safety property where at least one trace must have the property.

Compositional reasoning is very useful as it facilitates the proof of properties of larger systems by proving properties of their constituent parts. A technique known as simulation is also commonly used in I/O automata to show that a higher level abstraction, say automaton A, is equivalent to (exhibits the same external behaviour as) a lower level automaton say B. A formal account of what it means for A to simulate B can be given. This technique has been used to give hierarchical proofs of properties [53] of distributed systems. It can also be automated to some extent with the aid of theorem provers. Nipkow [60] uses the Isabelle theorem prover and Søgaard-Andersen *et al.* [73] use the Larch theorem prover to this end.

Das and Fekete [21] use I/O automata to model a transaction processing system. The components of the system are *transaction automata* which model the operations of a transaction, *local manager* automata that model a distributed commit protocol and *crashing object* automata that model resources that are accessed by the operations of transactions. Requirements are given for each of the automata that constitute the overall system. It is then shown that if all these requirements are met the composed system is correct. A particular instance of a commit protocol (2PC) automaton is given and it is shown that it meets their requirement specification. Perhaps the most interesting part of this work is its compositional approach.

I/O automata provide a framework that is both general and formal in which to specify, model and verify properties of distributed systems. Unfortunately, due to its generality, when modelling asynchronous message-passing systems, message-passing is modelled explicitly. Furthermore, to model systems with arbitrary numbers of processes I/O automata and their message passing connectives are usually parameterised. This means that automating proofs of properties modelled in this way becomes difficult. Our model presented later in this chapter addresses these problems by introducing *views*. Using views means that message passing need not be made explicit within the model.

As we have seen a notion of fairness is implicit in the I/O automata model. In our views based model we make no restrictions on executions but instead prefer to include this as part of the property being checked. This reduces the complexity of the model and furthermore accommodates different definitions of fairness such as those found in [59]. As we shall see later, in some cases, our modelling technique allows us to reason about systems with arbitrary numbers of processes. In so doing we will examine classes of properties that are preserved by compositional operators similar to the techniques used in I/O automata.

3.3 A Knowledge Theoretic Approach

A different approach to modelling and analysing commit protocols is rooted in knowledge theory [38]. The I/O automata technique provides an operational semantics, where the details and contents of messages and the state of each process and its communication medium are modelled explicitly. There are also fairness constraints placed on the actions of the automata. In contrast the knowledge based approach constructs arguments based on the information that must be present, at a particular site, or within a group of sites, in order, for example, that that site may commit or abort a transaction. In so doing it uses a very general model of distributed computation and then constructs arguments about the amount of information which a site must acquire, if for example blocking is to be avoided in the presence of site failure[5].

3.3.1 Modelling Distributed Systems

Hadzilacos [36] constructs a model of a distributed system that consists of a set of sites $\Pi = \{p, q, \ldots\}$ and a message buffer M. Sites in Π communicate by exchanging messages through the message buffer M. A site p can execute one of three types of actions, where m is the value of some message.

- $\texttt{SEND}(m, q)$ process p sends a message m to process q;
- $\texttt{RCV}(m, q)$ process p receives a message m from q, a null message λ is introduced to model the case where p tries to receive a message from q but there is no message in M for it to receive.
- `LOCAL` this models a process taking some internal step. A special `LOCAL` action `FAIL` exists to model crash failure. Once a process executes this action it can take no more actions.

Hadzilacos models an execution of a distributed system as a *run*. Informally we imagine that there is an omnipresent observer that samples the states of the processes and the message buffer at discrete real time instances $0, 1, 2 \ldots$. The state of a process, at a point in a run, is the (finite) sequence of actions it has executed up to that point in the run. The state of a message buffer is the set of messages (messages can be assumed to be unique by tagging each one with a unique identifier) that have been sent but have been neither delivered nor lost. We can think of the sampling instants as the ticks of a perfect clock available

[5] A later result introduces the possibility of message loss.

to the observer; this clock is a fictional device - in particular, we assume that processes do not have access to it.

We will now introduce some notation on sequences. Let x, y be sequences of actions. We write $e \in x$ to denote that action e is in x. $x \circ e$ denotes the sequence resulting by appending e to x. $x \leq y$ ($x < y$) denotes that x is a prefix (proper prefix) of y; in that case we write $y \setminus x$ to indicate the sequence whose concatenation to x equals y. We denote the length of sequence x, $\| x \|$. We adopt the convention that the letters p, q denote processes from Π and i, j, k, l denote natural numbers.

Formally a *run* is a function r mapping each pair (p, i) to a sequence of actions (the actions executed by p up to and including time i) and the pair (M, i) to a set of triples of the form (q, m, p) where m is a non-null message (the contents of the message buffer just before the actions at time $i + 1$ are performed). (q, m, p) indicates that q sent message m to p. A run r must satisfy the following properties:

- $r(p, 0)$ is the empty sequence and $r(p, i) \leq r(p, i+1)$ - initially, each process starts having executed no actions and the sequence of actions taken by a process can only be extended or remain unchanged with each clock tick.

- $\| r(p, i + 1) \setminus r(p, i) \| \leq 1$ - for each i, the clock ticks sufficiently often so that no process performs more than one action between successive ticks.

- If $\texttt{FAIL} \in r(p, i)$, no event may follow $\texttt{FAIL}$ in $r(p, i)$.

- If $\texttt{RCV}(m, q) = r(p, i + 1) \setminus r(p, i)$ and $m \neq \lambda$ then $(q, m, p) \in r(M, i)$ - a non-null message can be received only if it was in the message buffer in the previous time instant.

- $r(M, 0) = \emptyset$ and, for all $i \geq 0$, $r(M, i + 1) \subseteq r(M, i) \cup \{(q, m, p) : \texttt{SEND}(m, p) = r(q, i+1) \setminus r(q, i)\} \setminus \{(q, m, p) : \texttt{RCV}(m, q) = r(p, i+1) \setminus r(p, i)\}$ - the message buffer contains only messages that were sent but not yet received. If $\texttt{SEND}(m, q) \in r(p, i)$, $\texttt{RCV}(m, p) \notin r(p, i)$, we say m is a lost message.

Using this structure we can model a distributed system as the set of all possible behaviours of its constituent components, i.e. of the processes and the message buffer. Thus the distributed system is defined as a set of runs. Not all runs are valid and so we require some closure properties to capture the idea that processes can only effect each others behaviour through communication actions. We can informally state these as follows.

- S1 The ability of a process to perform a `LOCAL` or `SEND` action is determined by its own behaviour, not the behaviour of other processes.
- S2 The ability of a process to *attempt* to receive a message from some other process is determined by its own behaviour; the message it actually receives (possibly null) depends on other processes.
- S3 The behaviour of a process after it sends a message cannot determine the recipient's ability to receive that message.
- S4 Process q can prevent p from performing $\mathtt{RCV}(\lambda, q)$ only by sending messages to p.

At this point it is useful to define some further notation and terminology. A *point* is a pair (r, i) where r is a run and i is a natural number. We say a run s *extends* point (r, i) if for every $j \leq i$ and every p, $s(p, j) = r(p, j)$, and $s(M, j) = r(M, j)$ i.e., up to time i, all processes and the message buffer behave identically in r and s.

For any process p a relation $\approx_p$ is defined between points as follows: $(r, i) \approx_p (s, j)$ iff $r(p, i) = s(p, i)$. Informally this says that (r,i) and (s,j) look the same to p. The relation can be extended to sets of processes as follows: $(r, i) \approx_P (s, j)$ iff $r(p, i) = s(p, i)$ for every $p \in P$.

3.3.2 A Knowledge Logic

Hadzilacos makes use of a knowledge logic introduced by Halpern and Moses [37] in order to state precisely and succinctly various properties of distributed computations.

For a fixed set of processes Π and primitive propositions F, including the primitive propositions true and false, the set of formulae can be defined inductively, where $P \subseteq \Pi$, as follows:

- Every primitive proposition is a formula.
- If Φ_1 and Φ_2 are formulae, so are $\neg\Phi_1$, $\Phi_1 \vee \Phi_2$, $\Box\Phi_1$, $\Diamond\Phi_1$, $K_P\Phi_1$

We write $(r, i) \models \Phi$ to express that Φ is true at point (r, i). Truth of a formula Φ at a point is defined by structural induction on the syntax of Φ as follows.

- If Φ is a primitive proposition then its truth is defined by an interpretation function $I(\Phi)$ that specifies at which points a primitive proposition holds.
- $(r, i) \models \neg\Phi$ iff it is not the case that $(r, i) \models \Phi$.

- $(r, i) \models \Phi_1 \vee \Phi_2$ iff $(r, i) \models \Phi_1$ or $(r, i) \models \Phi_2$.

- $(r, i) \models \Diamond\Phi$ iff for every run s extending (r, i) there is some $j \geq i$, $(s, j) \models \Phi$ - i.e. Φ is true now or will eventually become true at some point, in *any* possible future.

- If $\Phi = \Box\Phi$, $(r, i) \models \Phi$ iff for every run s extending (r, i) and every $j \geq i$, $(s, j) \models \Phi$ - i.e. Φ is true now and will remain so in *any* possible future.

- $(r, i) \models K_P\Phi$ iff for every (s, j) such that $(s, j) \approx_P (r, i)$, $(s, j) \models \Phi$ i.e.,Φ is true at *every* point which, from P's (collective) point of view, are indistinguishable from the present point.

The definition of $K_P\Phi$ is known as the *total view interpretation* of logic; other definitions are proposed in [37]. The following facts follow from these definitions.

- If $(r, i) \models K_P\Phi$ then $(r, i) \models \Phi$ i.e. only truths are known.

- If $(r, i) \models K_P\Phi$ and $P \subseteq Q$ then $(r, i) \models K_Q\Phi$.

- If $(r, i) \models K_P\Phi$ and $(s, j) \approx_P (r, i)$ then $(s, j) \models K_P\Phi$.

In order to model a commit protocol some primitive propositions are introduced namely; $\texttt{YES}_p$ which is true at a point if p has voted `yes` (similarly `no`), $\texttt{COMMIT}_p$ which is true if a process has committed (similarly aborted). The proposition $\texttt{FAILED}_p$ is also introduced. A formula Φ is *local* to a set of processes P iff the truth or falsity of Φ is always known to P. That is, for all (r, i), $(r, i) \models K_P\Phi \vee K_P\neg\Phi$.

Using this knowledge logic Hadzilacos defines the problem of Atomic Commitment by restricting possible runs. For example, initially a process has not cast its vote and, until it does, it can vote either way. This can be stated formally as follows:

$$\text{for all points } (r, i) \text{ and for all } p \in \Pi,\ (r, 0) \models \neg(\texttt{YES}_p \vee \texttt{NO}_p)$$

To express the property of **AC2** that the decision to commit is reached only if all process' votes are `yes` we write

$$\text{for all points } (r, i) \text{ and for all } p \in \Pi,\ (r, i) \models \texttt{COMMIT}_p \rightarrow \bigwedge_{q \in \Pi} \texttt{YES}_q$$

Within this knowledge model Hadzilacos also defines what he calls *2PC level of knowledge*. A protocol has this property if, for all runs, a site commits if and

only if it is known by that site that all sites have voted **yes**.

$$\text{for all points } (r,i) \text{ and for all } p \in \Pi,\ (r,i) \models \texttt{COMMIT}_p \text{ iff } (r,i) \models K_p \bigwedge_{q\in\Pi} \texttt{YES}_q.$$

It is then shown that any protocol that solves the AC problem must exhibit this property. Suppose not. Thus, for some point (r,i) and $p \in \Pi$, $(r,i) \models \texttt{COMMIT}_p$ but $(r,i) \models \neg K_p \bigwedge_{q\in\Pi} \texttt{YES}_q$. Therefore there must exist some $(s,j) \approx_p (r,i)$ such that $(s,j) \not\models \bigwedge_{q\in\Pi} \texttt{YES}_q$. Since $\texttt{COMMIT}_p$ is local to p, $(r,i) \models \texttt{COMMIT}_p$ and $(s,j) \approx_p (r,i)$, it follows that $(s,j) \models \texttt{COMMIT}_p$. But then (s,j) contradicts **AC2**.

A *3PC level of knowledge* is also defined. It is shown that for a protocol to be non-blocking (when site but not communication failures may happen) a site commits if and only if it knows every non-failed site knows all sites voted **yes**. Using these techniques two interesting results are derived. Firstly, a non-existence proof for a non-blocking atomic commit protocol in the presence of communication failure is constructed. This supports Skeen's alternative proof. Secondly, a lower bound of $2(n-1)$ on the number of messages required by a protocol to solve the atomic commit problem in this model is given. In fact, linear 2PC [32] achieves this bound. Hazilacos makes no attempt to model recovery. When a site crashes perhaps unrealistically it never recovers.

The modelling approach used is general, formal and concise. It focuses on what is known and what must be known in a system to solve the atomic commit problem. The declarative style means that protocol descriptions are divorced from implementation details. One of its major strengths is that because it focuses on what level of knowledge is acquired by sites, it is not burdened by having to model message passing explicitly. This important observation motivates our views based model where we incorporate a site's knowledge of other sites state explicitly within that site's state. This abstracts the details of message passing while at the same time maintaining some of the desirable properties of an operational modelling approach.

3.4 The Calculus for Communicating Systems

Many process algebras have been proposed for modelling distributed systems. CCS [55] provides a framework in which to describe communicating processes or agents. Agents have state and perform actions which are either input, output or so called silent or τ actions[6]. After performing an action an agent changes

[6]The τ action is similar to the internal actions of I/O Automata.

state. Operators are defined allowing, amongst other things, the composition of agents. In a composed agent, communication can take place between two agents when one agent has an output action with the same name as another agents input action. In this case both actions can happen (sometimes called a handshake) to form a silent τ action. We can model a FIFO buffer below as:

$$\begin{array}{lcl} \text{FIFO}\langle\rangle & \stackrel{def}{=} & send(m).\text{FIFO}\langle m\rangle \\ \text{FIFO}\langle m_1,\ldots,m_n\rangle & \stackrel{def}{=} & \overline{recv}(m_n).\text{FIFO}\langle m_1,\ldots,m_{n-1}\rangle \\ & & +send(m).\text{FIFO}\langle m, m_1,\ldots,m_n\rangle \end{array}$$

A transition semantics is given for CCS in which agents exhibit behaviour as sequences of actions. For example for the model above we have the following transition semantics.

$$\begin{array}{lcr} \text{FIFO}\langle\rangle & \stackrel{send(m)}{\longrightarrow} & \text{FIFO}\langle m\rangle \\ \text{FIFO}\langle m_1,\ldots,m_n\rangle & \stackrel{\overline{recv}(m_n)}{\longrightarrow} & \text{FIFO}\langle m_1,\ldots,m_{n-1}\rangle \\ \text{FIFO}\langle m_1,\ldots,m_n\rangle & \stackrel{send(m)}{\longrightarrow} & \text{FIFO}\langle m, m_1,\ldots,m_{n-1}\rangle \end{array}$$

Equivalence of agents can be defined using bisimulation. This is similar to the simulation techniques used in I/O automata. For example, consider the two CCS agents that model a ticking clock below.

$$\begin{array}{l} Cl_1 \stackrel{def}{=} tick.Cl_1 + tick.tick.Cl_1 \\ Cl_2 \stackrel{def}{=} tick.Cl_2 \end{array}$$

Cl_1 is bisimulation equivalent to Cl_2 because any action Cl_1 performs can be matched by Cl_2 and furthermore the resulting agents reached after this action are themselves bisimilar.

Using the transition semantics one can unfold a CCS agent into a transition system. For example the transition system of Cl_1 and Cl_2 are given in figure 3.3. Behavioural properties of agents can be verified using model checking techniques

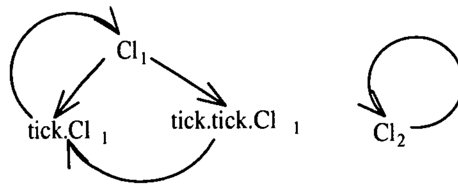

Figure 3.3: Transition systems generated from two simple of a simple CCS agents. All transitions are `tick` transitions.

on the resulting transition systems. For example the games-based model checking techniques of Stirling [77] are used to check modal μ-calculus properties of CCS

agents. An implementation of these techniques exists in the Edinburgh Concurrency Workbench [58].

To model a commit protocol, for example Centralised 2PC, using CCS we define a set of participant processes P_i as CCS agents. We also define a coordinator agent C. A protocol system is then the composition of these agents $P_1|P_2|\ldots|P_n|C$. Each process communicates with the coordinator by means of handshakes. Often we wish to model failure or message passing delay. In this case we must introduce some kind of environment like a message buffer. Agents now communicate through the message buffer. When modelling commit protocols using I/O Automata a similar strategy is used. Simple I/O Automata are defined to model components in the commit protocol and then they are composed to model the whole system.

CCS provides a very general modelling technique. Unfortunately this generality means that when used to model message passing systems such as commit protocols the state space of the resulting transition systems is very large even for very simple commit protocols. This state explosion problem, in the case of centralised commit protocols, seems to be due to two factors. Firstly if messages are modelled explicitly a coordinator must keep a state to reflect which out of the n participant agents have voted yes. There are 2^n different states between nobody voting yes and the state where everyone has voted yes. Although different techniques exist for reducing the number of states, the problem seems rooted in the requirement to explicitly model message passing with agent handshakes. Secondly, a naive approach to modelling arbitrary numbers of sites gives rise to infinite state spaces as we describe later in chapter 5.

The state space explosion problem aside, in basic CCS there is no mechanism for specifying the behaviour of individual agents using the pre-conditions and post-actions method, as there is in I/O Automata. This means it can be very difficult to write down succinctly the behaviour of complex protocols. Perhaps for these reason CCS has not been widely adopted as a modelling technique for commit protocols.

3.5 Comparing Techniques

Each of the models we describe has some similarities. When protocols are modelled using these techniques the behaviour of the sites taking part in a protocol are described by the possible steps or actions they might take. By composing groups of sites (often within an environment) we can generate runs or executions

as sequences of actions taken by the sites. The set of all possible runs captures the behaviour of the entire system. A property of the protocol being modelled therefore is defined as a proposition (often formalised in temporal logic) over these runs. For example, a property might be that in every run eventually the coordinator decides commit or abort.

Ideally, a good modelling technique for commit protocols should be: intuitive enough so that protocol designers can easily model their protocols; expressive enough to facilitate the modelling of a wide and rich variety of commit protocols and environments; precise enough to accurately describe the behaviour of the protocol being modelled; and designed in a way that allows support for automated reasoning techniques, such as model checking. The techniques we surveyed fulfil some but not all of these requirements. For example CCS provides a very well structured and precise account of communicating systems which has a formal transition semantics that can easily be used to generate transitions systems thus supporting automatic verification through model checking. Unfortunately, its generality means that it quickly becomes cumbersome and difficult to use for anything other than the simplest commit protocols. I/O automata allow more complex protocols to be modelled easily but the resulting state space and transition systems that express their behaviour can be very large. This is largely due to the fact that message passing behaviour must be made explicit. Knowledge based protocol analysis abstracts message passing but because the knowledge is not made part of the site's state it is more difficult to generate a transition semantics for the protocols being modelled. Table 3.2 describes these differences, including a comparison with the views based model developed as part of this thesis and described in the next section.

3.6 The Views Model

The I/O automata and CCS modelling techniques are highly operational in their semantics. This has the advantage that transition systems can easily be generated from the models to reflect behaviour. Unfortunately, when modelling message passing systems, because messages are usually modelled explicitly, very large and intractable state spaces often result. The knowledge based modelling approach, on the other hand, abstracts many of the details of message passing preferring to construct arguments based on the level of knowledge acquired by processes in the system. Unfortunately, the resulting knowledge based models lack the desirable operational qualities of CCS and I/O automata models. It seems that, for the

	I/O Automata	KB	CCS	Views
Suitability for modelling atomic commit	medium	medium	poor	good
Generality	high	medium	high	low
Generation of transition system				
Ease of transition system generation	medium	medium	easy	easy
Size of resulting state space	large/∞	medium	large/∞	small
Easily scalable to arbitrary size	no	yes	no	yes
Support for automated proof techniques				
Amenable to model checking	yes	medium	yes	yes
Amenable to theorem proving	yes	yes	yes	yes
Messages communication structures				
Message modelling	explicit	knowledge based	explicit	views based
Compositional operators				
Ease of composition	good	reasonble	good	yes

Table 3.2: Comparing the I/O automata, Knowledge based (KB), CCS and views based approaches to modelling commit protocols. We have not made clear how a transition system could be generated from a KB model. A process's state in a KB model consists of all the actions that process has taken up to that point and so this could provide method for producing a transition system for model checking.

case of commit protocols at least, a hybrid modelling technique, that captures the desirable abstractions of knowledge based models together with operational approach of CCS and I/O automata might be more suitable. This observation motivates our *views* based model which will be used throughout this thesis for modelling commit protocols.

We first describe the components of our model and then discuss how they can be used to model commit protocols. Finally, by way of a simple example, we model a 2PC protocol.

3.6.1 Processes, local state and views

We model a commit protocol as a system of processes[7]. Processes communicate by means of message passing[8]. Each process belongs to a particular class of processes, corresponding to the role it plays in the protocol. Processes have a set of local state variables, together these variables constitute a process's *internal state.* Unless otherwise stated the internal state variables of each process, within a particular class, are usually initialised with the same values. Each class has a set of rules that determine the behaviour of all processes that belong to that class.

Let p be a process from class P. We denote p's local state variable s as $p.s$. Each variable has a value. If p's variable s has value $\mathbf{x}$ we say $p.s = \mathbf{x}$ holds (at p).

Together with its local state variables a process may have a *view* of the internal state variables of other processes. This view is constructed from information it receives from these processes in the form of messages. We say,

$$@p(q.s = \mathbf{x})$$

holds at process p if the most up-to-date view p has of $q.s$ is $\mathbf{x}$. That is p has received a message from q[9] informing p that q's variable s *had* the value $\mathbf{x}$. It is important to note that if at some point, $q.s$ has value $\mathbf{x}$ then this does *not* imply that, $@p(q.s = \mathbf{x})$ holds. This is because the message reporting q's state change may not have arrived at p. Depending on assumptions about the message passing environment the message may never arrive. Similarly, if $@p(q.s = \mathbf{x})$, this does not imply that q's variable s still has value $\mathbf{x}$, merely that at some point in the past, q's variable s had value $\mathbf{x}$. A process always has an up-to-date view of its own state so, $p.s = \mathbf{x}$ implies $@p(p.s = \mathbf{x})$.

We often wish to express the fact that our view of at least one (some) process(es) within a particular class, P, has (have) reached a particular state $\mathbf{x}$, and so we introduce quantifiers for instance $@p(\exists q \in P,\ q.s = \mathbf{x})$ or $@p(\forall q \in P,\ q.s = \mathbf{x})$.

[7] In the literature the word site or agent is often used instead of process. In our context, each different term describes an entity with its own internal state, thread of control and message passing capabilities.

[8] In fact the particular mechanism used for communication is not relevant. It could equally well be shared memory. The important feature is that the communication may be asynchronous.

[9] p need not receive this message directly from q. In principle the information about q could be received from a third party.

3.6.2 Protocol rules

The behaviour of each process within a particular class is defined by a set of *protocol rules.* Each rule consists of a pre-condition and a post-action. Let $\mathbf{R}$ be a rule for a class of processes P, and p be a process from that class. The pre-condition of $\mathbf{R}$ makes assertions over p's local state variables together with p's *view* of remote process variables. If the pre-condition of rule $\mathbf{R}$ holds at p, we say $\mathbf{R}$ is applicable at p and then $\mathbf{R}$'s post-action may happen changing the local state at p. When p's local state is updated in the post-action of a rule, messages are sent[10] to any process that maintains a view of p's state, enabling them to update their view of p, if and when these messages arrive. For example, suppose we have a system consisting of a process p from class P and a set of processes from class Q. To express the behaviour that p may move to state $\mathbf{y}$ from initial state $\mathbf{x}$, if it believes some process from class Q to be in state $\mathbf{z}$, we write the following rule.

$$\mathbf{R}(p)\ \frac{s = \mathbf{x} \wedge @p(\exists q \in Q,\ q.s = \mathbf{z})}{s := \mathbf{y}}$$

The pre-condition of this rule is a conjunct of two clauses. The first clause state that the process executing the rule, p, has state variable s set to $\mathbf{x}$. The second clause states that p *views* some other process q in the Q class with its internal state variable set to $\mathbf{z}$. If this is the case p can execute the post-action and assign $\mathbf{y}$ to its internal state variable $p.s$.

In general a post-action may contain more than one assignment. It is assumed that all the assignments in the post-action are performed atomically (i.e. as one atom) and are ordered from left to right.

3.6.3 Environment rules

Let a process p have a view of variable s at a remote process q. When q assigns[11] a value $\mathbf{x}$, by executing an assignment in the post-action of a rule, a message is sent to p informing p that $q.s$ was updated.

At the point the message arrives and is delivered[12] at p, p's view of $q.s$ is updated. That is $@p(q.s = \mathbf{x})$ now holds at p. Again we model this behaviour using rules. We call these rules *environment rules* rather than protocol rules. Although the rule can be thought of as taking place at the process where the

[10] Messages are only sent to processes that are interested in that particular update.

[11] Sometimes a message is sent even if the assignment does not change the value of a variable but more normally only changes cause messages to be sent.

[12] The term "delivered" is often used to mean that the incoming message is processed rather than just residing in a buffer on the input device.

message arrives, say p, the pre-condition of the rule often makes reference to state that is not local to p. The post-action of the rule updates p's view of some remote state variable. If process p updates its view of $q.s$ to be $\mathbf{x}$ we write $@p(q.s := \mathbf{x})$ in the post-action of p's environment rule. A typical rule for updating p's view of a process q is as follows.

$$\mathbf{UV1}(p)\ \frac{q \in Q \wedge q.s = \mathbf{y} \wedge @p(q.s \neq \mathbf{y})}{@p(q.s := \mathbf{y})}$$

It is not always possible to update a process's view based solely on the current global state of variables within the system. This is because a process may change a state variable causing a message to be sent and then make a further assignment to the same variable causing a second message to be sent before the first message is delivered. We can still model this situation using rules. Suppose q changes state from $\mathbf{x}$ to $\mathbf{y}$ and then again from $\mathbf{y}$ to $\mathbf{z}$ then the following rule might be appropriate.

$$\mathbf{UV2}(p)\ \frac{q \in Q \wedge (q.s = \mathbf{y} \vee q.s = \mathbf{z}) \wedge @p(q.s \neq \mathbf{y}) \wedge @p(q.s \neq \mathbf{z})}{@p(q.s := \mathbf{y})}$$

In any exucution as the protocol progresses protocol rules are applied, environment rules however need not necessarily be applied. For example, to model message loss we say that rules such as **UV1** might be applied (representing a message arriving) or that they might not (representing the message being lost). If message loss is possible, when the system reaches a configuration (defined later) where only environment rules are applicable, since they need not always be applied, we can assume we might have reached the end of an execution. We can use this technique when modelling commit protocols to show that they might block.

3.6.4 Global state and executions

If we restrict our model to a fixed number of processes (we will investigate techniques to relax this condition later), a *protocol configuration* C can be modelled as a vector of internal states and views, with an entry for each process. We denote this $\langle s_1, \ldots, s_n \rangle$, where each s_i is the internal state and view of a process.

The system takes a step whenever a process within the composition takes a step by executing a protocol rule. Thus if process p_i is in state s_i and rule $\mathbf{R}(p_i)$ happens which we can write as $s_i \xrightarrow{\mathbf{R}(p_i)} s'_i$, then $C \xrightarrow{\mathbf{R}(p_i)} C'$ where $C' = \langle s_1, \ldots, s'_i, \ldots, s_n \rangle$[13]. A process (and therefore the system) can also take a step when its view is updated through the application of an environment rule.

[13]Sometimes we omit the name of the process that the rule is applied to.

An execution ρ, of a protocol therefore is a possibly infinite evolution of system configurations.

$$C_0 \xrightarrow{\mathbf{R_1}} C_1 \xrightarrow{\mathbf{R_2}} \ldots$$

where C_0 is the initial configuration obtained by composing processes in their initial states.

Sometimes the pre-condition of more than one rule might hold for a process allowing more than one rule to be applied at that process. In some cases we allow both rules to be applied leading to two new system states (for example when a participant can choose to vote `yes` or `no`) and in other situations we define a precedence over the rules. If rule **R1** has higher precedence than rule **R2**, **R2** can only be applied once **R1** can no longer be applied.

3.6.5 Modelling centralised two-phase commit

A very simple[14] 2PC protocol consists of a transaction coordinator and some number of participants. When the operations of a transaction have completed the commit protocol is invoked. The coordinator asks each participant to vote on the feasibility of the transaction. If *all* participants vote `yes` then the coordinator can decide to commit the transaction and thus sends a `commit` message to each participant. On receipt of this `commit` message a participant enters its commit state. If *any* participant votes `no` it enters its abort state and sends an `abort` message to its coordinator. When the coordinator receives this message it sends an `abort` message to each remaining participant. On receiving an `abort` message each participant enters its abort state.

To model this protocol we create two classes of processes, a coordinator class C and a participant class P. There is a single instance of the coordinator class c and we let p, q, range over elements of the participant class P. Both types of process have a single state variable s. In the case of the coordinator, c, this variable may take values **i** (initial), **c** (commit) or **a** (abort). In addition to these states participant processes, p, have a further state **w** (wait). In the centralised 2PC we model, all communication is between a participant and the coordinator. For this reason each participant maintains a view of its coordinator's state and the coordinator maintains a view of the state of each of the participants it is coordinating. Table 3.3 summarises this.

We now give the rules for both classes. The participant processes have four rules: the first two rules, **PVY** and **PVN**, allow a participants, p, to vote `yes` and enter their **w** state or vote `no` and enter their **a** state respectively. It should

[14]This is a restricted version which does not model either site failure or messages loss.

	Participant P	Coordinator C
State Values	**i, w, a, c**	**i, a, c**
Number	Many processes	Single Process
Views	View of coordinator	Views of participants

Table 3.3: Modelling centralised two-phase commit

be noted that this gives participants autonomy over whether to force abort or try to commit.

$$\mathbf{PVY}(p)\ \frac{s = \mathbf{i}}{s := \mathbf{w}} \quad \mathbf{PVN}(p)\ \frac{s = \mathbf{i}}{s := \mathbf{a}}$$

The structure of these rules is the pre-condition post-action style. In **PVY** if any process p is in a state such that $p.s = \mathbf{i}$ then the rule can be applied. This means that the post-action is applied to p changing $p.s$ to $\mathbf{w}$.

The next two rules, **PC** and **PA**, allow participants to enter either the **c** or **a** state if they view the coordinator in the **c** or **a** state respectively.

$$\mathbf{PC}(p)\ \frac{s = \mathbf{w} \wedge @p(c.s = \mathbf{c})}{s := \mathbf{c}} \quad \mathbf{PA}(p)\ \frac{s \in \{\mathbf{w}, \mathbf{i}\} \wedge @p(c.s = \mathbf{a})}{s := \mathbf{a}}$$

The coordinator class C has only two rules. The first allows it to enter the **c** state if its view of *all* of its participants shows that they have all entered their **w** state (i.e. they have all voted **yes**). The second rule allows a coordinator to enter the **a** state if its view of *any* of its participants shows that one has entered the **a** state (i.e. one of the participants has voted **no**). In this particular model of 2PC we do not allow the coordinator to vote.

$$\mathbf{CC}(c)\ \frac{s = \mathbf{i} \wedge @c(\forall p \in P,\, p.s = \mathbf{w})}{s := \mathbf{c}} \quad \mathbf{CA}(c)\ \frac{s = \mathbf{i} \wedge @c(\exists p \in P,\, p.s = \mathbf{a})}{s := \mathbf{a}}$$

In this simple example we will assume message delivery is reliable, messages are never lost and sites never crash. Later we will see how to model protocols with arbitrary numbers of participants but for now let us assume we have a single coordinator and two participants. The environment rules to update views therefore are very simple. If the view of a process's state is out of date, that is, the actual value at the remote process has changed then the view can be updated. We denote updating a view of $q.s$ at p to value $\mathbf{x}$, $@p(q.s := \mathbf{x})$.

The environment rules for the coordinator and participant processes respectively are given below.

$$\mathbf{CUV}(c)\ \frac{p.s = x \wedge @c(p.s = \mathbf{i})}{@c(p.s := x)} \quad x \in \{\mathbf{w}, \mathbf{a}\}$$

$$\mathbf{PUV}(p)\ \frac{c.s = x \wedge @p(c.s = \mathbf{i})}{@p(c.s := x)} \quad x \in \{\mathbf{c}, \mathbf{a}\}$$

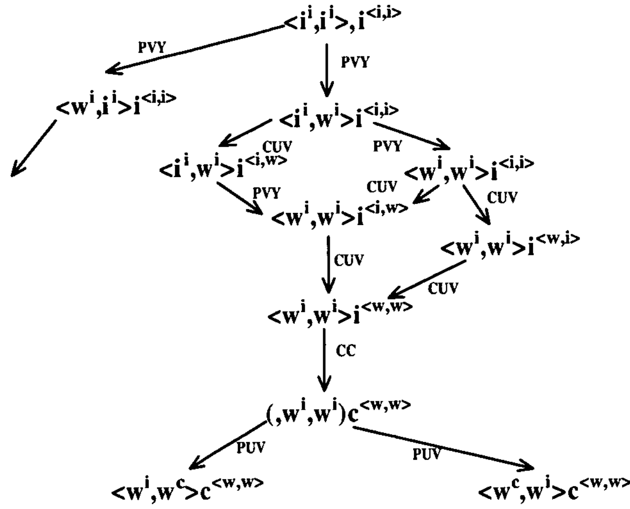

Figure 3.4: Execution fragment of a simple 2PC protocol

In some rules in order to simplify their presentation, we let x range over the values that a variable might take. For example the rule $\mathbf{PUV}(p)$ is shorthand for the two very similar rules

$$\frac{c.s = \mathbf{c} \wedge @p(c.s = \mathbf{i})}{@p(c.s := \mathbf{c})} \quad \frac{c.s = \mathbf{a} \wedge @p(c.s = \mathbf{i})}{@p(c.s := \mathbf{a})}$$

Let us represent the system state as a triple where the first entry is the state of the coordinator and the second and third entries are the states of the two participants. We write the views as superscripts. So $\mathbf{i}^{\langle \mathbf{w},\mathbf{i} \rangle}$ denotes the state of a coordinator with $c.s = \mathbf{i}$ and with views such that $@c(p_1 = \mathbf{w} \wedge p_2 = \mathbf{i})$ for the participants p_1 and p_2. Figure 3.4 shows some of the possible executions of our simple protocol. In the diagram we drop the index p_i on a rule since it can be ascertained from the transition.

Given our rules it is possible to generate a transition system that represents all the possible executions of the protocol. Using the transition system we can verify properties of our system. One such property is "if the coordinator knows (has views) that both participants are in **w** then eventually those participants commit by reaching **c**". For the simple protocol above this property does hold. We will return to the subject of verifying properties at length when we introduce games-based model checking in chapter 5.

3.7 Summary

We have discussed some general modelling techniques that have been used for modelling commit protocols and the environments in which they execute. A re-

quirement emerged from our survey of three general modelling techniques that seemed to suggest that a technique that captured the desirable operational features of CCS, the declarative style of knowledge based reasoning, and the precondition post-action specification technique of I/O automata would be most appropriate. This motivated the development of our views based model. Using an example we showed, how this technique can be used to model a simple 2PC protocol.

Chapter 4

A More Committed Three Phase Commit Protocol

4.1 Introduction

To recapitulate in the centralised 2PC protocol, a coordinator collects votes on whether or not participants can commit a transaction and broadcasts whether or not there is unanimity for commit. Problems arise with 2PC when site and/or network failures occur. Some working sites may become "blocked", they want to commit the transaction but they are unable to proceed, neither commit nor abort, until an external failure has been repaired. 3PC was developed which provides some protection against blocking under a restricted failure model. Skeen [71] recognised that 3PC could only protect against blocking for this restricted class of failure and developed a *quorum-based three-phase commit* protocol which we shall call Q3PC. If a network failure occurs preventing some processes from communicating but still allowing a quorum of processes to communicate, Q3PC ensures this quorum will not block.

Kiedar and Dolev [41] describe a *cascading* network failure. Such a failure occurs when there are several successive partial network failures, and possibly some repairs too, but the network is not totally failure free at any time during the failure period. There may be times of calm where some progress is made but more disruption soon follows. In Q3PC, it is possible that, after cascading network failures, a quorum of sites may form, yet those sites remain blocked. However the Enhanced 3PC Protocol (E3PC) [41] extends Q3PC in a way that ensures that a quorum never blocks.

We now proceed to use the views based modelling technique to describe, and reason about the behaviour of, these protocols. We first extend the simple 2PC protocol of the last chapter to include communication failure. We show in this

enriched model that 2PC might block. By adding some extra rules we show how 2PC can provide protection against blocking but for some communication failures it still cannot protect against blocking completely. We then go on to briefly model the basic 3PC by way of introduction to modelling complex quorum based 3PC protocols that will be central to this chapter. By enhancing 3PC with a recovery protocol (by the addition of more protocol rules) we can derive Skeen's Q3PC from 3PC which is the basis for our investigation into quorum based commit protocols. By enriching the internal state held at the participants of Q3PC we can derive a model of Kiedar and Dolev's E3PC [41] protocol that enhances Q3PC by protecting against cascading network failure. Once more by extending the protocol rules of E3PC we then derive an improved version of E3PC called X3PC in which a coordinator can use the distributed knowledge within a quorum to detect situations where it is possible to make progress towards committing a transaction. Like E3PC, in X3PC a connected quorum of sites never blocks, but X3PC will decide commit more often than E3PC and in no more attempts[1]. We will see that E3PC, Q3PC and X3PC have similar message-passing behaviour, but differ in the amount of state information that is exchanged in a message.

A pattern is starting to emerge. As the environment in which the processes interact becomes richer, with for example the possibility of communication failure the local states of the processes involved and the number and complexity of the protocol rules increases. With this increased richness comes the ability to reason about more complicated behaviour such as blocking. Figure 4.1 depicts this.

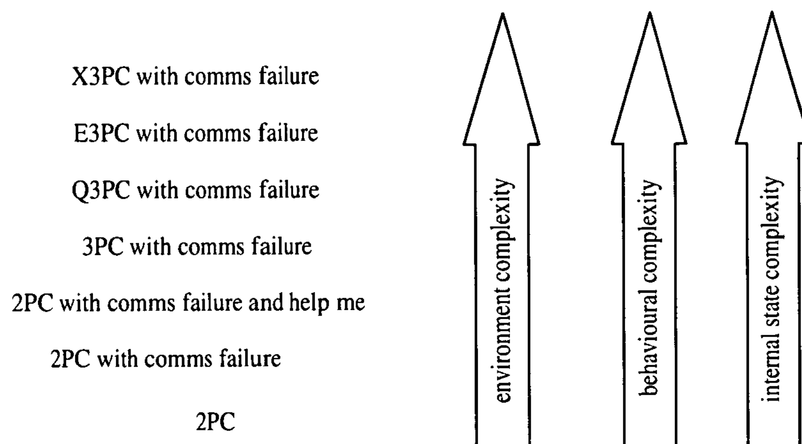

Figure 4.1: The increased complexity of a protocols environment is mirrored by the increased complexity of rules and the ability to reason about more complicated behaviour.

[1]An attempt is started when a coordinator updates it's last attempt counter.

4.2 Adding failure to 2PC

In the models of commit protocols we study we model communication failure by allowing processes to become disconnected from one another and partitioned into groups or components. All messages in transit at the time of the failure are assumed to be lost and it is assumed that processes can no longer communicate unless they are in the same component.

We model a communication network, which may fail and divide the set of all processes P (now and in the following we use the set P for all processes including the coordinator) arbitrarily, using a partition Par so that the following holds.

- $Par(p) = \{q \in P \mid q \text{ can communicate with } p\}$
- $\forall p \in P,\ p \in Par(p)$
- $\forall p \in P,$ if $p \in Par(q)$ then $Par(p) = Par(q)$

It follows from this definition that if $p \notin Par(q)$ then $Par(p) \cap Par(q) = \emptyset$. Sometimes we write Par as a set of disjoint subsets or components of P whose union is P. For example, let $P = \{p_1, p_2, p_3\}$ and $Par(p_1) = \{p_1, p_2\}$, $Par(p_2) = \{p_1, p_2\}$ and $Par(p_3) = \{p_3\}$, then we write $Par = \{\{p_1, p_2\}, \{p_3\}\}$. Using this representation we write $X \in Par$ to mean X is one of the components in the partition Par, thus p's component is $Par(p)$. In our example $\{p_1, p_2\} \in Par$. To model communication failure and repair Par may change $Par \rightarrow Par'$. Extending our example, suppose p_2 loses communication with p_1 and gains communication with p_3, can write this as follows.

$$\frac{}{\{\{p_1, p_2\}, \{p_3\}\} \rightarrow \{\{p_1\}, \{p_2, p_3\}\}}$$

or more generally as the environment rule:

$$\textbf{NET}\ \frac{}{Par \rightarrow Par'}$$

Since the pre-condition of **NET** is empty Par may change at any time, allowing communication failure at any point in a protocol execution. Initially $Par = \{P\}$.

To enhance the simple 2PC protocol of section 3.6.5 we need only extend the environment rules **CUV** and **PUV** that update the views of participants and coordinators. We simply add the extra clause $p \in Par(c)$ to both pre-conditions to reflect the fact that a coordinator (participant) can only update its view of

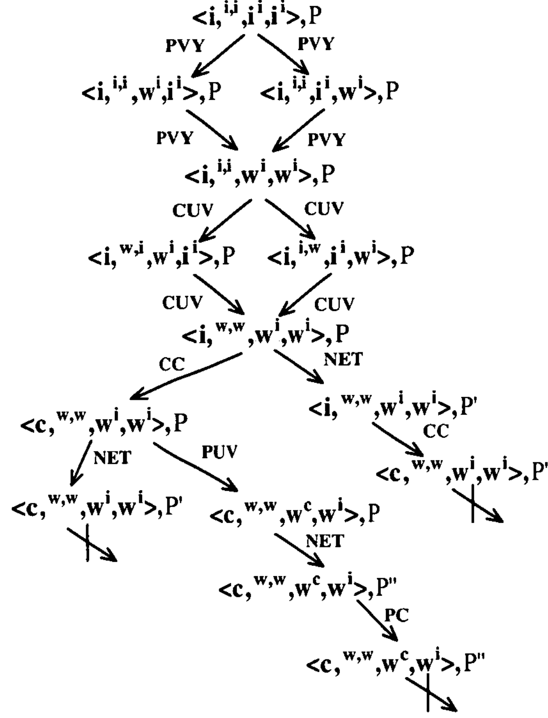

Figure 4.2: Part of the execution diagram of 2PC in the presence of network partitioning. **NET** events in these executions result in the protocol blocking. In the diagram $P = \{\{c, p_1, p_2\}\}$, $P' = \{\{c\}, \{p_1, p_2\}\}$ and $P'' = \{\{c, p_1\}, \{p_2\}\}$.

a participant (coordinator) if it is in the same component. Our rules are thus changed as follows.

$$\mathbf{CUV}(c)\ \frac{p.s = x \wedge @c(p.s = \mathbf{i}) \wedge p \in Par(c)}{@c(p.s := x)}\quad x \in \{\mathbf{w}, \mathbf{a}\}$$

$$\mathbf{PUV}(p)\ \frac{c.s = x \wedge @p(c.s = \mathbf{i}) \wedge p \in Par(c)}{@p(c.s := x)}\quad x \in \{\mathbf{c}, \mathbf{a}\}$$

4.2.1 2PC blocks

With the addition of network partitioning to the simple 2PC we see that our simple protocol might now block. Consider the following execution involving two participants and a coordinator. We use a similar notation for a configuration as in figure 3.4 by appending the current value of *Par* to form a configuration. Figure 4.2 shows several executions of 2PC. Network events in these executions result in the protocol blocking.

The leftmost execution can be written as follows.

$$\langle \mathbf{i}^{\mathbf{i},\mathbf{i}}, \mathbf{i}^{\mathbf{i}}, \mathbf{i}^{\mathbf{i}} \rangle \{\{c, p_1, p_2\}\} \stackrel{\mathbf{PVY}^2}{\longrightarrow} \langle \mathbf{i}^{\mathbf{i},\mathbf{i}}, \mathbf{w}^{\mathbf{i}}, \mathbf{w}^{\mathbf{i}} \rangle \{\{c, p_1, p_2\}\} \stackrel{\mathbf{CUY}^2}{\longrightarrow}$$
$$\langle \mathbf{i}^{\mathbf{w},\mathbf{w}}, \mathbf{w}^{\mathbf{i}}, \mathbf{w}^{\mathbf{i}} \rangle \{\{c, p_1, p_2\}\} \stackrel{\mathbf{CC}}{\longrightarrow} \langle \mathbf{c}^{\mathbf{w},\mathbf{w}}, \mathbf{w}^{\mathbf{i}}, \mathbf{w}^{\mathbf{i}} \rangle \{\{c, p_1, p_2\}\} \stackrel{\mathbf{NET}}{\longrightarrow}$$
$$\langle \mathbf{c}^{\mathbf{w},\mathbf{w}}, \mathbf{w}^{\mathbf{i}}, \mathbf{w}^{\mathbf{i}} \rangle \{\{c\}, \{p_1, p_2\}\} \not\rightarrow$$

The only rule applicable to the last configuration, in this execution is the environment rule **NET**. In other words no process can make progress towards reaching a commit or abort decision until the network is repaired and so they are deemed blocked.

4.2.2 Help-Me messages

In the last example, the leftmost branch in figure 4.2 the protocol blocked even though there was enough knowledge within the participants' component to commit because both participants voted `yes`. Unfortunately, for the simple 2PC described participants rely on their coordinator for communication. Once the coordinator is isolated in a different component all communications break down. One solution to this problem is that once a participant discovers that it has become disconnected from its coordinator it can send out a `help-me` message to other participants. If another participant in the same partition receives such a message, and has itself decided, it can help the undecided participant reach a decision. Furthermore, if a participant receives `help-me` messages from all other participants it is safe to deduce that all other sites have voted `yes` and thus itself commit. Many implementations of 2PC include `help-me` messages as a way of reducing blocking.

We can model `help-me` messages using views. In **RHQ** below, when a participant detects that it has become isolated from its coordinator it enters a help-me state, **h**. In **PH** any decided participant that views another participant in state **h** provides help by moving to either $\mathbf{c}_h$ or $\mathbf{a}_h$ depending on its decision.

$$\mathbf{RHQ}(p)\ \frac{s = \mathbf{w} \wedge c \notin Par(p)}{s := \mathbf{h}} \quad \mathbf{PH}(p)\ \frac{s = x \wedge @p(\exists q \in P,\, q.s = \mathbf{h})}{s := x_h} \quad x \in \{\mathbf{c}, \mathbf{a}\}$$

If a participant views all other participants in the help-me state, **h**, then it can safely deduce that all sites voted `yes` and move to **c**. This is modelled by the rule **PDC** below. Similarly, if it views a participant as having helped, states $\mathbf{c}_h$ or $\mathbf{a}_h$, it can commit or abort accordingly. This is modelled by the rule **PRH** below.

$$\mathbf{PDC}(p)\ \frac{s = \mathbf{h} \wedge @p(\forall q \in P,\, q.s = \mathbf{h})}{s := \mathbf{c}_h}$$

$$\mathbf{PRH}(p)\ \frac{s = \mathbf{h} \wedge @p(\exists q \in P,\, q.s = x_h)}{s := x} \quad x \in \{\mathbf{c}, \mathbf{a}\}$$

Once again we must provide environment rules for propagating `help-me` messages within a component which we call **PHUV** and help responses which we call **PRUV**.

$$\mathbf{PRUV}(p)\ \frac{s = \mathbf{h} \wedge q.s = x_h \wedge @p(q.s \neq x_h) \wedge p \in Par(q)}{@p(q.s := x_h)}\quad x \in \{\mathbf{c}, \mathbf{a}\}$$

$$\mathbf{PHUV}(p)\ \frac{s = x \wedge q.s = \mathbf{h} \wedge @p(q.s \neq \mathbf{h}) \wedge p \in Par(q)}{@p(q.s := \mathbf{h})}\quad x \in \{\mathbf{c}, \mathbf{a}, \mathbf{h}\}$$

Consider again the execution where 2PC blocked in section 4.2.1. The last system configuration in the execution was $(\mathbf{c}^{\mathbf{w},\mathbf{w}}, \mathbf{w}^{\mathbf{i}}, \mathbf{w}^{\mathbf{i}})\{\{c\}, \{p_1, p_2\}\}$. Using our help-me rules we see that **RHQ** can be applied by each participant to request help. After this each participant can apply environment rule **PHUV** to update its view of its cohort to be **h**. Finally **PDC** can be applied at both participants because each participant can deduce that, because all participants are in state **h**, they must all have voted `yes`, and so it is safe to commit.

In this enriched model a participant must maintain a view, not only of its coordinator, but also of the other participants. We therefore extend the participant state so that a participant's view is a pair of states (as it is in the coordinator) the first denotes its view of its coordinator and the second its view of the other participant. Using this notation we can extend the previous execution as follows.

$$\begin{array}{l}
\langle \mathbf{c}^{\mathbf{w},\mathbf{w}}, \mathbf{w}^{\mathbf{i},\mathbf{i}}, \mathbf{w}^{\mathbf{i},\mathbf{i}}\rangle\{\{c, p_1, p_2\}\} \stackrel{\mathbf{PUV}(p_1)}{\longrightarrow} \\
\langle \mathbf{c}^{\mathbf{w},\mathbf{w}}, \mathbf{w}^{\mathbf{c},\mathbf{i}}, \mathbf{w}^{\mathbf{i},\mathbf{i}}\rangle\{\{c, p_1, p_2\}\} \stackrel{\mathbf{PC}(p_1)}{\longrightarrow} \\
\langle \mathbf{c}^{\mathbf{w},\mathbf{w}}, \mathbf{c}^{\mathbf{c},\mathbf{i}}, \mathbf{w}^{\mathbf{i},\mathbf{i}}\rangle\{\{c, p_1, p_2\}\} \stackrel{\mathbf{NET}}{\longrightarrow} \\
\langle \mathbf{c}^{\mathbf{w},\mathbf{w}}, \mathbf{c}^{\mathbf{c},\mathbf{i}}, \mathbf{w}^{\mathbf{i},\mathbf{i}}\rangle\{\{c, p_1\}, \{p_2\}\} \not\rightarrow
\end{array}$$

And so we see that in this example blocking is prevented. Unfortunately, although `help-me` messages relieve the problem they do not entirely eliminate it. For example suppose that the NET event results in the more disruptive partition $\{\{c, p_1\}, \{p_2\}\}$, after p_1 has committed but before p_2 receives a commit message then, although p_2 can issue a `help-me` message this will not help it reach a decision until the network is repaired.

$$\langle \mathbf{c}^{\mathbf{w},\mathbf{w}}, \mathbf{w}^{\mathbf{i},\mathbf{i}}, \mathbf{w}^{\mathbf{i},\mathbf{i}}\rangle\{\{c\}, \{p_1\}, \{p_2\}\} \stackrel{\mathbf{RHQ}^2}{\longrightarrow} \langle \mathbf{c}^{\mathbf{w},\mathbf{w}}, \mathbf{h}^{\mathbf{i},\mathbf{i}}, \mathbf{h}^{\mathbf{i},\mathbf{i}}\rangle\{\{c\}, \{p_1\}, \{p_2\}\} \not\rightarrow$$

Finally, for completeness sake, we should note that if a participant becomes blocked because it is isolated from its coordinator, and then a **NET** event happens, rejoining that participant and its coordinator, then the coordinator should interpret a view of sate **h** as state **w**, similarly $\mathbf{c}_h$ and $\mathbf{a}_h$ states should be interpreted as **c** and **a** states. This requires some small changes to our existing rules but we omit the details.

4.3 Modelling 3PC

As we have seen in section 2.2 by adding a buffer state (**pc**) to the simple 2PC (without help-me rules) we can derive, 3PC, a protocol that is slightly more resilient to blocking than the basic 2PC. We can simply describe this protocol by introducing the new state **pc**, making changes to the rules **PC**, **CC** and introducing two new rules **PPC** and **CPC** as follows.

$$\mathbf{PPC}(p)\ \frac{s = \mathbf{w} \wedge @p(c.s = \mathbf{pc})}{s := \mathbf{pc}} \qquad \mathbf{CPC}(c)\ \frac{s = \mathbf{i} \wedge @c(\forall p \in P,\ p.s = \mathbf{w})}{s := \mathbf{pc}}$$

$$\mathbf{PC}(p)\ \frac{s = \mathbf{pc} \wedge @p(c.s = \mathbf{c})}{s := \mathbf{c}} \qquad \mathbf{CC}(c)\ \frac{s = \mathbf{pc} \wedge @c(\forall p \in P,\ p.s = \mathbf{pc})}{s := \mathbf{c}}$$

The addition of this state does provide some tolerance to blocking in the rather unrealistic case where failure is restricted to single site failure only. By adding timeout and crash actions to the rules of 3PC above, it is possible to model this particular environment and show that for this restricted case of failure 3PC is non-blocking. Since in practice it is impossible to restrict the types of failures that might occur, we will not proceed in this direction. Instead, we will investigate more realistic quorum based 3PC protocols that provide blocking tolerance for a more general class of failure.

4.4 Modelling quorum based commit protocols

In this section we provide a views based model for two existing quorum based three phase commit protocols namely Q3PC and E3PC. Since E3PC is an extension of Q3PC they share many of the same protocol rules. Both of these protocols have a similar structure. In their so called initial phase, before any failures, each protocol carries out a basic 3PC[2]. If a failure occurs then the protocols enter their termination phase. It is in their termination phase that Q3PC and E3PC differ. During this phase each protocol, where possible, attempts to reach a commit (abort) decision, by first moving processes to pre-commit (pre-abort) and then to commit (abort). We first model E3PC and then show how to simplify its protocol rules to derive Q3PC.

4.4.1 Views and process state

Unlike the commit protocol models we considered earlier all processes in our model are from a single class of processes P. In the protocols we consider any

[2]If no failures occur during the initial phase the executions of Q3PC and E3PC are identical to 3PC.

process p can assume the role of a coordinator. The identity of a processes's coordinator is stored in the internal state variable $p.c$. Thus if $p.c = q$ then q is a coordinating participant. Another important internal state variable of processes is $p.r$, which if true means p has detected a communication failure and is executing its termination phase.

Fig. 4.1 describes the state variables at each process $p \in P$. As before we include the internal state variable s, e.g. $p.s = \mathbf{c}$ means p is committed. In the case of E3PC we add the two counters $p.le$ and $p.la$ of [41]: $p.le = m$ means that p's last-elected counter is m. We will discuss how these counters are used to derive E3PC from Q3PC in detail later.

Later when we further extend E3PC we will employ an internal state object $p.h$ which records a history of attempts that process made to move to the pre-abort (**pa**) state during the termination phase: its details are discussed later. As in our previous example, processes have views about the states of other processes. For p, $@p(q.s = \mathbf{pc})$ means that p knows q was at some time[3] in **pc**. For p, $@p(q.la = m)$ means that p knows q's last attempt counter was at some time equal to m.

We say $@p(q.s \approx \mathbf{x})$ if $@p(q.s = \mathbf{x}) \wedge @p(q.le = p.le) \wedge @p(q.c = p.c)$ holds. Similarly we write $@p(q.s \equiv \mathbf{x})$ holds at p if $@p(q.s \approx \mathbf{x}) \wedge @p(q.la = p.la)$ holds. We use the notation $=$, $\approx$ and $\equiv$ to denote stronger and stronger versions of equality. $@p(q.s \approx \mathbf{x})$ means not only $@p(q.s = \mathbf{x})$ but also that $@p(q.le = p.le) \wedge @p(q.c = p.c)$, i.e. the state was reached during the last election and p believes q has the same coordinator. Stronger still $@p(q.s \equiv \mathbf{x})$ adds the further condition that p believes q is in the same attempt as p.

A new[4] feature of this model allows a view to be updated by a process locally in the post-action of a protocol rule in the same way as local state is updated. Updating a view allows a process to change its belief of what a remote processes's state might be. When views are updated the change does not produce messages; this disallows views of views.

When modelling quorum based commit protocols we will again adopt the partition based model of communication failure. This means we include in a configuration the partition Par and the environment rule **NET** that can disrupt the network at any time. All processes affected by a network disruption detect it and new coordinators are elected [29] in each new component of the new partition. During the election a coordinator in a group, X, can compute the maximum

[3] p may possibly still be in state **pc**.

[4] Previously, views were updated only in environment rules.

$p.s \in \{\mathbf{pa}, \mathbf{pc}, \mathbf{i}, \mathbf{w}, \mathbf{a}, \mathbf{c}\}$	state of a process
$p.c \in P$	p's coordinator
$p.le$	p's last elected counter
$p.la$	p's last attempt counter
$p.r \in \{\texttt{tt}, \texttt{ff}\}$	p has entered the recovery phase
$p.f \in \{\texttt{tt}, \texttt{ff}\}$	coordinator is collecting participant's state
$p.h$	history of attempts to move to **pa**

Table 4.1: **a**, **c**, represent the abort and commit decision states. **i** is the initial state before a process has voted, **w** is entered after a **yes** vote. **pc** and **pa** are entered during attempts to commit and abort respectively. During termination a proces must record when a coordinator is collecting participant states and behave accordingly. This is reflected in the value of the variable $p.r$.

last elected counter $me(X) = max_{q \in X}\{q.le\}$[5], within the component. When a participant adopts a new coordinator it updates its last elected counter to be $me + 1$ and begins the termination phase of the protocol. We model this change with **NET**(c) as follows where $me = max_{q \in X}\{q.le\}$, c is a process chosen from X during an election.

$$\frac{Par \rightarrow Par' \wedge X \in Par' \wedge X \notin Par}{\text{UPDATE}(c, X)}$$

where

$$\begin{aligned}
\text{UPDATE}(c, X) \stackrel{def}{=} \; & \textstyle\bigwedge_{q \in X}(q.c := c \wedge q.r := \texttt{tt} \wedge q.le := me + 1 \wedge & (1)\\
& @c(q.s := q.s) \wedge @c(q.le := q.le) \wedge @c(q.la := q.la) & (2)\\
& @q(c.f := \texttt{tt})) & (3)
\end{aligned}$$

The UPDATE macro is a little complicated and so requires some explanation. For each process q in X there are three parts to the update. The first part, line (1), sets process q's coordinator variable $q.c$ to be c, sets $q.r$ to `tt` and also sets $q.le$ to be $me + 1$. The next part, line (2), sets the elected coordinator, c's view of $q.s$, $q.le$ and $q.la$ to be the actual values of $q.s$, $q.le$ and $q.la$. Finally, in line (3), q's view of $c.f$ is set to be `tt`.

We can assume the post-action is atomic because if it is interrupted by another network event, $Par' \rightarrow Par''$, the leadership election can be restarted [29] and any partial changes ignored.

It should be noted that this is the first example of a rule that we have seen where, in the post-action, a view is updated at more than one process, in this case at c and at p. Clearly if these non-local actions are to be assumed to be atomic

[5] A coordinator need not store $me(X)$ as part of its state it need only calculate it during an election.

we require further justification. Although we will not elaborate on the details it is possible to achieve these updates atomically during a leadership election. An example application of the **NET** rule can be seen in figure 4.3 where it is applied twice to the fourth state in the diagram.

4.4.2 Updating views

Views are updated in a way which is similar to, but slightly different from, the way they were in the simple 2PC example presented earlier in section 3.6.3. If a process p has a view that is out of date with respect to the current state of another process *within the same component* of its partition (i.e. within $Par(p)$), that view may be updated. Notice this models the situation where messages within a component eventually arrive, provided the component does not change, but still allows message loss if a network event occurs re-partitioning the system and changing the component. The rules for updating the view of a coordinator and a participant process are given below. Notice a participant p does not update its view of a coordinator c if it views $c.f$ to hold, this models the situation where a coordinator is collecting state after a network event during an election.

$$\mathbf{UCV}(p)\ \frac{c = p \wedge q \in Par(p) \wedge q.\overrightarrow{z} = \overrightarrow{v} \wedge @p(q.\overrightarrow{z} \neq \overrightarrow{v})}{@p(q.\overrightarrow{z} := \overrightarrow{v})}\quad \overrightarrow{z} = \langle s, la, le, c\rangle$$

$$\mathbf{UPV}(p)\ \frac{c \neq p \wedge @p(c.f = \mathtt{ff}) \wedge c.\overrightarrow{z} = \overrightarrow{v} \wedge @p(c.\overrightarrow{z} \neq \overrightarrow{v})}{@p(c.\overrightarrow{z} := \overrightarrow{v})}\quad \overrightarrow{z} = \langle s, la, le, f, c\rangle$$

4.4.3 Protocol rules for E3PC

Initially all processes p start in the **i** state with $\neg p.f$ and their last attempt and last elected counters set to 0 and 1 respectively. All processes are connected and there exists a process which is the coordinator for all processes. More formally

$$(\forall p \in P,\ p.s = \mathbf{i}\ \wedge\ p.la = 0\ \wedge\ p.le = 1\ \wedge \neg p.f\ \wedge\ \neg p.r\ \wedge\ Par(p) = P)\wedge$$
$$(\exists q \in P,\ \forall p \in P,\ p.c = q)$$

The protocol rules are divided into four groups. One group for the initial phase where $\neg r$ holds at all processes and another for the termination phase where r holds. The rules are further divided between participant rules, where $c = p$ and coordinator rules, where $p \neq c$. Rules for coordinator processes begin with **C** and those for participants begin with **P**. Rules for the initial phase end with **I** and for the termination phase end with **T**.

A step is applicable to a process if the pre-condition for that step is satisfied at that process. The initial phase of E3PC is described by the following rules.

The first group of rules describe the behaviour of participants not executing their termination protocol. We therefore omit $\neg r \wedge (c \neq p)$ from the pre-conditions for the sake of brevity.

$$\textbf{PVYI}(p)\ \frac{s = \mathbf{i}}{s := \mathbf{w}} \qquad \textbf{PVNI}(p)\ \frac{s = \mathbf{i}}{s := \mathbf{a}}$$

$$\textbf{PPCI}(p)\ \frac{s \neq \mathbf{pc} \wedge @p(c.s = \mathbf{pc})}{s := \mathbf{pc}}$$

$$\textbf{PCI}(p)\ \frac{s \neq \mathbf{c} \wedge @p(c.s = \mathbf{c})}{s := \mathbf{c}} \qquad \textbf{PAI}(p)\ \frac{s \neq \mathbf{a} \wedge @p(c.s = \mathbf{a})}{s := \mathbf{a}}$$

In the rules **PVYI** and **PVNI** participants vote **yes** and **no** respectively moving to either wait (**w**) or abort (**a**). The rule **PPCI** allows a participant to enter pre-commit (**pc**) and the rules **PCI** and **PAI** allow a participants to decide commit (**c**) or abort (**a**). We now give the rules for coordinators not executing their termination protocol so this time we omit $\neg r \wedge (c = p)$ from each pre-condition.

$$\textbf{CPCI}(p)\ \frac{s = \mathbf{i} \wedge @p(\forall q \neq p \in P,\ q.s = \mathbf{w})}{s := \mathbf{pc} \wedge la := le} \qquad \textbf{CVNI}(p)\ \frac{s = \mathbf{i}}{s := \mathbf{a}}$$

$$\textbf{CCI}(p)\ \frac{s = \mathbf{pc} \wedge @p(\forall q \neq p \in P,\ q.s = \mathbf{pc})}{s := \mathbf{c}}$$

$$\textbf{CAI}(p)\ \frac{s = \mathbf{i} \wedge @p(\exists q \neq p \in P,\ q.s = \mathbf{a})}{s := \mathbf{a}}$$

CVNI allows a coordinator to vote no from its initial state **i**, **CPCI** moves a coordinator to **pc** while **CCI** and **CAI** allow a coordinator to move to **c** or **a** respectively.

In some cases more that one pre-condition may be true for a process. For example suppose a coordinator, p, is in state **i** with a view that all other processes have voted **yes**, or more formally $@p(\forall q \neq p \in P,\ q.s = \mathbf{w})$ holds, either **CNVI** or **CPCI** might happen. This reflects the fact that a coordinator (because it is also a participant in its own right) may itself abort a transaction even though all other processes voted **yes**. We do not, in this case but will sometimes, restrict this type of choice by providing a precedence on the rules.

We also make use of the predicate *isMaxAttemptCommittable* (defined in [41] which we rename IMAC(X)) over $X \subseteq P$. IMAC(X) is true at coordinator p where $ma = max\{m \mid @p(\exists q \in X,\ q.la \approx m)\}$ if

$$\forall q \in X,\ @p(q.la \approx ma) \Rightarrow @p(q.s \approx \mathbf{pc})$$

I.e., p believes no member of X with the greatest last attempt counter, within X, is in a state other than **pc**. We now present the rules of the termination

protocol. We first give rules for participants this time omitting $r \wedge (c \neq p)$ from each pre-condition.

$$\mathbf{PPCT}(p)\ \frac{la \neq le \wedge @p(c.s = \mathbf{pc} \wedge \neg c.f)}{la := le \wedge s := \mathbf{pc}}$$

$$\mathbf{PPAT}(p)\ \frac{la \neq le \wedge @p(c.s = \mathbf{pa} \wedge \neg c.f)}{la := le \wedge s := \mathbf{pa}}$$

$$\mathbf{PCT}(p)\ \frac{s \neq \mathbf{c} \wedge @p(c.s = \mathbf{c})}{s := \mathbf{c}} \quad \mathbf{PAT}(p)\ \frac{s \neq \mathbf{a} \wedge @p(c.s = \mathbf{a})}{s := \mathbf{a}}$$

In **PPCT** (**PPAT**) a participant moves to **pc** (**pa**). In **PCT** (**PAT**) a participant commits (aborts) when it views the coordinator as having committed (aborted). Finally we can give rules for the coordinator's behaviour during the termination phase of the protocol, this time we omit $r \wedge (c = p)$ from each precondition. In **CPCT** we make reference to the quorum predicate $Q(Par(p))$, this was defined in section 2.3. One can think of a quorum as a majority for the purposes of this chapter.

$$\mathbf{CC1T}(p)\ \frac{f \wedge @p(\exists q \in Par(p),\ q.s = \mathbf{c})}{s := \mathbf{c} \wedge f := \mathtt{ff}}$$

$$\mathbf{CA1T}(p)\ \frac{f \wedge @p(\exists q \in Par(p),\ q.s = \mathbf{a})}{s := \mathbf{a} \wedge f := \mathtt{ff}}$$

$$\mathbf{CPCT}(p)\ \frac{f \wedge Q(Par(p)) \wedge \mathrm{IMAC}(Par(p))}{la := le \wedge s := \mathbf{pc} \wedge f := \mathtt{ff}}$$

$$\mathbf{CPAT}(p)\ \frac{f \wedge Q(Par(p)) \wedge \neg\mathrm{IMAC}(Par(p))}{la := le \wedge s := \mathbf{pa} \wedge f := \mathtt{ff}}$$

$$\mathbf{CC2T}(p)\ \frac{\neg f \wedge \exists X \subseteq Par(p),\ (Q(X) \wedge @p(\forall q \in X,\ q.s \equiv \mathbf{pc}))}{s := \mathbf{c}}$$

$$\mathbf{CA2T}(p)\ \frac{\neg f \wedge \exists X \subseteq Par(p),\ (Q(X) \wedge @p(\forall q \in X,\ q.s \equiv \mathbf{pa}))}{s := \mathbf{a}}$$

If a process exists in the coordinator's component in a **c** or **a** state, then the coordinator propagates the value using rules **CC1T** and **CA1T**, which have highest precedence. If a quorum of **pc** states exist and IMAC holds, the rule **CPCT** is used to move participants to the **pc** state. Likewise, **CPAR** is used to advance processes to the **pa** state. If enough processes are in **pc** the coordinator decides commit with **CC2T** or if enough are in **pa** it moves to status **a** with rule **CA2T**.

4.4.4 Q3PC: Skeen's Quorum-based 3PC

Using this notation for modelling E3PC as a starting point, we can obtain a model of Q3PC by changing the pre-condition of **CPCT** and **CPAT** as follows.

$\mathbf{CPCT}'(p)\ f\ \wedge\ @p(\exists q,\ q.s = \mathbf{pc}) \wedge Q(\{r \in Par(p)\ \mid\ @p(r.s = \mathbf{pc} \vee r.s = \mathbf{w})\})$

$\mathbf{CPAT}'(p)\ f\ \wedge\ Q(\{r \in Par(p)\ \mid\ @p(r.s = \mathbf{pa} \vee r.s = \mathbf{w})\})$

The post-actions of the rules remain the same. Using these modified pre-conditions, if cascading network partitioning is possible this introduces the possibility of blocking. We will examine this further in the example in section 4.5.

4.4.5 Configurations and executions

As before a *configuration* C is a collection of processes with their internal state and views together with Par.

$$C = \langle s_1, \ldots, s_n \rangle, Par$$

An *execution* of a protocol is a sequence of configurations $C_1, \ldots, C_m, \ldots$ where C_{i+1} is derived from C_i by a protocol step, for example applying the rule **CA2T**, a network event **NET**, or an update of a view, for example **UPV**. A *decided* process is one in state **c** or **a**. A deciding configuration is one which has a quorum of decided processes and a deciding execution is one which contains a decided configuration.

4.5 E3PC's advantage over Q3PC

Example 1 See [41], three processes, p_1, p_2 and p_3, initially all connected, carry out E3PC. All processes vote **yes** and the first coordinator, p_1, moves to **pc**. A network partition causes p_2 and p_3 to become isolated. They are both in state **w**, and form a quorum. The second coordinator p_2 moves to **pa**, (rule **CPAT**) updating its last attempt counter. Another network event occurs and now p_2 rejoins p_1. Q3PC would now block, but E3PC can abort. See figure 4.3 towards the end of this chapter for a diagram of this execution.

□

We are now in a position to express the previous example 1 using our views based model. Figure 4.3 shows an execution sequence for the previous example up to the point where a coordinator decides **a**.

Interestingly, by replacing **CPCT** and **CPAT** with **CPCT′** and **CPAT′** respectively, thus deriving Q3PC from E3PC we see that in the seventh configuration represented in figure 4.3 neither **CPCT′** nor **CPAT′** apply. In fact no protocol rule applies to the seventh state and so even though p_1 and p_2 form a quorum they cannot make progress until a network event happens and so they are deemed blocked.

By examining the pre-conditions of **CC1T**, **CA1T**, **CPCT** and **CPAT** we can see that at least one of them must hold within a quorate component. This means that, unlike Q3PC, in E3PC a coordinator can make progress towards terminating a transaction within every quorate component.

4.6 Constructing X3PC from E3PC

We note that it would have been safe to commit rather than abort in example 1 of this chapter, because there is enough information for the coordinator in the last attempt to know the second attempt was unsuccessful and so view p_2*'s* **pa** state as its previous state **w**. This motivates the development of our new protocol X3PC.

We now show how to derive X3PC from E3PC. To do this we change the rules **CPAT** and **PPAT** and add two extra protocol rules, **CUV1T** and **CUV2T** which change a coordinator's view during the termination phase of the protocol. The update view rule allows a coordinator, p, to determine if a participant's earlier attempt to abort did not result in any process moving to **a**. The coordinator might be able to reach this conclusion in two ways.
CASE 1: Let q be a participant in state **pc** in coordinator p's component $Par(p)$ and let $q.la = i$. If the coordinator of attempt i is also in $Par(p)$ and that coordinator's status is not **a**, then p can safely assume no process reached **a** in attempt i where q moved to **pa**. Accordingly, p may adjust its view of $q.s$.
CASE 2: If the coordinator of attempt i is not present in $Par(p)$ but enough processes are present in $Par(p)$ that were also involved in attempt i, but did not move to **pa** during that attempt then the coordinator can deduce whether or not a quorum of processes moved to **pa** in attempt i. If not then the coordinator for attempt i could not have moved to **a**. Accordingly, p may safely adjust its view of $q.s$.

For processes to reason in these ways they must exchange extra information. Each process therefore keeps a history h. A process updates h when it enters the **pa** state. The history is indexed by last attempt number, i. If a process

moved to **pa** in attempt i, then it contains all of the processes involved in the i'th attempt denoted $h[i].involved$, the coordinator of the attempt $h[i].c$, and the process' previous state and last attempt counter before moving to **pa** are denoted respectively by $h[i].s_{prev}$ and $h[i].la_{prev}$. Initially, at all processes p, $\forall i,\ p.h[i] = \emptyset$. During an election a coordinator updates its view of all the histories of processes within its component. To reflect this in our rules we must extend the vector in the **NET** rule to include h so it becomes $\overrightarrow{z} = \langle s, la, le, h\rangle$. Finally we must change the post-action of the rules **CPAT**, and **PPAT** by replacing the post-action of **CPAT** with

$$\mathrm{SH}(p) \wedge p.la := p.le \wedge p.s := \mathbf{pa}$$

and the post-action of **PPAT** with

$$\mathrm{SH}(p) \wedge p.la := p.le \wedge p.s := \mathbf{pa}$$

where

$$\begin{aligned}\mathrm{SH}(p) \stackrel{def}{=} & h[le].involved = Par(p) \wedge h[le].c = p.c \wedge \\ & h[le].s_{prev} = p.s \wedge h[le].la_{prev} = p.la\end{aligned}$$

We are now in a position to define two new rules for coordinator p. Let

$$m \stackrel{def}{=} max\{t \mid \exists r \in Par(p),\ @p(r.la \approx t \ \wedge\ r.s \neq \mathbf{pa})\}$$

be the highest non **pa** attempt in a coordinator p's component. The rules attempt to change p's view of a participant q, if p's view of q's last attempt counter is greater than or equal to m, and p's view of q's state is **pa**. In both rules we omit $p.c = p \wedge p.f$ from the pre-conditions for the sake of brevity. The first rule **CUVT1** corresponds to CASE 1 above.

$$\mathbf{CUVT1}(p)\ \frac{@p(q.s = \mathbf{pa} \wedge q.la \geq m \wedge q.h[q.la].c = c' \wedge c'.s \neq \mathbf{a}) \wedge c' \in Par(p)}{@p(q.s := q.h[q.la].s_{prev} \wedge q.la := q.h[q.la].la_{prev})}$$

In the second rule **CUVT2** corresponding to CASE 2 above, let $L(q) \stackrel{def}{=} @p(q.h[q.la].involved)$ be coordinator p's view of q's involved set at attempt $q.la$ and $L'(q) \stackrel{def}{=} \{r \in Par(p) \mid @p(r.h[q.la].involved = \emptyset)\}$ be p's view of those processes in the current partition that were not involved in attempt $q.la$.

$$\mathbf{CUVT2}(p)\ \frac{q \in Par(p) \wedge @p(q.s = \mathbf{pa} \wedge q.la \geq m) \wedge \neg Q(L(q) - L(q'))}{@p(q.s := q.h[q.la].s_{prev} \wedge q.la := q.h[q.la].la_{prev})}$$

The rules "roll back" the view of q's **pa** state when there is enough information within the component of the current attempt to be sure that the earlier attempt, $q.la$ did not result in any process moving to state **a**. The pre-condition

of **CUVT1** ensures that the coordinator of attempt $q.la$ is in the current attempt with state not equal to **a**. The pre-condition of **CUVT2** ensures that the coordinator of attempt $q.la$ could not have moved to state **a** because a quorum of processes did not move to **pa** in that attempt. The rules consider processes with attempt numbers greater than or equal to the highest non pre-abort attempt. **CUVT1** has higher precedence than **CUVT2** so the precedence of all protocol rules for coordinators in the termination phase of X3PC is then (**CC1T**, **CA1T**)$\prec$(**CUVT1**, **CUVT2**)$\prec$ (**CPCT**, **CPAT**, **CC2T**, **CA2T**).

These update rules are by no means optimal. When initiating an attempt a coordinator could pass on all the attempt histories it collected to each of the participating processes which would put them in a better position to provide information to a future coordinator about the possible success of an abort attempt. We do not consider these further optimisations here.

In Example 1, it was the coordinator of the second attempt which returned to form a quorum with the first processes. At this point p_1 applies **CUVT1** to its view of p_2 updating this view so that $@p_1(p_2.s = \mathbf{w})$ and $@p_1(p_2.la = 0)$. Now rule **CPCT** allows p_1 to enter **pc**, rather than **pa**. If no more network events interrupt this component a commit decision is eventually reached. Figure 4.4 depicts these events.

4.7 X3PC Solves Atomic Commitment

In this section we appeal to the rules of our model to prove that executions of X3PC solve the atomic commit problem. We use the problem definition of Bernstein *et al.* [9] which can be found in figure 2.1 as a starting point. We will see that we can make more precise statements of what it means for our protocol to be correct than those made in this definition. In particular, since we have a more precise idea of what it means for communication to fail, we can give a more precise account of how resilient our protocol is to different types of failure.

We can divide any execution ρ of X3PC into attempts. The ith attempt is started within a component when the coordinator, p, of that component updates its last attempt counter to i. This happens in the rule **CPAT** in the case of an abort attempt or in the rule **CPCT** in the case of a commit attempt. We call the first attempt before the invocation of the termination phase attempt 0.

Lemma 1 *If a coordinator process decides* **c** *(***a***) during attempt $i \geq 0$ in an execution ρ then no process will decide* **a** *(***c***) during any attempt $j \geq i$.*

Proof Let attempt i be the first time any coordinator process, p, decides **c** (**a**). First consider the case p decides **a**. This must be in rule **CA1T**, **CAI**, **CVNI** or **CA2T**. Consider each case in turn.

- **CA1T**: By the pre-condition of **CA1T** p must have a view of some process $q \in Par(p)$ in state **a**, i.e. $@p(q.s = \mathbf{a})$ holds. For this to happen p's view must have been updated earlier in ρ by the rule **NET** or **UCV**. In either case for some q, $q.s = \mathbf{a}$ must have held earlier in ρ. q cannot be a coördinator, in this earlier attempt, since this contradicts our assumption that p was the first coordinator to decide so it must be a participant. This means q must decide in attempt 0 since no participant may decide in the termination phase before its coordinator. Clearly, if any participant decides **a** at attempt 0 then no process can reach **pc** and so even if subsequent **NET** events happen no process can decide **c**.

- **CAI** or **CVNI**: If a coordinator decides **a** during the initial phase (attempt 0) then by the pre-condition of **PPCI** and **PCI** and the new rules **UCV** and **UPV** no process exists up to this point in state **pc** or **c**. After this point either no **NET** event happens and so **c** is never reached or **NET** happens. After a **NET** event **CPCT** will never hold since IMAC will always fail since no process reached **pc** before leaving the initial phase and therefore **c** will not be reached by any process.

- Suppose at attempt $i > 0$, p decides **a** during rule **CA2T**. So by the pre-condition of **CA2T**, $\exists X \subseteq Par(p), Q(X) \wedge @p(\forall q \in X, q.s \equiv \mathbf{pa})$. Thus in attempt i a quorum of processes are in **pa** this change was conveyed to p with rule **UVC**. This took place during attempt i since $@p(q.s \equiv \mathbf{pa})$ holds at p only if p views q to have the same last attempt counter as itself. For the processes $q \in X$ to have moved to **pa** during attempt i their coordinator p must have moved to **pa**, earlier in ρ, during attempt i with the rule **CPAT**. Let us consider two cases from this point in ρ.

  - No further **NET** events happen. So within the component $Par(p)$ **CPCT** can never happen because its pre-condition cannot hold once the pre-condition of **CPAT** holds. So no process will reach **pc** during attempt i and so no process will reach **c**. Furthermore, since $Q(Par(p))$ for any component Y outside of $Par(p)$, $Q(Y)$ fails. This means in Y neither **CPAT** nor **CPCT** will be applicable and since $\neg p.f$ will hold for the new coordinator of Y, **CC2T**, **CA2T** will not apply and

CC1T CA1T will not apply by the assumption that p is the first coordinator to decide. Similarly, **PPCT**, **PPAT**, **PCT** and **PAT** are not applicable in Y.

- A further **NET** event happens. Consider any new component Y formed after this event. There are two cases to consider.
  - Y does not contain any process from X. It cannot therefore be quorate. Only rules **CC1T** or **CA1T** are applicable but this would mean some process must have decided before p did during attempt i violating our assumption.
  - Y does include a process q from X. q must have the highest last attempt in the component. To see this note that the **NET** rule ensures all processes from quorate components formed in ρ have monotonically increasing last elected counters. We now show that neither **CUV1T** nor **CUV2T** are applicable. If $p \in Y$ then clearly at the coordinator, p', for this attempt $@p'(p.s \approx \mathbf{a})$ and so **CUV1T** will not apply. Furthermore, $L(q) = X$ and $L'(q)$ cannot contain any $q \in X$ i.e. $L(q) \cap L'(q) = \emptyset$ therefore $L(q) - L'(q) = L(q)$ and thus $Q(L(q)-L'(q))$ holds making **CUV2T** inapplicable.

An exactly similar argument but slightly easier applies if in attempt i a process decides commit. An intersecting site in any subsequent quorate component will be in state **pc**, with a maximal last attempt counter, and so IMAC will hold, without the need for any applications of **CUV1T** or **CUV2T**.

□

Theorem 1 *X3PC satisfies AC1.*

Proof: Any decision must be the result of a successful attempt during the termination phase of the protocol or during the initial phase. It is clear that if any process decides **a** in the initial phase no process could reach state **pc** and thus all subsequent attempts will never result in **c**. Also if any process was to decide **c** in the initial phase *all* processes must be in state **pc** so in all subsequent attempts IMAC will hold preventing any process from deciding **a**.

In the termination phase a coordinator can only move to **a** (**c**) if the component of the attempt is quorate. Of course a coordinator may propagate **a** (**c**) decisions within a non-quorate group using rules **CC1T** and **CA1T** but the decision will remain consistent. In any execution ρ no two quorate attempts may be interleaved

so by lemma 1 after one successful attempt a decision value is locked for all future attempts

□

Lemma 2 *X3PC satisfies AC2.*

Proof: Follows from lemma 1.

□

Lemma 3 *X3PC satisfies AC3*

Proof: Let p be a process that decides **c** in an execution ρ of X3PC. There are two cases. Either it is a coordinator or it is not. If it is not, then it can decide commit as the consequence of one of the rules **PCI** or **PCT**. In either case the pre-condition $@p(c.s = \mathbf{c})$ must hold. So there must have been an earlier point in the execution when at p's $c.s = \mathbf{c}$. Thus some coordinator decided **c**, either during an earlier attempt or during the initial phase. If that coordinator decided **c** during the initial phase then all processes must have voted **yes** when applying rule **PVYI** therefore we may restrict our attention to earlier attempts in the termination phase.

Consider the first time a coordinator, p changes state to **pc** in the termination phase. Such an event must occur (if not then no process could reach a **c** decision).

When p entered **pc** it must have been because of rule **CPCT** a pre-condition of which is $\text{IMAC}(X) \wedge Q(X)$ for some subset X of $Par(p)$. For $\text{IMAC}(X)$ to hold we know that the coordinator p must have constructed a view of some $q \in X$ where $@p(q.s = \mathbf{pc})$, so q entered **pc** in an *earlier* attempt. By assumption this is the first such attempt in the termination phase where a coordinator p moves to **pc**, thus some process must have been in **pc** in the initial phase. By the pre-condition of rule **CPCI** and the **UCV** rule if some process was in state **pc** in the initial phase all processes must have voted **yes**.

□

Lemma 4 *X3PC satisfies AC4.*

Proof: If no **NET** rules occur then X3PC does not enter its termination phase so it behaves as 3PC.

□

Lemma 5 *X3PC satisfies AC5.*

Proof If even only a quorate group X becomes connected for sufficiently long they will reach a decision and if any process becomes connected to a decided process it too will decide. This follows from the fact that the pre-condition of at least one protocol rule holds until a decision is reached.

□

Theorem 2 *If using the rules of E3PC a commit decision is reached during the termination phase where no more than two quorums have formed then using the rules of Q3PC will also lead to a commit decision.*

Proof: In a committing run of E3PC there exists a configuration in the initial phase before the first network failure of the form

$$\overbrace{\mathbf{w}\ \mathbf{w}\ \ldots\ \mathbf{w}\ \mathbf{w}}^{n-m}\ \overbrace{\mathbf{pc}\ \mathbf{pc}\ \ldots\ \mathbf{pc}\ \mathbf{pc}}^{m}$$

where $m > 0$. This follows from the proof of Lemma 3. Consider the first time a quorate component X forms after a **NET** event. X must either consist of processes all in state **w**, or processes in **pc** and **w**, in the latter case, for both E3PC and Q3PC, the newly elected leader of X, will apply rules **CPCT** and participant processes will attempt to move to **pc**. In the former case both E3PC and Q3PC will carry out rule **CPAT** to attempt to move processes to **pa**. Let L be the set of processes that move to **pa** during this attempt. From this point another network event must occur, if not this quorum would decide abort. Consider the next quorum. By assumption this must be the last quorum to form and must result in a commit decision. So we know rule **CPCT** must be applicable in E3PC. Clearly no process from L could be present in this quorum as it would have a maximum attempt counter and then IMAC would not hold invalidating the pre-condition of **CPCT** so the quorum can only consist of processes in **pc** and **w** and at least one process must be in **pc**. We see then that Q3PC can also apply rule **CPCR** and behave in an identical way to E3PC producing a committing execution.

□

4.8 Performance Comparison

To compare the three protocols we considered runs from the point that all participants had voted **yes** and entered their **w** state and the coordinator had collected

these votes and changed state to **pc**. This is the most interesting initial condition because if any process votes no, or if all vote yes but the coordinator does not change to **pc**, before entering the recovery phase then all protocols will abort the transaction.

Each protocol was compared by examining random runs. Between each protocol step a network event could occur with uniform probability γ, causing the network to partition. We only considered network events which resulted in quorums being formed. The same failure pattern was applied to all protocols during each run. Where there was a choice of steps to apply (i.e. more than one process could take a step) one was picked at random. The protocol was deemed to have decided once a quorum of processes decided, or in the case of Q3PC blocked if no process could take a step. The results of 500 runs for seven processes and seven values of γ are presented in Fig 4.5.

The behaviour of E3PC and Q3PC is similar in executions when the outcome is commit. E3PC will often abort a transaction if it would block in Q3PC for an identical failure pattern. This is not generally true but by theorem 2 holds in the case where failures are followed by the formation of less than two quorums. E3PC is far superior to the Q3PC at avoiding blocking, especially when network disruption is high. In our experiments a run of Q3PC was deemed blocked if it could not take a step. The equivalent E3PC run would continue but might have undergone several more network partitions before reaching a decision. Interestingly if Q3PC was allowed to run for as long (i.e. allowing further network partitioning) as E3PC took to reach a decision it would still block in many cases.

X3PC will commit in all runs that E3PC commits. Especially under high network disruption, X3PC will commit many of the transactions that E3PC aborted. When network disruption is very high both E3PC and X3PC take many attempts to reach a decision. X3PC is more likely to decide to commit under high, rather than medium, disruption. This is because under high levels of random disruption more information can be exchanged between processes about unsuccessful attempts to abort. When a period of calm returns X3PC is then in a good position to move towards commit, whereas in E3PC there is a much greater chance that pre-abort attempts dominate.

4.9 Conclusions and Future Research Directions

It is possible to further optimise X3PC. Processes could store not only their own histories of attempt to pre-abort but also the histories of other processes.

Coordinators, after gathering this information, could distribute this information to all participants. This process could go on even in a non-quorate components. This would further improve the likelihood that pre-abort attempt could be rolled back using an update view rule.

Up to this point we have assumed that all processes have perfect knowledge of which other processes are in their own component. This is equivalent to having a perfect failure detector [11]. The protocol is also correct if processes only have access to an unreliable account of which other processes are in their component, as long as this knowledge is eventually reliable. For example, a process p may suspect that it has lost communication with another process q provided that eventually 1) all real lost connections are eventually suspected and 2) no process that is really connected is suspected indefinitely. If we assume this weaker model we must change **AC4** in the problem definition to:

> If all processes voted yes, and no process is ever *suspected* then the decision will be to commit.

Like E3PC, X3PC solves this weaker version, of the problem, called the Non-blocking Atomic Commit [35], using only an eventually perfect-failure detector [24, 11]. As in E3PC, X3PC will terminate once a quorum of processes become connected and no failures or suspicions occur for sufficiently long.

In a mobile computing or Internet environment where network disruption is common E3PC gives a greater availability of service than Q3PC. As applications use transaction semantics in increasingly varied ways it may become more difficult to restart transactions frequently. X3PC provides a protocol which combines the high availability of E3PC with a greater chance of committing a transaction even when network disruption is high.

Recently IBM and Microsoft have cooperated on a new protocol called SOAP. SOAP provides remote procedure calls over the Internet using HTTP. Because the HTTP protocol is stateless (each HTTP request appears as a new connection) X3PC would be a perfect protocol to build on top of SOAP as the basis of an Internet transaction processing system.

We have shown that our views based model is sufficient to model the execution of complex protocols within a rich environment that includes a notion of failure. The rules define precisely the actions of processes within a protocol and also the types of environment changes that might take place. By interleaving these rules we can generate a set of possible protocol executions and by appealing to the rules we can show properties over the set of all protocol executions.

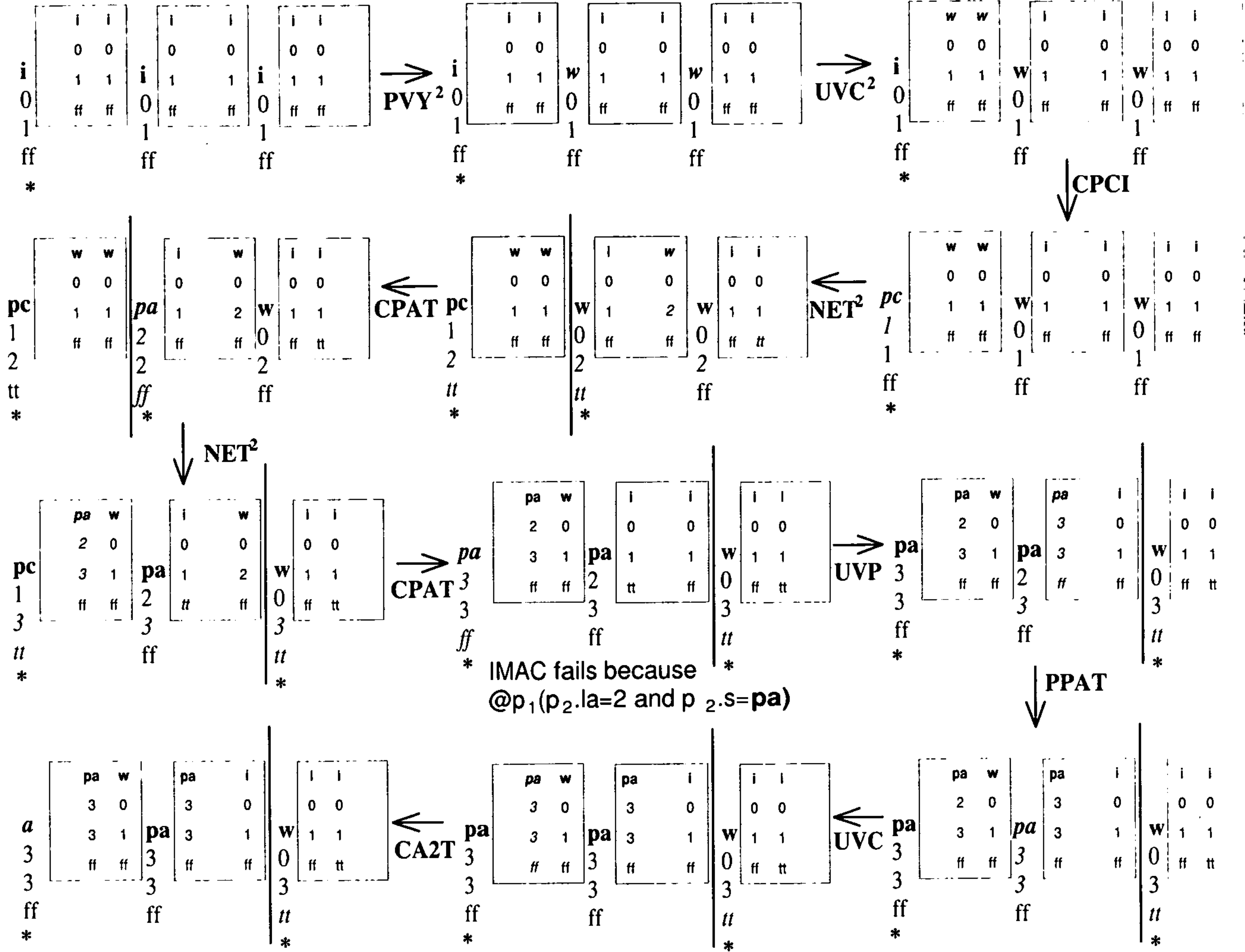

Figure 4.3: One execution of example 1 for three processes p_1,p_2 and p_3. The fifth configuration in the execution represents $Par = \{\{p_1\}, \{p_2, p_3\}\}$, $p_1.c = p_1$, $p_1.s =$ **pc**, $p_1.la = 1$, $p_1.le = 2$, $p_1.f =$ **tt** etc. Views are also represented for example in the fifth state at $@p_2(p_1.s \approx \mathbf{i})$ holds and, $@p_1(p_2.s \approx \mathbf{w} \wedge p_3.s \approx \mathbf{w})$ holds. Since a process' view of itself is always up-to-date it is omitted. When a rule changes the value of a variable we *italicise* the change in the next state, for instance at p_1, $p_1.la = 0 \rightarrow p_1.la = 1$ between the third and fourth states represented. Here, and in the rest of the thesis, when a rule **R** is applied n-times we write $\mathbf{R^n}$. We use a (*) to label a coordinator and thus omit *p.c* as a variable explicitly.

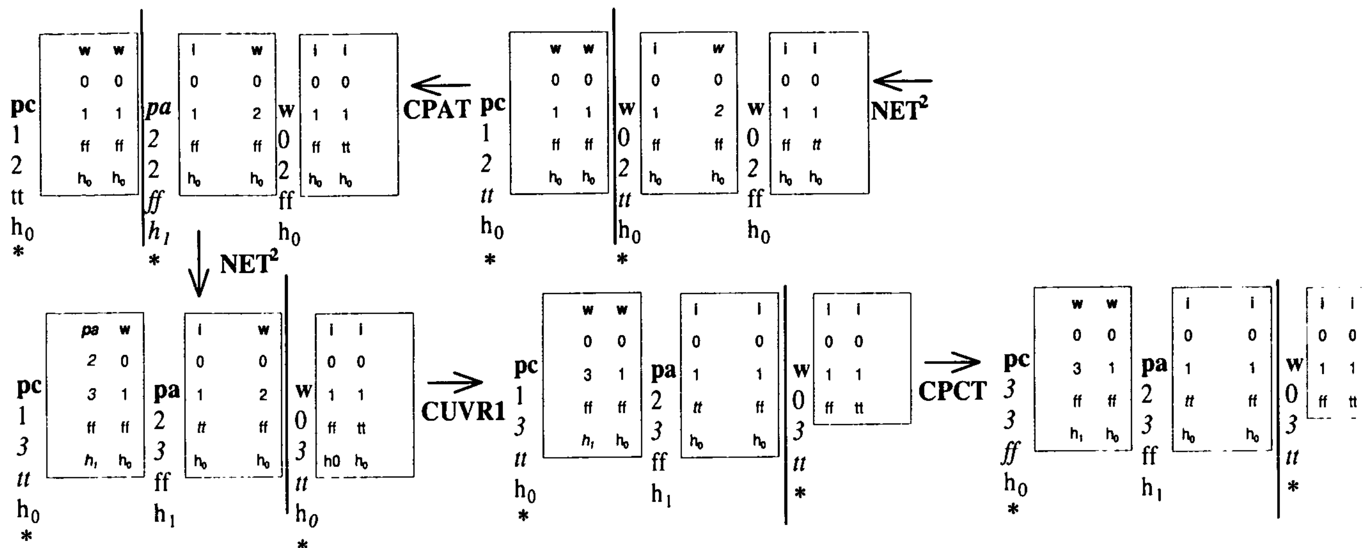

Figure 4.4: One execution of example 1 using X3PC. In the diagram h_0 is the empty history and $h_1[2].s_{prev} = \mathbf{w}$, $h_1[2].la_{prev} = 0$, $h_1[2].involved = \{p_2, p_3\}$ and $h_1[2].c = p_2$.

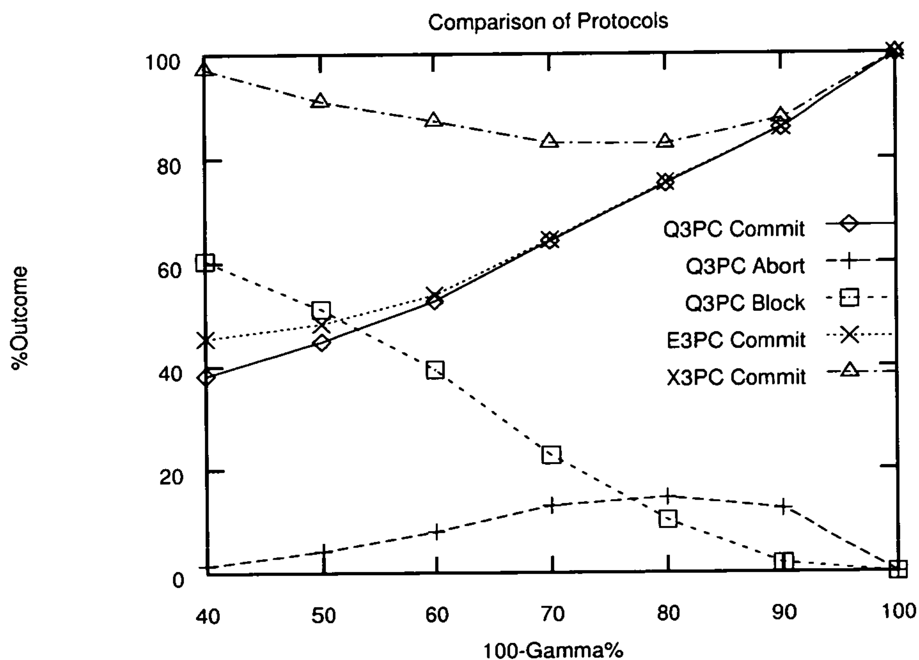

Figure 4.5: Comparison of E3PC, X3PC and Q3PC

Chapter 5

Model Checking Two-Phase Commit

5.1 Introduction

In this chapter we will present methods by which a transition system can be automatically generated from a views based model of a commit protocol. Each state in the resulting transition system will represent a configuration in a protocol execution. A transition from one state to another reflects the evolution of the system during a protocol execution. The entire transition system therefore models the possible behaviour that can occur during any protocol executions that were included in the views based model.

Temporal logics such as CTL [16] have been used extensively to formally express properties and capabilities of transition systems. Using these logics we can express properties of protocol executions. For example, that a path exists in a commit protocol's transition system to a state which has no successors and yet no commit or abort decision has been reached. This example captures the property that the protocol might block. It lacks the capability of taking a step from some reachable undecided state.

Many algorithms exist which take as input a labelled transition system[1] and a property expressed in a logic such as CTL and outputs whether or not[2] the property holds or fails for the transition system supplied. This technique is known as *model checking* [19].

Our strategy therefore, is to first generate transition systems from views based models that capture the behaviour of commit protocols and then verify properties of the resulting transition systems using model checking. We will see in this

[1] The entire transition system need not be supplied in every case.

[2] Some algorithms have the desirable property that if a property fails for a particular transition system, an explanation of why the property failed is given.

chapter that we can formally express many useful properties of commit protocols using CTL. A good example is atomicity, that is the safety property that in no protocol execution one participant can abort and another commit.

Model checking facilitates an iterative design process whereby a protocol can be modelled, a transition system automatically generated and then properties of that system model checked. If a desirable property fails due to a bug in the protocol (or model) this can be rectified and the process repeated. Of course this design process is only as good as the extent to which the model reflects the real system being modelled and the property reflects the designer's intended property.

In this chapter we present a model checking algorithm based on games [76] and use it to model check properties, expressed using a sub-logic of CTL, of the simple 2PC protocol presented in chapter 3.

Although model checking is a very useful and powerful technique it suffers from a major drawback. For complex protocols (exactly those that benefit from the support of automated verification) the size of transition systems generated are often very large or even infinite. This is known as the *state space explosion* problem [17]. For many properties the time and space required to check a property depend on the size of the transition system being checked. To circumvent this problem *abstraction techniques* [18] have been devised to reduce the size of the resulting transition systems. We will see in this chapter that our views based modelling technique accommodates a very natural abstraction technique that allows us, to some extent, to alleviate the state space explosion problem.

5.2 Modelling Two-Phase Commit

Recall the simple model of 2PC from section 3.6.5. We restate the rules of this simple protocol.

$$\mathbf{PVY}(p) \quad \frac{s = \mathbf{i}}{s := \mathbf{w}} \qquad \mathbf{PVN}(p) \quad \frac{s = \mathbf{i}}{s := \mathbf{a}}$$

$$\mathbf{PC}(p) \quad \frac{s = \mathbf{w} \wedge @p(c.s = \mathbf{c})}{s := \mathbf{c}} \qquad \mathbf{PA}(p) \quad \frac{s \in \{\mathbf{w}, \mathbf{i}\} \wedge @p(c.s = \mathbf{a})}{s := \mathbf{a}}$$

$$\mathbf{CC}(c) \quad \frac{s = \mathbf{i} \wedge\ @c(\forall p \in P, p.s = \mathbf{w})}{s := \mathbf{c}} \qquad \mathbf{CA}(c) \quad \frac{s = \mathbf{i} \wedge\ @c(\exists p \in P, p.s = \mathbf{a})}{s := \mathbf{a}}$$

$$\textbf{CUV} \quad \frac{p.s = x \wedge @c(p.s = \mathbf{i})}{@c(p.s := x)} \quad x \in \{\mathbf{w}, \mathbf{a}\}$$

$$\textbf{PUV} \quad \frac{c.s = x \wedge @p(c.s = \mathbf{i})}{@p(c.s := x)} \quad x \in \{\mathbf{c}, \mathbf{a}\}$$

In this model there are two classes of processes, the coordinator class C, and the participant class P. Processes from each class have a single state variable s. In the case of the coordinator this variable may take values **i** (initial), **c** (commit) or abort **a** (abort). In addition to these states participants have a further state **w** (wait). Table 3.3 of section 3.6.5 summarises this. We will use the notation x^y to denote a participant (coordinator) that is in state x with a view that its coordinator (participants) is (are) in state(s) y. For example in the case of a participant in state **w** with a view that its coordinator is in state **i** we write $\mathbf{w}^{\mathbf{i}}$. In this case the view is of a single state but this is not always the case. For example the coordinator keeps a view of the states of all the processes it is coordinating. For example, if the coordinator, in state **i**, has a view of three participants that are in states **i**, **w**, **i** respectively, we might write $\mathbf{i}^{\langle \mathbf{i},\mathbf{w},\mathbf{i} \rangle}$. It will always be clear from the context when we use the notation x^y whether we are describing the state of a participant or a coordinator. When representing multiple states we will sometimes use vectors as in this example and sometimes sets or multi-sets.

5.3 Generating Transition Systems from Rules

For the commit protocols in which we are interested, the number of processes within the participant class P may be large. We would like to construct arguments about protocol executions involving n participant processes, which are valid for all $n > 0$. In order to generate a transition system from our set of rules we must find a way to represent the state of all n processes within a particular class. In this section we discuss two approaches to this problem, the second more abstract than the first. The first approach is the most concrete but it results in very large transition systems, whereas the second approach is the most abstract representation. By means of an argument based on simulation we will show that for particular classes of properties it is sufficient to show that if properties hold (fail) in the abstract system then they also hold (fail) in the concrete system.

5.3.1 A Concrete approach

A simple approach to representing the states of n processes in a system uses a vector of length n. We have already used this approach in section 3.6.5 where a system configuration C was represented by the composition of a fixed number of process states together with a coordinator, $\langle s_1, \ldots, s_n \rangle, t$, where s_i is the state of p_i and t is the state of c. We call this representation CON. For example we can represent four processes p_1, p_2, p_3, p_4 and a coordinator, c, where p_1, p_2, p_3 are in state $\mathbf{w}^\mathbf{i}$, p_4 is in state $\mathbf{i}^\mathbf{i}$ and the coordinator c is in state $\mathbf{i}$ with an up-to-date view of its participants as

$$\langle \mathbf{w}^\mathbf{i}, \mathbf{w}^\mathbf{i}, \mathbf{w}^\mathbf{i}, \mathbf{i}^\mathbf{i} \rangle \mathbf{i}^{\langle \mathbf{w},\mathbf{w},\mathbf{w},\mathbf{i} \rangle}$$

Not all combinations of state vectors are valid in CON. In order to restrict configurations to just those that are valid, we define a partial order on the states a process may take. This order is

$$\mathbf{i} < \mathbf{w} < \mathbf{c}, \mathbf{a}$$

A CON configuration

$$\langle x_i^{z_1}, \ldots, x_n^{z_n} \rangle y^{\langle u_1, \ldots u_n \rangle}$$

is valid iff $u_i \leq x_i$ and $z_i \leq y$, $1 \leq i \leq n$. An example of a configuration that is *not* valid is

$$\langle \mathbf{i}^\mathbf{i}, \mathbf{i}^\mathbf{i}, \mathbf{i}^\mathbf{i}, \mathbf{i}^\mathbf{i} \rangle \mathbf{i}^{\langle \mathbf{w},\mathbf{w},\mathbf{w},\mathbf{i} \rangle}$$

From this point onwards in the thesis we restrict our attention to valid CON configurations only. One can see that the validity condition we place on CON configurations is preserved by our rules for 2PC.

When a rule is applied to a process in the vector a new vector is produced. Thus if rule **R** happens at a process p_i in the vector $\langle s_1, \ldots, s_i, \ldots, s_n \rangle$ which we can write as $s_i \xrightarrow{\mathbf{R}} s_i'$, then we have the following transition in our representation

$$\langle s_1, \ldots, s_i, \ldots, s_n \rangle, t^{\langle \ldots \rangle} \xrightarrow{\mathbf{R}} \langle s_1, \ldots, s_i', \ldots, s_n \rangle, t^{\langle \ldots \rangle}$$

In fact we can allow any type of rule that produces valid future configurations.

Generating a transition system in this way is straightforward. The applicability of a rule, **PVY**, **PVN**, **PC** or **PA**, is tested by evaluating its pre-condition for each participant process in the vector. If the pre-condition holds, a post-action can be applied to that process resulting in a new vector. In order to update a view at a participant, we apply the rule **PUV** by checking if the coordinator state is different from the participant's view, if so we update the view. The coordinator

rules are similar. The pre-conditions of the rules **CC** and **CA** can be determined by examining the view vector of the coordinator, and an appropriate change made to the coordinator's state, if a rule is applicable. Finally **CUV** is applicable if in the view vector of the coordinator, the view of a participant, is out of date; we give an example of this below.

$$\langle \mathbf{w}^{\mathbf{i}}, \mathbf{w}^{\mathbf{i}}, \mathbf{w}^{\mathbf{i}}, \mathbf{i}^{\mathbf{i}} \rangle \mathbf{i}^{\langle \mathrm{w,w,i,i} \rangle} \xrightarrow{\mathbf{CUV}} \langle \mathbf{w}^{\mathbf{i}}, \mathbf{w}^{\mathbf{i}}, \mathbf{w}^{\mathbf{i}}, \mathbf{i}^{\mathbf{i}} \rangle \mathbf{i}^{\langle \mathrm{w,w,w,i} \rangle}$$

CON has a major drawback. Suppose we use it to represent n participant processes and a coordinator and now consider all the possible executions that result from the simple 2PC protocol where each participant votes **yes** up until the point that all participants have voted. Figure 5.1 (a) depicts this. This gives rise to greater than $n!$ transitions. To see this note the number of transitions is the solution to the recurrence relation $S_n = n(S_{n-1} + 1)$ and $n(S_{n-1} + 1) > n(S_{n-1})$. In fact matters are worse if we allow participants to vote **no** or include transitions whereby a coordinator updates its view.

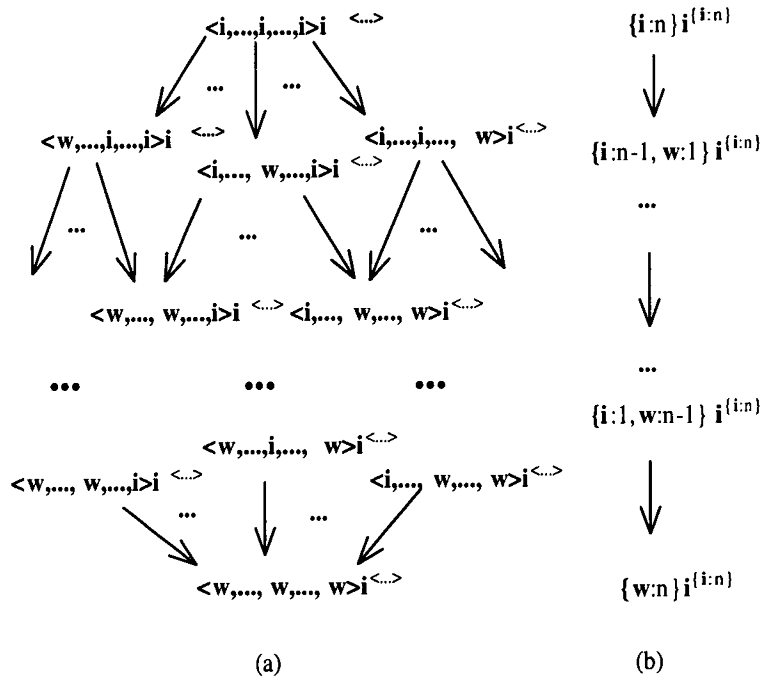

Figure 5.1: In (a) the state space explosion that results from using the CON representation. The MULTI representation in (b) reduces this.

CON produces unmanageable state spaces for even small fixed numbers of participant processes.

5.3.2 Multi-set representation

We can represent a set of n processes as a multi-set. To do this we list, in any order, each different state together with the number of times it occurs. Following on from the previous example, the four processes, three in $\mathbf{w}^{\mathbf{i}}$, and one in $\mathbf{i}^{\mathbf{i}}$, together with a coordinator can be represented as follows.

$$\{\mathbf{w}^{\mathbf{i}} : 3, \mathbf{i}^{\mathbf{i}} : 1\}\mathbf{i}^{\{\mathrm{w}:3,\mathrm{i}:1\}}$$

We call this representation scheme MULTI. The size of a MULTI configuration is the number of participants represented in the participant multi-set. In the example above therefore the size is 4.

It is useful to be able to map between CON and MULTI representations. To do this we define a function f : CON $\rightarrow$ MULTI and a relation $G \subseteq$ MULTI $\times$ CON. To simplify their definition we first define auxiliary functions α and α' that map an n length vectors of states to a multi set of states as follows.

$$\alpha(\langle s_1, \ldots, s_n\rangle) = \{s_i : c_i \mid s_i \text{ occurs } c_i \text{ times in } \langle s_1, \ldots, s_n\rangle\}$$

$$\alpha'(\langle s_1^{t_1}, \ldots, s_n^{t_n}\rangle) = \{s_i^{t_i} : c_i \mid s_i^{t_i} \text{ occurs } c_i \text{ times in } \langle s_1^{t_1}, \ldots, s_n^{t_n}\rangle\}$$

We now define function f and its inverse relation G using α and α'.

$$f(\tilde{s}, t^{\tilde{v}}) \stackrel{def}{=} \alpha'(\tilde{s}), t^{\alpha(\tilde{v})}$$

$$G \stackrel{def}{=} f^{-1}$$

We use the symbol $\tilde{s}$ to denote the state of a group of n processes using one of the representations (either a vector or a multi-set), t denotes the state of the coordinator and $\tilde{v}$ the coordinator's view, again using one of our representations. Although the notation is a little heavyweight the transformations are really very simple. Consider the examples below.

$$\langle \mathbf{w}^{\mathbf{i}}, \mathbf{w}^{\mathbf{i}}, \mathbf{w}^{\mathbf{i}}, \mathbf{i}^{\mathbf{i}}\rangle, \mathbf{i}^{\langle \mathrm{w,w,i,i}\rangle} \in G(\{\mathbf{w}^{\mathbf{i}} : 3, \mathbf{i}^{\mathbf{i}} : 1\}, \mathbf{i}^{\{\mathrm{w}:2,\mathrm{i}:2\}})$$

$$f(\langle \mathbf{i}^{\mathbf{i}}, \mathbf{w}^{\mathbf{i}}, \mathbf{w}^{\mathbf{i}}, \mathbf{i}^{\mathbf{i}}\rangle, \mathbf{i}^{\langle \mathrm{w,i,i,w}\rangle}) = \{\mathbf{w}^{\mathbf{i}} : 2, \mathbf{i}^{\mathbf{i}} : 2\}, \mathbf{i}^{\{\mathrm{w}:2,\mathrm{i}:2\}}$$

Again not all possible combinations of multi-sets constitute valid MULTI representations. We say a MULTI representation $\tilde{s}$ is valid if there exists a valid CON representation $\tilde{s}'$ such that $\tilde{s}' \in G(\tilde{s})$. Again we restrict our attention to MULTI representations that are valid from this point on.

Later when we come to show how to model check properties of protocols we will be particularly interested in classes of statements where the truth or

falsity is preserved by these transformations. These will form atomic sentences in the temporal logics we use to express properties. For example the property "all participants are in state $\mathbf{w^i}$" is preserved by our function f and relation G whereas the statement "p_4 is in state $\mathbf{i^i}$" is not preserved because the MULTI representation does not retain which participant processes are in which states. We will show that there is a simulation relationship between the MULTI and CON.

We now address an important question, when can a protocol rule be applied to a MULTI representation and what effect does it have on that representation? A rule can be applied to a MULTI representation iff there exists a corresponding CON representation where the same rule can be applied. Suppose a participant process, p_i in state s_i moves to state s_i' then in our multi-set representation we decrease the counter associated with s_i and increase the counter[3] associated with s_i'. In our example if the participant process in state $\mathbf{i^i}$ moves to $\mathbf{w^i}$ during rule **PVY** we have the following transition in a MULTI representation.

$$\{\mathbf{w^i} : 3, \mathbf{i^i} : 1\}\mathbf{i}^{\{\mathrm{w:3,i:1}\}} \xrightarrow{\mathbf{PVY}} \{\mathbf{w^i} : 4\}\mathbf{i}^{\{\mathrm{w:3,i:1}\}}$$

The rules **PVN**, **PC**, **PA** are similar. The rule **PUV** is applicable to a participant process when its view of the coordinator is out of date. The **CC** rule is applicable when the coordinator views all participants in state $\mathbf{w}$ so in the MULTI representation this means the view consists of the multi-set $\{\mathbf{w} : n\}$. Similarly, the **CA** rule is applicable if the coordinator views any participant in state $\mathbf{a}$. This corresponds to $\mathbf{a}$ being in the coordinator's view multi-set with any positive count. **CUV** is applicable when the the coordinator's view multi-set is out of date, i.e. the count of participants viewed to be in a state $x \in \{\mathbf{a}, \mathbf{w}\}$ is less than the count of processes in that state[4]. This is exemplified below.

$$\{\mathbf{w^i} : 3, \mathbf{i^i} : 1\}\mathbf{i}^{\{\mathrm{w:2,i:2}\}} \xrightarrow{\mathbf{CUV}} \{\mathbf{w^i} : 3, \mathbf{i^i} : 1\}\mathbf{i}^{\{\mathrm{w:3,i:1}\}}$$

Fact 1: The sum of the counters in any MULTI configuration is preserved by the application of any of the protocol rules. This follows from the observation that participants and coordinators are neither created nor destroyed by the rules.

Proposition 1 If a rule $\mathbf{R}$ is applicable to a CON representation $\tilde{s}_1, t_1^{\tilde{v}_1}$ to reach $\tilde{s}_2, t_2^{\tilde{v}_2}$ then $\mathbf{R}$ is applicable to $f(\tilde{s}_1, t_1^{\tilde{v}_1})$, a MULTI representation, to derive a state $\tilde{s}_4, t_4^{\tilde{v}_4}$ and $\tilde{s}_2, t_2^{\tilde{v}_2} \in G(\tilde{s}_4, t_4^{\tilde{v}_4})$. This is depicted below, strictly speaking the arrow labeled G should be broken but typographical restrictions prevented this.

[3]If a counter is decreased to 0 that element is removed. If no element for s_i' exists it is added with counter value 1.

[4]This follows from the fact that the state must be valid.

$$\begin{array}{ccc} & \text{CON} & \\ \tilde{s}_1, t_1^{\tilde{v}_1} & \xrightarrow{\mathbf{R}} & \tilde{s}_2, t_2^{\tilde{v}_2} \\ | & & \uparrow \\ f & & G \\ \downarrow & & | \\ f(\tilde{s}_1, t_1^{\tilde{v}_1}) & \overset{\mathbf{R}}{- - \rightarrow} & \tilde{s}_4, t_4^{\tilde{v}_4} \\ & \text{MULTI} & \end{array}$$

Proof: First consider the case where a participant process applies $\mathbf{R}$. Let s_i be the state in the CON representation $\tilde{s}_1$ that changes to s_i' after $\mathbf{R}$ is applied. Let us also assume that the pre-condition of $\mathbf{R}$ is determined only by the state s_i, this is true for rules **PVY**, **PVN**, **PC** and **PA**. Let l, m be the number of s_i and s_i' states in $\tilde{s}$ respectively. $\mathbf{R}$ is applicable to the MULTI representation $f(\tilde{s}_1, t_1^{\tilde{v}_1})$ by definition. Furthermore, after application l is decreased by one and m is increased by one so $\tilde{s}_2, t_2^{\tilde{v}_2} \in G(\tilde{s}_4, t_4^{\tilde{v}_4})$ as required.

If **PUV** is applicable to some p_i in state s_i within $\tilde{s}_1$ it will also be applicable to $f(\tilde{s}_1, t_1^{\tilde{v}_1})$ by definition. Clearly after the application $\tilde{s}_2, t_2^{\tilde{v}_2} \in G(\tilde{s}_4, t_4^{\tilde{v}_4})$ as required.

If **CC** is applicable then $\tilde{v}_1 = \langle \mathbf{w}, \ldots, \mathbf{w} \rangle$ and after **CC** is applied $t_2 = \mathbf{c}$ and so **CC** will also be applicable to $f(\tilde{s}_1, t_1^{\tilde{v}_1})$ by definition. The multi-set view $\tilde{v}_3$ will be $\{\mathbf{w} : n\}$. After the application $t_4 = \mathbf{c}$ assuring $\tilde{s}_2, t_2^{\tilde{v}_2} \in G(\tilde{s}_4, t_4^{\tilde{v}_4})$ as required.

If **CA** is applicable then $\tilde{v}_1$ will contain some $\mathbf{a}$ and so **CA** will be applicable to $f(\tilde{s}_1, t_1^{\tilde{v}_1})$ because the multi-set view will contain some $\mathbf{a}$. After the application $t_4 = \mathbf{a}$ as required.

Finally if **CUV** is applicable then some p_i exists with value $x \in \{\mathbf{c}, \mathbf{a}\}$ in $\tilde{s}_1$ such that the coordinator's view of p_i in $\tilde{v}_1$ is not x. By definition **CUV** is applicable to $f(\tilde{s}_1, t_1^{\tilde{v}_1})$. The count of processes in the coordinator's view that are in state x will be less than the number of processes in state x in $\tilde{s}_3$. Therefore after the application $\tilde{s}_4, t_4^{\tilde{v}_4}$ will have the required property that $\tilde{s}_2, t_2^{\tilde{v}_2} \in G(\tilde{s}_4, t_4^{\tilde{v}_4})$.

□

Corollary 1 If in CON $C_1 \xrightarrow{\mathbf{R_1}} C_2 \xrightarrow{\mathbf{R_2}} \ldots \xrightarrow{\mathbf{R_{n-1}}} C_n \xrightarrow{\mathbf{R_n}} \ldots$ then in MULTI

$$\begin{array}{ccccccccc} C_1 & \xrightarrow{\mathbf{R_1}} & C_2 & \xrightarrow{\mathbf{R_2}} & \ldots & \xrightarrow{\mathbf{R_{n-1}}} & C_n & \xrightarrow{\mathbf{R_n}} \\ | & & \uparrow & & & & \uparrow & \\ f & & G & & & & G & \\ \downarrow & & | & & & & | & \\ f(C_1) & \xrightarrow{\mathbf{R_1}} & C_2' & \xrightarrow{\mathbf{R_2}} & \ldots & \xrightarrow{\mathbf{R_{n-1}}} & C_n' & \xrightarrow{\mathbf{R_n}} \end{array}$$

such that $C_i \in G(C_i')$ for $2 \leq i \leq n$.

□

Proposition 2 If a rule $\mathbf{R}$ is applicable to a MULTI representation $\tilde{s}_1, t_1^{\tilde{v}_1}$ to reach $\tilde{s}_2, t_2^{\tilde{v}_2}$ then $\mathbf{R}$ is applicable to *all* CON representations in $G(\tilde{s}_1, t_1^{\tilde{v}_1})$. Moreover for any $\tilde{s}_4, t_4^{\tilde{v}_4}$ that results from the application of $\mathbf{R}$ $f(\tilde{s}_4, t_4^{\tilde{v}_4}) = \tilde{s}_2, t_2^{\tilde{v}_2}$. This is depicted as follows.

$$\begin{array}{ccc} & \text{CON} & \\ \tilde{s}_3, t_3^{\tilde{v}_3} & \overset{\mathbf{R}}{\dashrightarrow} & \tilde{s}_4, t_4^{\tilde{v}_4} \\ \uparrow & & | \\ G & & f \\ | & & \downarrow \\ \tilde{s}_1, t_1^{\tilde{v}_1} & \overset{\mathbf{R}}{\longrightarrow} & \tilde{s}_2, t_2^{\tilde{v}_2} \\ & \text{MULTI} & \end{array}$$

Proof: Again consider the case where a participant process applies $\mathbf{R}$. Let s_i be the state in the MULTI representation $\tilde{s}_1$ that changes to s_i' after $\mathbf{R}$ is applied. Let us also assume that the pre-condition of $\mathbf{R}$ is determined only by the state s_i, this is true for rules **PVY**, **PVN**, **PC** and **PA**. Clearly some process is in state s_i in all possible $\tilde{s}_3$ so $\mathbf{R}$ is applicable as it only depends on the state s_i. Furthermore, after $\mathbf{R}$ is applied to $\tilde{s}_3, t_3^{\tilde{v}_3}$ resulting in $\tilde{s}_4, t_4^{\tilde{v}_4}$, one less process will be in s_i and one more in s_i' therefore $\tilde{s}_2, t_2^{\tilde{v}_2} = f(\tilde{s}_4, t_4^{\tilde{v}_4})$ as required.

If **PUV** is applicable to some process in state s_i within $\tilde{s}_1$ it will also be applicable to all $\tilde{s}_3, t_3^{\tilde{v}_3}$ because applicability depends only on the state s_i which will be present in $\tilde{s}_3$ and the state of the coordinator c_3 which is preserved by G. Clearly after application $\tilde{s}_2, t_2^{\tilde{v}_2} = f(\tilde{s}_4, t_4^{\tilde{v}_4})$ as required.

If **CC** is applicable then $\tilde{v}_1 = \{\mathbf{w} : n\}$ and and after the application of **CC** $t_2 = \mathbf{c}$ and so **CC** will be applicable to all $\tilde{s}_3, t_3^{\tilde{v}_3}$ because the resulting view vectors for the coordinator, $\tilde{v}_3$ will be $\langle \mathbf{w}, \ldots, \mathbf{w} \rangle$. After the application $t_4 = \mathbf{c}$ assuring $\tilde{s}_2, t_2^{\tilde{v}_2} = f(\tilde{s}_4, t_4^{\tilde{v}_4})$ as required.

If **CA** is applicable then $\tilde{v}_1$ will contain some $\mathbf{a}$ and so **CA** will be applicable to all $\tilde{s}_3, t_3^{\tilde{v}_3}$ because $\tilde{v}_3$ will contain $\mathbf{a}$. After the application $t_4 = \mathbf{a}$ assuring $\tilde{s}_2, t_2^{\tilde{v}_2} = f(\tilde{s}_4, t_4^{\tilde{v}_4})$ as required.

Finally if **CUV** is applicable then some participant exists with state $x \in \{\mathbf{c}, \mathbf{a}\}$ in $\tilde{s}_1$ with count m and in the coordinator's view $\tilde{v}_1$ the count of participants in state x is less than m. This means for all $\tilde{s}_3, t_3^{\tilde{v}_3} \in G(\tilde{s}_1, t_1^{\tilde{v}_1})$ a processes in state x will be in $\tilde{s}_3$ and the coordinator's view of that process in $\tilde{v}_3$ will be $\mathbf{i}$ therefore **CUV** will be applicable resulting in a $\tilde{v}_4$ with the required property that $\tilde{s}_2, t_2^{\tilde{v}_2} = f_n(\tilde{s}_4, t_4^{\tilde{v}_4})$.

□

Corollary 2 If in MULTI $C'_1 \xrightarrow{\mathbf{R_1}} C'_2 \xrightarrow{\mathbf{R_2}} \ldots \xrightarrow{\mathbf{R_{n-1}}} C'_n \xrightarrow{\mathbf{R_n}} \ldots$ then in CON for all $C_1 \in G(C'_1)$

$$\begin{array}{ccccccccc} C_1 & \xrightarrow{\mathbf{R_1}} & C_2 & \xrightarrow{\mathbf{R_2}} & \ldots & \xrightarrow{\mathbf{R_{n-1}}} & C_n & \xrightarrow{\mathbf{R_n}} \\ \uparrow & & | & & & & | & \\ G & & f & & & & f & \\ | & & \downarrow & & & & \downarrow & \\ C'_1 & \xrightarrow{\mathbf{R_1}} & C'_2 & \xrightarrow{\mathbf{R_2}} & \ldots & \xrightarrow{\mathbf{R_{n-1}}} & C'_n & \xrightarrow{\mathbf{R_n}} \end{array}$$

such that $C'_i = f(C_i)$ for $2 \le i \le n$.

□

Corollary 1 and 2 are useful as they allow us to infer properties of executions in the CON representation by showing that they hold or fail for executions using the MULTI representation. Informally, if a property fails for an execution using the MULTI representation then by corollary 1 the same execution exists in CON, and so that property must also fail in CON. Similarly, if a property holds in MULTI no execution in CON provides a counter example because by corollary 2 the same counter example would then also exist in the MULTI representation. Of course in order for this technique to work we must restrict our atomic sentences to those preserved by f and G. A formal proof of this argument is given in theorem 3.

5.4 Expressing Properties of Protocols

In the previous sections we have shown how to generate a labelled transition systems from a views model. We can generate transition systems using various levels of abstraction and have proved simulation relationships between the different abstractions, for a particular model 2PC. In this section we show how to verify or refute temporal properties of protocols by model checking the resulting transition systems. For example, suppose we want to show that a commit protocol has the desirable property of atomicity. That is, in any execution, if one participant decides commit (abort) no other participant may decide abort (commit). To show this we need to verify that in no execution (traversal of the transition system from the initial state) can we reach a state where one participant is in **a** while another is in **c**. Our views based model has the advantage that we can make "knowledge statements". For example we can express that if the coordinator believes (has a view) that all participants have voted **yes** then the decision will be to commit. More generally, for any property Φ we say

$$C \models \Phi$$

if property Φ holds of a configuration C. Since temporal logic expresses the capability of actions over time, we are able to determine properties of protocol executions by determining if a temporal property holds of the initial configuration C_0. For example, in order to express that a commit protocol never aborts we express that C_0 has the property that in no reachable configurations from C_0 can we reach a state where a process aborts.

The temporal logic we use to express properties is a subset and slight variant of computation tree temporal logic, CTL due to Clarke, Emerson and Sistla [16]. We call this subset CTL$^-$. In the following let $\mathbf{R}$ ranges over rules.

In this and the next section we define CTL$^-$ relative to some arbitrary configuration scheme C and set of paths, over these configurations, P. The set of paths P is suffix closed, that is if $C_0 \xrightarrow{\mathbf{R}_1} C_1 \xrightarrow{\mathbf{R}_2} C_2 \xrightarrow{\mathbf{R}_3} \in \mathsf{P}$ then $C_1 \xrightarrow{\mathbf{R}_2} C_2 \xrightarrow{\mathbf{R}_3} \in \mathsf{P}$. Later we will "plug in" our configurations schemes CON and MULTI and the paths generated from the protocol rules, to show how to apply CTL$^-$ to prove properties of our protocol. For now though the discussion of CTL$^-$ is independent of a particular configuration scheme.

A property in our logic can be expressed as follows.

$$\begin{array}{rl} \Phi \ ::= & \mathtt{tt} \mid X \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid [\mathbf{R}^+]\Phi \\ & \mathrm{A}(\Phi_1 \,\mathrm{U}\, \Phi_2) \mid \mathrm{E}(\Phi_1 \,\mathrm{U}\, \Phi_2) \end{array}$$

The definition of satisfaction between a configuration C_0 and a formula proceeds by induction on the formula.

$$\begin{array}{lll}
C_0 \models \mathtt{tt} & & \\
C_0 \models X & \text{iff} & X \text{ holds of } C_0 \text{ with respect to some valuation function} \\
C_0 \models \neg\Phi & \text{iff} & C_0 \not\models \Phi \\
C_0 \models \Phi \wedge \Psi & \text{iff} & C_0 \models \Phi \text{ and } C_0 \models \Psi \\
C_0 \models [\mathbf{R}^+]\Phi & \text{iff} & \forall C_1,\ C_0 \xrightarrow{\mathbf{R}^+} C_1,\ C_1 \models \Phi \\
C_0 \models \mathrm{A}(\Phi \,\mathrm{U}\, \Psi) & \text{iff} & \text{for all runs } C_0 \xrightarrow{\mathbf{R}_1} C_1 \xrightarrow{\mathbf{R}_2} \ldots \\
& & \text{there is } i \geq 0 \text{ with } C_i \models \Psi \text{ and} \\
& & \text{for all } j : 0 \leq j < i,\ C_j \models \Phi \\
C_0 \models \mathrm{E}(\Phi \,\mathrm{U}\, \Psi) & \text{iff} & \text{for some run } C_0 \xrightarrow{\mathbf{R}_1} C_1 \xrightarrow{\mathbf{R}_2} \ldots \\
& & \text{there is } i \geq 0 \text{ with } C_i \models \Psi \text{ and} \\
& & \text{for all } j : 0 \leq j < i,\ C_j \models \Phi
\end{array}$$

We let $\mathtt{ff}$ abbreviate $\neg\mathtt{tt}$, $\langle\mathbf{R}^+\rangle\Phi$ abbreviate $\neg[\mathbf{R}^+]\neg\Phi$ and $\Phi \vee \Psi$ abbreviate $\neg(\neg\Phi \wedge \neg\Psi)$, and so the derived clauses for these abbreviations are

$$\begin{array}{lll}
C_0 \not\models \mathtt{ff} & & \\
C_0 \models \langle\mathbf{R}^+\rangle\Phi & \text{iff} & C_0 \xrightarrow{\mathbf{R}^+} C_1,\ C_1 \models \Phi \\
C_0 \models \Phi \vee \Psi & \text{iff} & C_0 \models \Phi \text{ or } C_0 \models \Psi
\end{array}$$

A useful formula is $\neg\text{E}(\texttt{tt}\,\text{U}\,\Psi)$ this expresses the property that in all runs Ψ *fails* at all points. We can derive a definition for this formula as follows.

$$C_0 \models \neg\text{E}(\texttt{tt}\,\text{U}\,\Psi) \quad \text{iff} \quad \begin{array}{l} \text{for all runs } C_0 \xrightarrow{\mathbf{R}_1} C_1 \xrightarrow{\mathbf{R}_2} \dots \\ \text{for all } i \geq 0,\ C_i \models \neg\Psi \end{array}$$

Strong liveness properties are expressed using $A(\Phi\,\text{U}\,\Psi)$. The formula $A(\texttt{tt}\,\text{U}\,\Psi)$ means in all paths eventually Ψ. In CTL^- we often write this as $AF(\Psi)$.

Strong safety properties are expressed using $\neg E(\Phi\,\text{U}\,\Psi)$. For example if we were using the paths generated by our 2PC protocol and X was the sentence "A participant is in **c** and another is in **a**", then the formula $\neg E(\texttt{tt}\,\text{U}\,X)$, which is often abbreviated as $\neg EF(X)$, expresses the inability in any execution to reach a state where X holds i.e. atomicity is ensured.

The more expressive logic CTL includes an explicit next operator $\Box\Phi$. This expresses that Φ will be true in all next states. More formally it is defined as follows.

$$C_0 \models \Box\Phi \quad \text{iff} \quad \forall\mathbf{R}, \forall C_1, C_0 \xrightarrow{\mathbf{R}} C_1\, C_1 \models \Phi$$

We do not wish to include this operator in our logic because it introduces a form of counting. A possible further abstraction of MULTI removes counters from the configurations. So as to future proof our logic and safe gaurd its applicability we use a logic that does not allow counting properties to be expressed.

5.5 Games-Based Model Checking

The "property checking game" $G(C,\Phi)$, when C is a system configuration and Φ is a formula, is played by two players, players R (the refuter) and V (the verifier). Player R attempts to show that C fails to have the property Φ whereas player V wishes to establish that the property holds of C.

A play of the game $G(C_0,\Phi_0)$ is a finite or infinite length sequence of the form

$$(C_0,\Phi_0)\dots(C_n,\Phi_n)\dots$$

where each formula Φ_i is a subformula[5] of Φ_0, and each C_i is a configuration. If part of a play is $(C_0,\Phi_0)\dots(C_j,\Phi_j)$ then the next move and which player makes it depends on the main connective of the formula Φ_j. We write $C_j \xrightarrow{\mathbf{R}}$ if $\exists\mathbf{R}$, $\exists C'$, $C_j \xrightarrow{\mathbf{R}} C'$. All the possibilities are presented below.

- if $\Phi_j = \Psi_1 \wedge \Psi_2$ then player R chooses one of the conjuncts Ψ_i, $i \in \{1,2\}$: the state C_{j+1} is C_j and Φ_{j+1} is Ψ_i.

[5] A formula is the composition of subformulas using connectives. For example, Φ and Ψ are subformulas of the formula $\Phi \wedge \Psi$.

- if $\Phi_j = (\Psi_1 \vee \Psi_2)$ then player V chooses one of the disjuncts Ψ_i, $i \in \{1, 2\}$: the state C_{j+1} is C_j and Φ_{j+1} is Ψ_i.
- if $\Phi_j = [\mathbf{R}^+]\Psi$ then player R chooses any non-zero number of $\mathbf{R}$ transitions $C_j \xrightarrow{\mathbf{R}^+} C_{j+1}$ and Φ_{j+1} is Ψ.
- if $\Phi_j = \langle\mathbf{R}^+\rangle\Psi$ then player V chooses any non-zero number of $\mathbf{R}$ transitions $C_j \xrightarrow{\mathbf{R}^+} C_{j+1}$ and Φ_{j+1} is Ψ.
- if $\Phi_j = \neg\neg\Psi$ then C_{j+1} is C_j and Φ_{j+1} is Ψ.
- if $\Phi_j = \mathrm{A}(\Psi_1 \mathrm{U} \Psi_2)$ and not $(C_j \xrightarrow{\mathbf{R}})$ then $\Phi_{j+1} = \Psi_2$ and $C_{j+1} = C_j$.
- if $\Phi_j = \mathrm{A}(\Psi_1 \mathrm{U} \Psi_2)$ and $C_j \xrightarrow{\mathbf{R}}$ then player V chooses;
  1. $\Phi_{j+1} = \Psi_2$ and $C_{j+1} = C_j$ or
  2. allows player R to choose
     (a) $\Phi_{j+1} = \Psi_1$ and $C_{j+1} = C_j$ or
     (b) any rule $\mathbf{R}$ and a transition such that $C_j \xrightarrow{\mathbf{R}} C_{j+1}$ and $\Phi_{j+1} = \Phi_j$.
- if $\Phi_j = \neg\mathrm{A}(\Psi_1 \mathrm{U} \Psi_2)$ and not$(C_j \xrightarrow{\mathbf{R}})$ then $\Phi_{j+1} = \neg\Psi_2$ and $C_{j+1} = C_j$.
- if $\Phi_j = \neg\mathrm{A}(\Psi_1 \mathrm{U} \Psi_2)$ and $C_j \xrightarrow{\mathbf{R}}$ then player R chooses;
  1. $\Phi_{j+1} = \neg\Psi_2$ and $C_{j+1} = C_j$ or
  2. allows player V to choose
     (a) $\Phi_{j+1} = \neg\Psi_1$ and $C_{j+1} = C_j$ or
     (b) any rule $\mathbf{R}$ and a transition such that $C_j \xrightarrow{\mathbf{R}} C_{j+1}$ and $\Phi_{j+1} = \Phi_j$.
- if $\Phi_j = \mathrm{E}(\Psi_1 \mathrm{U} \Psi_2)$ and not$(C_j \xrightarrow{\mathbf{R}})$ then $\Phi_{j+1} = \Psi_2$ and $C_{j+1} = C_j$.
- if $\Phi_j = \mathrm{E}(\Psi_1 \mathrm{U} \Psi_2)$ and $C_j \xrightarrow{\mathbf{R}}$ then player V chooses;
  1. $\Phi_{j+1} = \Psi_2$ and $C_{j+1} = C_j$ or
  2. allows player R to choose
     (a) $\Phi_{j+1} = \Psi_1$ and $C_{j+1} = C_j$ or
     (b) force player V to choose a transition such that $C_j \xrightarrow{\mathbf{R}} C_{j+1}$ and $\Phi_{j+1} = \Phi_j$.
- if $\Phi_j = \neg\mathrm{E}(\Psi_1 \mathrm{U} \Psi_2)$ and not$(C_j \xrightarrow{\mathbf{R}})$ then $\Phi_{j+1} = \neg\Psi_2$ and $C_{j+1} = C_j$.
- if $\Phi_j = \neg\mathrm{E}(\Psi_1 \mathrm{U} \Psi_2)$ and $C_j \xrightarrow{\mathbf{R}}$ then player R chooses;

Player R wins

1. The play is $(C_0, \Phi_0) \ldots (C_n, \Phi_n)$ and

   - $\Phi_n = \texttt{ff}$ or
   - $\Phi_n = \langle \mathbf{R}^+ \rangle \Psi$ and not $\exists C'$, $C_n \xrightarrow{\mathbf{R}+} C'$ or
   - $\Phi_n = X$ and X is not true at configuration C_n.

2. The play $(C_0, \Phi_0) \ldots (C_n, \Phi_n) \ldots$ has infinite length and there is an until formula $\mathrm{A}(\Psi_1 \mathrm{U} \Psi_2)$ or $\mathrm{E}(\Psi_1 \mathrm{U} \Psi_2)$ which occurs infinitely often.

Player V wins

1. The play is $(C_0, \Phi_0) \ldots (C_n, \Phi_n)$ and

   - $\Phi_n = \texttt{tt}$ or
   - $\Phi_n = [\mathbf{R}^+] \Psi$ and not $\exists C'$, $C_n \xrightarrow{\mathbf{R}+} C'$ or
   - $\Phi_n = X$ and X is true at configuration C_n.

2. The play $(C_0, \Phi_0) \ldots (C_n, \Phi_n) \ldots$ has infinite length and there is a negated until formula $\neg\mathrm{A}(\Psi_1 \mathrm{U} \Psi_2)$ or $\neg\mathrm{E}(\Psi_1 \mathrm{U} \Psi_2)$ which occurs infinitely often.

Figure 5.2: Winning conditions

1. $\Phi_{j+1} = \neg\Psi_2$ and $C_{j+1} = C_j$ or
2. allows player V to choose
   (a) $\Phi_{j+1} = \neg\Psi_1$ and $C_{j+1} = C_j$ or
   (b) force player R to choose a transition such that $C_j \xrightarrow{\mathbf{R}} C_{j+1}$ and $\Phi_{j+1} = \Phi_j$

The rules appear to be complicated because of the presence of negation in the logic. The refuter chooses the next position when the formula has the form $\Psi_1 \wedge \Psi_2$ or $[\mathbf{R}^+]\Psi$ and the verifier chooses when it has the form $(\Psi_1 \vee \Psi_2)$ or $\langle \mathbf{R}^+ \rangle \Psi$. In the case of $A(\Psi_1 \mathrm{U} \Psi_2)$ and $E(\Psi_1 \mathrm{U} \Psi_2)$ an initial choice is made by player V but V may choose to defer this choice to R. Likewise in the cases $\neg A(\Psi_1 \mathrm{U} \Psi_2)$ and $\neg E(\Psi_1 \mathrm{U} \Psi_2)$ player R has an initial choice but may choose to defer this choice to player V. As there are no choices in the remaining rules neither player is responsible for them. The first of these reduces a double negation.

Figure 5.2 captures when a player is said to win a play of a game. Player R wins if a blatantly false position is reached and player V wins if a blatantly true position is reached. Condition 2 identifies which player wins an infinite length

play. For any infinite length play of a CTL$^-$ game there is only one until formula or negation of an until formula which occurs infinitely often. It is this formula which decides who wins. If it is an until formula then it is the refuter that wins and if it is the negation of an until formula then it is the verifier that wins. To detect when a formula and configuration can appear infinitely often we need only detect repeats.

A strategy for a player is a family of rules which tell the player how to move. It suffices to consider history-free strategies, whose rules do not depend upon previous positions in the play. For player R rules have the following form.

- At position $(C, \Phi_1 \wedge \Phi_2)$ choose (C, Φ_i) where $i \in \{1, 2\}$.
- At position $(C, [\mathbf{R}^+]\Phi)$ choose (C', Φ) where $C \xrightarrow{\mathbf{R}^+} C'$.
- At position $(C, \neg A(\Phi_1 \mathrm{U} \Phi_2))$ or $(C, \neg E(\Phi_1 \mathrm{U} \Phi_2))$ choose one of the two options discussed earlier possibly deferring a choice to V.

The verifier rules have a similar form.

- At position $(C, \Phi_1 \vee \Phi_2)$ choose (C, Φ_i) where $i \in \{1, 2\}$.
- At position $(C, \langle\mathbf{R}^+\rangle\Phi)$ choose (C', Φ) where $C \xrightarrow{\mathbf{R}^+} C'$.
- At position $(C, A(\Phi_1 \mathrm{U} \Phi_2))$ or $(C, E(\Phi_1 \mathrm{U} \Phi_2))$ choose one of the two options discussed earlier possibly deferring a choice to R.

A player uses the strategy π in a play if all her moves in the play obey the rules in π. The strategy π is winning if the player wins every play in which she uses π. The following result provides an alternative account of the satisfaction relation between system configurations and formulas.

Proposition 3 1. *If $\Phi \in$ CTL$^-$ then $C \models \Phi$ iff player V has a history-free winning strategy for $G(C, \Phi)$.*

2. *If $\Phi \in$ CTL$^-$ then $C \not\models \Phi$ iff player R has a history-free winning strategy for $G(C, \Phi)$.*

Proof: Follows from [77].

□

5.6 Applying CTL$^-$ to CON and MULTI

In this section we set about model checking CTL$^-$ properties of the 2PC protocol using the different representations CON and MULTI. Recall we defined CTL$^-$ with reference to a set of valid configurations and paths over these configurations. When checking properties using the CON representation, the configurations are those of CON and the paths are those generated by the application of rules in CON. Likewise for MULTI configurations we use the MULTI configuration and paths generated by applications of the rules in MULTI.

At this point we will introduce labelled atomic sentences $X, Y, \ldots$. We restrict these sentences to those preserved by the transformation, G and f that we used previously to map between CON and MULTI. More formally X is preserved if for any MULTI configuration C_1

$$C_1 \models X \quad \text{iff} \quad \forall C_2 \in G(C_1),\, C_2 \models X$$

and for any CON representation C_1

$$C_1 \models X \quad \text{iff} \quad f(C_1) \models X$$

Examples of atomic sentences, X, that are preserved include, "the coordinator is in state **c**" or, "all participants are in state **a**".

The next theorem expresses the relationship between model checking using our two representations. They allow us to model check properties in the more abstract and efficient representation of MULTI and then infer properties in the more concrete representation of CON.

Theorem 3 For any configuration C

$$C \models \Phi \text{ in CON iff } f(C) \models \Phi \text{ in MULTI}$$

Proof The proof is by induction on the structure of Φ and makes use of corollaries 1 and 2.

Base Cases: If $\Phi = \texttt{tt}$ trivially true. If $\Phi = X$ then if follows because X is preserved by f and G.

Induction:

case $\Phi = \neg\Psi$.

> $C \models \neg\Psi$ iff $C \not\models \Psi$ therefore by the induction hypothesis iff $f(C) \not\models \Psi$ iff $f(C) \models \Phi$.

case $\Phi = \Psi_1 \wedge \Psi_2$.

$C \models \Psi_1 \wedge \Psi_2$ iff $C \models \Psi_1$ and $C \models \Psi_2$ therefore by the induction hypothesis iff $f(C) \models \Psi_1$ and $f(C) \models \Psi_2$ and iff $f(C) \models \Phi$.

case $\Phi = [\mathbf{R}^+]\Psi$.

$\Rightarrow$ Assume $C \models [\mathbf{R}^+]\Psi$ in CON but $f(C) \not\models [\mathbf{R}^+]\Psi$ in MULTI so $f(C) \xrightarrow{\mathbf{R}^+} C_3$ and $C_3 \not\models \Psi$ in MULTI. By corollary 2 if $f(C) \xrightarrow{\mathbf{R}^+} C_3$ in MULTI then $C \xrightarrow{\mathbf{R}^+} C_2$ in CON with $f(C_2) = C_3$. Applying the induction hypothesis $C_2 \not\models \Psi$ and so $C \not\models [\mathbf{R}^+]\Psi$ a contradiction.

$\Leftarrow$ Assume $f(C) \models [\mathbf{R}^+]\Psi$ in MULTI but $C \not\models [\mathbf{R}^+]\Psi$ in CON so $C \xrightarrow{\mathbf{R}^+} C_2$ in CON and $C_2 \not\models \Psi$. By Corollary 1 $f(C) \xrightarrow{\mathbf{R}^+} C_3$ in MULTI and $C_2 \in G(C_3)$. Applying the induction hypothesis $C_3 \not\models \Psi$ and so $f(C) \not\models [\mathbf{R}^+]\Psi$ a contradiction.

case $\Phi = A(\Psi_1 \mathrm{U} \Psi_2)$ or $\Phi = E(\Psi_1 \mathrm{U} \Psi_2)$. We prove the case for $\Phi = A(\Psi_1 \mathrm{U} \Psi_2)$, $\Phi = E(\Psi_1 \mathrm{U} \Psi_2)$ being very similar.

$\Rightarrow$ Assume $C \models A(\Phi_1 \mathrm{U} \Phi_2)$ in CON but $f(C) \not\models A(\Phi_1 \mathrm{U} \Phi_2)$ in MULTI. If $f(C) \not\models A(\Phi_1 \mathrm{U} \Phi_2)$ then there exists a path $f(C) = C_0 \xrightarrow{\mathbf{R_0}} C_1 \xrightarrow{\mathbf{R_1}} \ldots \xrightarrow{\mathbf{R_{n-1}}} C_n \xrightarrow{\mathbf{R_n}} \ldots$ such that either Φ_1 fails before Φ_2 holds or Φ_2 never holds. Consider the first case the second being similar. So there exists $n \geq 0$ such that $C_n \not\models \Phi_1$ and $C_n \not\models \Phi_2$ and furthermore for each $i < n$, $C_i \models \Phi_1$ and $C_i \not\models \Phi_2$. For each prefix of the path apply Corollary 2 and so for each $D \in G(f(C))$ there is a path in CON $D_0 \xrightarrow{\mathbf{R_0}} D_1 \xrightarrow{\mathbf{R_1}} \ldots \xrightarrow{\mathbf{R_{n-1}}} D_n \xrightarrow{\mathbf{R_n}} \ldots$ such that $f(D_i) = C_i$. Applying induction hypothesis $D_n \not\models \Phi_1$ and $D_n \not\models \Phi_2$ and furthermore for each $i < n$, $D_i \models \Phi_1$ and $D_i \not\models \Phi_2$, therefore $C \not\models A(\Phi_1 \mathrm{U} \Phi_2)$ a contradiction.

$\Leftarrow$ Assume $f(C) \models A(\Psi_1 \mathrm{U} \Psi_2)$ in MULTI but $C \not\models A(\Phi_1 \mathrm{U} \Phi_2)$ in CON. If $C \not\models A(\Phi_1 \mathrm{U} \Phi_2)$ then there exists a path $C = C_0 \xrightarrow{\mathbf{R_0}} C_1 \xrightarrow{\mathbf{R_1}} \ldots \xrightarrow{\mathbf{R_{n-1}}} C_n \xrightarrow{\mathbf{R_n}} \ldots$ in CON such that either Φ_1 fails before Φ_2 holds or Φ_2 never holds. For each prefix of the path apply corollary 2 and so there exists a path $f(C) = D_0 \xrightarrow{\mathbf{R_0}} D_1 \xrightarrow{\mathbf{R_1}} \ldots \xrightarrow{\mathbf{R_{n-1}}} D_n \xrightarrow{\mathbf{R_n}} \ldots$ in MULTI such that $C_i \in G(D_i)$. Applying induction hypothesis at each point in the MULTI path we have Φ_1 fails before Φ_2 holds or Φ_2 never holds, therefore $f(C) \not\models A(\Phi_1 \mathrm{U} \Phi_2)$. The cases for $\Phi = E(\Phi_1 \mathrm{U} \Phi_2)$ is very similar.

□

Theorem 3 allows us to model check properties in the more compact MULTI representation and then infer properties in concrete representations. If for example a property holds in MULTI we know that property holds in CON.

5.7 Checking CTL$^-$ Properties of Two Phase Commit

As a simple example let ONE(x) be the set of atomic sentences "there exists a participant p with $p.s = x^*$", $p.s = x^*$ means that the participant is in state x with any view of the coordinator, and let ALL(x) be the set of atomic formaulas "for all participants p, $p.s = x^*$, where $x \in \{\mathbf{a}, \mathbf{c}, \mathbf{w}, \mathbf{i}\}$. We can see that both ONE(x) and ALL(x) are preserved by the abstraction translations, f and G.

The safety property $\neg EF(\text{ONE}(\mathbf{c}) \wedge \text{ONE}(\mathbf{a}))$ expresses the property that in any protocol execution a state is never reached where one participant has committed and another has aborted.

We model checked this property, using an implementation of the games based algorithm above, against our simple two-phase commit protocol. As expected the verifier V produced a history free winning strategy. If we add the following rule to our system

$$\mathbf{PVY}(p)\ \frac{s = \mathbf{i}}{s := \mathbf{c}}$$

and check the same property we find that the refuter has a winning strategy. This is to be expected because participants are now able to unilaterally commit as well as abort.

A more complicated property makes use of views. Suppose we wish to verify that in any run if the coordinator has a view that all participants have voted **yes** then eventually all participants will commit. Let Z be the property that the coordinator's view is that all participants are have voted **yes**. Z is preserved, because f and G maintain the validity of the coordinators view of participant's state. We are are then required to model check the following formula.

$$\neg EF(Z \wedge \neg AF(\text{ALL}(\mathbf{c})))$$

Since this is an example of a liveness property we can see by figure 5.2 that if a negated until formula is repeated then the verifier has a winning strategy.

In chapter 2 we saw an axiomatic specification for the correctness of a commit protocol due to Berstein *et al.* [9]. We can use our model checker to verify these axioms. We have already verified **AC1** which is atomicity. **AC2** is implicit in our model since a participant can only vote once. **AC3** states that "if any participant decides **c** then all participants voted **yes**". This follows from the property that in no run can any participant vote **no** and then from that point some site eventually commit which we express as follows.

$$\neg EF(\text{ONE}(\mathbf{a}) \wedge EF(\text{ONE}(\mathbf{c})))$$

Since we do not model failure we can interpret **AC4** as the statement "If all participants vote **yes**, then the decision will be commit" which we can express as follows.

$$\neg EF(\text{ALL}(\mathbf{w}) \wedge \neg AF(\text{ALL}(\mathbf{c})))$$

AC5 deals with failure which is not modelled in our simple 2PC but its analogy might be that eventually a decision is reached by all participants which we can express as follows.

$$AF(\text{ALL}(\mathbf{c}) \vee \text{ALL}(\mathbf{a}))$$

Games-based model checking has the advantage of producing a strategy that proves or refutes the property being checked. This provides a protocol designer with a design cycle. For example, if a safety property were to fail a designer can "play" the game and see exactly why the property fails. This then leads to modifications of the original protocol and re-verification. Another useful property of the games-based approach is that, because the game is played only until a winning strategy is found, in many cases the entire transition system need not be generated ie. the transition system can be generated on demand.

5.8 Conclusions

We have shown how a simple views based model can be used to construct a labelled transition system using different abstraction techniques. The most concrete approach led to large state spaces. A slightly less concrete approach reduced the state space. Using a simulation argument we show that for many properties model checking in the abstract system is equivalent to model checking in the concrete system. We defined a sub-logic CTL$^-$ that captured these properties and also developed algorithms to automatically generate transition systems from views based models, so that our models could support automatic model checking of these properties.

Because our views based model captures a processes belief of another processes state explicitly within its own state, we are able to express belief type properties within CTL$^-$. For example, we can express properties like "If the coordinator *believes* that all participants have voted **yes** then it will eventually enter its **c** state."

We will see in following chapters that our technique could be used for more complex protocols. Another avenue to explore is to introduce failure for example the partitioning network model of the previous chapter, into our models. If we were to do this many interesting properties of protocols might emerge.

Chapter 6

Diluting ACID

6.1 Introduction

We have seen the role a commit protocol plays in guaranteeing the atomicity of a set of actions in a transaction. Equally important is the role it plays when interacting with concurrency control mechanisms to guarantee a particular level of transaction isolation, i.e. the 'I' in the ACID properties of transactions.

So far, we have only looked at transactions from a single transaction point-of-view. If only one transaction happens at a time, then each transaction occurs serially and does not have to contend with interference from other transactions. However, with many transaction execute at the same time, transactions can occur simultaneously and each transaction has the potential to interfere with another one. Transactions that have the potential of interfering with one another are said to be interleaved or parallel transactions. When transactions are interleaved, mechanisms often exist to protect them interfering with one another. The level to which they are allowed to interfere is the level of transaction isolation that they attain. Informally, transactions that run totally isolated from each other are said to be serializable, which means that the results of running them simultaneously will be the same as the results of running them one right after another (serially). In this chapter we will provide a formal definition of some different types of transaction isolation.

The life-cycle of a distributed transaction can be thought of as an operational phase, where transactions perform read and write accesses to data objects followed by[1] the invocation of a commit protocol. Up until this point our models focused on the commit phase of a single transaction while ignoring the operational phase. In order to study the role of commit protocols in transaction isolation we must enrich our models in two ways. Firstly, we must include read and write accesses

[1] We will see in the next chapter than sometimes these phases can be overlapped.

to simple data objects prior to the invocation of a commit protocol, and secondly, we must also model the concurrent interaction of many transactions rather than just considering the case of a single transaction.

Before we introduce a views based model of a very simple distributed transaction processing environment, that includes these features, we must first define what we mean by transaction isolation. Unfortunately, the current literature is inconsistent on the subject. Most definitions appeal to specific concurrency control techniques, in particular lock management schemes such as strict two phase locking (S2PL). In response to this, different levels of isolation were defined in part of the ANSI SQL 92 standard [3] and many database and transaction processing vendors have implemented this facility, or something similar. In order to remain neutral to particular concurrency control implementations the ANSI community constructed a definition based on "phenomena". In their definition, a level of isolation is achieved when a set of phenomena, is prevented from occurring. For example, a basic phenomenon is that one transaction cannot write to the same data object as another, until the first transaction has terminated, that is committed or aborted.

In order to simplify our account of transaction isolation we first consider the case when our transactions are not distributed. For this centralised case transactions do not require a commit protocol. Although it may seem we have departed from the analysis of the behaviour of commit protocols, we will see that the framework we construct in this chapter for centralised transactions can be generalised to thė distributed case. In the next chapter we will re-introduce commit protocols and study the role they play in providing levels of transaction isolation using our views based modelling techniques combined with games-based model checking. Another motivating factor for this aside comes from the realisation that the current literature on transaction isolation is rather unclear. To analyse the role of commit protocols in transaction isolation we must have a clear idea of what attaining a particular isolation level actually means.

A classical theory of serializability (the highest level of transaction isolation[2]), and the related theory of recoverability has been in existence for quite some time. In the next section 6.2 we will we will briefly survey some relevant literature in the field. We base our discussion on the seminal work found in [9] and [34]. In section 6.3 we introduce some notation for reasoning about the order of a transaction's actions. In section 6.4 we extend the classical theory of conflict seri-

[2]Unfortunately, the ANSI community muddied the waters by calling this level of isolation REPEATABLE READ and using the term SERIALIZABLE when predicates (discussed later) are included.

alizability to include the commit or abort outcome of a transaction. Berenson *et al.* [6] criticise the ANSI standard and give a more precise phenomena based definition that corrects some of the deficiencies they discovered. In section 6.5 we describe their work and in turn criticise their phenomenon based definitions of isolation and construct our own definition that we claim better captures isolation levels. In section 6.6 we prove that our new definition coincides with the extended definition of conflict serializability we gave in section 6.4. For completeness we enrich our model with predicate read and write accesses allowing us to complete our discussion by defining the four levels of transaction isolation (see table 6.1), discussed in the ANSI standard, using the theory we develop.

6.2 Classical Recoverability and Serializability Theory

Concurrency control is the activity of coordinating the actions of transactions that access shared data objects, and therefore potentially interfere with one another. Recovery is the activity of ensuring that system failures do not corrupt data. Concurrency control and recovery problems arise in the design of hardware, operating systems, realtime systems, middleware systems, and database systems among others. In order to hide many of the details of the particular system we are interested in, a transaction is modelled as sequence of four different types of actions, sometimes called a schedule[3]. We write $\mathtt{read}_t(x)$ to denote a transaction t, reading a data object x and $\mathtt{write}_t(x, v)$[4] to denote a write of value v to x. Once t has completed all of its read and write accesses it terminates by either committing the transaction by performing a `commit` action, or aborting the transaction by performing an `abort` action.

6.2.1 Recoverability

Recoverability guarantees that data objects contain all the effects of committed transactions and none of the effects of uncommitted[5] ones. If transactions never abort, recoverability is rather simple to ensure because all transaction eventually commit, the access of each transaction can just be carried out as they arrive. To understand recovery therefore we must look at the processing of aborts. When a transaction aborts all the effects of the transaction must be wiped out. The

[3]Strictly speaking a schedule is really a partial order.

[4]The actual values read or written are not always important to our discussion of concurrency control and recoverability and so they are sometimes omitted.

[5]An uncommitted transaction is either aborted or not yet terminated.

effects of a transaction t are twofold: effects on data objects due to write accesses carried out by t; and effects on other transactions, namely, transactions that read values written by t. For example, suppose the initial values of data objects x and y are[6] 2, and suppose transactions t_1 and t_2 issue accesses that are executed in the following order.

$$\texttt{write}_1(x,3)\ \texttt{read}_2(x)\ \texttt{write}_2(y,3)$$

Now suppose t_1 aborts. This means that the $\texttt{write}_1(x,3)$ must be undone, restoring x to its initial value 2 and because t_2 read the value of x written by t_1 it must also be aborted. This is sometimes referred to as a *cascading abort* because $\texttt{write}_2(y,3)$ must also be undone due to t_1's abort.

Because a committed transaction cannot be subsequently aborted it is important, given the possibility of cascading aborts, that a transaction is not committed when there is a chance it may later be required to be aborted. Therefore, a transaction, t, cannot commit until all the transactions that wrote values read by t are guaranteed not to abort, that is, are themselves committed. Executions that satisfy this condition are called *recoverable.* More formally we say a transaction t_j reads x from transaction t_i in an execution, if

1. t_j reads x after t_i has written to it and;
2. t_i does not abort before t_j reads x and;
3. every transaction (if any) that writes to x between the time t_i writes it and t_j reads it, aborts before t_j reads it.

A transaction t_j reads from t_i if t_j reads any data object from t_i. An execution is recoverable if, for every transaction t that commits, t's commit follows the commit of every transaction from which t read. Consider the following example.

$$\texttt{write}_1(x,2)\ \texttt{read}_2(x)\ \texttt{write}_2(y,3)\ \texttt{commit}_2$$

This is not recoverable, because t_2 read x from t_1 and yet the commit of t_2 does not follow the commit of t_1. A problem would arise if t_1 were now to abort.

Enforcing recoverability does not remove the possibility of cascading aborts. Consider the example below.

$$\texttt{write}_1(x,2)\ \texttt{read}_2(x)\ \texttt{write}_1(y,3)\ \texttt{abort}_1$$

[6] In all our examples the data objects we consider will be simple integer values but the results we derive are valid for any data types.

Although this execution is recoverable it requires cascading aborts because t_1 aborted and so to remain recoverable t_2 must now abort. To ensure that cascading aborts are avoided we must place the following further restriction on the order of actions. Every transaction must read only those values that were written by committed transactions. Thus, only committed transactions can affect other transactions. This means that each $\texttt{read}(x)$ must be delayed until all transactions that have previously issued a $\texttt{write}(x, v)$ have terminated. In so doing recoverability is also achieved. From a practical viewpoint, a further restriction is required. Consider the following execution sequence, assuming the value of x before the execution is 0.

$$\texttt{write}_1(x, 1)\ \texttt{write}_2(x, 3)\ \texttt{abort}_2\ \texttt{abort}_1$$

Aborting t_2 then t_1 should remove all the effects of both t_1 and t_2 and thus should restore the value of x to be 0. A very natural implementation is to undo a write by restoring its previous value. In this case when t_2 aborts it will be restored to 1 and then when t_1 aborts it will be restored to 0 as required. Consider the case where the abort actions are transposed as follows.

$$\texttt{write}_1(x, 1)\ \texttt{write}_2(x, 3)\ \texttt{abort}_1\ \texttt{abort}_2$$

We see now that simply restoring previous values fails. In order to prevent this from happening and ensuring that writes can be undone by restoring previous values we must place a further restriction on the order of actions. We must delay a write access on x until all transaction that have previously written x are either committed or aborted. This is similar to the requirement for avoiding cascading aborts. Executions that satisfy both these conditions are called *strict* [9].

From this short discussion we see that in order to implement the semantics of commit and abort, executions sequences must be recoverable. Due to practical considerations it is often preferable that they are also cascadeless. If in addition to this the undo of writes is to be implemented by replacing the value before the write took place then executions should also be strict.

6.2.2 Serializability

When the accesses of two or more transactions execute concurrently causing their actions to be interleaved, the interleaving can cause incorrect behaviour. To understand this consider the simple transaction that deposits a sum of money in a bank account with balance b in figure 6.1.

```
deposit(amount){
        t := read(b);
        t := t+amount;
        write(b, t);
        commit;
}
```

Figure 6.1: A simple deposit transaction.

Suppose the account has a balance of £1000 and customer 1 (transaction t_1) deposits £100 into this account, at the same time that customer 2 (transaction t_2) deposits £100, 000 into the same account. A possible execution sequence might be as follows.

$$\texttt{read}_1(b)\ \texttt{read}_2(b)\ \texttt{write}_2(b, £101,000)\ \texttt{commit}_2\ \texttt{write}_1(b, £1100)\ \texttt{commit}_1$$

The result is that the balance b contains £1100, although t_2 executed successfully its deposit of £100, 000 was lost. Clearly this is incorrect.

To avoid this and similar behaviour, the kinds of interleaving between transaction accesses must be controlled. One possible way to accomplish this is to insist that the actions of transactions are not interleaved. This leads to *serial* executions, i.e. all actions of a transaction t_i are either strictly before or after those of another transaction t_2. Unfortunately, this leads to very poor performance since the level of concurrency is severely restricted. A better approach is to only allow executions that have the *same effect as* a serial execution. Such an execution is called *serializable*. There are several different definitions of serializable [9, 10] but the most widely accepted is that of conflict serializability.

Two accesses, from different transaction t_i and t_j, are said to conflict if they both operate on the same data item and at least one of them is a write. Thus $\texttt{read}_i(x)$ conflicts with $\texttt{write}_j(x, v)$, while $\texttt{write}_i(x, v)$ conflicts with both $\texttt{read}_j(x)$ and $\texttt{write}_j(x, v)$. Using this definition of conflicting accesses execution sequences, σ_1 and σ_2 are defined to be equivalent when

1. they inlcude the same actions; and

2. they order conflicting accesses of *non-aborted* transactions in the same way; that is, for any conflicting access o_i and o_j belonging to t_i and t_j (respectively) where neither t_i nor t_j abort, if o_i is before o_j in σ_1 then o_i is before o_j in σ_2.

A conflict serializable history is now defined as one which is equivalent to a

serial history. In the following example the execution

$$\texttt{read}_1(x)\ \texttt{write}_2(x,3)\ \texttt{read}_1(y)\texttt{write}_2(y,4)\ \texttt{commit}_2\ \texttt{commit}_1$$

is conflict equivalent to the serial execution

$$\texttt{read}_1(x)\ \ \texttt{read}_1(y)\ \texttt{commit}_1\ \texttt{write}_2(x,3)\ \texttt{write}_2(y,4)\ \texttt{commit}_2$$

and is thus serializable whereas

$$\texttt{write}_2(x,3)\ \texttt{read}_1(x)\ \texttt{read}_1(y)\texttt{write}_2(y,4)\ \texttt{commit}_2\ \texttt{commit}_1$$

is not conflict equivalent to any serial schedule and so is not conflict serializable.

In much of the literature on the classical theory of serializability [9, 34] there is an assumption that a mechanism exists to ensure schedules are recoverable, and often that they avoid the possibility of cascading aborts[7]. Under this assumption the classical theory need not distinguish between two schedules, say σ_1 and σ_2, where the only difference is that, in σ_1 a transaction, t_2, reads a value for a data object that was written by another transaction, t_1, *before* t_1 aborted because such schedules are disallowed by recoverability constraints.

$$\sigma_1 : \ \texttt{write}_1(x,3)\,\texttt{read}_2(x)\,\texttt{commit}_2\,\texttt{abort}_1 \quad \sigma_2 : \ \texttt{write}_1(x,2)\,\texttt{abort}_1\,\texttt{read}_2(x)\,\texttt{commit}_2$$

In the above example σ_1 is disallowed by recoverability constraints because t_2 reads x from t_1 before t_1 terminates.

In the extensions to the classical theory we make in section 6.4 we do not presuppose a technique exists to ensure recoverability and so our account of isolation is independent of recoverability assumptions. In order to do this we include in the theory of conflicting actions the context of the outcome (i.e. either commit or abort) of the transactions in which these actions appear.

6.3 Modelling simple schedules

Let t_i, t_j denote transactions and d, d' denote data objects. It is assumed that $t_i \neq t_j$, unless otherwise stated, but we do not assume $d \neq d'$. A transaction, t_i, consists of actions. These actions are divided into four categories. The first two categories are read and write actions, which we call *accesses*, and which we denote r_i, w_i, respectively, or o_i to denote either a r_i or w_i. The second category are

[7] If cascading aborts are required by the concurrency control mechanism in place to ensure recoverability a mechanism is assumed to exist to detect when they are required and perform the necessary transaction aborts.

commit and abort actions which we call *terminals* and denote c_i, a_i respectively or e_i to denote either c_i or a_i. When a transaction commits, the changes it has made to the data objects are made durable, and the values it has read are returned to the user. If a transaction aborts, all write actions are undone leaving any data objects with the value that they would have had if the transaction had never executed, furthermore no read values are returned.

Accesses		Terminals	
$w_i[d]$	t_i writes d	c_i	t_i commits
$r_i[d]$	t_i reads d	a_i	t_i aborts

We assume each type of access within a transaction is to a unique data object[8] and also that exactly one terminal for each transaction occurs exactly once[9]. A *schedule*, s, is the sequence[10] of actions generated by transactions as they execute concurrently. We say $o_i[d] \prec o'_j[d]$ if an action $o_i[d]$ is earlier than an action $o'_j[d]$ in s. In any schedule no terminal of a transaction precedes an action of that transaction. An example of a schedule is $w_1[d]\, r_2[d]\, w_1[d']\, c_1\, c_2$. A *serial schedule* is one in which all actions of one transaction are completed before any action of another transaction is started, for instance $w_1[d]\, w_1[d']\, c_1\, r_2[d]\, c_2$.

By slightly abusing notation we write c_i (a_i) is true over a schedule if action c_i (a_i) happens at some point. We use $w_i[d] \prec c_j$, to denote that t_j commits and does so after a write action of t_i on data object d. We write $w_i[d] \prec e_j$, to mean that either t_j aborts or commits but does so after a write action by t_i, on data object d. Similarly, we write $r_j[d] \prec (a_i \wedge c_j)$, to say t_i aborts after a read of d by t_j and also t_j commits, note there is no assumed order between a_i and c_j in this case. Finally, we write $r_i[d] \prec w_j[d] \wedge (a_i \wedge c_j)$, to say t_i aborts and t_j commits and that $r_i[d]$ is before $w_j[d]$. It should be noted this allows a_i before or after $w_j[d]$.

6.4 Extended Conflict Serializability

To define serializability we must first define an equivalence over schedules. The most common and useful definition is that of conflict equivalence [9], which we discussed earlier in section 6.2. Unfortunately, this definition fails to capture the inequivalence of schedules containing aborting transactions. For example, in this

[8]The results in this chapter do not depend on this but it is a useful notational convenience.

[9]Schedules with this property are often called *complete schedules*.

[10]A schedule is also sometimes defined as a poset of actions, and sometimes called a history. We choose to define it as a sequence in order to keep it consistent with [6].

definition the following two schedules are defined to be equivalent.

$$w_1[x]\, r_2[x]\, a_1\, c_2 \equiv w_1[x]\, a_1\, r_2[x]\, c_2$$

The classical definition of conflict equivalence requires the ordering of conflicting accesses from committing transactions to be maintained, but says nothing about the ordering of actions of aborting transactions. To capture this behaviour we extend the classical definition of conflict equivalence by first extending the definition of a conflict.

Two transactions t_i and t_j are said to *conflict* on a data object, d, if both access d and transposing the order of these accesses on d *might* result in either a different value being read by one of the transactions, or a different value resulting at d after the transactions terminate. We enumerate all the possible types of conflict that can occur in a schedule below.

I $r_i[d] \prec w_j[d] \wedge (c_i \wedge c_j)$

II $w_i[d] \prec r_j[d] \wedge (c_i \wedge c_j)$

III $w_i[d] \prec w_j[d] \wedge (c_i \wedge c_j)$

IV $r_i[d] \prec w_j[d] \wedge (c_i \wedge a_j)$

V $w_i[d] \prec r_j[d] \prec (a_i \wedge c_j)$

We can see that we have extended the classical notion of a conflict to include the context of the outcome of the transactions. If we remove this context the five conflict types collapse into the classical three conflict types: read/write, write/write and write/read, that we saw in section 6.2.

At first sight type IV might not look like a conflict. If we reorder the accesses $w_j[d]$ and $r_i[d]$ we arrive at either

$$w_j[d] \prec a_j \prec r_i[d] \prec c_i$$

or

$$w_j[d] \prec r_i[d] \prec (c_i \wedge a_j).$$

depending on whether or not a_i happens before or after a_j. The first reordering does not change the values read or written by the transactions but in the second re-ordering it does. If recoverability mechanisms are assumed to be in place the second type of schedule could not occur because in the case $a_j \prec c_i$ t_j's abort action would cascade and cause t_i to abort and the case $c_i \prec a_j$, t_i cannot be

allowed to commit until t_j had terminated. Type IV conflicts are included because we seek a definition that does not assume recoverability.
The schedule

$$r_1[d]\, w_2[d]\, w_2[d']\, r_1[d']\, c_1\, a_2$$

has two conflicts, the first between $r_1[d]$ and $w_2[d]$, an instance of IV above, and the second between $w_2[d']$ and $r_1[d']$ which is an instance of V above. The extended definition of conflict equivalence naturally follows from the extended definitions of conflicts.

Definition 1 Schedules σ and σ' are *conflict equivalent* iff

- σ and σ' have the same actions and
- for each conflict of type $C \in \{I, \ldots, V\}$ involving actions o_i, o_j, e_i, e_j, in σ the same conflict of type C appears in σ' involving the same actions o_i, o_j, e_i, e_j.

□

Definition 2 Schedule σ is conflict serializable if it is conflict equivalent to some serial schedule.

□

Our definition of conflict serializability coincides with classical theory [9], in the case when the committed projection[11] of schedules is considered. However we can now judge equality between schedules containing aborting transactions. For example, under our new definition

$$w_1[x]\, r_2[x]\, a_1\, c_2 \not\equiv w_1[x]\, a_1\, r_2[x]\, c_2$$

because the write-read conflict (type V) on the left hand side does not exist on the right hand side.

Although conflict serializability has been defined only on complete schedules (i.e. those where all transactions in the schedule eventually either abort or commit) we can extend the definition to incomplete schedules. Any incomplete schedule can be extended to a complete schedule by aborting all the transactions without a terminal. We call the resulting schedule the *aborting-completion*. We now

[11] The committed projection of a schedule is obtained by removing any action from the schedule that belongs to a transaction that does not commit in that schedule.

define an incomplete schedule as serializable iff its aborting-completion is serializable.

In real systems failures can truncate schedules at any point. In many database systems, after failure and upon recovery (in the centralised case) active transactions are aborted. For this reason a useful property of any serializability definition over schedules is that if σ' is a prefix of a serializable schedule σ, then σ' is serializable.

Proposition 4 Any prefix of a complete conflict serializable schedule is conflict serializable.

Proof Let σ denote a complete serializable schedule and σ_{ser} denote a serial schedule that is conflict equivalent to σ. Let σ' denote any prefix of σ. We will construct a serial schedule σ'_{ser} from σ_{ser} that is conflict equivalent to the aborting-completion of σ', which we denote σ'_{com}. This shows that any prefix of a complete serializable schedule is serializable. We do this in two steps.

S1 For each $a_i \in \sigma'_{com}$ such that $c_i \in \sigma_{ser}$, replace c_i in σ_{ser} with a_i to form σ''.

S2 If any action appears in σ'' but not in σ'_{com} remove the action from σ'' to form σ'_{ser}.

We will now show that σ'_{ser} is a serial schedule that is conflict equivalent to σ'_{com}. Clearly σ'_{ser} is serial and has the same actions as σ'_{com} because of the way it was constructed from σ'_{ser}. We must now show that if a conflict of type C appears in σ'_{com} it also appears in σ'_{ser}.

If σ'_{com} has a conflict of type I, II or III, then it will also be in σ_{ser} because it was in σ. Steps **S1** and **S2** will not remove this conflict so it will also be present in σ'_{ser}.

Suppose σ'_{com} has a conflict of type IV or V, then either it was in σ', and by a similar argument to the one above will be in σ'_{ser}, or a new conflict will have been formed when the abort completion of σ' was taken to give σ'_{com}. If a new conflict was formed of type IV (V) in σ'_{com} then a conflict of type I (II) must have been present in s so it will also be present in σ_{ser} and will be changed to a conflict of type IV (V) by step **S1** when constructing σ_{ser} as required.

□

6.5 Redefining Phenomena

As pointed out by Berenson *et al.* the phenomena based definitions of isolation levels proposed in the ANSI standard [3] are ambiguous and incomplete. Berenson *et al.* give more precise definitions in response to these deficiencies. We restate these improvements in our notation and extend them a little further. Essentially we are looking for sufficient conditions, which are as weak as possible, which prevent cycles of conflicts, as defined above, occuring in a schedule.

Berenson *et al.* considered two possible interpretations of the ANSI Dirty Read phenomenon; a strict (**P1** below) and a loose interpretation. They argued that the strict interpretation was required to prevent the classical inconsistent analysis problem exemplified in the history σ_1 below. We use the notation $r_2[x = 50]$ to denote the action of t_2 reading data object x as having a value 50. Similarly $w_1[y = 90]$ denotes t_1 writing a value of 90 to data object y.

$$\sigma_1: \quad r_1[x = 50]w_1[x = 10]r_2[x = 10]r_2[y = 50]c_2r_1[y = 50]w_1[y = 90]c_1$$

$$\mathbf{P1}: \quad w_i[d] \prec r_j[d] \prec e_i$$

Clearly, the intention is to disallow the situation where t_j reads the changes made by t_i before they are committed. However, it is not always unsafe to do so. In fact, it is only unsafe in the case that t_i aborts after t_j read d and also when t_j commits. For example, consider the serializable schedule $w_i[d]\, r_j[d]\, c_i\, a_j$ which is disallowed by **P1**.

We propose that the loose interpretation of the phenomenon, below, more accurately captures the idea of a Dirty Read. We rename this **NP1** for consistency but it is identical to the loose interpretation called **A1** in [6].

$$\mathbf{NP1}: \quad w_i[d] \prec r_j[d] \prec (c_j \wedge a_i)$$

Unfortunately, the loose interpretation still admits schedules with the inconsistent analysis problem exemplified by history σ_1. This problem is better captured by the introduction of a new phenomenon we call **NP2L** (see below). Furthermore, we argue that this phenomenon should be disallowed at the higher ANSI REPEATABLE READ isolation level but not at the READ COMMITTED level c.f. [6]. The inconsistent analysis problem arises from transaction t_2 reading an inconsistent view of the data objects x and y. Item x is read after t_1 has updated it and y is read before t_1 has updated it. The problem therefore is better described as a Fuzzy or Non-Repeatable read not as a Dirty Read. From a user's perspective the value read by t_2 in σ_1 is not one that is later aborted as it is in

the case of a Dirty Read. Rather the values reflect partial changes made by other transactions. It should therefore be admitted at the READ COMMITTED level but excluded at the REPEATABLE READ level. Another example of the Fuzzy read problem appears in σ_2, a history that is symmetric to σ_1.

$$\sigma_2: \quad r_2[x=50]r_1[x=50]w_1[x=10]r_1[y=50]w_1[y=90]c_1r_2[y=90]c_2$$

To prevent this problem Berenson *et al.* defined phenomena **P2** which we state below.

$$\mathbf{P2}: \quad r_i[d] \prec w_j[d] \prec e_i$$

Again the intention is to prevent inconsistent reads of data objects by ensuring no other transaction t_j may change the value of a data object once read by t_i until after t_i has terminated. It is not always unsafe to do this. For example the schedule, $r_i[d]\, w_j[d]\, a_i\, c_j$, is serializable but not allowed by **P2**.

In our definition we replace **P2** with two phenomena **NP2R** and **NP2L** to capture the two symmetric phenomena that lead to Fuzzy reads of data objects. **NP2L** captures the problems of inconsistent analysis found in σ_1 (thus allowing us to use the loose interpretation **NP1** admitting more schedules at the lower ANSI READ COMMITTED level) and **NP2R** captures the Fuzzy read problem of σ_2.

$$\mathbf{NP2R}: \quad r_i[d] \prec w_j[d] \prec (c_i \wedge c_j)$$

$$\mathbf{NP2L}: \quad w_i[d] \prec r_j[d] \prec (c_i \wedge c_j)$$

Although excluding phenomena **NP2L**, and **NP2R** from schedules allows more serializable schedules than disallowing **P2**, they still disallow some serializable schedules. For example, the schedule $r_i[d]\, w_j[d]\, c_i\, c_j$ is serializable but disallowed by **NP2R**. This raises the following question. Can we simply characterise using our notation a phenomenon that captures only the schedules that read inconsistent views and no more? The answer to this is no. Such a definition would need to include reachability in the associated conflict graph of a schedule, but this type of property is not expressible in our notation.

The ANSI standard did not disallow schedules containing so called "dirty writes". This was identified and correctly rectified by the addition of the **P0** Phenomenon in [6]. This phenomenon can also be weakened to **NP0** (below) if we are only interested in isolation properties. In practice its stricter form **P0** is more useful for recoverability and consistency reasons.

$$\mathbf{P0}: \quad w_i[d] \prec w_j[d] \prec e_i$$

$$\mathbf{NP0}: \quad w_i[d] \prec w_j[d] \prec (c_i \wedge c_j)$$

Using these phenomena we provide definitions for the lowest three isolations levels; see Table 6.1 towards the end of this chapter.

6.6 Disallowing Phenomena Provides Conflict Serializability

We now show that if a schedule exhibits none of the phenomena **NP0**, **NP1**, **NP2L** or **NP2R** then it will be conflict serializable[12] as defined in definition 2 of section 6.4. We first prove the following lemma.

Lemma 6 *If a conflict exists between two transactions, t_i and t_j ($t_i \neq t_j$), on data object d which we can write generically as*

$$o_i[d] \prec o_j[d] \wedge e_i \wedge e_j$$

in a schedule s and phenomena **NP0, NP1, NP2L, NP2R** *do not occur over the actions of this conflict then either ($e_i \prec o_j[d]$ and $e_i = c_i$) or $e_j = a_j$.*

Proof By case analysis of the types of conflict.

I $r_i[d] \prec w_j[d] \wedge (c_i \wedge c_j)$ but **NP2R** does not occur so $c_i \prec w_j[d]$, as required.

II $w_i[d] \prec r_j[d] \wedge (c_i \wedge c_j)$ but **NP2L** does not occur so $c_i \prec r_j[d]$, as required.

III $w_i[d] \prec w_j[d] \wedge (c_i \wedge c_j)$ but **NP0** does not occur so $c_i \prec w_j[d]$, as required.

IV $r_i[d] \prec w_j[d] \wedge (c_i \wedge a_j)$ but $e_j = a_j$, as required.

V $w_i[d] \prec r_j[d] \prec (a_i \wedge c_j)$ but **NP1** does not occur which rules out this type of conflict completely.

Lemma 7 *If transaction t_i aborts in a schedule, s, that contains no* **NP1** *phenomena then no conflict can exist in s between t_i and some other transaction, say t_j, that would order the accesses of t_j after t_i.*

Proof The only possible conflict candidate to order t_j after t_i is a conflict of type V but this conflict is excluded by **NP1**.

Theorem 4 All schedules, s, which do not exhibit phenomena **NP0**, **NP1**, **NP2L**, and **NP2R** are conflict serializable.

[12] This is equivalent to the ANSI Isolation level REPEATABLE READ.

Proof Suppose s is not conflict serializable. Let $G = (V, E)$ be the directed conflict graph constructed from s as follows. The vertices of G are the transactions in s and an edge (t_i, t_j) is in E if there is a conflict between t_i and t_j $(t_i \neq t_j)$ and the accesses of this conflict are ordered $o_i[d] \prec o_j[d]$. Clearly, s is serializable iff G is acyclic (a proof of this is a special case of proposition 6 in the next chapter).

Suppose s is not serializable. Without loss of generality let the smallest cycle in the conflict graph G be denoted by

$$t_1 \xrightarrow{d_1} t_2 \xrightarrow{d_2} \ldots \xrightarrow{d_{m-1}} t_m \xrightarrow{d_m} t_1$$

By Lemma 7 no conflict ordering $t_i \xrightarrow{d_i} t_{i+1} \ : \ 1 \leq i < m$ in the cycle exists where $e_{i+1} = a_{i+1}$ (so all conflicting transactions in the cycle commit).

By Lemma 6 in each conflict $c_i \prec o_{i+1}[d] \ : \ 1 \leq 1 < m$ and also $o_i \prec c_i \ : \ 1 \leq i \leq m$ because all actions must be before their terminals, thus we can order the conflicts in the graph as follows.

$$o_1[d_1] \prec e_1 \prec o_2[d_1] \prec e_2 \prec o_2[d_2] \prec e_3 \ldots \prec e_m \prec o_1[d_m]$$

This leads to a contradiction since $e_1 \prec o_1[d_m]$ may not occur in s, so s is serializable.

□

6.7 Enriching Schedules with Predicate Accesses

We now extend our model with some new types of accesses. Given a predicate P we add a new action, $r_i[P]$, to denote a read of the set of data objects that fulfill P. For example, P might be "all employees that are male", so that $r_i[P]$ denotes transaction t_i reading all those employees that are male. We also add two types of write actions $w_i[insert\ y\ in\ P]$ and $w_i[delete\ y\ in\ P]$, these denote actions that insert or delete a new data object, y, in a way that could change the values returned by a $r_i[P]$ access[13]. We write $w_i[y\ in\ P]$ to denote either an insert or a delete access. In our example above $w_i[y\ in\ P]$ might be inserting or deleting a male employee. In this extended model, a phenomenon known as a phantom may occur. We restate an example from [6] that exemplifies this.

Example 2 *Transaction t_i performs a* `<search-condition>` *to find the list of active employees. Then transaction t_j performs an insert of a new active employee and then updates z, the count of employees in the company. Following this t_i reads*

[13] Item y does not have to directly satisfy P for this to be true.

the count of employees as a check and finds a discrepancy. The schedule can be written as:-

$$r_i[P]\, w_j[insert\ d\ in\ P]\, r_j[z]\, w_j[z]\, c_j\, r_i[z]\, c_i$$

Berenson *et al.* provide the following definition of a phantom.

$$\textbf{P3}: \quad r_i[P] \prec w_j[d\ in\ P] \prec e_i$$

Strictly speaking this does not completely characterise all phantom phenomena. Consider Example 3 below.

Example 3 *Transaction t_i deletes an active employee. Transaction t_j then reads the count of active employees z, this will include the one previously deleted by t_i. Transaction t_j then reads the set of all active employees, this will not include the employee deleted by t_i, and then commits. Finally t_i updates the count of new employees and commits. The schedules can be written as follows.*

$$w_i[delete\ y\ in\ P]\, r_j[z]\, r_j[P]\, c_j\, r_i[z]\, w_i[z]\, c_i$$

The schedule in example 3 contains a phantom not disallowed by **P3** therefore strictly speaking the characterisation of phantom phenomena, **P3**, given by Berenson *et al.* appears to permit some kinds of phantoms. Furthermore, it is claimed this phenomena based definition is equivalent the locking based definition of serializable isolation LOCKING SERIALIZABLE they give [6]. In this definition predicate write locks are not released until the transaction commits or aborts, which would prevent the problem in Example 3.

It appears that Berenson *et al.*'s phenomenon based definition of SERIALIZABLE isolation admits schedules with phantoms, and that it is not equivalent to the locking based definition they provide which does not allow phantoms.

This discrepancy seems to originate from an assumption that a predicate read access $r[P]$ will conflict with any previous writes (be they deletes or inserts) to data objects satisfying the predicate. Implementations do exist to detect exactly this. In some a flag is set in all index entries when a row is deleted, this flag is later garbage collected. Similar implementation details could solve the problem exemplified in Example 2. Rather than make reference to implementations it seems more sensible to define a complete set of phenomena that capture the behaviour of both examples. We therefore define phenomena **NP3R**, and **NP3L** in an analogous way to **NP2R** and **NP2L** as alternatives to **P3**. We also define a

predicate form of the dirty read and dirty write phenomena which we call **NP2$\frac{1}{2}$** and **NP2$\frac{1}{4}$** respectively.

$$\textbf{NP3R}: \quad r_i[P] \prec w_j[d\ in\ P] \prec (c_i \wedge c_j)$$

$$\textbf{NP3L}: \quad w_i[d\ in\ P] \prec r_j[P] \prec (c_i \wedge c_j)$$

$$\textbf{NP2}\tfrac{1}{2}: \quad w_i[d\ in\ P] \prec r_j[P] \prec (c_j \wedge a_i)$$

$$\textbf{NP2}\tfrac{1}{4}: \quad w_i[d\ in\ P] \prec w_j[d\ in\ P] \prec (c_i \wedge c_j)$$

We are now in a position to define isolation levels in terms of all our new phenomena see Table 6.1.

Isolation Level	**P0** **NP2$\frac{1}{4}$**	**NP1**	**NP2R** **NP2L**	**NP3R** **NP3L** **NP2$\frac{1}{2}$**
READ UNCOMMITTED	–	+	+	+
READ COMMITTED	–	–	+	+
REPEATABLE READ	–	–	–	+
SERIALIZABLE	–	–	–	–

Table 6.1: Definition of isolation levels. [+] denotes a phenomena that is allowable at a particular isolation level whereas [–] denotes that the phenomena is not allowed in any schedules achieving this isolation level.

6.8 Conclusion

In order to discuss the role of commit protocols in providing different transaction isolation levels we need to clarify what we mean by a particular transaction isolation level. In this chapter we provided a definition of conflicting actions that includes the commit or abort outcome of the transactions involved in the conflict. Under this extended definition we need not make any recoverability assumptions about schedules. This leads to a definition of conflict serializability that is independent of recoverability. This definition of conflict serializability does not include the phantom phenomenon because it is defined over schedules that do not include reads and writes over predicates. It is therefore equivalent to the ANSI definition REPEATABLE READ.

Secondly, we provided a definition of the lowest three ANSI isolation levels based on phenomena. Our phenomena are weaker than those proposed in [6] and thus admit more serializable schedules. Furthermore, we argued that the **NP2L**

phenomena should be excluded at the higher REPEATABLE READ level of isolation but not at the READ COMMITTED level thus admitting more schedules at the READ COMMITTED level.

Lastly, we enrich schedules to include read and write actions on predicates. Within these enriched schedules we discuss Phantom Phenomena and characterise them in a way that is independent of predicate concurrency control mechanisms. Excluding these phantom phenomena results in a level of isolation termed SERIALIZABLE by the ANSI community.

Many textbooks state that isolation and serializability are synonymous [10, 34]. We argued in this chapter that isolation is really a sufficient but not necessary condition for serializability. Indeed, the isolation levels defined in the literature exclude many serializable schedules. We are now in a position to generalise the theory we developed in this chapter to include distributed transactions and to discuss the role of commit protocols in transaction isolation.

Chapter 7

Verifying Isolation Levels in Distributed Transactions

7.1 Introduction

When transactions perform accesses on data objects that are distributed across a set of autonomous sites, we call those transactions distributed. Many middleware systems now support distributed transactions such as CORBA transaction services [61] or Microsoft's DTC [20], as well as more traditional distributed database management systems. Distributed transactions must fulfill the following additional requirements over centralised transactions:

- Distributed transactions must be *atomic* across sites, that is if t_i commits (aborts) at one site it must not abort (commit) at another. This statement is of course trivially true in the centralised case.
- A distributed transaction must have the ability to unilaterally abort at a site if, for example, an access at that site caused a deadlock or violated an integrity constraint. This means that all the accesses required for a transaction at a site must be completed before that site can determine if it is willing or prepared to commit. If a site were prepared to commit before completing its accesses future accesses may require the transaction to be aborted, but the site has already announced its willingness to commit.

We have seen in previous chapters the role a commit protocol plays in providing distributed transactions with these additional properties. In this chapter we will study its role in providing distributed transactions with a particular level of isolation. In the previous chapter we defined isolation levels for centralised transactions. In this chapter we will generalise these ideas to distributed transactions. We have shown that the exclusion of **NP** phenomena from a schedule was

sufficient to ensure serializability in the centralised case. A distributed schedule can be modelled as a vector of local schedules with one entry for each site where a data objects reside. In the case of distributed schedules, the exclusion of **NP** phenomena at each local schedule *alone* is not enough to ensure distributed serializability. Another property is required. We define such a property and call it *synchpoint prepare.* As we have seen, a commit protocol is used to ensure the atomicity of distributed transactions. We will see how commit protocols play another vital role in providing this synchpoint prepare property, and thus a level of transaction isolation.

In order to increase distributed transaction concurrency in a system, some transaction processing systems such as Microsoft's DTC [20] support an optimisation whereby read locks are released immediately after the read has been performed. Under this scheme the synchpoint prepare property is lost. In this chapter we will use our views based modelling technique to show the effect this optimisation has on the isolation level attained.

7.2 Modelling Distributed Schedules

When transactions are distributed, data objects are distributed across a set of sites S. We let $SITE(d)$ denote the unique site to which data object d belongs. Read and write accesses as before are denoted $r_i[d]$, $w_i[d]$ as in the previous chapter. Once the accesses of a transaction have completed, at a site, they can be terminated at that site, we denote these terminals in distributed schedules $c_i[s]$, $a_i[s]$ making reference to the site at which they occur.

The accesses of concurrently executing distributed transactions produce actions that execute locally at each site $s \in S$. We refer to these local execution, at s, as the transactions *local schedule* at s.

We discussed at length in chapters 4 to 5 how a commit protocol is used to provide atomicity. We have seen that in a commit protocol, before any site can commit, that site must acquire (usually by the receipt of messages) knowledge that each site involved in the transaction is willing or prepared to commit. We saw in chapter 2 that this property is true for all 2PC protocols[1] and was called the 2PC-level of knowledge. It follows from these observations that in every distributed transaction, each local site must perform a prepare action before committing a transaction[2]. We denote transaction t_i's prepare action at site s, $p_i[s]$. At each

[1]Unless some special mechanism, such as compensating transactions is in place.

[2]In the early prepare variant of two phase commit this action is implicit after each access is performed. In the read-only optimisation the prepare action is not followed by a final commit,

site, all transactions must perform their prepare action before committing, that is $p_i[s] \prec c_i[s]$, holds for each local schedule. A prepare action is neither a terminal nor an access, the table below summarises this.

Accesses		Prepare action		Terminals	
$w_i[d]$	t_i writes d	$p_i[s]$	t_i prepares at s	$c_i[s]$	t_i commits at s
$r_i[d]$	t_i reads d			$a_i[s]$	t_i aborts at s

Definition 3 A *complete distributed schedule*, ρ is a vector of complete local schedules $\rho = \langle \sigma_{s_1}, \ldots, \sigma_{s_n} \rangle$, with one entry in the vector for each site, where data objects reside, in the distributed system.

□

For notational convenience we write distributed schedules as a list of local schedules. An example is given below, where $s = SITE(d)$ and $s' = SITE(d')$.

$$\begin{array}{llllllll} \sigma_s : & r_1[d] & r_2[d] & p_1[d] & w_2[d] & p_2[d] & c_1[s] & c_2[s] \\ \sigma_{s'} : & r_2[d'] & w_2[d'] & r_1[d'] & p_1[d] & p_2[d] & c_2[s'] & c_1[s'] \end{array}$$

We use the same notational conventions as we did in centralised schedules, that if one action is to the right of another in the same row then the rightmost, of the two actions, took place later. If however two actions are in different rows we can only determine their relative order if we know some information about the global order of events. For example, if we know that all accesses across all sites are finished before any prepare actions are taken at any sites, then we are often able to determine the order of events in the distributed schedule.

In the example above if we assume that all accesses are finished before any prepare actions are taken then we know that $r_1[d']$ in sequence $\sigma_{s'}$ is before $c_2[s]$ in sequence σ_s by applying transitivity to the facts that $r_1[d']$ must be before $p_1[s]$ and that $p_1[s]$ is before $c_2[s]$ by the local order in the local schedule σ_s.

Definition 4 A *complete distributed schedule* is *serial* if there exists a total order of the transactions such that if t_i precedes t_j in the order, then all of t_i's accesses precede all of t_j's accesses in each local schedule where both appear [8].

□

---

if no writes have been performed at the site on behalf of the transaction. These variants were discussed in chapter 2

Definition 5 A prefix of a distributed schedule is the vector of (possibly empty) prefixes of the original local schedules. A prefix $\rho' = (\sigma'_{s_1}, \ldots, \sigma'_{s_n})$ of $\rho = (\sigma_{s_1}, \ldots, \sigma_{s_n})$ is *admissible* provided

$$\text{if } c_i[s_j] \in \sigma'_{s_j} \text{ then } \forall s_k \in S, \text{ if } o_i[d] \in \sigma_{s_k} \text{ then } o_i[d] \in \sigma'_{s_k}$$

If a transaction commits in a prefix of a schedule, then all accesses of that transaction from the original schedule are present in the prefix.

□

In distributed transactions processing systems logging to stable storage ensures that the prefix reconstructed on recovery from a failure of any site is always admissible. Since we want to concentrate on isolation we will assume that all prefixes are admissible. By modelling failure, and the actions taken upon recovery this assumption could be verified.

Definition 6 The *abort-completion* of a prefix of a distributed schedule ρ, is formed by extending each local schedule σ_s of ρ in the following way. An $a_i[s]$ is appended to σ_s for each transaction t_i with an access but no terminal in σ_s, provided t_i has not committed in any other local schedule of ρ. If it has been committed elsewhere and some access $o_i[d]$ exists in σ_s we append $c_i[s]$ to σ_s.

Clearly, an abort-completed prefix of a complete distributed schedule is a complete distributed schedule. Furthermore, because any prefixes must be admissible, any transaction that commits in the abort-completed prefix must contain all the actions that it did in the original schedule.

7.3 Serializability of Distributed Schedules

Definition 7 Distributed schedules ρ and ρ' are *conflict equivalent* iff

- ρ and ρ' have the same actions and
- for each conflict of type $C \in \{I, \ldots, V\}$ involving accesses $r_i[d]\, w_j[d]$ $(w_i[d]\, w_j[d])$ $(w_i[d]\, r_j[d])$ and terminals $e_i[s]$, $e_j[s]$, in ρ, the same conflict of type C appears in ρ' involving the same accesses $r_i[d]\, w_j[d]$ $(w_i[d]\, w_j[d])$ $(w_i[d]\, r_j[d])$ and terminals $e_i[s]$, $e_j[s]$.

□

Definition 8 A complete distributed schedule ρ is serializable if it is conflict equivalent to some complete distributed serial schedule [8].

□

Just as in the centralised case of section 6.4 in chapter 6 we define a prefix of a complete distributed schedule to be serializable iff its abort-completion is serializable.

Proposition 5 All prefixes of complete serializable schedules are serializable.

Proof Let $\rho = (\sigma_{s_1}, \ldots, \sigma_{s_k})$ be a complete distributed serializable schedule and let ρ_{pref} be any admissible prefix of ρ. Let ρ_{com} be the abort-completion of ρ_{pref}.

If accesses $o_i[d]$ and $o_j[d]$ exist in ρ_{com} then they also exist in ρ and furthermore they are in the same order in each of the schedules. The terminals e_i, e_j in ρ_{com} need not be the same (or be in the same order) as e_i, e_j in ρ. We do however know that if e_i (e_j) is c_i (c_j) in ρ then it will also be c_i (c_j) in ρ_{com}, from the way ρ_{com} is constructed and the atomicity of transactions in ρ. It thus follows for each conflict ordering t_i before t_j in ρ_{com}, at some site s_k, involving accesses $o_i[d]$ and $o_j[d]$, there exists a conflict ordering t_i before t_j involving the same accesses $o_i[d]$, $o_j[d]$ at s_k in ρ. ρ_{com} will therefore have a subset of the conflicts found in ρ. Since ρ is serializable then so is ρ_{com}.

□

As in the centralised case a graph theoretic characterisation exists for the conflict serializability of a distributed schedule. Let G be a directed graph whose nodes are the transactions of a schedule ρ. If a conflict exists in a schedule we add a directed edge between $t_i \rightarrow t_j$ to the graph G.

Proposition 6 *If G is acyclic then ρ is serializable.*

Proof Suppose no cycle exists in G. We will now show ρ is serializable. If no cycle exists then a topological sort of G will provide a total order, such that if t_i is before t_j in the total order either no conflict exists between t_i and t_j or a conflict exists ordering t_i *before* t_j. By definition therefore ρ is conflict equivalent to a serial schedule.

□

Proposition 7 *If a complete distributed schedule ρ is serializable then each complete local schedule is also serializable.*

Proof If ρ is serializable then it is conflict equivalent to a serial distributed schedule ρ'. This means each local schedule of ρ is conflict equivalent to the corresponding local serial schedule of ρ'

□

We can see in the example below the converse of Proposition 7 does not necessarily hold (*c.f.* [62]).

$$\begin{array}{lllllll} \sigma_s : & r_1[d] & p_1[s] & c_1[s] & w_2[d] & p_2[s] & c_2[s] \\ \sigma_{s'} : & w_2[d'] & p_2[s'] & c_2[s'] & r_1[d'] & p_1[s'] & c_1[s'] \end{array}$$

In σ_s $t_1 \rightarrow t_2$ because of the conflict $r_1[d]w_2[d]$ and in $\sigma_{s'}$, $t_2 \rightarrow t_1$ because of the conflict $w_2[d']r_1[d']$. That said, each local schedule, σ_1, σ_2, is serializable.

7.3.1 Local rules for distributed serializability

In section 6.6 of the previous chapter we saw that the absence of phenomena **P0**, **NP1**, **NP2L**, **NP2R** gives rise to serializable executions of local schedules. Unfortunately, serializability of each local schedule does not imply the serializability of a distributed schedule. We require a further condition to achieve this.

Definition 9 A distributed transaction obeys *synchpoint prepare* if for all sites $s \in S$ all accesses of the transaction are before any prepare action. More formally:

$$\forall o_i \in \rho,\ o_i \rightsquigarrow p_i[s]$$

where $\rightsquigarrow$ is the order of events in the distributed system.

□

Although it is often impossible for processes in a distributed system to know the global order of events, $\rightsquigarrow$, this does not mean that the order does not exist. Consider an omnipresent observer who samples the global state of the system at regular time intervals. Provided the samples are frequent enough that observer can determine the order of events.

We saw in chapter 2 a read-only optimisation for 2PC. In this optimisation if a transaction performs only read accesses of data objects at a site then after it is asked to prepare it can vote `read-only` and need no longer be involved in the transaction. In a distributed transaction therefore, the point at which a site performs a prepare action can mark the end of the transaction at that site.

For this reason in a distributed transaction we should change the **NP2R** rule as follows, where $SITE(d) = s$.

$$\mathbf{NP2R'}: \; r_i[d] \prec w_j[d] \prec (p_i[s] \wedge c_j[s])$$

Whereas the absence of **NP2R** in a schedule stipulates that $c_i[s]$ must be before $w_j[d]$, the absence of **NP2R′** in a schedule stipulates the weaker condition that $p_i[s]$ must be before $w_j[d]$. It is thus possible for $c_i[s]$ to be after $w_j[d]$ as long as $p_i[s]$ is before $w_j[d]$. If we are using strict two phase locking for concurrency control this translates to saying that it is safe to release shared locks at prepare time.

Lemma 8 *Let t_i and t_j be two distributed transactions that participate in a conflict of type I, II or III on a data object d at site s which we write as*

$$\begin{array}{llll} \text{I} & r_i[d] \prec w_j[d] \wedge (c_i[s] \wedge c_j[s]) & \text{II} & w_i[d] \prec r_j[d] \wedge (c_i[s] \wedge c_j[s]) \\ \text{III} & w_i[d] \prec w_j[d] \wedge (c_i[s] \wedge c_j[s]) & & \end{array}$$

If phenomena **P0**, **NP1**, **NP2L**, **NP2R′** *do not occur over the actions of the particular conflict then in the case of conflict types I and II $c_i[s] \prec o_j[d]$, and for conflict type III $p_i[s] \prec o_j[d]$. Recall $o[d]$ is shorthand for $r[d]$ or $w[d]$.*

□

Proof For conflict types I and II see the proof of lemma 6 in the previous chapter. For type III note that the absence of **NP2R′** in any schedule means $p_i[s] \prec o_j[d]$ as required.

□

Theorem 5 *Any complete distributed schedule which obeys synchpoint prepare and contains no local phenomena of the type* **P0**, **NP1**, **NP2L** *and* **NP2R′** *is serializable.*

Proof If a schedule ρ is not serializable then (by proposition 6) a cycle of conflicts between transactions must exist. Each conflict must be on a particular data object so we can denote this cycle as $t_1 \xrightarrow{d_1} t_2 \xrightarrow{d_2} \ldots t_m \xrightarrow{d_m} t_1$.

No conflict in this cycle can be of type V because type V conflicts $w_i[d] \prec w_i[d] \prec (a_i[s] \wedge c_j[s])$ are disallowed by **NP1**. Furthermore, no conflict $t_i \xrightarrow{d_i} t_{i+1}$ can be of type IV because this implies a_{i+1} and so no conflict can order any transaction after t_{i+1} which gives rise to a contradiction because t_{i+1} is part of a conflict cycle.

Using lemma 8 and the fact that $p_i[s] \prec c_i[s]$ in any transactions we have $p_i[s] \prec o_j[d]$. We also know that our transactions have the synchpoint prepare property and so we can derive the following order on actions in ρ. The notation $\downarrow$ denotes a vertical version of $\leadsto$ which was not available to the author.

$$
\begin{array}{ccccccccccccc}
o_1[d_1] & \prec & p_1[s_1] & \prec & o_2[d_1] & \prec & e_2[d_1] & & & & & & \\
 & & & & \downarrow & & & & & & & & \\
 & & o_2[d_2] & \prec & p_2[s_2] & \prec & o_3[d_2] & \prec & e_3[s_2] & & & & \\
 & & & & & & \downarrow & & & & & & \\
 & & & & o_3[d_3] & \prec & p_3[s_3] & \prec & o_4[d_3] & \prec & e_4[s_3] & & \\
 & & & & & & & \ddots & \ddots & \ddots & & & \\
 & & & & & & & & \downarrow & & & & \\
 & & & & & & o_m[d_m] & \prec & p_m[s_m] & \prec & o_1[d_m] & \prec & c_1[s_m]
\end{array}
$$

This gives rise to the contradiction that $o_1[d_m]$ is after $p_1[s_1]$ which violates the synchpoint prepare property.

□

We can see if exclusive locks are held until commit time and read locks held until prepare time[3] then **P0**, **NP1**, **NP2L** and **NP2R′** will be prevented. If in addition to this the commit protocol used has the synchpoint prepare property then distributed transactions will be serializable.

7.4 Modelling Distributed Transactions

In this section we use our views based modelling technique to model a very simple distributed transaction processing system in which concurrent transactions perform read and write accesses to distributed data objects.

The processes in our system are from two different classes. The data object class D and the transaction class T. Let t and d range over the classes T and D respectively.

Each transaction t has the following local state. First there is a set of write accesses to be performed denoted $t.W$. This contains the identities of the data objects to be written to. If initially $d \in t.W$ then transaction t requests a write access on data object d. Similarly if $d \in t.R$, then t requests a read access on d. In our simple model each transaction can carry out at most one access at each data object which can be either a read or a write access. We also assume only one data object per site, however the results in this chapter do not rely on this restriction. The set $t.A$ contains the protocol actions that t requests of the data objects. For example if $\mathbf{p}_d \in t.A$ then t requests data object d to prepare.

The data objects in our model have an acknowledgement set $d.X$. If $\mathbf{w}_t \in d.X$, for example then data object d is replying with a **yes** vote to transaction t.

[3] Recall in our simple model each data object undergoes either a read access or a write access but not both. In the more complex model when data objects are both read and written to exclusive locks are held until the transaction terminates.

Similarly if $\mathbf{x}_t \in d.X$ then d is acknowledging t's request to either read or write. In addition to this data objects maintain a set, $d.S$, to record those transaction that hold a shared lock on their data. Data objects also record which transaction holds an exclusive lock on their data using the variable $d.e$. If $d.e = \perp$ then no transaction holds an exclusive lock at d. If $t \in d.S$ for example then t holds a shared lock on data object d, similarly if $d.e = t$ then t holds an exclusive lock on data object d.

Transaction objects communicate with data objects by viewing sets. For instance,

$$@d(d \in t.R)$$

means d views the set $t.R$ as containing the element d. In this case data object d has received a read request from transaction t requesting a read access on data object d. Similarly, $@t(\mathbf{x}_t \in d.X)$ then transaction t has received an acknowledgement for its access at d. Table 7.4 summarises the states of transaction and data object processes.

Name	Description	Range	Initial Value	Viewable
$t.R$	Transaction read requests	2^D	$\subseteq D$	Y
$t.W$	Transaction write requests	2^D	$\subseteq D$	Y
$t.A$	Transaction protocol actions	$2^{\{\mathbf{p},\mathbf{w},\mathbf{a}\}}$	$\emptyset$	Y
$d.X$	Data object acknowledgements	$2^{\{\mathbf{x},\mathbf{w},\mathbf{a}\}}$	$\emptyset$	Y
$d.S$	Data object shared locks	2^T	$\emptyset$	N
$d.e$	Data object exclusive lock	T	$\emptyset$	N

Table 7.1: Local state at transaction and data object processes.

7.4.1 Protocol Rules

First we consider four rules that govern how a transaction process and a data object communicate when performing either a read or write access. In our very simple model a transaction knows at the start exactly what it will want to read and write. In practice this is not always the case.

$$\mathbf{DR}(d)\ \frac{@d(d \in t.R) \wedge t \notin S \wedge (e = t \vee e = \perp)}{S := S \cup \{t\} \ \wedge\ X := X \cup \{\mathbf{x}_t\}} \qquad \mathbf{TR}(t)\ \frac{@t(\mathbf{x}_t \in d.X) \wedge d \in R}{R := R - \{d\}}$$

$$\mathbf{DW}(d)\ \frac{@d(d \in t.W) \wedge e = \perp \wedge S = \emptyset}{e := t \ \wedge\ X := X \cup \{\mathbf{x}_t\}} \qquad \mathbf{TW}(t)\ \frac{@t(\mathbf{x}_t \in d.X) \wedge d \in W}{W := W - \{d\}}$$

In the rule **DR** a data object, once it has viewed a read request, will carry out that request if no other transaction holds an exclusive lock. Once carried out a

shared lock is added and an acknowledgement is sent, by updating $d.X$. **DR** is matched by rule **TR**. In **TR**, if t views an acknowledgement to a read access from d it removes d from its read request set $t.R$. Similarly the rules **DW** and **TW** allow the communication of a write access. This time an exclusive locking policy is employed. In order for data objects to receive requests from transactions for reads and writes they must be able to update their views of a transaction's $t.R$ and $t.W$ sets. In the following rule let O be either set W or set R.

$$\mathbf{DUVO}(d)\ \frac{d \in t.O \wedge @d(d \notin t.O)}{@d(t.O := t.O \cup \{d\})}$$

Similarly **TUV** allows a transaction process to update its view of a data object when that object acknowledges an action or votes. **DUV** allows a data object to update its view for prepare, commit or abort requests.

$$\mathbf{TUV}(t)\ \frac{z_t \in d.X\ \wedge\ @t(z_t \notin d.X)}{@t(d.X := \{z_t\})}\ z \in \{\mathbf{x}, \mathbf{w}, \mathbf{a}\}$$

$$\mathbf{DUV}(d)\ \frac{z_t \in t.A\ \wedge\ @d(z_t \notin t.A)}{@d(t.A := t.A \cup \{z_t\})}\ z \in \{\mathbf{p}, \mathbf{c}, \mathbf{a}\}$$

In the next rule a transaction process starts to prepare the transaction. We define $\mathrm{DATA}_t \overset{def}{=} t.R \cup t.W$, $\mathrm{PREPARE}_t \overset{def}{=} \{\mathbf{p}_{d'} \mid d' \in \mathrm{DATA}_t\}$, $\mathrm{COMMIT}_t \overset{def}{=} \{\mathbf{c}_{d'} \mid d' \in t.W\}$ and $\mathrm{ABORT}_t \overset{def}{=} \{\mathbf{c}_{d'} \mid d' \in t.W\}$, where $t.R$ and $t.W$ are their values before any **TW** or **TR** rules take place.

$$\mathbf{TP}(t)\ \frac{R = \emptyset \wedge W = \emptyset}{A := \mathrm{PREPARE}_t}$$

The next three rules model the response a data object makes to a prepare request. Either it votes **yes** to transaction t modelled by rules **DSVY** and **DEVY** (**DSVY** for a read-only response) or votes not and releases all locks.

$$\mathbf{DSVY}(d)\ \frac{t \in S \wedge @d(\mathbf{p}_d \in t.A)}{X := X \cup \{\mathbf{w}_t\} - \{\mathbf{x}_t\} \wedge S := S - \{t\}}$$

$$\mathbf{DEVY}(d)\ \frac{e = t \wedge @d(\mathbf{p}_d \in t.A)}{X := X \cup \{\mathbf{w}_t\} - \{\mathbf{x}_t\}}$$

The next two rules model an abort response from a data object when it views a prepare request. Again we model the case where the data object holds a shared lock separately, **DSVN**, to the case where the data object holds an exclusive lock

DEVN.

$$\mathbf{DSVN}(d)\ \frac{t \in S \wedge @d(\mathbf{p}_d \in t.A)}{X := X \cup \{\mathbf{a}_t\} - \{\mathbf{x}_t\} \wedge S := S - \{t\}}$$

$$\mathbf{DEVN}(d)\ \frac{e = t \wedge @d(\mathbf{p}_d \in t.A)}{X := X \cup \{\mathbf{a}_t\} - \{\mathbf{x}_t\} \wedge e = \bot}$$

Finally we model a transaction process deciding either commit or abort using the rules **TC** and **TA**, and also, once it has viewed this decision, a data object using the rules **DC**, **DA**, to decide accordingly.

$$\mathbf{TC}(t)\ \frac{@t(\forall d \in \mathrm{DATA}_t,\ \mathbf{w}_t \in d.X)}{t.A := \mathrm{COMMIT}_t} \qquad \mathbf{TA}(t)\ \frac{@t(\exists d \in \mathrm{DATA}_t,\ \mathbf{a}_t \in d.X)}{t.A := \mathrm{ABORT}_t}$$

$$\mathbf{DC}(d)\ \frac{e = t \wedge @d(\mathbf{c}_d \in t.A)}{e := \bot} \qquad \mathbf{DA}(d)\ \frac{e = t \wedge @d(\mathbf{a}_d \in t.A)}{e := \bot}$$

7.5 An example execution

We now give an example execution involving two data objects d_1, d_2 and two transaction objects t_1 and t_2. Initially $t_1.R = \{d_1, d_2\}$, $t_1.W = \emptyset$ and $t_2.R = \{d_1\}$, $t_2.W = \{d_2\}$ modelling an initial configuration where t_1 attempts read operations on both d_1 and d_2 and t_2 attempts a read on d_1 and a write on d_2. To model a configuration we compose the states of t_1, t_2, d_1, d_2. The state of a transaction object t is represented as a triple, using a superscript for t's view of a data object's acknowledgement sets written as $t.R\ t.W\ t.A^{d_1.X\ d_2.X}$. Using this representation initially the state of t_1 is $\{d_1, d_2\}\emptyset\emptyset^{\emptyset\emptyset}$. Similarly we represent the state of a data objects d as $d.S\ d.e\ d.X^{t_1.R\ t_1.W\ t_1.A\ t_2.R\ t_2.W\ t_2.A}$ and so the initial state of d_1 is $\emptyset\bot\emptyset^{\emptyset\emptyset\emptyset\emptyset\emptyset\emptyset}$.

An execution of a the system can be found in figure 7.5. In this execution t_1 and t_2 both perform read operations on d_1 after which t_2 performs a write operation on d_2. After this t_2 holds an exclusive lock on d_2 and so t_1's read operation cannot take place. After receiving acknowledgements, t_2 proceeds to issue a prepare to d_1 and d_2. They respond with **yes** votes and t_2 decides commit. On receiving commit d_2 can release its exclusive lock allowing t_1 to read d_2. d_2 acknowledges this read and t_1 prepares the transaction. Both d_1 and d_2 respond to this prepare by releasing their locks and finally t_1 commits.

$\{d_1, d_2\}\emptyset\emptyset^{\emptyset\emptyset}$	$\{d_1\}\{d_2\}\emptyset^{\emptyset\emptyset}$	$\emptyset\bot\emptyset^{\emptyset\emptyset\emptyset\emptyset\emptyset\emptyset}$	$\emptyset\bot\emptyset^{\emptyset\emptyset\emptyset\emptyset\emptyset\emptyset}$
$\{d_1, d_2\}\emptyset\emptyset^{\emptyset\emptyset}$	$\{d_1\}\{d_2\}\emptyset^{\emptyset\emptyset}$	$\downarrow \mathbf{DUVO}(d_1)$ $\emptyset\bot\emptyset^{\{d_1\}\emptyset\emptyset\{d_1\}\emptyset\emptyset}$	$\downarrow \mathbf{DUVO}(d_2)$ $\emptyset\bot\emptyset^{\{d_2\}\emptyset\emptyset\emptyset\{d_2\}\emptyset}$
$\{d_1, d_2\}\emptyset\emptyset^{\emptyset\emptyset}$	$\{d_1\}\{d_2\}\emptyset^{\emptyset\emptyset}$	$\downarrow \mathbf{DR}(d_1)$ $\{t_1\}\bot\{\mathbf{x}_{t_1}\}^{\{d_1\}\emptyset\emptyset\{d_1\}\emptyset\emptyset}$	$\downarrow \mathbf{DW}(d_2)$ $\emptyset t_2\{\mathbf{x}_{t_2}\}^{\{d_2\}\emptyset\emptyset\emptyset\{d_2\}\emptyset}$
$\{d_1, d_2\}\emptyset\emptyset^{\emptyset\emptyset}$	$\{d_1\}\{d_2\}\emptyset^{\emptyset\emptyset}$	$\downarrow \mathbf{DR}(d_1)$ $\{t_1, t_2\}\bot\{\mathbf{x}_{t_1}, \mathbf{x}_{t_2}\}^{\{d_1\}\emptyset\emptyset\{d_1\}\emptyset\emptyset}$	$\emptyset t_2\{\mathbf{x}_{t_2}\}^{\{d_2\}\emptyset\emptyset\emptyset\{d_2\}\emptyset}$
$\downarrow \mathbf{TUV}(t_1)$ $\{d_1, d_2\}\emptyset\emptyset^{\{\mathbf{x}_{t_1}\}\emptyset}$	$\downarrow \mathbf{TUV}(t_2)$ $\{d_1\}\{d_2\}\emptyset^{\{\mathbf{x}_{t_1}\}\{\mathbf{x}_{t_1}\}}$	$\{t_1, t_2\}\bot\{\mathbf{x}_{t_1}, \mathbf{x}_{t_2}\}^{\{d_1\}\emptyset\emptyset\{d_1\}\emptyset\emptyset}$	$\emptyset t_2\{\mathbf{x}_{t_2}\}^{\{d_2\}\emptyset\emptyset\emptyset\{d_2\}\emptyset}$
$\mathbf{TR}(t_1)\downarrow$ $\{d_2\}\emptyset\emptyset^{\{\mathbf{x}_{t_1}\}\emptyset}$	$\downarrow \mathbf{TR}(t_2) \downarrow \mathbf{TW}(t_2)$ $\emptyset\emptyset\emptyset^{\{\mathbf{x}_{t_1}\}\{\mathbf{x}_{t_1}\}}$	$\{t_1, t_2\}\bot\{\mathbf{x}_{t_1}, \mathbf{x}_{t_2}\}^{\{d_1\}\emptyset\emptyset\{d_1\}\emptyset\emptyset}$	$\emptyset t_2\{\mathbf{x}_{t_2}\}^{\{d_2\}\emptyset\emptyset\emptyset\{d_2\}\emptyset}$
$\{d_2\}\emptyset\emptyset^{\{\mathbf{x}_{t_1}\}\emptyset}$	$\downarrow \mathbf{TP}(t_2)$ $\emptyset\emptyset\{\mathbf{p}_{d_1}, \mathbf{p}_{d_2}\}^{\{\mathbf{x}_{t_2}\}\{\mathbf{x}_{t_2}\}}$	$\{t_1, t_2\}\bot\{\mathbf{x}_{t_1}, \mathbf{x}_{t_2}\}^{\{d_1\}\emptyset\emptyset\{d_1\}\emptyset\emptyset}$	$\emptyset t_2\{\mathbf{x}_{t_2}\}^{\{d_2\}\emptyset\emptyset\emptyset\{d_2\}\emptyset}$
$\{d_2\}\emptyset\emptyset^{\{\mathbf{x}_{t_1}\}\emptyset}$	$\emptyset\emptyset\{\mathbf{p}_{d_1}, \mathbf{p}_{d_2}\}^{\{\mathbf{x}_{t_2}\}\{\mathbf{x}_{t_2}\}}$	$\downarrow \mathbf{DUV}(d_1)$ $\{t_1, t_2\}\bot\{\mathbf{x}_{t_1}, \mathbf{x}_{t_2}\}^{\{d_1\}\emptyset\emptyset\{d_1\}\emptyset\{\mathbf{p}_{d_1}\}}$	$\downarrow \mathbf{DUV}(d_2)$ $\emptyset t_2\{\mathbf{x}_{t_2}\}^{\{d_2\}\emptyset\emptyset\emptyset\{d_2\}\{\mathbf{p}_{d_2}\}}$
$\{d_2\}\emptyset\emptyset^{\{\mathbf{x}_{t_1}\}\emptyset}$	$\emptyset\emptyset\{\mathbf{p}_{d_1}, \mathbf{p}_{d_2}\}^{\{\mathbf{x}_{t_2}\}\{\mathbf{x}_{t_2}\}}$	$\downarrow \mathbf{DSVY}(d_1)$ $\{t_1\}\bot\{\mathbf{x}_{t_1}, \mathbf{w}_{t_2}\}^{\{d_1\}\emptyset\emptyset\{d_1\}\emptyset\{\mathbf{p}_{d_1}\}}$	$\downarrow \mathbf{DEVY}(d_2)$ $\emptyset t_2\{\mathbf{w}_{t_2}\}^{\{d_2\}\emptyset\emptyset\emptyset\{d_2\}\{\mathbf{p}_{d_2}\}}$
$\{d_2\}\emptyset\emptyset^{\{\mathbf{x}_{t_1}\}\emptyset}$	$\downarrow \mathbf{TUV}(t_2) \downarrow \mathbf{TUV}(t_2)$ $\emptyset\emptyset\{\mathbf{p}_{d_1}, \mathbf{p}_{d_2}\}^{\{\mathbf{w}_{t_2}\}\{\mathbf{w}_{t_2}\}}$	$\{t_1\}\bot\{\mathbf{x}_{t_1}, \mathbf{w}_{t_2}\}^{\{d_1\}\emptyset\emptyset\{d_1\}\emptyset\{\mathbf{p}_{d_1}\}}$	$\emptyset t_2\{\mathbf{w}_{t_2}\}^{\{d_2\}\emptyset\emptyset\emptyset\{d_2\}\{\mathbf{p}_{d_2}\}}$
$\{d_2\}\emptyset\emptyset^{\{\mathbf{x}_{t_1}\}\emptyset}$	$\downarrow \mathbf{TC}(t_2)$ $\emptyset\emptyset\{\mathbf{c}_{d_2}\}^{\{\mathbf{w}_{t_2}\}\{\mathbf{w}_{t_2}\}}$	$\{t_1\}\bot\{\mathbf{x}_{t_1}, \mathbf{w}_{t_2}\}^{\{d_1\}\emptyset\emptyset\{d_1\}\emptyset\{\mathbf{p}_{d_1}\}}$	$\emptyset t_2\{\mathbf{w}_{t_2}\}^{\{d_2\}\emptyset\emptyset\emptyset\{d_2\}\{\mathbf{p}_{d_2}\}}$
$\{d_2\}\emptyset\emptyset^{\{\mathbf{x}_{t_1}\}\emptyset}$	$\emptyset\emptyset\{\mathbf{c}_{d_2}\}^{\{\mathbf{w}_{t_2}\}\{\mathbf{w}_{t_2}\}}$	$\{t_1\}\bot\{\mathbf{x}_{t_1}, \mathbf{w}_{t_2}\}^{\{d_1\}\emptyset\emptyset\{d_1\}\emptyset\{\mathbf{p}_{d_1}\}}$	$\downarrow \mathbf{DUV}(d_2)$ $\emptyset t_2\{\mathbf{w}_{t_2}\}^{\{d_2\}\emptyset\emptyset\emptyset\{d_2\}\{\mathbf{c}_{d_2}\}}$
$\{d_2\}\emptyset\emptyset^{\{\mathbf{x}_{t_1}\}\emptyset}$	$\emptyset\emptyset\{\mathbf{c}_{d_2}\}^{\{\mathbf{w}_{t_2}\}\{\mathbf{w}_{t_2}\}}$	$\{t_1\}\bot\{\mathbf{x}_{t_1}, \mathbf{w}_{t_2}\}^{\{d_1\}\emptyset\emptyset\{d_1\}\emptyset\{\mathbf{p}_{d_1}\}}$	$\downarrow \mathbf{DC}(t_2)$ $\emptyset\bot\{\mathbf{w}_{t_2}\}^{\{d_2\}\emptyset\emptyset\emptyset\{d_2\}\{\mathbf{c}_{d_2}\}}$
$\{d_2\}\emptyset\emptyset^{\{\mathbf{x}_{t_1}\}\emptyset}$	$\emptyset\emptyset\{\mathbf{c}_{d_2}\}^{\{\mathbf{w}_{t_2}\}\{\mathbf{w}_{t_2}\}}$	$\{t_1\}\bot\{\mathbf{x}_{t_1}, \mathbf{w}_{t_2}\}^{\{d_1\}\emptyset\emptyset\{d_1\}\emptyset\{\mathbf{p}_{d_1}\}}$	$\downarrow \mathbf{DR}(d_2)$ $\{t_1\}\bot\{\mathbf{x}_{t_1}\mathbf{w}_{t_2}\}^{\{d_2\}\emptyset\emptyset\emptyset\{d_2\}\{\mathbf{c}_{d_2}\}}$
$\downarrow \mathbf{TUV}(t_1)$ $\{d_2\}\emptyset\emptyset^{\{\mathbf{x}_{t_1}\}\{\mathbf{x}_{t_2}\}}$	$\emptyset\emptyset\{\mathbf{c}_{d_2}\}^{\{\mathbf{w}_{t_2}\}\{\mathbf{w}_{t_2}\}}$	$\{t_1\}\bot\{\mathbf{x}_{t_1}, \mathbf{w}_{t_2}\}^{\{d_1\}\emptyset\emptyset\{d_1\}\emptyset\{\mathbf{p}_{d_1}\}}$	$\{t_1\}\bot\{\mathbf{x}_{t_1}\mathbf{w}_{t_2}\}^{\{d_2\}\emptyset\emptyset\emptyset\{d_2\}\{\mathbf{c}_{d_2}\}}$
$\downarrow \mathbf{TR}(t_1)$ $\emptyset\emptyset\emptyset^{\{\mathbf{x}_{t_1}\}\{\mathbf{x}_{t_2}\}}$	$\emptyset\emptyset\{\mathbf{c}_{d_2}\}^{\{\mathbf{w}_{t_2}\}\{\mathbf{w}_{t_2}\}}$	$\{t_1\}\bot\{\mathbf{x}_{t_1}, \mathbf{w}_{t_2}\}^{\{d_1\}\emptyset\emptyset\{d_1\}\emptyset\{\mathbf{p}_{d_1}\}}$	$\{t_1\}\bot\{\mathbf{x}_{t_1}\mathbf{w}_{t_2}\}^{\{d_2\}\emptyset\emptyset\emptyset\{d_2\}\{\mathbf{c}_{d_2}\}}$
$\downarrow \mathbf{TP}(t_1)$ $\emptyset\emptyset\{\mathbf{p}_{d_1}, \mathbf{p}_{d_2}\}^{\{\mathbf{x}_{t_1}\}\{\mathbf{x}_{t_2}\}}$	$\emptyset\emptyset\{\mathbf{c}_{d_2}\}^{\{\mathbf{w}_{t_2}\}\{\mathbf{w}_{t_2}\}}$	$\{t_1\}\bot\{\mathbf{x}_{t_1}, \mathbf{w}_{t_2}\}^{\{d_1\}\emptyset\emptyset\{d_1\}\emptyset\{\mathbf{p}_{d_1}\}}$	$\{t_1\}\bot\{\mathbf{x}_{t_1}\mathbf{w}_{t_2}\}^{\{d_2\}\emptyset\emptyset\emptyset\{d_2\}\{\mathbf{c}_{d_2}\}}$
$\emptyset\emptyset\{\mathbf{p}_{d_1}, \mathbf{p}_{d_2}\}^{\{\mathbf{x}_{t_1}\}\{\mathbf{x}_{t_2}\}}$	$\emptyset\emptyset\{\mathbf{c}_{d_2}\}^{\{\mathbf{w}_{t_2}\}\{\mathbf{w}_{t_2}\}}$	$\downarrow \mathbf{DUV}(d_1)$ $\{t_1\}\bot\{\mathbf{x}_{t_1}, \mathbf{w}_{t_2}\}^{\{d_1\}\emptyset\mathbf{p}_{d_1}\{d_1\}\emptyset\{\mathbf{p}_{d_1}\}}$	$\downarrow \mathbf{DUV}(d_2)$ $\{t_1\}\bot\{\mathbf{x}_{t_1}\mathbf{w}_{t_2}\}^{\{d_2\}\emptyset\mathbf{p}_{d_2}\emptyset\{d_2\}\{\mathbf{c}_{d_2}\}}$
$\emptyset\emptyset\{\mathbf{p}_{d_1}, \mathbf{p}_{d_2}\}^{\{\mathbf{x}_{t_1}\}\{\mathbf{x}_{t_2}\}}$	$\emptyset\emptyset\{\mathbf{c}_{d_2}\}^{\{\mathbf{w}_{t_2}\}\{\mathbf{w}_{t_2}\}}$	$\downarrow \mathbf{DSVY}(d_1)$ $\emptyset\bot\{\mathbf{w}_{t_1}, \mathbf{w}_{t_2}\}^{\{d_1\}\emptyset\mathbf{p}_{d_1}\{d_1\}\emptyset\{\mathbf{p}_{d_1}\}}$	$\downarrow \mathbf{DSVY}(d_2)$ $\emptyset\bot\{\mathbf{w}_{t_1}\mathbf{w}_{t_2}\}^{\{d_2\}\emptyset\mathbf{p}_{d_2}\emptyset\{d_2\}\{\mathbf{c}_{d_2}\}}$
$\downarrow \mathbf{TUV}(t_1) \downarrow \mathbf{TUV}(t_1)$ $\emptyset\emptyset\{\mathbf{p}_{d_1}, \mathbf{p}_{d_2}\}^{\{\mathbf{w}_{t_1}\}\{\mathbf{w}_{t_2}\}}$	$\emptyset\emptyset\{\mathbf{c}_{d_2}\}^{\{\mathbf{w}_{t_2}\}\{\mathbf{w}_{t_2}\}}$	$\emptyset\bot\{\mathbf{w}_{t_1}, \mathbf{w}_{t_2}\}^{\{d_1\}\emptyset\mathbf{p}_{d_1}\{d_1\}\emptyset\{\mathbf{p}_{d_1}\}}$	$\emptyset\bot\{\mathbf{w}_{t_1}\mathbf{w}_{t_2}\}^{\{d_2\}\emptyset\mathbf{p}_{d_2}\emptyset\{d_2\}\{\mathbf{c}_{d_2}\}}$
$\downarrow \mathbf{TC}(t_1)$ $\emptyset\emptyset\emptyset^{\{\mathbf{w}_{t_1}\}\{\mathbf{w}_{t_2}\}}$	$\emptyset\emptyset\{\mathbf{c}_{d_2}\}^{\{\mathbf{w}_{t_2}\}\{\mathbf{w}_{t_2}\}}$	$\emptyset\bot\{\mathbf{w}_{t_1}, \mathbf{w}_{t_2}\}^{\{d_1\}\emptyset\mathbf{p}_{d_1}\{d_1\}\emptyset\{\mathbf{p}_{d_1}\}}$	$\emptyset\bot\{\mathbf{w}_{t_1}\mathbf{w}_{t_2}\}^{\{d_2\}\emptyset\mathbf{p}_{d_2}\emptyset\{d_2\}\{\mathbf{c}_{d_2}\}}$

Figure 7.1: An execution of a multiple transaction commit protocol.

7.6 Verifying Isolation Levels

In this section we will prove that the **NP** phenomena defined earlier in chapter 6 are prevented by our rules. We will then go on to show that the prepare synchpoint property holds and thus deduce that any distributed schedule, of our model, is indeed serializable. To do this we must first define when a sequence of rules in an execution constitutes a phenomenon. Clearly if data object d executes rule **DR** (**DW**) when $@d(d \in t.R)$ ($@d(d \in t.W)$) holds in the rule's pre-condition then a read (write) at d for transaction t has occurred which we previously denoted $r_t[d]$ ($w_t[d]$). Similarly, a data object d acts on the receipt of a prepare, commit or abort request in the rules **DSVY** or **DEVY**, **DC** and **DA**, we say t prepares, commits or aborts at t which we previously denoted p_t, c_t and a_t.

The execution in figure 7.5 therefore gives rise to the following distributed schedule. In our simple model we only allow one object per site and so we do not need to distinguish between data objects and the sites where they reside.

$$\begin{array}{llllll} \sigma_{d_1}: & r_{t_1}[d_1] & r_{t_2}[d_1] & p_{t_2} & p_{t_1} & \\ \sigma_{d_2}: & w_{t_2}[d_2] & p_{t_2} & c_{t_2} & r_{t_1}[d_2] & p_{t_2} \end{array}$$

Proposition 8 **NP0**, **NP1**, **NP2L**, **NP2R**$'$ *cannot occur in any schedule produced by the rules of our model.*

Proof We will consider each phenomenon in turn and show that they are prevented by the rules of our system.

> **NP0**: $w_i[d] \prec w_j[d] \prec (c_i \wedge c_j)$. $w_i[d]$ happens when d applies the **DW**(d) rule. In the post-action of this rule $e := t_i$. The pre-condition of **DW**(d) contains the clause $e = t$ and therefore while $e = t_i$, d cannot apply **DW** for any other transaction t_j. The only other rules that change the value of e are **DC**(d), **DA**(d) and **DEVN**(d). We need not consider the cases **DA**(d) and **DEVN**(d) since c_i must occur in **NP0**. **DC** corresponds to c_i so we know $w_j[d] \not\prec c_i$ as required.

> **NP1**: $w_i[d] \prec r_j[d] \prec (c_j \wedge a_i)$ and **NP2L**: $w_i[d] \prec r_j[d] \prec (c_i \wedge c_j)$. A very similar argument to the one presented in **NP0**. This time while $e = t_i$, d cannot apply rule **DR** for any other transaction t_j.

> **NP2R**$'$: $r_i[d] \prec w_j[d] \prec (p_i \wedge c_j)$ $r_i[d]$ happens when d applies the **DR**(d) rule. In the post-action of this rule $S := S \cup \{t_i\}$. The pre-condition of **DW**(d) contains the clause $S = \emptyset$ and therefore while $t_i \in S$, d cannot apply **DW** for any other transaction t_j. The only rules that remove t_i from

S are **DSVY**(d) and **DSVN**(d). **DSVN**(d) can not happen since t_j must commit and **DSVY**(d) corresponds to p_i so we know $w_j[d] \not\prec p_i$ as required.

□

Proposition 9 *Synchpoint prepare is guarenteed in any execution produced by the rules of our system.*

Proof We must show for any transaction t *all* read and write accesses for t are before *any* prepare action. Suppose this is not the case then there exists an execution where a prepare rule for t, either **DSVY** or **DEVY**, is before a read or write rule, either **DR** or **DW**, for t. In the pre-condition of both rules **DSVY** and **DEVY** we have the clause $@d(\mathbf{p}_d \in t.A)$. For this to happen d must have applied the rule **DUV** to update its view of transaction t's $t.A$ set and before that t must have applied the rule **TP**(t). The pre-condition to **TP** is $R = \emptyset \wedge W = \emptyset$. Clearly in order for both these sets to be empty t must have executed **TR** or **TW** for each of its read and write accesses, the pre-condition to both these rules contains the clause $@t(\mathbf{x}_t \in d.R)$ and so all read and write rules **DR** and **DW** for t must have already been applied.

□

Theorem 6 *All distributed schedules produced by the rules of our system are serializable.*

Proof By proposition 8, no **NP** phenomena are present by proposition 9, synchpoint prepare is guaranteed and so by theorem 5 all distributed schedules are serializable.

□

7.7 Overlapping Prepare

To increase transaction throughput, in transaction processing systems, many different strategies have been used. Many rely on increasing transaction concurrency. As we have seen before, in the read-only optimisation, an effective strategy to this end is for a distributed transaction to release its locks as early as possible. The transactions in our model release their read locks as soon as they have received acknowledgements for *all* their accesses (i.e. at prepare time). Another strategy is to send a `prepare` message to any read-only site as soon as the transaction

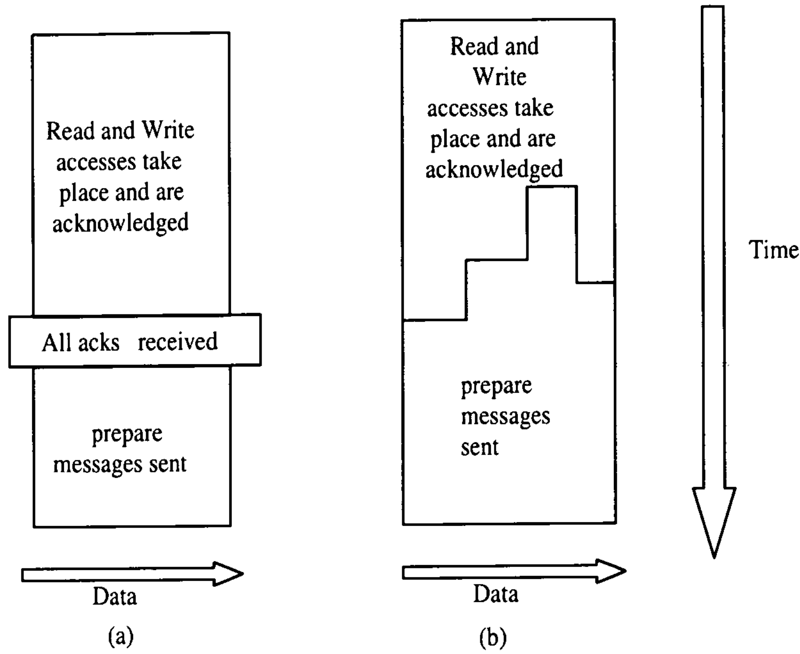

Figure 7.2: In (a) a transaction collects acknowledgements for all accesses before any prepare messages are sent. In (b) prepare messages are overlapped with accesses to increase transaction concurrency.

has received an acknowledgements *from that site only*, that all read accesses have taken place. In this strategy the transaction process need not wait until it has received acknowledgements from every site before sending a prepare message. In Microsoft's Distributed transaction manager [20], just such a facility exists. Figure 7.2 depicts the two different strategies.

It is interesting to model this new strategy using our rules, which we do by making some small changes. First we add $A := A \cup \{\mathbf{p}_d\}$ to the post-action of **TR**(t) thus.

$$\mathbf{TR}(t)\ \frac{@t(\mathbf{x}_t \in d.X) \wedge d \in R}{R := R - \{d\} \wedge A := A \cup \{\mathbf{p}_d\}}$$

This change means a transaction can start to prepare a read-only data object as soon as it receives an acknowledgement of its read. The only other change we must make is to the definition of PREPARE_t. This now becomes $\text{PREPARE}_t \stackrel{def}{=} \{\mathbf{p}_{d'} \mid d' \in t.W\}$. This means that in the post-action of **TP**, prepare messages will only be sent to data objects that have been written to.

It is interesting to ask, what level of transaction isolation is now attained. The level of isolation claimed by Microsoft for this type of strategy in their distributed transaction server is READ COMMITTED. Clearly, full or serializable isolation

is not attained; consider the example below.

$$\begin{array}{llllllll} \sigma_{d_1}: & r_{t_1}[d_1] & p_{t_1} & w_{t_2}[d_1] & c_{t_2} & & & \\ \sigma_{d_2}: & & & w_{t_2}[d_2] & c_{t_2} & r_{t_1}[d_2] & c_{t_1} & \end{array}$$

If we assume in this example that $w_{t_2}[d_1]$ and $w_{t_2}[d_2]$ happen at d_1 and d_2 at the same time then the schedule is not serializable. t_1 reads the value of d_1 before t_2 writes to it, whereas t_1 reads the value of d_2 after t_2 writes to it.

Interestingly, the proof of proposition 8 still holds. The problem here is that we have lost the synchpoint prepare property. It is no longer the case that all accesses for a transaction must take place before any prepare actions. The lack of synchpoint prepare results in distributed schedules no longer being serializable (each local schedule however does remain serializable). Although it was not performed we could model check this version of our model to automatically verify that the synchpoint property does not hold.

7.8 Conclusions

In this chapter we studied the way a commit protocol is used within a distributed transaction to provide a level of transaction isolation. In chapter 6 we saw how to define transaction isolation levels for transaction schedules produced by non-distributed transactions. In this chapter we extended these definitions for distributed transaction schedules. In particular we introduced the idea of when a transaction prepares within a distributed transaction.

Using these extensions we formally defined distributed serializability. The **NP** rules of chapter 6 were extended for distributed schedules and **NP2R** was modified to take account of the prepare point in a distributed transaction. In the non-distributed case the exclusion of **NP** phenomena guaranteed serializability, but this exclusion alone does not guarantee serializability for distributed transactions. If a further condition, that we defined and named synchpoint prepare, holds then distributed schedules are serializable.

In order to study the role of commit providing a mechanism whereby the **NP** phenomena are excluded and synchpoint prepare property is provided, we constructed a very simple views based transaction processing system model. This model incorporated centralised 2PC. In this model we showed that the **NP** phenomena are indeed excluded. We also showed how 2PC helps provide synchpoint prepare and concluded that schedules of this model are serializable.

A commercial optimisation technique to increase transaction concurrency is often used, for example in Microsoft's DTC [20]. In this optimisation a transaction's accesses and prepare messages are overlapped. By changing our model

slightly we capture this situation and show that the synchpoint prepare property is lost. Thus the extra transaction concurrency is at the cost of a loss of serializability in the resulting schedules.

The chapter demonstrates how our modelling technique can be used to model the more complex situation where many concurrent transaction interact while executing a commit protocol. We see how we can formally describe this complex behaviour and analyse properties of this behaviour, for example the level of transaction isolation attained within the system.

Chapter 8

Conclusions and Future Research Directions

8.1 Introduction

In this last chapter we will summarise the work presented in this dissertation. In so doing we will highlight the research contributions made and discuss further possible research directions.

The thesis centres around the views based modelling technique presented. During this summary therefore, we will support our claim that the modelling technique is suitable for modelling and analysing commit protocols and the environments in which they execute.

In particular we will highlight evidence that, as claimed in the introduction, our views based model is formal enough to support rigourous arguments about the behavioural properties of commit protocols as well as supporting automated techniques such as model checking. Our views based model is flexible enough to model a wide variety of commit protocols, and is scalable in the sense that the arguments constructed using the model can be applied to arbitrary numbers of processes.

8.2 Commit Literature and Modelling Techniques

We started the dissertation by reviewing some literature on the problem of atomic commit. This review concentrated on the types of environments in which commit executes describe some formal models that have been used for commit protocols.

Bernstein *et al.* provide an early definition of the problem of commit and introduce us to 2PC. The subject of blocking has attracted much academic interest and it was discussed with the introduction of Skeen and Stonebreaker's formal

model of a commit protocol. In order to give a flavour of the different environments and versions of commit protocols we described a whole host of different commit protocols and protocol optimisations. We saw as the protocols became more complex the models used in their description became richer. For example the presumed abort 2PC optimisation reduces the extent to which logging needs to be carried out. A model of presumed abort therefore must model a log if we are to reason about its behaviour. Paralleling these richer environments we also saw richer behavioural properties expressed. For example, if network failure is modelled then we can pose questions about blocking. It soon came apparent that because of the huge diversity in the different environments, it is difficult to define the problem of commit precisely for all environments. Furthermore it is difficult to encompass all these models using one modelling technique.

A key component of the environment in which atomic commit executes is the extent to which failure is modelled. Many different models of failure have been proposed. The diversity of failure assumptions again adds another dimension to the task of modelling the environments in which commit protocols execute.

Over the years atomic commit protocols have found applications in many different areas. In fact, the atomicity property of commit protocols is so useful we find commit or commit style protocols in the most unlikely areas. Finding new applications for atomic commit protocols, particularly in e-commerce and Internet applications, is a whole area of active research.

Although many different models have been proposed, no one model, or modelling technique, seems to be generally applicable. Furthermore, assumptions about the distributed environment in which the atomic commit protocols are studied varies widely. This is particularly true of assumptions made about site and communication failure. For these reasons it is difficult to mould the various models of commit into a hierarchy.

By way of introducing our views based model we studied three general modelling techniques used to model asynchronous message passing distributed systems. I/O automata have been widely used to model and specify distributed systems. These models support a powerful pre-condition post-action style specification of the behaviour of processes within a system. Knowledge based models were introduced by Hadzilacos. In these models processes communicate by sending messages but a more declarative semantics is given to the processes within the system. The knowledge based technique is formal, concise and provides a good abstraction to the explicit message passing details that are found in many other models. The calculus for communicating systems is a very general technique for

modelling distributed systems of agents. Agents communicate by handshaking. A formal transition semantics is given for CCS agents which means that properties of agents can be easily automatically verified using model checking. Unfortunately it is difficult to specify complex protocols using CCS. Message passing details are explicit which often results in very large transition systems.

Our views based technique draws upon some of the strengths of these models. We use the expressive pre-condition post-action methods of I/O automata to specify behaviour. The main novelty is the incorporation of message passing within a processes state in the form of views. A pre-condition can contain a clause of the form:

$$@p(q.s = \mathbf{x})$$

meaning that p's most up-to-date knowledge of q's state variable s is that its value is $\mathbf{x}$.

In all of these models an execution of a protocol is expressed using a sequence of configurations. A configuration captures the global state of the processes in the system and the environment in which they execute. By taking steps (applying rules) a configuration evolves. A protocol execution therefore is a sequence of configurations.

8.2.1 Future Research Directions

Although there is a large amount of academic research on atomic commit protocols, almost all protocols commercially in use are based around centralised two-phase commit. The subject of blocking has received much attention from academics but in practice it is enough to take steps, such as the help-me messages we modelled in chapter 4, to relieve the problem and live with its consequences. The family of three-phase commit protocols are unlikely to be of use within traditional environments where high transaction throughput is very important. This may soon change however. Wireless communication systems and Internet communication increases the likelihood of partitioning networks, and highly variable message propagation times. In e-commerce style transactions, throughput may not be of paramount importance over reliability.

These new transaction processing environments are likely to give rise to a whole new set of requirements. Already we see requirements for commit style protocols which involve parties that may not trust one another [44, 54], reaching agreement.

8.3 Putting our Model to Work

In order to demonstrate the flexibility of our views based model we used it to model the simple centralised 2PC. We achieved this using only six protocol rules.

Our simple model did not include any types of failure, and so, in order to model failure we introduced the partitioning model of communication failure where processes become disconnected into groups. We showed in this enriched model that it is possible to formally capture properties like blocking and we produced protocol executions, in our model, that did indeed block.

By adding a buffer state to 2PC we arrive at 3PC which is the basis for a family of commit protocols called quorum based three phase commit protocols. In this more complex class of protocols blocking is avoided in quorate connected components. We use our model to describe a quorum version of 3PC, called Q3PC, and model the recent extended version of Q3PC called E3PC.

Our model allows us to reason about the behaviour quite naturally even for the more complex protocols such as E3PC. This lead to the innovation of our new quorum based 3PC, which we named X3PC. A views based model of X3PC is derived from a model of E3PC by enriching the state of a process in the E3PC model and adding some protocol rules. We formally prove that X3PC provides a solution to the atomic commit problem, demonstrating the model's applicability in this arena. The traditional proof technique we use makes reference to the order that rules of the protocol obey in any protocol execution. We show that X3PC provides an identical level of tolerance to blocking as E3PC but that it commits in many of the failure scenarios in which E3PC aborts.

By deriving quite complex commit protocols, in a non-trivial environment, where communication failure is possible, we see that our views based model provides us with a flexible modelling technique. We also demonstrate that it provides good support for reasoning about commit protocol properties by proving behavioural properties of the most complex protocol presented, X3PC.

8.3.1 Future Research Directions

Our modelling technique was tailored to commit protocols but it can be used for many other types of protocols involving groups of homogenous agents. It is particularly useful in protocols involving agreement amongst these agents. It is an interesting research question to define the types of protocols that it is applicable to, its strengths and limitations.

In order to further increase the performance of X3PC one could include, in a

process's state, histories of other processes' histories. This would further increase a terminations protocol's chances of reaching commit over abort. More analysis could be performed to analyse X3PC's performance. One strategy similar to the techniques of Peleg and Wool [63] would be to consider the behaviour of X3PC under all possible partitioning scenarios.

It is possible to generate a transition system, as we did in chapter 5 from a model of X3PC. The state of a process in X3PC is much larger than the simple example we considered in chapter 5 but in theory the same principles apply. Using the abstraction techniques we discussed it should be possible to automatically verify properties of complex protocols such as X3PC using the same techniques.

8.4 Automatic Verification Techniques

It is possible to express the behaviour of a commit protocol by expressing all its possible execution paths as a labelled transition system. Using a views based model of a simple centralised 2PC protocol as a starting point we showed how a transition system can be automatically generated. The size of resulting transition system depends on how the global state of the system, the configuration, is represented. We described how to generate a transition system using two different representations. CON, a concrete representation, produces very large transition systems, MULTI gives rise to smaller ones.

Using these representations we describe a simulation technique that allows us to reason about the behaviour in the concrete representation, CON using the abstract representation MULTI.

A sub-logic of CTL we name CTL^- is defined. Using this logic we can formally express properties of commit protocols. The logic CTL^- cannot express counting properties. Its inability to express these properties means that the properties it does capture are preserved between our representations. This means we can show, for our simple 2PC model, that if a CTL^- property holds (fails) by model checking in the MULTI representation then it will also hold (fail) if the CON representation was used.

Using this strategy we verify many properties of the simple views based model automatically using a games based model checking algorithm. This demonstrates that our model supports model checking techniques for automated verification.

8.4.1 Future Research Directions

The success of the abstraction techniques we discussed depend on the fact that our processes communicate using views and also that they are largely homogeneous in their behaviour. They all follow the same set of rules.

Many different protocols can be modelled and model checked using our views based models . For example, recently Chkliaev [13] formally prove properties of a non-blocking atomic broadcast protocol first proposed by Babaoğlu and Toueg [4], using a automated proof tool called PVS. Our techniques are highly suitable for this protocol and some preliminary research has been carried out in to model check CTL^- properties of this protocol. Another avenue to explore involves leadership election protocols in partitioning networks. The extent to which our techniques are applicable is an open research question.

Although we only model check a very simple protocol in this thesis it should be clear how we can extend our models to allow the model checking of more complex systems. In particular we could quite easily add partition failure to our simple two-phase commit and model check the resulting system.

8.5 Commit and Isolation

Commit protocols help ensure distributed transactions are atomic they also play a role in providing transaction isolation. In order to study the role of a commit protocol in this area we must first define what it means for a distributed transaction to attain a particular level of isolation.

Unfortunately, definitions of isolation levels in the literature, for even non-distributed transactions are not clear. A good starting point is the ANSI 92 standard, but the definitions found within are too ambiguous for our purposes. A critical analysis of the ANSI definitions was given by Berenson *et al.*, where clearer definitions were presented. We built on these definitions and further refined them. Like Berenson *et al.* our definitions were based around phenomena. For a specific isolation level particular sets of phenomena are disallowed in a schedule. These types of definitions are completely independent of assumptions about particular concurrency and recoverability mechanisms (e.g. locking) that might be in place. Furthermore they are weaker, than those presented by Berenson *et al.*, in the sense that at a particular isolation level more concurrency is permitted

Once we were able to define what it meant for a centralised transaction to attain a particular isolation level, we could then proceed to generalise our definitions to distributed transactions. In so doing, we saw that although excluding

phenomena in local schedules provided serializable isolation in each local schedule is does not provide a level of serializable isolation for the complete distributed schedule. It turns out that, if we supplement this with an extra condition we named synchpoint prepare, it is enough to ensure a serializable level of isolation for distributed transactions.

Using our views based model we modelled a very simple transaction processing system. The system included a two phase commit protocol. We showed that the phenomenon defined in the previous chapter are excluded in any execution of our system. Furthermore, we showed that 2PC does provide the synchpoint prepare property and thus we can conclude only serializable schedules are produced.

A common optimisation overlaps the accesses of distributed transactions with their commit phase. This increases transaction concurrency and thus performance. We made some changes to our model to reflect this situation and then showed that the synchpoint prepare property was lost.

Once again we have demonstrated the power of our modelling technique. By defining isolation level as a restriction on the possible ordering of rules within a protocol execution we are able to verify that a protocol supports a particular level of isolation. We can appeal directly to the rules of our protocol in order to verify that certain phenomena are prevented. Furthermore, in the case that synchpoint prepare is not provided we can exhibit a counter example in the form of a protocol execution.

8.5.1 Future Research Directions

Model checking techniques could be applied to these and similar models. The same abstraction methods could be used as those found in chapter 5 to provide automated checking of properties for arbitrary numbers of concurrently executing transactions. Although in the isolation properties counting does count to a certain extent processes are still largely anonymous. A possible strategy therefore would be to name two transactions and two data objects and then model all others generically as a third anonymous transaction and data object. We would then be required to show that this abstraction preserves certain properties. In particular if an isolation phenomenon exists in the multi–process model it will also exist in our abstraction.

It is also interesting to consider other levels of transaction isolation and how they are defined. As new applications such as Internet transaction processing and mobile e-commerce protocols become more prevalent it is likely that these lower levels of isolation will become more important. For example, in mobile commerce,

hand held mobile devices may perform transactions while periodically connected to larger systems guaranteeing only READ UNCOMMITTED levels of isolation to ensure large levels of concurrency.

8.6 Concluding Remarks

Throughout this dissertation we have seen how the views based modelling technique can be used to model and analyse the behaviour of commit protocols. The process of modelling a commit protocol often provides insight into its behaviour which can lead to further improvements. An example of this was the derivation of X3PC from E3PC. A novel feature of our technique allows us to incorporate message passing details as views within a process's state. This provides a a solid and scalable basis on which to generate transition systems that describe protocol behaviour, which in turn supports abstraction techniques, that allow automated proofs of properties of systems using model checking. In order to study the role of commit in providing transaction isolation we first constructed a formal account of transaction isolation. Our model then captured a simple transaction processing system and we were able to prove different levels of isolation were supported within this system.

The techniques presented in this thesis are of little direct practical use for today's protocol designers and software engineers. They are however a step in the right direction. The extent to which these and similar tools are adopted will largely depend on how easy they are to use and how useful their output is. It is unlikely they will be palatable in their current state but they could form the basis of a suite of software tools. Model checking has found commercial applications in hardware design and is being used successfully by companies like Intel. The extra effort involved with model checking is justified in this case due to the large costs associated with making changes to hardware once in production. As protocols are embedded as firmware in devices such as smart phones a formal techniques such as model checking will likely be justified on a similar basis.

Bibliography

[1] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Heartbeat: a timeout-free failure detector for quiescent reliable communication. In *pro-WDAG97*, pages 126–140, September 1997.

[2] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. On the weakest failure detector for quiescent reliable communication. Technical Report TR97-1640, Cornell University, Computer Science, July 18, 1997.

[3] ANSI. ANSI x3.135-1992. *American National Standard for Information Systems–Database Language–SQL*, November 1992.

[4] Ö. Babaoğlu and S. Toueg. Non-blocking atomic commitment. In *Distributed systems*, pages 147–168. ACM Press/Addison-Wesley, second edition, 1993.

[5] A. Basu, B. Charron-Bost, and S. Toueg. Simulating reliable links with unreliable links in the presence of process crashes. In Ö. Babaoğlu and K. Marzullo, editors, *Distributed Algorithms - Proceedings of the Tenth International Workshop, WDAG'96*, pages 105–122. Springer, 1996.

[6] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. *ACM SIGMOD Record*, 24(4), 1995.

[7] P. Bernstein, W. Emberton, and V. Trehan. DECdta–digital's distributed transaction processing architecture. *Digital Technical Journal*, 3(1):Winter 1991, 1991.

[8] P.A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, June 1981.

[9] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.

[10] P.A. Bernstein and E. Newcomer. *Principles of Transaction Processing.* Morgan-Kaufmann, San Mateo, CA, 1997.

[11] T.D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[12] D. Cheung and T. Kameda. Site-optimal termination for a distributed database under netwrok partitioning. In *Proceedings of the 4th ACM-SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 111–121, Minaki, Ontario, August 1985.

[13] Dmitri Chkliaev, Peter van de Stok, and Jozef Hooman. Mechanical verification of a non-blocking atomic commitment protocol. In *Proceedings of the International Workshop on Distributed System Validation and Verification*, pages 96–103, 2000.

[14] P.K. Chrysanthis, G. Samaras, and Y.J. Al-Houmaily. Recovery and performance of atomic commit processing in distributed database systems. In V. Kumar and M. Hsu, editors, *Recovery Mechanisms in Database Systems*, chapter 13, pages 370–416. Prentice-Hall, New Jersey, 1998.

[15] G. Clark, S. Gilmore, and J. Hillston. Specifying performance measures for PEPA. *Lecture Notes in Computer Science*, 1601:211–227, 1999.

[16] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[17] Edmund M. Clarke. Temporal logic model checking: Two techniques for avoiding the state explosion problem. *Lecture Notes in Computer Science*, 531, 1991.

[18] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM*, pages 343–354, 1992.

[19] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

[20] Microsoft Corporation. *Microsoft Distributed Transaction Coordinator Resource Manager Implementation Guide*, January 1996. Version 6.5.

[21] R. Das and A. Fekete. Modular reasoning about open system: a case study of distributed commit. In *Proceedings of the Seventh International Workshop on Software Specification and Design*, pages 30–39, Redondo Beach, CA, December 1993. IEEE Computer Society Press.

[22] J. Davidson. *An introduction to TCP/IP.* Springer-Verlag, 1988.

[23] L. Davis. Peering at the LU 6.2 choice. *Datamation*, 36(3):49–50, 52, February 1990.

[24] D. Dolev, R. Friedman, I. Keidar, and D. Malkhi. Failure detectors in omission failure environments. Technical Report 96-1608, Dept. of Computer Science, Cornell University, September 1996.

[25] D. J. Faber and F. R. Heinrich. The structure of a distributed computer system–the distributed file system. In *Proc.1st Internat.Conf.on Computer Communications*, October 1972.

[26] Robert Felice. Implementing the CCITT cyclic redundancy check. *C Users Journal*, 8(9):61–77, September 1990.

[27] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty processor. *Journal of the ACM*, 32(2):374–382, 1985.

[28] M. Franklin. Concurrency control and recovery. *Handbook of Computer Science*, pages 334–368, 1996.

[29] H. Garcia-Molina. Elections in a distributed computing system. *IEEE Transactions on Computers*, C-31(1):48–59, January 1982.

[30] D. Gawlick, M. Haynie, and A. Reuter. A distributed, high–performance, high–availability implementation of SQL. In *NonStop SQL*, volume 359 of *Lecture Notes in Computer Science*, New York, N.Y., 1989. Springer-Verlag.

[31] D. Gawlick and D. Kinkade. Varieties of concurrency control in IMS/VS fast path. In *IEEE CS Technical Com. on Database Engineering Bulletin*, volume 8, June 1985.

[32] J. Gray. Notes on database operating systems. In *Operating Systems An Advanced Course*, pages 394–481. LNCS Springer Verlag, Berlin, 1978.

[33] J. Gray. A comparison of the byzantine agreement problem and the transaction commit problem. In B. Simon and A. Spector, editors, *Fault Tolerant Distributed Computing*, pages 10–17. Springer Verlag, Berlin, 1990. Lecture Notes in Computer Science, 448.

[34] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, San Mateo, CA, 1993.

[35] R. Guerraoui. Revisiting the relationship between non-blocking atomic commitment and consensus. In J.-M. Helary and M. Raynal, editors, *Proceedings of the 9th International Workshop on Distributed Algorithms*, pages 87–100. Springer Verlag, 1995.

[36] V. Hadzilacos. A knowledge theoretic analysis of atomic commitment protocols. In *ACM Principles of Database Systems (PODS)*, pages 129–134, 1987.

[37] J. Y. Halpern and Y. O. Moses. Knowledge and common knowledge in distributed environments. In *Proceedings of the 3rd ACM Conference on Principles of Distributed Computing.* ACM Press, 1984.

[38] Joseph Halpern and Yoram Moses. Knowledge and common knowledge in a distributed environment. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, pages 50–61, 1984.

[39] R. W. Hamming. *Coding and Information Theory.* Prentice-Hall, 1986.

[40] M. Hesselgrave. Considerations for building distributed transaction processing systems on unix system v. In Washington, editor, *Proceedings of UNIFORM*, January 1990.

[41] I. Keidar and D. Dolev. Increasing the resilience of atomic commit, at no additional cost. In *Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 245–254, 1995.

[42] T. Kempster, G. Brebner, and P. Thanisch. A transactional approach to network management. In *Pproceedings of the 1999 Workshop on Databases in Telecommunications*, pages 224–252. Springer-Verlag, 1999.

[43] T. Kempster, C. Stirling, and P. Thanisch. A more committed quorum-based three phase commit protocol. In *LNCS: The Twelth International Symposium on Distributed Computing*, pages 246–257, 1998.

[44] T. Kempster, C. Stirling, and P. Thanisch. A critical analysis of the transaction internet protocol. In *Proceedings of the Second International Conference on Telecommunications and Electronic Commerce (ICTEC)*, pages 245–271, 1999. Longer version to Journal of Electronic Commerce Research.

[45] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[46] B. Lampson and D. Lomet. A new presumed commit optimisation for two phase commit. Research Note CRL 93/1, Digital Equipment Corporation, Cambridge Research Laboratory, 1 Kendall Square, Cambridge, MA 02139, February 1993.

[47] B. W. Lampson. Atomic transactions. In *Distributed Systems–Architecture and Implementation*, volume 105 of *Lecture Notes in Computer Science*, pages 246–265. Springer-Verlag, New York, N.Y., 1981. This is a revised version of Lampson and Sturgis's unpublished *Crash Recovery in a Distributed Data Storage System.*

[48] C.-S. Li, C.J. Georgiou, and K.W. Lee. A hybrid multilevel control scheme for supporting mixed traffic in broadband networks. *IEEE Journal on Selected Areas in Communications*, 14(2):306–316, February 1996.

[49] S. Luan and V.D. Gligor. A fault-tolerant protocol for atomic broadcast. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):271–285, July 1990.

[50] N. A. Lynch, M. Merritt, W. E. Weihl, and A. Fekete. *Atomic Transactions.* Morgan Kaufmann, San Mateo, 1994.

[51] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, 1989.

[52] Nancy A. Lynch. *Distributed Algorithms.* Morgan-Kaufmann, San Francisco, CA, 1996.

[53] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In Fred B. Schneider, editor, *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 137–151, Vancouver, BC, Canada, August 1987. ACM Press.

[54] Moses Ma. Agents in e-commerce. *Communications of the ACM*, 42(3):79–91, March 1999.

[55] R. Milner. A calculus of communicating systems. *Lecture Notes in Computer Science*, 92, 1980.

[56] C. Mohan and D. Dievendorff. Recent work on distributed commit protocols, and recoverable messaging and queuing. *Data Engineering Bulletin*, 17(1):22–28, March 1994.

[57] C. Mohan, B. Lindsay, and R. Obermark. Transaction management in the R* distributed database management system. *ACM Transactions on Database Systems*, 11(4):378–396, December 1986.

[58] F. Moller and P. Stevens. The Edinburgh Concurrency Workbench. Technical report, University of Edinburgh, July 1999. On-line, http://www.dcs.ed.ac.uk/home/cwb/doc/manual.pdf.

[59] N. Francez. *Fairness*. Springer-Verlag, New York, 1987.

[60] T. Nipkow. Formal verification of data type refinement — theory and practice. In J. W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness, REX Workshop, Mook, The Netherlands*, volume 430 of *Lecture Notes in Computer Science*, pages 561–591. Spring-Verlag, May/June 1989.

[61] Robert Orfali and Dan Harkey. *Client-Server Programming with Java and CORBA*. La Nuova Italia, Firenze, Italy, 19xx.

[62] M.T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[63] David Peleg and Avishai Wool. The availability of quorum systems. *Information and Computation*, 123(2):210–223, December 1995.

[64] J Postel. Internet datagram protocol RFC791. *USC/Information Sciences Institute, RFC 791*, September 1981.

[65] S. Ramsay, P. Thanisch, R. Pooley, S. Gilmore, and J. Numenmaa. Interactive simulation of distributed transaction processing commit protocols. In *Proceedings of the Third UK Simulation Society Conference*, April 1997.

[66] A. Ricciardi, A. Schiper, and K. Birman. Understanding partitions and the "no partition" assumption. In *Proceedings of the 4th IEEE Computer Society Workshop on Future Trends in Distributed Computing Systems (FTDCS-4)*, pages 354–360, Lisbon, Portugal, September 1993.

[67] I. Rojas. General marking-dependent rates and probabilities in gspns. In *Proceedings of UK Performance Engineering of Computer and Telecommunications Systems*, pages 138–152. Springer, 1995.

[68] G. Samaras, K. Britton, A. Citron, and C. Mohan. Two-phase commit optimisations in a commercial distributed environment. *Distributed and Parallel Databases*, 3(4):325–360, October 1995.

[69] D. Skeen. Nonblocking commit protocols. In *Proceedings of the ACM SIGMOD Conference on the Management of Data (SIGMOD'81)*, pages 133–142, 1981.

[70] D. Skeen. Crash recovery in a distributed database system. Technical report, University of California at Berkeley, 1982.

[71] D. Skeen. A quorum-based commit protocol. In *Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 69–80, February 1982.

[72] D. Skeen and M. Stonebraker. A formal model of crash recovery in a distributed system. *IEEE Transactions on Software Engineering*, SE-9(3):220–228, May 1983.

[73] J. Søgaard-Andersen, S. Garland, J. Guttag, N. A. Lynch, and A. Pogosyants. Computer-assisted simulation proofs. In C. Courcoubetis, editor, *Proceedings of the 5th International Conference on Computer Aided Verification,* Elounda, Greece, volume 697 of *Lecture Notes in Computer Science*, pages 305–319. Springer-Verlag, 1993.

[74] P.M. Spiro, A.M. Joshi, and T.K. Rengarajan. Designing an optimized transaction commit protocol. *Digital Technical Journal*, 3(1):1–10, 1991.

[75] J.W. Stamos and F. Cristian. A low-cost atomic commit protocol. In *Proceeding of the Ninth Symposium on Reliable Distributed Systems*, pages 66–75. IEEE Comput. Soc. Press, 1990.

[76] C. Stirling. Local model checking games. In Insup Lee and Scott A. Smolka, editors, *Proceedings of the 6th International Conference on Concurrency Theory (CONCUR'95)*, volume 962 of *LNCS*, pages 1–11, Berlin, GER, August 1995. Springer.

[77] C. Stirling. Bisimulation, modal logic and model checking games. *Logic Journal of the IGPL*, 7(1):103–124, 1999.

[78] M. Stonebraker. Concurrency control in distributed ingres. *IEEE Transactions on Software Engineering*, 5(3):188–194, 1979.

[79] Andrew S. Tanenbaum. *Computer Networks (Third Edition)*. Prentice–Hall, Upper Saddle River, NJ 07458, 1996.

[80] P. Thanisch. Atomic commit in concurrent computing. *IEEE Concurrency*, 8(4):34–41, December 2000.

[81] J.D. Tygar. Atomicity in e-commerce. *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing*, pages 8–26, May 1996.