Copyright © and Moral Rights for this thesis are retained by the author and/or other copyright owners. A copy can be downloaded for personal non-commercial research or study, without prior permission or charge. This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the copyright holder/s. The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the copyright holders.

When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given e.g.

AUTHOR (year of submission) "Full thesis title", University of Southampton, name of the University School or Department, PhD Thesis, pagination
Rigorous Design of Distributed Transactions

by

Divakar Singh Yadav

Thesis for the degree of Doctor of Philosophy

February 2008
Database replication is traditionally envisaged as a way of increasing fault-tolerance and availability. It is advantageous to replicate the data when transaction workload is predominantly read-only. However, updating replicated data within a transactional framework is a complex affair due to failures and race conditions among conflicting transactions. This thesis investigates various mechanisms for the management of replicas in a large distributed system, formalizing and reasoning about the behavior of such systems using Event-B. We begin by studying current approaches for the management of replicated data and explore the use of broadcast primitives for processing transactions. Subsequently, we outline how a refinement based approach can be used for the development of a reliable replicated database system that ensures atomic commitment of distributed transactions using ordered broadcasts.

Event-B is a formal technique that consists of describing rigorously the problem in an abstract model, introducing solutions or design details in refinement steps to obtain more concrete specifications, and verifying that the proposed solutions are correct. This technique requires the discharge of proof obligations for consistency checking and refinement checking. The B tools provide significant automated proof support for generation of the proof obligations and discharging them. The majority of the proof obligations are proved by the automatic prover of the tools. However, some complex proof obligations require interaction with the interactive prover. These proof obligations also help discover new system invariants. The proof obligations and the invariants help us to understand the complexity of the problem and the correctness of the solutions. They also provide a clear insight into the system and enhance our understanding of why a design decision should work.

The objective of the research is to demonstrate a technique for the incremental construction of formal models of distributed systems and reasoning about them, to develop the technique for the discovery of gluing invariants due to prover failure to automatically discharge a proof obligation and to develop guidelines for verification of distributed algorithms using the technique of abstraction and refinement.
## Contents

**Acknowledgements** ix

1  **Introduction** 1
   1.1  Motivation .............................................. 1
   1.1.1  Data Replication ...................................... 1
   1.1.2  Broadcast Primitives .................................. 2
   1.2  Why use Formal Methods for Data Replication ? .......... 4
   1.3  Related Work ............................................. 5
   1.4  Our Contributions ......................................... 8
   1.5  Chapter Outline ........................................... 8

2  **Background** 11
   2.1  Preliminaries ............................................. 12
   2.1.1  A Database Transaction ................................. 12
   2.1.2  Long Running Transactions and Compensations .......... 12
   2.1.3  Distributed Transactions ................................ 13
   2.1.4  Updating Distributed Data ............................... 13
   2.2  Failures in Distributed Databases ......................... 14
   2.2.1  Commit Protocols ....................................... 14
   2.2.2  Variants of Two Phase Commit Protocol ................. 15
   2.3  Message Ordering Properties .............................. 15
   2.3.1  Reliable Broadcast ..................................... 15
   2.3.2  FIFO Order ............................................. 16
   2.3.3  Local Order ............................................. 17
   2.3.4  Causal Order ............................................ 17
   2.3.5  Total Order ............................................. 19
   2.3.6  Total Causal Order ..................................... 20
   2.4  Logical Clocks ............................................. 21
   2.4.1  Lamport’s Clocks .................................... 21
   2.4.2  Vector Clocks .......................................... 24
   2.5  Event-B .................................................... 27
   2.5.1  Modelling Approach in Event-B ......................... 28
   2.5.1.1  An Event-B System ................................ 28
   2.5.1.2  Gluing Invariants ................................ 29
   2.5.2  Event-B Notation ....................................... 31
   2.6  Conclusions ............................................... 33
CONTENTS

5.4.2 Proving Transitivity Property ........................................ 90
5.5 Total Order Refinements .................................................. 92
  5.5.1 First Refinement: Introducing the Sequencer ....................... 93
  5.5.2 Second Refinement: Refinement of Order event ..................... 94
  5.5.3 Third Refinement: Introducing Sequence Numbers ................... 95
  5.5.4 Fourth Refinement: Introducing Control Messages .................. 97
  5.5.5 Fifth Refinement: Introducing Receive Control Event ............... 99
5.6 Conclusions ........................................................................... 100

6 Causally and Totally Ordered Broadcast ............................... 102
  6.1 Introduction ......................................................................... 102
  6.2 Mechanism for building a Total Causal Order ......................... 102
    6.2.1 Overview of the Refinement Chain ................................ 104
  6.3 Abstract Model of Total Causal Order Broadcast ...................... 105
    6.3.1 Abstract Variables .................................................... 105
    6.3.2 Events in the abstract model ....................................... 106
    6.3.3 Verification of Ordering Properties ............................... 109
  6.4 First Refinement of Total Causal Order ................................. 112
    6.4.1 Events in the First Refinement .................................... 112
    6.4.2 Constructing Gluing Invariants ................................... 114
      6.4.2.1 Relationship of abstract causal order and vector clock rules 115
      6.4.2.2 Relationship of abstract total order and sequence number 116
      6.4.2.3 Gluing Invariants ............................................... 116
  6.5 Second Refinement: Replacing Sequence Number by the Vector Clocks 117
  6.6 Further Refinements ....................................................... 119
  6.7 Conclusions ........................................................................... 120

7 Liveness Properties and Modelling Guidelines ......................... 122
  7.1 Introduction ......................................................................... 122
  7.2 Liveness in the Event-B Models ......................................... 122
    7.2.1 Feasibility .............................................................. 123
    7.2.2 Non-Divergence ....................................................... 124
    7.2.3 Enabledness Preservation ......................................... 127
  7.3 Guidelines for an Event-B Development ................................. 138
    7.3.1 General Methodological Guidelines for Modelling in Event-B 138
    7.3.2 Guidelines for Discharging Proof Obligations using B Tools .. 145
  7.4 Conclusions ........................................................................... 147

8 Conclusions ............................................................................. 149
  8.1 Summary ............................................................................. 149
  8.2 Comparison with other Related Work ................................... 153
  8.3 Future Work ........................................................................ 154

A Distributed Transactions ......................................................... 157

B TimeOut .................................................................................... 174

C Causal Order Broadcast .......................................................... 176
D  Total Order Broadcast  181
E  Total Causal Order Broadcast  188
Bibliography  197
# List of Figures

2.1 FIFO order ................................................................. 16  
2.2 Local order ............................................................... 17  
2.3 Broadcast not related by causal precedence ...................... 19  
2.4 Total Order and a Causal Order ................................. 20  
2.5 Total Order but not a Causal Order ............................. 20  
2.6 Lamport Clock ............................................................ 23  
2.7 Lamport Clock : Broadcast System ............................. 24  
2.8 Vector Clocks .............................................................. 26  
2.9 Event-B Machine .......................................................... 28  
3.1 Abstract Model of Transactions in Event-B ...................... 38  
3.2 Events of Abstract Transaction Model- I ....................... 40  
3.3 Events of Abstract Transaction Model- II ....................... 41  
3.4 Initial part of Refinement .............................................. 43  
3.5 Events of Update Transaction ......................................... 44  
3.6 Refinement : Coordinator Site Events-I ......................... 45  
3.7 Refinement : Coordinator Site Events - II ....................... 46  
3.8 Refinement : Coordinator Site Events - III ....................... 47  
3.9 Refinement : Participating Site Events -I ....................... 49  
3.10 Refinement : Participating Site Events -II ....................... 49  
3.11 Refinement : Participating Site Events -III ....................... 50  
3.12 Gluing Invariants-I .................................................... 51  
3.13 Gluing Invariants -II .................................................. 52  
3.14 Gluing Invariants -III ................................................ 53  
3.15 Gluing Invariants -IV ................................................ 54  
3.16 Gluing Invariants -V .................................................. 56  
3.17 The New events : A Reliable Broadcast ......................... 58  
3.18 Events IssueWriteTran and BeginSubTran : A Reliable Broadcast .... 59  
3.19 Gluing Invariants -VI .................................................. 61  
3.20 Refined Local Commit and Local Abort events : A Reliable Broadcast .... 62  
3.21 Event Site Failure ..................................................... 63  
3.22 Event TimeOut ............................................................ 64  
4.1 Abstract Model of Broadcast ........................................ 69  
4.2 Causal Order Broadcast : Initialization ......................... 70  
4.3 Causal Order Broadcast : Events .................................... 71  
4.4 Causal Order : CASE-I .................................................. 72  
4.5 Causal Order : CASE-II ................................................ 72
<table>
<thead>
<tr>
<th>Figure Number</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.6</td>
<td>Invariants-I</td>
<td>73</td>
</tr>
<tr>
<td>4.7</td>
<td>Invariants-II</td>
<td>75</td>
</tr>
<tr>
<td>4.8</td>
<td>Second Refinement : Refinement with Vector Clocks</td>
<td>77</td>
</tr>
<tr>
<td>4.9</td>
<td>Invariants-III</td>
<td>79</td>
</tr>
<tr>
<td>4.10</td>
<td>Fourth Refinement</td>
<td>80</td>
</tr>
<tr>
<td>5.1</td>
<td>Unicast Broadcast variant</td>
<td>84</td>
</tr>
<tr>
<td>5.2</td>
<td>Broadcast Broadcast variant</td>
<td>84</td>
</tr>
<tr>
<td>5.3</td>
<td>Unicast Unicast Broadcast</td>
<td>85</td>
</tr>
<tr>
<td>5.4</td>
<td>TotalOrder Abstract Model: Initial Part</td>
<td>85</td>
</tr>
<tr>
<td>5.5</td>
<td>TotalOrder Abstract Model : Events</td>
<td>86</td>
</tr>
<tr>
<td>5.6</td>
<td>Invariants-I</td>
<td>89</td>
</tr>
<tr>
<td>5.7</td>
<td>Invariants-II</td>
<td>92</td>
</tr>
<tr>
<td>5.8</td>
<td>TotalOrder Refinement-I</td>
<td>93</td>
</tr>
<tr>
<td>5.9</td>
<td>TotalOrder Refinement-I : Invariants</td>
<td>94</td>
</tr>
<tr>
<td>5.10</td>
<td>TotalOrder Refinement-II : Refined Order Event</td>
<td>95</td>
</tr>
<tr>
<td>5.11</td>
<td>TotalOrder Refinement-II : Invariants</td>
<td>95</td>
</tr>
<tr>
<td>5.12</td>
<td>TotalOrder Refinement-III</td>
<td>96</td>
</tr>
<tr>
<td>5.13</td>
<td>TotalOrder Refinement-III : Invariants</td>
<td>97</td>
</tr>
<tr>
<td>5.14</td>
<td>TotalOrder Refinement-IV</td>
<td>98</td>
</tr>
<tr>
<td>5.15</td>
<td>TotalOrder Refinement-IV : Invariants</td>
<td>99</td>
</tr>
<tr>
<td>5.16</td>
<td>Refinement-V : Receive Control</td>
<td>99</td>
</tr>
<tr>
<td>5.17</td>
<td>TotalOrder Refinement-V : Invariants</td>
<td>100</td>
</tr>
<tr>
<td>6.1</td>
<td>Execution Model of a Total Causal Order Broadcast</td>
<td>103</td>
</tr>
<tr>
<td>6.2</td>
<td>TotalCausalOrder: Initial Part</td>
<td>106</td>
</tr>
<tr>
<td>6.3</td>
<td>TotalCausalOrder: Events-I</td>
<td>107</td>
</tr>
<tr>
<td>6.4</td>
<td>TotalCausalOrder: Event-II</td>
<td>108</td>
</tr>
<tr>
<td>6.5</td>
<td>Invariants-I : Abstract Model</td>
<td>110</td>
</tr>
<tr>
<td>6.6</td>
<td>Invariants-II : Abstract Model</td>
<td>110</td>
</tr>
<tr>
<td>6.7</td>
<td>Invariants-III : Abstract Model</td>
<td>111</td>
</tr>
<tr>
<td>6.8</td>
<td>First Refinement- Part I</td>
<td>112</td>
</tr>
<tr>
<td>6.9</td>
<td>First Refinement - Part II</td>
<td>113</td>
</tr>
<tr>
<td>6.10</td>
<td>Gluing Invariants-IV : First Refinement</td>
<td>117</td>
</tr>
<tr>
<td>6.11</td>
<td>Second Refinement : SendControl</td>
<td>118</td>
</tr>
<tr>
<td>6.12</td>
<td>Second Refinement : TODeliver</td>
<td>119</td>
</tr>
<tr>
<td>6.13</td>
<td>Second Refinement : Gluing Invariant</td>
<td>119</td>
</tr>
<tr>
<td>6.14</td>
<td>Causal Deliver Event</td>
<td>120</td>
</tr>
<tr>
<td>7.1</td>
<td>Feasibility of Initialization and Event</td>
<td>123</td>
</tr>
<tr>
<td>7.2</td>
<td>Concrete Transaction States in the Refinement</td>
<td>125</td>
</tr>
<tr>
<td>7.3</td>
<td>Variant</td>
<td>126</td>
</tr>
<tr>
<td>7.4</td>
<td>Events Decreasing a Variant</td>
<td>126</td>
</tr>
<tr>
<td>7.5</td>
<td>Invariant used in variant Proofs</td>
<td>127</td>
</tr>
<tr>
<td>7.6</td>
<td>Transaction States in the Abstract Model</td>
<td>129</td>
</tr>
<tr>
<td>7.7</td>
<td>Order and BeginSubTran events</td>
<td>131</td>
</tr>
<tr>
<td>7.8</td>
<td>Transaction States in the Refinement-I</td>
<td>132</td>
</tr>
<tr>
<td>7.9</td>
<td>Transaction States in the Refinement-II</td>
<td>133</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Relational Notations ............................................. 32
2.2 Function Notations ............................................. 33

3.1 Events Code ..................................................... 51
3.2 Proof Statistics- Distributed Transactions ..................... 65

4.1 Proof Statistics- Causal Order Broadcast ......................... 81

5.1 Proof Statistics- Total Order Broadcast .......................... 101

6.1 Events Code ..................................................... 110
6.2 Proof Statistics- Total Causal Order Broadcast ................. 121

8.1 Proof Statistics- Overall ......................................... 154
Acknowledgements

I would like to acknowledge my deep sense of gratitude to my supervisor Professor Michael Butler for his valuable help, guidance and encouragement. He gladly accepted all the pains in going through my work again and again, and giving me opportunity to learn essential research skills. His ability to quickly understand the depth of the problem and suggesting a clear solution has always surprised me. This thesis would not have been possible without his insightful and critical suggestions, his active participation in constructing right models and a very supportive attitude.

I am also thankful to Commonwealth Scholarship Commission in the United Kingdom for funding my studies and British Council for looking after the welfare issues. Specially, thanks to Irene Costello and Rosalind Grimmit at Commonwealth Commission and Sue Davis at British Council for their quick attention whenever I was in need. I would also like to thank Professor Joao Marques-Silva for his constructive comments and useful advices. Comments of Professor Alexander Romanovsky and Dr. Robert J. Walters were very helpful in addressing some important issues in the thesis. Also, thanks to Elisabeth Ball, John Colley and Andy Edmunds for reading parts of the thesis and their valuable suggestions.

I gratefully acknowledge the support of every one at DSSE group for extending their cooperation, providing a very stimulating research environment and making my whole stay at Southampton a memorable experience. Informal discussions on B with Reza, Andy, Lis, Stephane, John, Colin, K.D., Mar Ya and Shamim were very enjoyable, be it a coffee room or B user group meetings or B Bay. Last but not the least, I thank my parents for their blessings, wife Gayatri for her support and daughters Surabhi and Ankita for bringing cheerful moments at home.

Divakar Singh Yadav
Dependable Systems and Software Engineering Group
School of Electronics and Computer Science
University of Southampton
Southampton, UK
Chapter 1

Introduction

1.1 Motivation

Various modern day distributed transactional information systems based on distributed databases are large and fairly complex due to their underlying mechanisms for transaction support. These systems, classified as business critical systems, take advantage of data distribution and are expected to exhibit high degrees of dependability. Any failure in these systems may lead to financial losses in addition to the potential loss of the trust of customers. Formal rigorous reasoning about the algorithms and mechanisms beneath such systems is required to precisely understand the behavior of such systems at the design level.

1.1.1 Data Replication

Due to the rapid advances in communication technology, the last decade has witnessed the development of several complex distributed information systems for banks, stock exchanges, electronic commerce, and airline/rail reservation, to name a few. The emergence of such applications has opened up new opportunities for integrating advances in database systems with the advances in the communication technology. In such systems, it is not uncommon to store a copy of a database (replication) or to store part of the database (fragmentation) at several sites for fault-tolerance and efficiency. A distributed database system can be thought of as a collection of several sites where data is distributed across these sites. These sites communicate by exchange of messages and cooperate with each other for the successful completion of global computation which may read or write to the data at several sites. With respect to the data distribution, from a user perspective, a distributed database should behave like a centralized database. This view of distributed databases implies that the user should be able to query the database without worrying about the distribution of the data. With respect to the updates, this
view of a distributed database requires that the transactions must be executed as an atomic action regardless of fragmentation and replication [105].

Replication improves availability in a distributed database system [53]. A replicated database system can be defined as a distributed system where copies of the database are kept across several sites. Data access in a replicated database can be done within a transactional framework. It is advantageous to replicate the data if the transaction workload is predominantly read only. However, during updates, the issue of keeping the replicas in a consistent state arises due to race conditions among conflicting update transactions. The strong consistency criterion in the replicated database requires that the database remains in a consistent state despite transaction failures. The possible causes of transaction failures include bad data input, time outs, temporary unavailability of data at a site and detected deadlocks.

In addition to providing fault-tolerance due to failures, one of the important issues to be addressed in the design of replica control protocols is consistency. The One-Copy Equivalence [19, 97] criteria states that a replicated database is in a mutually consistent state only if all copies of data objects logically have the same value. The One-Copy Serializability [19] is the highest correctness criterion for replica control protocols. It is achieved by coupling the consistency criteria of one-copy equivalence and providing serializable execution of transactions. In order to achieve this correctness criterion, it is required that interleaved execution of transactions on replicas be equivalent to serial execution of those transactions on one-copy of a database. One copy equivalence and serial execution together provide one-copy serializability which is supported in a read anywhere write everywhere approach [118]. Though serializability is the highest correctness criteria, it is too restrictive in practice. Various degrees of isolation to address this problem have been studied in [63].

### 1.1.2 Broadcast Primitives

A distributed system is a collection of distinct sites that are spatially separated and cooperate with each other towards the completion of a distributed computation. The design and verification of distributed applications is a complex issue due to the fact that the communication primitives available in these system are too weak. The inherent limitation of these systems is that there does not exist a system wide common global clock and they do not share common memory. Due to these limitations the up-to-date state of the entire system is not available to any process or site. These systems communicate with each other by exchange of messages which are delivered after arbitrary time delays [121]. This problem can be dealt with by relying on group communication or broadcast primitives that provide higher ordering guarantees on the delivery of messages. The implementations of these group communication primitives has also been investigated for different distributed systems such as Isis [21], Totem [94], Trans [91], Amoeba [128]
and Transis [10]. The protocols in these systems use varying broadcast primitives and address group maintenance, fault-tolerance and consistency services. Several approaches have been proposed for the management of replicated data using group communication primitives [9, 55, 63, 98, 100, 115, 125]. The transaction mechanism in the management of replicated data is also considered in [9, 14, 98, 114].

There exist several broadcast protocols based on varying group communication primitives that satisfy different higher ordering guarantees for the messages [38, 52, 125]. The weakest among them is reliable broadcast. A reliable broadcast eventually delivers the messages to all participating sites and imposes no restriction on the order in which the messages are delivered to those sites. Stronger variants of a reliable broadcast impose additional requirements on the order in which messages are delivered such as FIFO order, local order, causal order, total order and total causal order\(^1\). A causal order broadcast is a reliable broadcast that preserves the causality among the messages and the messages are delivered to the processes respecting the causality among the messages. The notion of causality is based on the causal precedence relation (\(\rightarrow\)) defined by Lamport [75]. A causal order broadcast combines the properties of both FIFO and local order. A total order broadcast is a reliable broadcast that satisfies the total order requirement and requires that all processes eventually deliver the same sequence of messages [52] irrespective of their sender(s). Similarly, a total causal order broadcast combines the properties of both total and causal order and requires that the messages are delivered to the processes respecting both total and causal order.

The introduction of transactions based on group communication primitives represents an important step towards extending the power and generality of group communication for design and implementation of reliable fault-tolerant distributed computing applications [114]. In a replicated database, users interact with the database using transactions. A read-only transaction may read the data locally at the site of submission, while an update transaction needs to access data at several sites. If a replicated database uses a reliable broadcast without ordering guarantees, the operations of conflicting update transactions may arrive at different sites in different orders due to race conditions\(^2\). This may lead to the formation of deadlocks among conflicting transactions involving several sites. The blocking of the update transactions at a site is usually resolved by aborting the transactions by timeouts. This problem can be addressed effectively by processing transactions over a stronger notion of reliable broadcast protocol that provides higher order guarantees on message delivery [9, 125]. The abortion of conflicting transactions can be avoided by using a total order broadcast which delivers and executes the conflicting operations at all sites in the same order, thus ensuring a serial execution of conflicting update transactions at replicas. Similarly, a causal order broadcast captures conflict as causality and the transactions executing conflicting operations are executed

\(^1\)The informal specifications of various ordering properties are given in Chapter 2.
\(^2\)The race conditions on the conflicting update transactions are explained in Section 3.2 in Chapter 3.
at all sites in the same order. Processing update transactions over a total causal order broadcast not only delivers the operations in a total order at the participating sites, but also preserves the causal precedence relationship among the update transactions.

1.2 Why use Formal Methods for Data Replication?

Database replication is traditionally envisaged as a way of increasing fault-tolerance and availability. There exists a vast literature on the management of replicated data [53] dealing with various aspects such as fault-tolerance, consistency, performance and scalability. Despite the abundance of work in this area, little work has been implemented in commercial products. One of the important reasons is that most replica control mechanisms are complex, under-specified and difficult to reason about. As a result many commercial products take a pragmatic approach for data replication which allows the replicas to be in inconsistent states [61, 63], tolerates inconsistency among the replicas due to lazy replication [101] and leaves solving inconsistencies to the user [124]. Group communication has been proposed as a powerful mechanism for the management of replicated databases. The existing work on the development of formal specifications of group communications services, ordering and reliability properties is often complicated, difficult to understand and sometime ambiguous [34, 42, 96]. Application of formal methods to this problem to provide clear specifications and formal verification of the critical properties is rare. It is desirable that the models of distributed systems be precise and reasonably compact, and one expects that all the aspects of the system must be considered in the proofs.

The dependability of distributed systems is an important design criterion for developing new distributed services or updating existing ones. In principle, the dependability of a system is the ability to avoid service failures that are more frequent and more severe than is acceptable. The dependability of the system encompasses the following attributes [13]: the readiness for service (availability), the continuity of service (reliability), absence of catastrophic consequences on the users and environment (safety), absence of improper system alterations (integrity), and ability to undergo modifications and repairs (maintainability). Reliability refers to both resilience of a system to various type of failures and its capability to recover from them [97]. A resilient system is tolerant of failures and can continue to provide the service even when failure occurs. A recoverable database system is one that can get to a consistent state by moving back to a previous consistent state (backward recovery) or moving forward to a new consistent state (forward recovery). One of the approaches for dealing with the failures in a distributed system is exception handling. The coordinated atomic action (CA action) [137] concept is a unified scheme for coordinating complex concurrent activities and supporting error recovery between multiple interacting components in a distributed object system. The
problem of exception handling in distributed systems where exceptions may be raised simultaneously in different processing nodes is addressed in [138].

These issues related to dependability must be addressed in the design, architecture and component infrastructure of a system. It is not possible to simply add a fault-tolerance module later on to make the system fault-tolerant [57]. A system can be designed to be fault-tolerant by ensuring that exhibits well defined behavior which facilitates the actions suitable for recovery. For example, in replicated data updates, the effect of an update transaction must not be visible until it commits at all sites and a replica should receive the updates in the same order they were sent. Formal Methods provides a systematic approach to the development of complex systems. They provide a framework for specification of the system under development and verification of desirable properties. Advantages and disadvantages of formal methods in industrial practice and the degree of formalism to use is considered in [54, 58, 112]. Until now, formal methods were considered suitable for design and development of safety critical and mission critical system such as train systems [3, 4], embedded controllers for railways [26], and a steam boiler [30]. Currently, computer science researchers are collaborating to enhance and develop the verification technologies that demonstrate high reliability and productivity in software development. One such long term research project, called the verified software grand challenge [135], is targeted towards developing a roadmap for integrating tools and techniques for verification and demonstrating the feasibility of applying formal methods to large scale industrial software development.

This thesis investigates various mechanisms for the management of replicas in a large distributed system, formalizing and reasoning about the behavior of such systems. Our approach to modelling and formal reasoning about fault-tolerant distributed transactions for replicated databases is based on Event-B [92].

1.3 Related Work

There exists a vast literature in the area of transactional information systems [134], distributed algorithms [85, 87], concurrency control [19], distributed databases [32] and group communication [38]. There also exists a plethora of algorithms and protocols covering several aspects of database transactions, replication and distributed databases showing the complexity of the problem. However, the application of formal methods for providing precise specifications of the problem, their solutions and proof of correctness is still an important issue. Some formal methods have been applied to the problems in this area and we outline some of that work.

I/O Automata, a formal method, was originally developed to describe and reason about distributed systems [47, 86]. The I/O automation model is a labelled transition system consisting of sets of states which also include the set of initial states, a set of actions and
a set of transitions. The operations of I/O automation are described by its executions and traces. Executions in the I/O automation are alternating sequences of states and actions, while the traces are sequences of input and output actions occurring in the actions. One automation implements another if its traces are also traces of the other. The proof method supported in this method for reasoning about the system involves invariant assertions. An invariant assertion is defined as a property of the state of a system that is true in all executions. Most notably, the work done so far using I/O Automata has been carried out by hand [42, 47]. Some of the significant work done using I/O Automata includes modelling and verification of sequentially consistent shared object systems in a distributed network [41]. In order to keep the replicated data in a consistent state, a combination of total order multicast and point to point communication is used. In [40], I/O automata are used to express lazy database replication. The authors present an algorithm for lazy database replication and prove the correctness properties relating to performance and fault-tolerance. In [42, 104] the specification for group communication primitives is presented using I/O automata under different conditions such as partitioning among the group and dynamic view oriented group communication. A series of invariants relating state variables and reachable states are proved using the method of induction.

Temporal Logic of Actions (TLA) [72, 78] is a method for specifying and reasoning about concurrent algorithms. In TLA, a system is specified by a formula [77]. Temporal logic formula contain variables to represent quantities that change with time and constants to represent the quantities that do not change with time. A TLA formula is defined on system behavior. A system satisfies a formula if the formula is true for every behavior corresponding to a possible execution of the system. $\text{TLA}^+$ is a language for writing a TLA specification which includes the operators for defining data structures for large specifications. $\text{TLA}^+$ specifications are supported by tools such as TLC, a model checker and simulator and SANY, a parser and semantic analyzer for specifications. The major work carried out using TLA includes formalizing the Byzantine Generals problem and providing a proof of correctness of the solution [79], the remote procedure call and memory specification problem [88] and distributed algorithms like lazy caching [70].

The Z notation [123, 136] has also been applied to develop formal specification of a database system. Z is a formal specification notation based on set theory and first order predicate logic to express model-based specifications. A notion of schema is central to Z specifications. A system specification in Z consists of state variables, initialization, and a set of operations on state variables. The invariants are expressed on state variables to represent the conditions which must always be satisfied. There exist a number of industrial-level tools for formatting, type-checking and aiding proofs in Z. In [11, 12], Z is used to formally specify a database system to illustrate transaction decomposition. In the Z specifications, they outline the necessary steps to obtain transaction decomposition to increase concurrency and reason about interleaving with other transactions.
The necessary properties are added in the form of invariants and they provide proof of correctness by hand to show that invariants are preserved by the specifications.

In [67, 68], an approach for modelling long running transactions using the NT/PV model is presented. In NT/PV, a long running transaction is modelled by a set of sub-transactions, a partial order among sub-transactions, inputs and outputs. A transaction is said to execute correctly if it begins execution in a state which satisfies its input conditions, executes its sub-transactions consistently in a partial order and terminates by leaving the database in a state which satisfies its output conditions.

In [130], a set theoretic model is proposed to verify ordering properties of a multicast protocol. Three types of ordering properties, local order, causal order and total order are considered. Formal results are presented that define a set of circumstances under which a total order satisfies the causal relationship among the messages. Formal results in the form of theorems are provided and they can be applied to a system to prove the ordering properties on messages in that system.

A refinement based approach to developing distributed systems in Event-B is outlined in [24]. The correspondence between the action-based formalism and the abstract B machines is outlined in this work. The action system formalism [15] is a state-based approach to distributed computing. An action system models a reactive system with guarded actions on state variables. In [24], the author outlines how the reactive refinement and decomposition of action systems can be applied to abstract machines and how this approach is related to step-wise refinement in Event-B. The refinement approach has been applied to the development of a secure communication system [25]. The aim was to carry out a development from initial abstract specifications of security services to a detailed design in the refinement steps. The authors have also demonstrated an effective way to combine B and CSP specifications.

In [22] important contributions are made towards development of a refinement rule which allows actions to be introduced in a refinement step and a decomposition rule which allows a system model to be decomposed into parallel subsystems. Use of refinement and decomposition rules in the development of telecommunications systems is outlined in [23]. Other important work carried out using the refinement approach includes the Mondex purse system in Event-B [31], verification of the IEEE 1394 tree protocol distributed algorithm [7], development of a train system [4], rigorous development of fault-tolerant agent systems [71] and modelling web based systems in B [110]. The case study on Mondex illustrates modelling strategies and the guidelines to achieve a high degree of automatic proofs.
1.4 Our Contributions

In this thesis, we present a model driven approach using Event-B for the construction of formal models of distributed transactions and broadcast protocols for a replicated database system. We outline how a refinement based approach can be used for the development of a reliable replicated database system that ensures atomic commitment of update transactions using broadcast primitives. Our approach of specification and verification is based on the technique of abstraction and refinement. This formal technique, supported in Event-B, consists of describing rigorously the problem in the abstract model and introducing the solution or design details in refinement steps. Through the refinement we verify that the detailed design of a system in the refinement conforms to the initial abstract specifications. We have used the industrial level B tool Click’n’Prove [6] for the generation of proof obligations and discharge them using the automatic and interactive prover.

In our approach, we model abstract behavior of a distributed algorithm in the abstract model and propose the solutions in the refinement step using concrete variables. The B tool generates proof obligations relating abstract and concrete variables for refinement checking. In order to discharge these proof obligations we need to add a series of new gluing invariants to the model. These gluing invariants demonstrate the relationship of abstract and concrete variables. The discovery of these new gluing invariant provides a clear insight to the system and support precise reasoning about why a specific solution proposed in the refinement is a correct solution of abstract problem. The aim of the work presented in the thesis is outlined below.

- To demonstrate the application of a technique for incremental construction of formal models of distributed systems and to reason about them.
- To develop the technique for the discovery of gluing invariants due to prover failures to automatically discharge a proof obligation.
- To investigate the applicability of ordered broadcasts for processing transactions in a replicated database.
- To develop guidelines for formal design of distributed transactional systems by means of abstraction and refinement.

1.5 Chapter Outline

The thesis is organized into eight chapters. The summary of each chapter is outlined below.
In Chapter 2, an overview of replicated data updates and the related problems is presented. Subsequently, informal specifications of various ordered broadcast are discussed. Later in the chapter, we address the notion of logical time. A background of logical clocks, such as Lamport’s clock and the vector clock is presented. The subtle issues related to the consistency of logical clocks are also addressed. At the end of the chapter, an overview of an approach to formal development of distributed systems using Event-B is outlined.

In Chapter 3, we present a formal approach to modelling and analyzing a distributed transaction mechanism for replicated databases using Event-B. In our abstract model, an update transaction modifies the abstract one-copy database through a single atomic event. In the refinement, an update transaction consists of a collection of interleaved events updating each replica separately. The transaction mechanism on the replicated database is designed to provide the illusion of an atomic update of a one-copy database. Through the refinement proofs, we verify that the design of the replicated database preserves the one-copy equivalence consistency criterion despite transaction failures at a site. The various events in the refinement are triggered within the framework of the two phase commit protocol. The system allows the sites to abort a transaction independently and keeps the replicated databases in a consistent state. A series of invariants discovered while discharging the proofs is also presented which provides a clear insight into why our model of the replicated database preserves consistency despite transactions aborting at a site.

In the subsequent refinement steps, we introduce explicit messaging among the sites and demonstrate how various messages are exchanged among the sites within the framework of two phase commit protocol. A notion of a reliable broadcast is adopted in our model to represent communication among the sites. We also present the specification of \textit{TimeOut} operation that aborts a transaction by timeouts. Chapters 4, 5 and 6 present incremental development of stronger variants of reliable broadcast protocol and in Chapter 7 (Section 7.2.3) we outline how the stronger notion of broadcast can be used to define an abstract ordering on the transactions.

In Chapter 4, abstract specifications of \textit{causal order} broadcast are presented. The causal order on the messages is defined by combining the properties of both \textit{FIFO} and \textit{local} order. We also outline how an abstract causal order is constructed by the sender. In the refinement we introduce the notion of \textit{vector clocks}. The abstract causal order in the abstract model is replaced by the vector clock rules. In this process we also discover some interesting invariants which define the relationship between abstract causal order and the vector clock rules. This formal study precisely reasons about how an abstract causal order on the messages can correctly be implemented by a system of vector clocks.
– In Chapter 5, we present an incremental development of a system of total order broadcast. The key issues with respect to the total order, such as how to build a total order on the messages and what information is necessary for defining a total order, are also addressed. In this development we first present the abstract specifications of the total order broadcast. Subsequently, in the refinement steps, we introduce a sequencer based approach to implement the total order.

– In Chapter 6, after establishing the invariants for a system of causal order broadcast and total order broadcast, we present a formal development of a system of total causal order broadcast which satisfies both a total and a causal order on the message delivery. In the refinements we outline how the abstract total order and causal order can correctly be implemented by a vector clock system. In the further refinements we also outline how the requirement of the generation of a sequence number can be eliminated by employing the vector clocks. The various invariants relating abstract total order, causal order, sequence numbers and vector clocks are also given.

– In Chapter 7, the liveness issues related to the model of distributed transactions are addressed. We briefly outline the construction of the proof obligations to ensure enabledness preservation and non-divergence. Lastly, we present the general guidelines for formal development of a distributed system using Event-B.

– In Chapter 8, we present our conclusions, compare our approach with other related work and outline future work.
Chapter 2

Background

The term *distributed system* has been defined and characterized in number of ways in various contexts in the past couple of years.

- Ozsu and Valduriez [97] define a distributed system as *a collection of autonomous processing elements (not necessarily homogenous) that are connected by a computer network and that cooperate in performing their assigned tasks.*

- Tanenbaum and Van Steen [129] give a loose characterization of a distributed system as *a collection of independent computers that appears to its user as a single coherent system.*

- Singhal and Shivratri [121] describe a distributed system as *a system consisting of several computers that do not share memory or a clock; communicate by exchange of messages; and each computer has its own memory and runs its own operating system.*

- Korth, Silberschatz and Sudershan [119] define a distributed database system as *a system consisting of loosely coupled sites that share no physical component; database systems that run on each site are independent of each other; and a transaction may access the data at one or more sites.*

- Gray and Reuter [50] define a distributed database system as *a database system that provides transparent access to replicated and partitioned data.*

We take a collective view of these definitions.
Chapter 2 Background

2.1 Preliminaries

2.1.1 A Database Transaction

A database transaction can be defined as a collection of actions that make consistent transformations of database state while preserving system consistency [97]. A database transaction is a unit of work which contains operations performing reads, writes or updates to a data object. A typical database transaction is said to have \textit{ACID} (atomicity, consistency, isolation, durability) properties. The \textit{atomicity} property requires either all or none of the operations of the transaction are executed. The transaction commits only when all of the operations are done, otherwise it aborts. The \textit{consistency} property requires that the execution of a transaction must leave the database in a consistent state. The \textit{isolation} property requires that following a schedule in which the execution of multiple transactions is interleaved has same effect as if they were executed in some serial order. This also implies that incomplete transaction updates are not visible to concurrent transactions. The \textit{durability} condition requires that once the transaction commits, all of its effects survive system failures and results are permanent.

2.1.2 Long Running Transactions and Compensations

A \textit{long running transaction} may be defined as a transaction which takes longer time to complete execution [45] than traditional ACID transactions. In traditional database systems, an execution is serializable if it is equivalent to a serial execution. The traditional notion of serializability as a correctness criterion is too restrictive and a bottleneck for long running transactions [60, 68]. In order to avoid this bottleneck different kinds of extended transaction models such as \textit{nested transactions} [95], \textit{SAGA} [46], \textit{cooperative transactions} [68] are suggested which use a relaxed notion of serializability. Compensation has been proposed as a mechanism for handling failures in long running transactions. If any activity needs to be rolled back, a compensatory action is taken to semantically undo the effect of the committed transactions. The transactions that are executed to semantically undo the effects of a committed transaction are called compensating transactions.

A formal approach for modelling compensation in business processes can be found in [28]. StAC [27] is a formal language developed for the design of component based enterprise systems which exclusively deals with compensation. Though compensation is an important concept used for handling failures during long business activities, compensating transactions are not enough to meet all the requirements of modern business-to-business interaction. Some of these requirements may be found in [51].
2.1.3 Distributed Transactions

In a distributed database system, a given transaction is submitted at one site, but it can access data at other sites as well [97, 105]. A distributed transaction (global transaction) can be defined as a transaction accessing data located at other sites as well. Each site maintains transaction coordinator, transaction manager and lock manager processes. The coordinated actions of all of these processes ensures execution of a distributed transaction. A transaction coordinator is responsible for starting the execution of transactions that originate at the site. The coordinator is also responsible for distributing sub-transactions at appropriate sites for execution. The coordinator monitors and coordinates the termination of each global transaction that originates at that site, which may result in the transaction being committed at all sites or aborted at all sites. The role of transaction manager is to maintain a log for recovery purposes and participate in coordinating the concurrent execution of the transactions executing at that site. The role of lock manager is to receive lock requests from the transaction manager and lock/unlock the data items at that site.

The transactions in a distributed system may be processed over broadcast protocols [8, 56, 64, 65, 99, 102]. Broadcast protocols that provide ordering guarantees have been proposed as a mechanism to propagate updates to the replicas in a distributed database. The broadcast protocols also provide serialization to updates at all sites [9, 62, 106].

2.1.4 Updating Distributed Data

The transparency requirement of distributed data requires that the user must view the distributed data as a centralized database. The issue of fragmentation and replication should be addressed at the system level. In the case of the update of replicas, it is necessary to keep the replicas logically in a identical state. Failing to do that may lead the database into an inconsistent state. There exist two approaches for updating replicas. In synchronous replication, all the copies of replicas must be updated before an update transaction commits, while in asynchronous replication, the replicas are updated in a progressive manner and a transaction may view different values of replicas. Voting and read one write all (ROWA) [53, 97, 105] are two important techniques for replica management that ensure all replicas are in identical state.

In the voting technique, a transaction writes to a majority of replicas before it commits. This ensures that a read-only transaction reads the correct value even though it may observe the different values for the same data. In ROWA, a read-only transaction reads one copy, but a write is performed to all copies before a transaction commits. This technique is suitable when the transaction workload is predominantly read-only. However, ROWA suffers from an important drawback. If a single copy of the replica is unavailable then update transactions cannot commit. An alternative to ROWA, which addresses
this problem, is called Read One Write All Available (ROWA-A). In this protocol, all available copies of the replica are updated when the update transaction commits. The copies which were unavailable need to enforce the write when they are available. A review of different variants of ROWA may be found in [53].

2.2 Failures in Distributed Databases

A robust design of a reliable replicated database system needs to identify the type of failures a system may suffer. There are four types of failure called transaction failures, site failures, media failures and communication link failures [97, 121].

Transaction Failures: These failures may occur for several reasons. The possible causes of transaction failures are bad data input, timeouts, race conditions or a formation of a deadlock. Most deadlock detection protocols require one of the transactions to abort if a deadlock occurs. The usual approach used to deal with transaction failures is to abort the transaction. Log based recovery techniques and shadow paging are two important techniques to facilitate database recovery due to failures.

Site Failures: The main reasons for site failures are hardware failures, processor or memory failures or failures of system software. Site failures in distributed systems result in the inaccessibility of resources located on that site. This failure may interrupt any distributed transaction executions that are accessing the resources located at this site.

Media Failures: These failures occur due to the failure of secondary storage devices (e.g., disk failure) containing the whole or part of the database. The reason for these failures varies from errors in the operating system and hardware faults, to faults in a disk controller. In the event of media failures, the data at that site becomes inaccessible and this may cause rollback of the transactions attempting to read or write to data objects.

Communication Link Failures: Communication link failures include errors caused in messaging, improperly ordered messages, loss or duplication of messages or a total failure of communication links. Failure of communication links may also divide the distributed system into several disjoint partitions, called network partitioning.

2.2.1 Commit Protocols

In distributed databases a transaction may be processed at various sites. A higher number of components in a distributed system implies a higher probability of component failure during execution of a distributed transaction. In order to maintain the global atomicity of a transaction, it is required that a distributed transaction commit at all sites or at none of the sites. Gray addressed the issue of ensuring global atomicity despite failures in [49]. Commit protocols provide a framework to ensure global atomicity in the
presence of failures. The application of commit protocols for distributed transaction management in Oracle, a commercial database management system, is discussed in [48].

The two phase commit protocol [49] is a basic protocol which provides fault-tolerance for distributed transactions. This ensures global atomicity through the exchange of messages among the participating and coordinating sites. This protocol ensures global atomicity in the presence of transaction failures as every site writes an appropriate record to its log and can take suitable action in case of recovery. A major limitation of this protocol is that it is blocking because, in the case of coordinator site failures, participants wait for its recovery.

2.2.2 Variants of Two Phase Commit Protocol

Many variants of the two phase commit protocol have been proposed to improve its performance [81, 93, 122]. The presumed commit protocol is optimized to handle general update transactions while the presumed abort optimizes read-only transactions. Levi and others presented an optimistic two phase nonblocking commit protocol [81] in which locks acquired on data objects at a site are released when the site is ready to commit. In the case of an abort of a distributed transaction, a compensating transaction is executed at that site to undo the updates. A three phase commit protocol [122] is a nonblocking commit protocol where failures are restricted to site failures only. All variants of the two phase commit protocol assume that mechanisms, such as, maintaining the database log and local recovery, are present locally at each site. There also exist a number of communication paradigms in which commit protocols may be implemented. In a centralized two phase commit protocol no messages are exchanged among participating sites. The exchange of messages takes place only between the coordinator site and the cohorts. In a nested two phase commit protocol cohorts may exchange messages among themselves. A distributed two phase commit eliminates the second phase as the coordinator and cohorts exchange messages through broadcasting.

2.3 Message Ordering Properties

In this section we outline the informal specifications of the message ordering properties for a broadcast system.

2.3.1 Reliable Broadcast

The concept of a reliable broadcast is central to ordered broadcasts. Various definitions of the ordering properties have been discussed in [17, 21, 38, 126]. In [52], Hadzilacos and Toueg say that a reliable broadcast satisfies the following properties:
– **Validity:** If a correct process broadcasts a message \( m \) then the sender eventually delivers \( m \).

– **Agreement:** All correct processes deliver the same set of messages, i.e., if a process delivers a message \( m \) then all correct processes eventually deliver \( m \).

– **Integrity:** For any message \( m \), every correct process delivers \( m \) at most once and only if \( m \) was previously broadcast by sender(\( m \)).

A reliable broadcast is defined in terms of two primitives called *broadcast* and *deliver*. A reliable broadcast imposes no restriction on the order in which messages are delivered to the processes. However, many applications require a stronger notion of reliable broadcast that provides ordering guarantees on message delivery. A reliable broadcast can be used to deliver messages to processes following a *fifo order*, *local order*, *causal order*, *total order* or a *total causal order*, providing ordering guarantees on the message delivery. An informal specification of these ordering properties is given below.

### 2.3.2 FIFO Order

*If a particular process broadcasts a message \( M_1 \) before it broadcasts a message \( M_2 \), then each recipient process delivers \( M_1 \) before \( M_2 \).*

A *fifo broadcast* is defined as a reliable broadcast that delivers the messages in *fifo order*. As shown in Fig. 2.1, process \( P_1 \) first broadcasts \( M_1 \) followed by \( M_2 \). The fifo order is said to be preserved by the system if all processes deliver \( M_1 \) before delivering \( M_2 \). The delayed message \( M_1 \), shown as a dotted line, violates the fifo order.

\[\text{Figure 2.1: FIFO order}\]

---

\(^1\) A correct process is defined as a non failed process [52, 107]. A process may fail due to crash failure, omission failures or Byzantine failure. It also assumes a reliable communication network i.e., there is no loss, generation or garbling of messages in the communication network [107].
2.3.3 Local Order

*If a process delivers M1 before broadcasting the message M2, then each recipient process delivers M1 before M2.*

As shown in Fig. 2.2, process P2 delivers M1 before it broadcasts M2. The local order is said to be preserved by the system if all processes deliver M1 before delivering M2. The delayed message M1, shown as a dotted line, violates the local order.

2.3.4 Causal Order

*If the broadcast of a message M1 causally precedes the broadcast of a message M2, then no correct process delivers M2 unless it has previously delivered M1.*

The notion of capturing causality in a distributed system was first formalized by Lamport in [75] and further extended in [76]. It is based on the notion of the happened before relationship that captures the causal relationships among the events. The execution of a process in a distributed system can be characterized by the sequences of the events and these events can be either internal events or message events. An internal event represents a computation milestone achieved in a process, whereas message events signify exchange of messages among the processes. The message send and message receive are message events respectively occurring at a process sending a message and receiving a message.

The happened before relation (→) [75] between any two events of a distributed computation is defined as \( a \rightarrow b \), where event \( a \) happened before \( b \). Events \( a \) and \( b \) are either of following,

- \( a, b \) are internal events of the same process such that \( a, b \in P_i \) and \( a \) happened before \( b \).
\(a, b\) are message events of different processes such that \(a \in P_i, b \in P_j\), where \(a\) is a \textit{message send} event occurring at process \(P_i\) and \(b\) is a \textit{message receive} event occurring at \(P_j\) while sending a message \(m\) from process \(P_i\) to \(P_j\).

The \textit{happened before} relation (\(\rightarrow\)) can be extended to the \textit{causal precedence} (or \textit{precedes}) relationship to define a global causal ordering on the messages. A message \(m_i\) \textit{precedes} \(m_j\) if the message send event \(\text{send}(m_i)\) at process \(P_i\) \textit{happened before} the message send event \(\text{send}(m_j)\) at a process \(P_j\). A message \(m_i\) \textit{causally precedes} \(m_j\) if either of following holds,

\[- \text{the broadcast event of } m_i \text{ \textit{causally precedes} the broadcast of } m_j.\]
\[- \text{the receive event of } m_i \text{ \textit{causally precedes} the broadcast of } m_j.\]

The \textit{happened before} relation is transitive i.e. if event \(a\) happened before \(b\) and \(b\) happened before \(c\) then \(a\) is said to have happened before \(c\).

\[a \rightarrow b \land b \rightarrow c \Rightarrow a \rightarrow c\]

Not all events in a system are causally related. Event \(a\) \textit{causally affects} event \(b\) only if \(a \rightarrow b\). The events which do not causally affect other events are characterized as concurrent events. Both \textit{causally related events} and \textit{concurrent events} can be defined using this relation. The two events \(a\) and \(b\) are causally related if either \(a \rightarrow b\) or \(b \rightarrow a\). Two events \(a\) and \(b\) are concurrent \((a \parallel b)\) if \(a \nrightarrow b\) and \(b \nrightarrow a\). The \textit{causally related events} and \textit{concurrent events} may be defined as follows.

\begin{align*}
\text{Causally Related Events :} & \quad a \rightarrow b \lor b \rightarrow a \\
\text{Concurrent Events} (a \parallel b) : & \quad \neg (a \rightarrow b) \land \neg (b \rightarrow a)
\end{align*}

Therefore for any two events \(a\) and \(b\) there exist three possibilities i.e., either \(a \rightarrow b\) or \(b \rightarrow a\) or \(a \parallel b\).

\textbf{Parallel Messages}

The two messages \(M1\) and \(M2\) are defined as parallel messages \((M1 \parallel M2)\) when no partial ordering exist among them i.e., \(\neg (M1 \rightarrow M2) \land \neg (M2 \rightarrow M1)\) holds.

The causal order is defined by combining the properties of both fifo and local order \([52]\). A \textit{causal order broadcast} is a reliable broadcast that satisfies the \textit{causal order} requirement. A causal order broadcast delivers messages respecting their causal precedence.
However, if the broadcast of any two messages is not related by causal precedence (parallel messages), then it does not impose any requirement on the order in which they can be delivered. As shown in the Fig. 2.3, the broadcast of messages $M_1$ and $M_2$ are not related by a causal precedence relationship and the causal order broadcast delivers them to the processes in arbitrary order.

![Figure 2.3: Broadcast not related by causal precedence](image)

### 2.3.5 Total Order

*If two processes $P_1$ and $P_2$ both deliver the messages $M_1$ and $M_2$ then $P_1$ delivers $M_1$ before $M_2$ if and only if $P_2$ delivers $M_1$ before $M_2$.*

A total order broadcast\(^2\) is a reliable broadcast that satisfies the total order requirement. The agreement and total order requirements of a total order broadcast imply that all correct processes eventually deliver the same sequence of messages [52]. Since a total order defines an arbitrary ordering on the delivery of messages, it does not satisfy causal relations. The two cases given below illustrate the relationship between causal order and total order.

In the first case, as shown in Fig. 2.4, all messages are delivered conforming to both the causal and the total order. The broadcast of a message $M_1$ *causally precedes* the broadcast of $M_2$ and each recipient process delivers $M_1$ before delivering $M_2$. Similarly, the broadcast of message $M_2$ *causally precedes* broadcast of $M_3$ and each recipient process delivers $M_2$ before delivering $M_3$. Therefore, the system delivers the messages respecting the causal order. It can also be noticed that since all processes deliver the messages in the same sequence, i.e., $M_1$, $M_2$ followed by $M_3$, the delivery order also conforms to the total order.

\(^2\)The total order broadcast is also known as atomic broadcast. Both of the terms are used interchangeably. The former is preferred over the later as the term atomic suggests the agreement property rather than total order.
In the second case, as shown in Fig. 2.5, a broadcast satisfies a total order on the messages, but does not preserve the causal relationships among them. All processes deliver the same sequence of messages, i.e., each process delivers $M1$ followed by $M3$, and lastly $M2$. Thus the delivery order conforms to the total order property. However, the delivery order does not respect the causal order for the following reason. Since the broadcast of $M2$ causally precedes the broadcast of $M3$, each recipient should deliver $M2$ before delivering $M3$. It can be noticed that each process delivers $M3$ before delivering $M2$, violating the causal order.

### 2.3.6 Total Causal Order

A total causal order broadcast\(^3\) is a reliable broadcast that satisfies both causal and total order. A total causal order broadcast is the strongest variant of a reliable broadcast which has been used as an important mechanism to address fault-tolerance in distributed systems [74, 116]. An example of a total causal order broadcast is illustrated in the Fig. 2.4.

\(^3\)A reliable broadcast that satisfies both causal and total order is also known as a causal atomic broadcast.
Chapter 2 Background

2.4 Logical Clocks

A distributed system is a collection of computers that are spatially separated. A distributed computation is composed of a finite set of processes where the actions of a process can be modelled as a collection of events produced by a process during its life cycle. The concept of temporal ordering of events is integral to the design and development of these systems. The causal precedence relation is an important concept for reasoning, analyzing and drawing inferences about distributed computations. Knowledge of the causal relationship among various events occurring at different processes helps designers solve a variety of problems related to distributed computation, such as ensuring fairness of a distributed mutual exclusion algorithm, maintaining consistency in replicated databases and distributed deadlock detection algorithms. Such knowledge is also useful in constructing a consistent state for resuming execution in distributed debugging, building a checkpoint for distributed recovery and detecting inconsistencies in replicated databases [109].

In distributed systems no built-in mechanism for a system wide clock exists and a causal precedence relation cannot be captured accurately. Logical clocks have been proposed as a viable solution to address this problem. Due to the absence of a common global clock and shared memory, the various processes in distributed systems communicate with each other by exchange of messages. More precisely, during a distributed computation, the processes produce and receive the messages from the cooperating processes. These messages are delivered after arbitrary delays. A class of problems related to such message passing systems may be solved by defining global ordering on the messages. Logical clocks such as Lamport clocks [75] and vector clocks [43, 90] provide a mechanism to ensure globally ordered delivery of the messages. The causal ordering of messages was proposed and discussed in [20, 75] and the protocols proposed in [21, 113] use logical clocks to maintain the causal order of messages. A critical review of logical clocks can be found in [18, 109]. Vector clocks were used by other researchers in [21, 69, 89, 133] to design and develop distributed systems.

2.4.1 Lamport’s Clocks

In Lamport’s logical clock system [75], a clock is defined as a function which assigns a number to an event. For every process $P_i$ there exists a clock $C_i$ which essentially maps an event to an integer. Suppose the set $E_{P_i}$ defines the sequence of events produced by a process $P_i$ as,

$$E_{P_i} = \{ e_{i1}, e_{i2}, e_{i3}, e_{i4}, ..., e_{in} \}$$

The clock function $C_i$ may be defined as follows,

$$C_i : E_{P_i} \rightarrow \mathbb{N}$$

where $\mathbb{N}$ is a set of natural numbers.
The logical clock present at every process takes monotonically increasing values and assigns a number to every event, called a timestamp of that event. Formally, a clock \( C_i \) present at a process \( P_i \) assigns a timestamp \( C_i(a) \) to an event \( a \) where \( a \in E_{P_i} \). The correctness criterion for this clock may be defined as follows.

- For any two internal events \( a \) and \( b \) occurring in a process \( P_i \), \( a \rightarrow b \Rightarrow C_i(a) < C_i(b) \).
- If \( a \) and \( b \) are message sent and message receive events of a message \( m \) occurring in the processes \( P_i \) and \( P_j \) respectively then \( C_i(a) < C_j(b) \).

In this system every message is also timestamped before sending it to a recipient process. This timestamp is equal to the timestamp of the message sent event of that message at the sender’s process. The correctness criterion outlined above can be guaranteed by following two implementation rules.

- The clock \( C_i \) at process \( P_i \) is incremented between two successive internal events as follows.
  \[ C_i = C_i + d, \text{ where } (d = 1). \]
- If \( a \) and \( b \) are message sent and message receive events of a message \( m \) occurring in the processes \( P_i \) and \( P_j \) respectively, then the message \( m \) is time-stamped as \( C_m := C_i(a) \). The \( C_i(a) \) is obtained by applying the previous rule. The timestamp \( C_j \) of a recipient process of the message is updated as below.
  \[ C_j := \max (C_j, C_m + d), \text{ where } (d = 1). \]

**Consistency of Lamport Clocks**

Raynal and Singhal introduced the notion of the consistency of logical clocks in [109]. Let \( E_1 \) and \( E_2 \) be any two events generated by a process(es). A logical clock is consistent if the following criteria is satisfied.

\[ E_1 \rightarrow E_2 \Rightarrow C(E_1) < C(E_2). \]

A logical clock is strongly consistent if following criteria is satisfied.

\[ E_1 \rightarrow E_2 \iff C(E_1) < C(E_2) \]

Lamport clocks are consistent due to their monotonically increasing values. However, the limitation of Lamports clocks is that they are not strongly consistent. By comparing
the timestamp of any two events, which occurred in different processes, it cannot be
guaranteed whether these events are casually related or not, i.e.,

\[ C_i(a) < C_j(b) \Rightarrow a \rightarrow b. \]

Consider Fig. 2.6 where occurrences of the internal and message events happening in
a set of processes are shown. The scalar timestamps of various events are also shown
in the diagram. Consider the message \( M_1 \) sent from process \( P_1 \) to \( P_2 \). The message
sent event \( E_{11} \) in a process \( P_1 \) happened before message receive event \( E_{21} \) in a process
\( P_2 \). The timestamp of these two events assigned by the logical clock system are 1 and 2
respectively. Since these events are casually related, it satisfies the following consistency
criterion defined over message events.

\[ E_{11} \rightarrow E_{21} \Rightarrow TS(E_{11}) < TS(E_{21}) \]

Similarly, consider events \( E_{21} \) and \( E_{31} \) occurring in process \( P_2 \) and \( P_3 \) respectively.
The timestamp generated by the Lamport clock system for these events are 2 and 1
respectively. Since these events are not causally related, it can not be determined if one
happened before another. Therefore, Lamport clocks are not strongly consistent. i.e.,

\[ TS(E_{31}) < TS(E_{21}) \Rightarrow E_{31} \rightarrow E_{21} \]

The reason for this behavior is that the process does not keep information about whether
advancement in the clock happened due to a internal event or a message event. Clocks

\[ P1 \]
\[ E11 \] \( M_1 \)
\[ E12 \] \( E13 \)
\[ P2 \]
\[ E21 \] \( E22 \) \( E23 \)
\[ M_3 \]
\[ P3 \]
\[ E31 \] \( E32 \)

\[ \textcircled{\textnormal{Internal Events}} \] \( \bullet \) \( \textnormal{Message Events} \)

\textbf{Figure 2.6: Lamport Clock}
at each process advance independently due to occurrences of events at that process. De-
spite this shortcoming, Lamport’s clock have been found suitable to solve some classical
distributed computing problems, such as, distributed mutual exclusion [75]. However,
the main advantage of Lamport’s clock is that upon receipt of any message a process
updates its logical time (clock) to more than the time of the previously known event at
the sender process.

The advancement of a Lamport clock is shown in the Fig. 2.6. Time stamps for the events
are obtained by applying the implementation rules. The same rules can be applied to
obtain the timestamp of events in a broadcast system. The advancement due to Lamport
clock in a broadcast system is given in Fig. 2.7.

2.4.2 Vector Clocks

One of the major limitations of Lamport clocks is that they are not strongly consistent
which means that by comparing the timestamp of two events it can not be decided
whether the events are casually related. This problem was addressed in the vector
clocks proposed by Fidge and Mattern in [43] and [90]. The vector clock overcomes the
limitation of Lamport clocks and the timestamps generated by the vector clock system
may be compared to decide the causal order of occurrence of two events.

In a system of vector clocks, every process maintains a vector of size $N$ to represent
the logical time at that process, where $N$ is equal to the total number of processes in
that system. Let each process $P_i$ maintain a vector clock $VT_i$. $VT_i$ can be defined as a
function which assigns to every event a vector called, a vector timestamp.

Suppose set $E_{P_i}$ defines the sequence of events produced by process $P_i$ then a clock
function $VT_i$ may be defined as follows,
\( E_{P_i} = \{ E_{i1}, E_{i2}, E_{i3}, E_{i4} \ldots \ldots E_{in} \} \)

\( VT_i : E_{P_i} \rightarrow \mathbb{V}, \) where \( \mathbb{V} \) is set of vector of integers of size \( N \).

On occurrence of an event, a process uses the following two rules to update their clocks.

1. On occurrence of an internal event, process \( P_i \) updates its own time (the \( i^{th} \) entry of the vector) by updating \( VT_i(i) \) as follows:

\[
VT_i(i) := VT_i(i) + 1
\]

If the event is of sending a message \( M \) from process \( P_i \) to \( P_j \) then a message timestamp \( VT_M \) is generated as follows:

\[
VT_M(k) := VT_i(k), \quad \forall \, k \in (1..N).
\]

2. If the event is of receiving a message \( M \) from process \( P_i \) to \( P_j \), the recipient process \( P_j \) updates its vector clock \( VT_j \) as follows:

\[
VT_j(k) := \text{Max}(VT_j(k), VT_M(k)), \quad \forall \, k \in (1..N), \, k \neq j
\]

Process \( VT_j \) updates its own clock \( VT_j(j) \) as follows:

\[
VT_j(j) := VT_j(j) + 1
\]

As mentioned in the first implementation rule, on occurrence of an internal event, a process \( P_i \) updates its own time \( VT_i(i) \). Therefore, \( VT_i(i) \) represents a local logical time at process \( P_i \). The entry \( VT_i(j) \) (\( i \neq j \)) represents the process \( P_i \)'s latest knowledge of time at process \( P_j \).

**Consistency of Vector Clocks**

A system of vector clocks is strongly consistent. In a system of vector clocks, vector timestamps of two events may be compared to find if these two events are casually related. Rules for comparing the timestamp of two events were proposed in [90] and these properties were further investigated by Raynal and Singhal in [109]. They proposed the following criterion to compare the vector timestamp of two events.

Let the vector timestamp of events \( a \) and \( b \) be \( VT_a \) and \( VT_b \) respectively. The following holds.

\[
VT_a = VT_b \iff \forall i \cdot VT_a(i) = VT_b(i)
\]

\[
VT_a \neq VT_b \iff \exists i \cdot VT_a(i) \neq VT_b(i)
\]

Similarly, the following relations compare timestamps to show if there exists a casual order among the events or if they are concurrent.
\[ \forall i \cdot VT_a(i) \leq VT_b(i) \text{ and } \exists k \cdot VT_a(k) < VT_b(k) \]

\[ VT_a \parallel VT_b \iff \neg (VT_a < VT_b) \land \neg (VT_b < VT_a) \]

In the case that two events \( a \) and \( b \) occurred in the same process \( P_i \), their causality order satisfies the following property.

\[ a \rightarrow b \iff VT_i(a) < VT_i(b) \]

Similarly, if the events \( a \) and \( b \) occurred in process \( P_i \) and \( P_j \) respectively, their causality order satisfies the following property.

\[ a \rightarrow b \iff VT_i(a) < VT_j(b) \]

\[ a \parallel b \iff \neg (VT_i(a) < VT_j(b)) \land \neg (VT_j(b) < VT_i(a)) \]

This also implies that after comparing the timestamp of two events occurring in different processes we can determine if they causally affect each other or they are concurrent.

**Advancement of Vector Clock: An Example**

The advancement of the vector clock is illustrated in the Fig. 2.8. Let \( VT_1, VT_2 \) and \( VT_3 \) be vector clocks present at processes \( P_1, P_2 \) and \( P_3 \) respectively.

The event \( E_{11} \) in process \( P_1 \) which is a message send event of sending a message \( M_1 \) to process \( P_2 \). Similarly, the event \( E_{21} \) is a message receive event occurring due to receipt of a message \( M_1 \) at process \( P_2 \).
Since $E_{11}$ is the first event produced by process $P_1$, it updates its clock $VT_1$ following the first implementation rule of vector clock as $VT_1(1):= VT_1(1)+1$. Therefore, the vector timestamp of event $E_{11}$ is generated as $VT_1(E_{11}):=[100]$.

The message $M_1$ is timestamped as $VT_M := VT_1(E_{11})=[100]$.

Upon receipt of message $M_1$, the clock at process $P_2$ is updated as

$$VT_2(k) := \text{Max}(VT_2(k), VT_M(k)), \; \forall \; k \in (1..N), \; k \neq 2$$

The clock $VT_2(2)$ is updated as $VT_2(2) := VT_2(2)+1$ according to second implementation rule. Therefore, the process $P_2$ assigns a vector timestamp to event $E_{21}$ as $VT_2(E_{21}) := [110]$.

The event $E_{12}$ happened after the occurrence of the message sent event $E_{11}$ in the process $P_1$. Process $P_1$ generates the vector timestamp for event $E_{12}$ by advancing the clock $VT_1$ as $VT_1(1) := VT_1(1)+1=2$. Therefore, the timestamp of event $E_{12}$ is generated as $VT_1(E_{12}) := [200]$.

Similarly, consider a message $M_2$ sent from a process $P_2$ to a process $P_1$. The event $E_{22}$ in a process $P_2$ and the event $E_{13}$ in a process $P_1$ are the outcome of the message sent and the message receive events of a message $M_2$. The timestamp for message send and message receive events of $M_2$ are generated as follows.

- The message $M_2$ is timestamped as $VT_{M_2} := VT_2(E_{22})=[120]$.
- Upon receipt of message $M_2$, the clock at process $P_1$ is updated as,

$$VT_1(k) := \text{Max}(VT_1(k), VT_{M_2}(k)), \; \forall k \in (1..N), \; k \neq 1.$$

The $VT_1(1)$ is updated as $VT_1(1) := VT_1(1)+1=3$. Therefore, the process $P_1$ assigns a vector timestamp to event $E_{13}$ as $VT_1(E_{13}) := [320]$.

Event $E_{31}$ is the first event occurring in the process $P_3$. It is assigned the vector timestamp $[001]$ following the first rule. The timestamp for the message $M_3$ and its associated events $E_{32}$ and $E_{23}$ are assigned as $[002]$, $[002]$ and $[132]$ respectively by the vector clock system as outlined above.

### 2.5 Event-B

Event-B [2, 7, 92] is a formal technique consisting of describing the problem, introducing solutions or details in refinement steps to obtain more concrete specifications and verifying that proposed solutions are correct. Event-B, a variant of B [1], was designed for developing distributed systems.
2.5.1 Modelling Approach in Event-B

A specific development in this approach is made of a series of further refined models. In Event-B, a system is modelled in terms of an abstract state space using variables with set theoretic types and the events that modify state variables. The events consist of guarded actions occurring spontaneously rather then being invoked. At each refinement step, new variables may be introduced and abstract variables may be removed. Each model is made of static properties (invariants) and dynamic properties (events). A list of state variables is modified by a finite list of events. The events are guarded by predicates and these guards may be strengthened at each refinement step. The invariants are properties that must be satisfied by the variables and maintained by the activation of events.

We have used the Click’n’Prove [6] B tool for proof management which provides an environment for the generation of proof obligations for consistency checking and refinement checking. This tool also provides an automatic and an interactive prover. The majority of the proof obligations are proved by the automatic prover. However, some of the complex proof obligations need to be proved interactively.

2.5.1.1 An Event-B System

The notions of abstract machine and refinement are central to an Event-B system. An abstract machine consists of sets, constants and variable clauses modelled as set theoretic constructs. The invariants and properties are defined as first order predicates. The event system is defined by its state and contains a number of events. The state of the system is defined by the variables. The constants and variables are constrained by the conditions defined in the properties and invariant clauses. Each event in the abstract model is composed of a guard and an action. The events are modelled using generalized substitution which includes constructs like assignment ($x := E(x)$) and guarded statement ($WHEN G \ THEN \ S \ END\). A typical abstract machine is outlined in the Fig. 2.9.

\[
\begin{align*}
\text{MACHINE} & \quad M \\
\text{SETS} & \quad S_1,S_2,S_3... \\
\text{CONSTANTS} & \quad C \\
\text{PROPERTIES} & \quad P \\
\text{VARIABLES} & \quad v_1,v_2,v_3... \\
\text{INVARIANTS} & \quad I \\
\text{INITIALISATION} & \quad \text{init} \\
\text{EVENTS} & \\
E1 & \equiv \text{WHEN } G_1 \ \text{THEN } S_1 \ \text{END}; \\
E2 & \equiv \text{WHEN } G_2 \ \text{THEN } S_2 \ \text{END}; \\
& \ldots
dd
\end{align*}
\]

**Figure 2.9:** Event-B Machine
In the guarded statement \((\text{WHEN } G \text{ THEN } S \text{ END})\), the guard \((G)\) of the event is expressed as a first order predicate. The actions of an event are specified as simultaneous assignments of state variables using substitution statements \((S)\). Events occur spontaneously whenever their guard holds (true) and they are executed atomically. After building a model of a system as an abstract machine, it must be proved that a system is consistent with respect to the invariant properties of the system. The consistency of the machine is shown by proving that each event of the system preserves the invariant.

### 2.5.1.2 Gluing Invariants

In an incremental development approach for system modelling we begin with an abstract definition of the problem. The system is built in several stages by gradually introducing the details in the refinement steps. An abstract machine can be refined by adding new events and by adding or removing variables. A refined system state must relate to the abstract one by an abstraction relation. This abstraction relation is defined by an invariant known as the gluing invariant. This invariant defines the relationship between abstract state variables and concrete (refined) state variables. More precisely, if a statement \(S\) that acts on variable \(x\), is refined by another statement \(T\) that acts on variable \(y\) under invariants \(I\) then we write \(S \sqsubseteq_I T\). The invariant \(I\) is called the gluing invariant and it defines the relationships between \(x\) and \(y\). Each event of the abstract model is refined to one or more corresponding concrete events. A concrete event is said to refine a corresponding abstract one, if it is obtained by strengthening the guards of the abstract one and the gluing invariant is preserved by the joined actions of both events.

Replacing the abstract variable by the concrete variable in the refinement results in proof obligations that are generated by the B tools. These proof obligations are associated with the events in the refinement. The B tool helps to factorize large and complex proof obligations into simpler proof obligations. In most cases the majority of the proof obligations are proved by the automatic prover. However, in some cases we need to prove them by interaction. The B tools also remembers the proved and unproved proofs in the form of a proof tree. In some cases, in order to prove the unproved proof obligations we may have to add gluing invariants to the model. In these cases the unproved proof obligations guide us to construct the gluing invariants. The addition of new gluing invariants can result in the generation of further proof obligations which may require the addition of new gluing invariants. After several stages of invariant strengthening we expect to arrive at a set of invariants which is sufficient to discharge all proof obligations.

In our case studies given in the Chapter 3, 4, 5 and 6, we outline the construction of the invariants by inspection of the proof obligations, generated by the B Tool. A model is said to be consistent with respect to a discovered invariant, only if the invariant holds on the initial state given by initialization clause of B machine, and the activation of
each event preserves the invariant. An invariant constructed \textit{incorrectly} may discharge the some of the proof obligations. However, the additional proof obligations generated by the B Tool associated with other events and initialization, cannot be discharged. In the modelling guidelines presented in Section 7.3 of Chapter 7, we addressed the issue of the prover’s failure to discharge a proof obligation and recommend model checking to precisely understand what is wrong with the newly constructed invariant.

The addition of an appropriate invariant is a key to proving the correctness of the refinement. In this approach not only do proof obligations and the interactive prover together guide the construction of new gluing invariants, but it also has the consequence that the form of gluing invariants closely matches the form of proof obligations, thereby, making the mechanical proof much easier and in many cases completely automatic.

\textbf{Consistency and Refinement Checking}

Informally, \textit{safety properties} express that something \textit{bad} will not happen during system execution. With regards to the safety properties, the existing tools generate proof obligations for following.

1. \textit{Consistency Checking} : Consistency of a machine is established by proving that the actions associated with each event modifies the variables in such a way that the invariants are preserved under the hypothesis that the invariants hold initially and the guards are true. The existing B tools generate proof obligations for consistency checking. By discharging these proof obligations we ensure that machine is consistent with respect to the invariants.

2. \textit{Refinement Checking} : The refinement of a machine consists of refining its state and events. The gluing invariants relate the state of the refinement, represented by the concrete variables, to the abstract state, represented by the abstract variables. An event in the abstraction may be refined by one or more events, and the tool generates the proof obligations to ensure that gluing invariants are preserved by actions of the events in the refinement.

Discharging the proof obligations generated due to consistency checking means that actions of the events do not violate the invariants. Discharging the proof obligations due to the refinement checking implies that each reachable concrete state is also reachable in the abstraction.

\textbf{Non-Divergence and Enabledness Preservation}

It is sometimes useful to state that the model of the system under development is \textit{non-divergent} and \textit{enabledness} preserving. The issues relating to these properties are
currently being addressed in the new generation of Event-B tools being developed [44, 92]. These properties are informally defined below.

1. **Non-Divergence**: In an incremental development approach using Event-B, new events and the variables can be introduced in the refinement steps. Each new event of the refinement refines a *skip* event in the abstraction and defines actions on the new variables. Proving the non-divergence requires us to prove that the new events do not take control forever. This constraint requires proof of a condition on a *variant*. A variant clause contains a positive integer expression and every new event introduced in the refinement should decrease the value of this expression. This mechanism guarantees that new events cannot diverge, since the variant expression cannot be decreased indefinitely.

2. **Enabledness Preservation**: By enabledness preservation, we mean that whenever some event (or group of events) is enabled at the abstract level then the corresponding event (or group of events) is eventually enabled at the concrete level. This property can be proved by stating that the guards of abstract event implies the disjunction of the guards of the refined events and the disjunction of the guards of new events.

The non-divergence and enabledness preservation properties with respect to our model of transactions are further addressed in the Chapter 7.

### 2.5.2 Event-B Notation

The Event-B notations are based on set theoretic notation and most of it is self-explanatory. However, the frequently used notations in our models are outlined here to increase the readability of the thesis. The Event-B notations are broadly classified as relational notation (Table 2.1) and function notation (Table 2.2).

#### Relational Notations

The *relation* is the most important structure used in Event-B specifications to maintain the relationship between two sets. Some of the important relational notations and their meaning is given in following table.

Let A and B be two sets. The notation \( \leftrightarrow \) defines the set of relations between A and B as:

\[
A \leftrightarrow B = \mathcal{P}(A \times B)
\]

where \( \times \) is cartesian product of A and B. A mapping of element \( a \in A \) and \( b \in B \) in a relation \( R \in A \leftrightarrow B \) is written as \( a \mapsto b \). The *domain* of a relation \( R \in A \leftrightarrow B \) is the
Chapter 2 Background

### Notations

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mapsto$</td>
<td>mapping</td>
</tr>
<tr>
<td>$\text{dom}(R)$</td>
<td>domain of relation $R$</td>
</tr>
<tr>
<td>$\text{ran}(R)$</td>
<td>range of relation $R$</td>
</tr>
<tr>
<td>$\prec$</td>
<td>domain restriction</td>
</tr>
<tr>
<td>$\succ$</td>
<td>range restriction</td>
</tr>
<tr>
<td>$\preceq$</td>
<td>domain anti-restriction</td>
</tr>
<tr>
<td>$\preceq$</td>
<td>overridden operator</td>
</tr>
<tr>
<td>$R[A]$</td>
<td>relational image of $R$ over set $A$</td>
</tr>
</tbody>
</table>

Table 2.1: Relational Notations

A relation $R \in A \leftrightarrow B$ can be projected on a domain $U \subseteq A$ called *domain restriction* ($\prec$) defined as:

$$U \prec R = \{ a \mapsto b \mid a \mapsto b \in R \land a \in U \}$$

*Domain anti-restriction* ($U \preceq R$) is defined as:

$$U \preceq R = \{ a \mapsto b \mid a \mapsto b \in R \land a \notin U \}$$

Similarly, *range restriction* ($\succ$) is the projection of $R$ whose second element is in $V \subseteq B$. The range restriction is defined as:

$$R \succ V = \{ a \mapsto b \mid a \mapsto b \in R \land b \in V \}$$

The *relational image* $R[U]$ where $U \subseteq A$ is defined as:

$$R[U] = \{ b \mid a \mapsto b \in R \land a \in U \}$$

The *relational inverse* ($R^{-1}$) of a relation $R$ is defined as:

$$R^{-1} = \{ b \mapsto a \mid a \mapsto b \in R \}$$

If $R_0 \in A \leftrightarrow B$ and $R_1 \in A \leftrightarrow B$ are relations defined on sets $A$ and $B$, the *relational over-ride* operator ($R_0 \lhd R_1$) replaces certain mappings in relation $R_0$ by those in
relation \( R_1 \).

\[ R_0 \uplus R_1 = (\text{dom}(R_1) \uplus R_0) \cup R_1 \]

**Function Notations**

Event-B extensively uses the notion of functions. A *function* is a relation having some special properties. A *partial function* from set \( A \) to \( B \) (\( A \rightarrow B \)) is a relation which relates an element in \( A \) to *at most* one element in \( B \). A partial function \( f \in A \rightarrow B \) satisfies the following:

\[ \forall (a, b_1, b_2). (a \in A \land b_1 \in B \land b_2 \in B \Rightarrow (a \rightarrow b_1 \in f \land a \rightarrow b_2 \in f) \Rightarrow b_1 = b_2) \]

Similarly a *total function* \( f \in A \rightarrow B \) is a partial function where \( \text{dom}(f) = A \). Given \( f \in A \rightarrow B \) and \( a \in \text{dom}(f) \), \( f(a) \) represents the unique value that \( a \) is mapped to by \( f \).

An *injective function* never maps two different elements of the source set to the same element of the target set. Injective functions may be of two types, *partial injection* or *total injection*. A partial injection from set \( A \) to \( B \) (\( A \mapsto B \)) may be defined as:

\[ A \mapsto B = \{ f | f \in A \rightarrow B \land f^{-1} \in B \rightarrow A \} \]

A total injection is a partial injective function which is also a total function defined as

\[ A \mapsto B = (A \mapsto B) \cap (A \rightarrow B) \]

Some of the important function notations and their meaning is given in Table 2.2. A more detailed explanation of these operations may be found in [1, 117].

<table>
<thead>
<tr>
<th>Notations</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \leftrightarrow )</td>
<td>partial function</td>
</tr>
<tr>
<td>( \rightarrow )</td>
<td>total function</td>
</tr>
<tr>
<td>( \mapsto )</td>
<td>partial injection</td>
</tr>
<tr>
<td>( \mapsto \rightarrow )</td>
<td>total injection</td>
</tr>
</tbody>
</table>

**Table 2.2:** Function Notations

**2.6 Conclusions**

In this chapter, we outlined the different issues related to replicated data updates, fault-tolerance, consistency and failures in a distributed database system. Capturing the causal precedence relation among the different events occurring in distributed systems is a key to the success of distributed computation. The concept of logical clocks addresses this problem. Logical clocks such as the Lamport clock and vector clock can be used
to solve a variety of problems relating to distributed mutual exclusion, consistency in replicated databases, distributed debugging, checkpointing and failure recovery. In a system of Lamport clocks, a clock at a process is represented by an integer value. The advantage of Lamport clocks is that messages *piggyback* less information but they suffer from the disadvantage that they are not strongly consistent. In vector clocks, a clock at a process is represented by a vector of integers whose size equals the number of processes in the system. The advantage of vector clocks over Lamport clocks is that they are strongly consistent. However, they suffer from the disadvantage that all messages *piggyback* a vector and message overheads are likely to increase. An approach to the reduction of vector timestamp has been addressed in [16, 120].

Finally, we have outlined our approach to modelling and reasoning about distributed system using Event-B. Event-B [2, 7, 92] is a formal technique consisting of describing a problem, rigorously introducing the solutions or design details in refinement steps to obtain more concrete specifications and verifying that the proposed solutions are correct. The approach to specifying the system and verification is based on the technique of abstraction and refinement. There exist several industrial level tools to support B development such as Click’n’Prove [6], Atelier B [127], and the B-Toolkit [33] which provide an environment for generation of proof obligations for *consistency checking* and *refinement checking*. Recently, a new generation RODIN B tool [5] has been developed which provides specific support for Event-B development. Applications of the B method to develop models of distributed systems include modelling a web based systems [110], a secure communication system [25], verification of one-copy equivalence criterion in a distributed database system [142], verification of the IEEE 1394 tree protocol distributed algorithm [7], a Mondex Purse [31]. The general modelling approach for distributed systems may be found in [24].
Chapter 3

Distributed Transactions

3.1 Introduction

In this chapter, we formally develop an abstract model of transactions in Event-B for a one-copy database. The notion of a replicated database is introduced in the refinement of the abstract model. The replica control mechanism considered in the refinement allows both update and read-only transactions to be submitted at any site. In our abstract model, an update transaction modifies the abstract one-copy database through a single atomic event. In the refinement, an update transaction consists of a collection of interleaved events updating each replica separately. The transaction mechanism on the replicated database is designed to provide the illusion of an atomic update of a one-copy database. Through our refinement proof we verify that this is indeed the case. A read-only transaction reads the values from a replica locally at the site of submission. Transaction failure is represented by sites aborting transactions. A site may decide to abort an update transaction due to race conditions among conflicting transactions. We address the one-copy equivalence consistency criterion through this refinement. By verifying the refinement, we verify that the design of the replicated database confirms to the one-copy database abstraction despite transaction failures at a site.

The remainder of this chapter is organized as follows: Section 3.2 describes the system model informally, Section 3.3 presents an abstract Event-B model of transactions considering the database as single logical entity, Section 3.4 presents a refinement of the abstract Event-B model introducing details of the replicated database, Section 3.5 presents some properties of system given as gluing invariants detailing the relationship between the single copy and the replicated database, Section 3.6 presents another refinement where explicit messaging is introduced. In this refinement, we show how a reliable broadcast can be used to ensure transaction execution. In Section 3.7 we address site failures and transaction abortion using a timeout and finally Section 3.8 concludes the chapter.
3.2 System Model

In this section, we present an informal model of a replicated database. Our system model consists of a set of sites and data objects. Users interact with the database by starting transactions. We consider the case of full replication and assume all data objects are updateable. The read anywhere write everywhere replica control mechanism is considered for updating replicas. A transaction is considered as a sequence of read/write operations executed atomically, i.e., a transaction will either commit or abort the effect of all database operations.

3.2.1 Transaction Types

Let the sequence of read/write operations issued by the transaction $T_i$ be defined by a set of objects $\text{objectset}(T_i)$ where $\text{objectset}(T_i) \neq \emptyset$. Let the set $\text{writeset}(T_i)$ represent the set of objects to be updated such that $\text{writeset}(T_i) \subseteq \text{objectset}(T_i)$. The following types of transactions are considered for this model of a replicated database.

- **Read-Only Transactions**: These transactions are submitted locally to the site and commit after reading the requested data object locally. A transaction $T_i$ is defined as a read-only transaction if $\text{writeset}(T_i) = \emptyset$.

- **Update Transactions**: These transactions update the requested data objects. The effects of update transactions are global, thus when committed, all replicas of data objects maintained at all sites must be updated. In case of abort, none of the sites update the data object. A transaction $T_i$ is a update transaction if its $\text{writeset}(T_i) \neq \emptyset$.

- **Conflicting Update Transactions**: Two update transactions $T_i$ and $T_j$ are in conflict if the sequence of operations issued by $T_i$ and $T_j$ are defined on sets of objects, i.e., $\text{objectset}(T_i) \cap \text{objectset}(T_j) \neq \emptyset$.

In order to meet the strong consistency requirement where each transaction reads the correct value of a replica, conflicting transactions need to be executed in isolation. We consider the case of general isolation [39], where the sequence of operations issued by conflicting transactions are executed in isolation at all participating sites. In our model, we ensure this property by not issuing a transaction at a site if there is a conflicting transaction that is active at that site. In our model the transactions are executed within the framework of the two phase commit protocol [48, 49] as follows.

- A read-only transaction $T_i$ is executed locally at the initiating site of $T_i$ (also called the coordinator site of $T_i$) by acquiring locks on the data object defined by $\text{objectset}(T_i)$. 

An update transaction $T_i$ is executed by broadcasting its operations to the participating sites. On delivery, a participating site $S_j$ initiates a sub-transaction $T_{ij}$ by acquiring locks on $\text{objectset}(T_i)$. If the objects are currently locked by another transaction, $T_{ij}$ is blocked.

The coordinator site of $T_i$ waits for the commit/abort vote messages from all participating sites. A global commit message is broadcast by the coordinator site of $T_i$ only if it receives local commit messages from all participating sites and a global abort message is broadcast if there is at least one vote-abort message from participating sites.

### 3.2.2 Race Conditions

In a replicated database that uses a reliable broadcast, conflicting operations of the transactions may arrive at different sites in different orders. Since operations of update transactions are executed by sending update messages to all participating sites, every participating site obtains a lock on the requested data object and retains the lock until the transaction globally commits using a two phase commit. This may lead to the blocking of conflicting transactions and the sites may abort one or more of the conflicting transaction by timeouts. For example, consider two conflicting update transactions $T_i$ and $T_j$ initiated at site $S_i$ and $S_j$ respectively. Both of the transactions may be blocked in the following scenario:

- $S_i$ starts transaction $T_i$ and acquires locks on $\text{objectset}(T_i)$ at site $S_i$. Site $S_i$ broadcasts an update messages for $T_i$ to participating sites. Similarly, another site $S_j$ starts a transaction $T_j$, acquires locks on $\text{objectset}(T_j)$ at site $S_j$ and broadcasts an update messages for $T_j$ to participating sites.

- The site $S_i$ delivers an update message for $T_j$ from $S_j$ and $S_j$ delivers an update message for $T_i$ from $S_i$. The $T_j$ is blocked at $S_i$ as $S_i$ waits for vote-commit from $S_j$ for $T_i$. Similarly, $T_i$ is blocked at $S_j$ waiting for vote-commit from $S_i$ for $T_j$.

In order to recover from the above scenario where two conflicting transactions are blocked, either or both transactions may be aborted by the sites. This problem can be removed by assuming a stronger notion of reliable broadcast that provides higher order guarantees on message delivery [9, 125]. The abortion of the conflicting transaction can be avoided by using a total order broadcast which delivers and executes the conflicting operations at all sites in the same order. Similarly, a causal order broadcast captures conflict as causality and transactions executing conflicting operations are executed at all sites in the same order. Processing update transactions over a total causal order broadcast not only delivers the operations in a total order at the participating site, but the causal precedence relationship among the update transaction is also preserved.
3.3 Abstract Model of Transactions in Event-B

The abstract data model of transactions is given in Fig. 3.1 as a B machine. The abstract model maintains a notion of a central or one-copy database. The abstract database is modelled as a total function from objects to values:

\[
database \in OBJECT \rightarrow VALUE
\]

In practice a database will be partial, but for simplicity we avoid dealing with the errors caused by trying to read undefined objects and instead focus on errors caused by sites failing to commit a transaction. An individual transaction will involve a set of objects \( \text{readset} \subseteq OBJECT \). It will read from a partial projection of the database \((\text{pdb})\) on to \text{readset}, i.e.,

\[
\text{pdb} = \text{readset} \triangleleft \text{database}
\]

If it is an update transaction it will write to a subset of \text{readset} and the new values of the objects to be written may depend on the existing values of the objects in \text{readset}. Let the set of objects to be written be \text{writeset} where \( \text{writeset} \subseteq \text{readset} \). So we model an update to a database as function that takes a partial database (representing the current values of the objects in \text{readset}) and yields a partial database (representing the new values of the objects in \text{writeset}).

\[
\text{VALIDUPDATE} (\text{update,readset}) = (\text{dom}(\text{update})= \text{readset} \rightarrow \text{VALUE} \land \text{ran}(\text{update}) \subseteq \text{readset} \rightarrow \text{VALUE} )
\]

\[
\begin{align*}
\text{MACHINE} & \quad \text{Replica} \\
\text{DEFINITIONS} & \quad \text{PartialDB} == (OBJECT \rightarrow VALUE) ; \\
& \quad \text{UPDATE} == (\text{PartialDB} \rightarrow \text{PartialDB}) ; \\
& \quad \text{ValidUpdate} (\text{update,readset}) == (\text{dom}(\text{update})= \text{readset} \rightarrow \text{VALUE} \\
& \quad \quad \quad \land \text{ran}(\text{update}) \subseteq \text{readset} \rightarrow \text{VALUE} )
\end{align*}
\]

\[
\begin{align*}
\text{SETS} & \quad \text{TRANSACTION; OBJECT; VALUE;} \\
\text{TRANSSTATUS} & = \{\text{COMMIT, ABORT, PENDING}\} \\
\text{VARIABLES} & \quad \text{trans, transstatus, database, transeffect, transobject}
\end{align*}
\]

\[
\begin{align*}
\text{INVARIANT} & \quad \text{trans} \subseteq \text{P(TRANSACTION)} \\
& \quad \land \text{transstatus} \subseteq \text{trans} \rightarrow \text{TRANSSTATUS} \\
& \quad \land \text{database} \subseteq \text{OBJECT} \rightarrow \text{VALUE} \\
& \quad \land \text{transeffect} \subseteq \text{trans} \rightarrow \text{UPDATE} \\
& \quad \land \text{transobject} \subseteq \text{trans} \rightarrow \text{P_1(OBJECT)} \\
& \quad \land \forall t.(t \in \text{trans} \Rightarrow \text{ValidUpdate} (\text{transeffect}(t), \text{transobject}(t))) )
\end{align*}
\]

\[
\begin{align*}
\text{INITIALISATION} & \quad \text{trans} ::= \emptyset \quad || \quad \text{transstatus} ::= \emptyset \\
& \quad || \quad \text{transeffect} ::= \{\} \quad || \quad \text{transobject} ::= \{\} \\
& \quad || \quad \text{database} ::= \emptyset \subseteq \text{OBJECT} \rightarrow \text{VALUE}
\end{align*}
\]

Figure 3.1: Abstract Model of Transactions in Event-B
A transaction is a read-only transaction if its writeset = ∅. Thus, for a read-only transaction, its update function maps a partial database defined over readset to an empty set. The update function is defined as below,

\[ \text{UPDATE} \equiv \text{PartialDB} \rightarrow \text{PartialDB} \]

where \( \text{PartialDB} \equiv \text{OBJECT} \rightarrow \text{VALUE} \)

An update function \( \text{update} \) maps a partial database \((pdb1)\) where \( pdb1 = \text{readset} \subset \text{database} \) to another partial database \((pdb2)\) where \( \text{dom}(pdb2) = \text{writeset} \). The update function \( \text{update} \in \text{UPDATE} \) updates the database as follows,

\[ \text{database} := \text{database} \leftarrow \text{update}(pdb1) \]

As shown above, a database is written by reading the values from a partial database \((pdb1)\) defined over readset. The data objects to be updated in the database are defined as \( \text{update}(pdb1) \) which represent the computation associated with the transaction. We say that an update \( \text{update} \in \text{UPDATE} \) is valid with respect to a set of objects readset whenever,

\[ \text{dom}(\text{update}) = \text{readset} \rightarrow \text{VALUE} \]
\[ \text{ran}(\text{update}) \subseteq \text{readset} \rightarrow \text{VALUE} \]

A brief description of our abstract data model of transactions in Fig. 3.1 is given below.

- \( \text{TRANSACTION, SITE, OBJECT} \) and \( \text{VALUE} \) are defined as a deferred sets. The \( \text{TRANSSTATUS} \) is an enumerated set containing values \( \text{COMMIT}, \text{ABORT} \) and \( \text{PENDING} \). These values are used to represent the global status of transactions.

- The database is represented by a variable \( \text{database} \) as a total function from \( \text{OBJECT} \rightarrow \text{VALUE} \). A mapping, \( (o \mapsto v) \in \text{database} \), indicates that object \( o \) has value \( v \) in the database.

- The variable \( \text{trans} \) represents the set of started transactions. The variable \( \text{transstatus} \) maps each started transaction to \( \text{TRANSSTATUS} \).

- The variable \( \text{transobject} \) is a total function which maps a transaction to a set of objects. The set \( \text{transobject}(t) \) represents the set of data objects read by a transaction \( t \). The set of objects written to by \( t \) will be a subset of \( \text{transobject}(t) \).

- The variable \( \text{transeffect} \) is a total function which maps each transaction to an object update function \( \text{UPDATE} \) as previously described.

- A transaction \( t \) is a read-only transaction if \( \text{ran}(\text{transeffect}(t)) = \{\emptyset\} \), i.e., each partial database is mapped to the empty partial database.
– The invariant \( t \in \text{trans} \Rightarrow \text{ValidUpdate} (\text{transeffect}(t), \text{transobject}(t)) \) indicates that all updates must be valid.

### 3.3.1 Starting a Transaction

The event \( \text{StartTran}(tt) \), given in Fig 3.2, models starting a new transaction \( tt \). The \( \text{updates} \) and \( \text{objects} \) are event parameters constrained by the guard of the event. The guard given in the \( \text{WHERE} \) statement ensures that \( tt \) is fresh. The action of the event sets the variables \( \text{transobject}(tt) \) and \( \text{transeffect}(tt) \) so that \( \text{transobject}(tt) \) is a non empty set of objects and \( \text{transeffect}(tt) \) is some valid update on the objects. A transaction \( tt \) is considered as read-only if \( \text{ran} (\text{transeffect}(tt)) \) is an empty set and it is considered an update transaction if \( \text{ran} (\text{transeffect}(tt)) \) contains at least one mapping of the form \( (o \rightarrow v) \). The action of the event also sets the status of transaction \( tt \) to \( \text{PENDING} \).

### 3.3.2 Commitment and Abortion of Update Transactions

The event \( \text{CommitWriteTran}(tt) \) models commitment of an update transaction. As a consequence of the occurrence of this event, the abstract database is updated with the

\[
\text{StartTran}(tt \in \text{TRANSACTION}) \equiv \\
\text{ANY} \quad \text{updates, objects} \\
\text{WHERE} \quad tt \in \text{trans} \\
\quad \wedge \quad \text{updates} \in \text{UPDATE} \\
\quad \wedge \quad \text{objects} \in P_1 (\text{OBJECT}) \\
\quad \wedge \quad \text{ValidUpdate} (\text{updates}, \text{objects}) \\
\text{THEN} \\
\quad \text{trans} := \text{trans} \cup \{tt\} \\
\quad || \quad \text{transstatus}(tt) := \text{PENDING} \\
\quad || \quad \text{transobject}(tt) := \text{objects} \\
\quad || \quad \text{transeffect}(tt) := \text{updates} \\
\text{END};
\]

\[
\text{CommitWriteTran}(tt \in \text{TRANSACTION}) \equiv \\
\text{ANY} \quad \text{pdb} \\
\text{WHERE} \quad tt \in \text{trans} \\
\quad \wedge \quad \text{transstatus}(tt) = \text{PENDING} \\
\quad \wedge \quad \text{ran} (\text{transeffect}(tt)) \neq \{\emptyset\} \\
\quad \wedge \quad \text{pdb} = \text{transobject}(tt) \triangleq \text{database} \\
\text{THEN} \\
\quad \text{transstatus}(tt) := \text{COMMIT} \\
\quad || \quad \text{database} := \text{database} \triangleq \text{transeffect}(tt)(\text{pdb}) \\
\text{END};
\]

Figure 3.2: Events of Abstract Transaction Model- I
effects of the transaction and its status is set to commit. The B specification of this event is given in Fig 3.2.

The event AbortWriteTran(tt) models an abort of an update transaction. As a consequence of the occurrence of this event, the transaction status is set to abort and its effects are not written to the database. The Event-B specification of this event is given in Fig 3.3.

3.3.3 Commitment of Read-Only Transactions

The event ReadTran(tt), given in Fig 3.3, models commitment of a read-only transaction tt. A pending read-only transaction tt commits after reading the objects from the abstract database defined by variable transobject(tt). A read-only transaction commits by returning the values of the objects as a partial database.

\[
\text{AbortWriteTran}(tt \in \text{TRANSACTION}) \equiv \\
\text{WHEN } tt \in \text{trans} \\
\quad \land \text{transstatus}(tt) = \text{PENDING} \\
\quad \land \text{ran}(\text{transeffect}(tt)) \neq \{\emptyset\} \\
\text{THEN } \text{transstatus}(tt) := \text{ABORT} \\
\text{END;}
\]

\[
\text{val} \leftarrow \text{ReadTran}(tt \in \text{TRANSACTION}) \equiv \\
\text{WHEN } tt \in \text{trans} \\
\quad \land \text{transstatus}(tt) = \text{PENDING} \\
\quad \land \text{ran}(\text{transeffect}(tt)) = \{\emptyset\} \\
\text{THEN } \text{val} := \text{transobject}(tt) \sqsubset \text{database} \\
\quad \land \text{transstatus}(tt) := \text{COMMIT} \\
\text{END;}
\]

Figure 3.3: Events of Abstract Transaction Model- II

3.4 Refinements of the Transactional Model

3.4.1 Overview of the Refinement Chain

In the Section 3.3 we outlined the abstract model of transactions. An overview of the refinement chain is outlined below.

L1 This level consists of the abstract model of transactions. In the abstract model, an update transaction modifies the abstract one-copy database in a single atomic event. This level is presented in Section 3.3.
L2 We introduce the notion of replicated databases in this refinement. In this refinement, an update transaction consists of a collection of interleaved events updating each replica separately. The transaction mechanism on the replicated database is designed to provide the illusion of an atomic update of a one-copy database. Through our refinement proof we verify that this is indeed the case. This level is presented in Section 3.4.2.

L3 In this refinement we outline the simplification of the event of Global Commit. This is shown by strengthening the guards of CommitWriteTran event. This level is presented in Section 3.4.5.

L4 In this refinement we introduce the notion of messages. The various messages, corresponding to the two phase commit protocol, are introduced in this refinement that illustrates the integration of our transaction model with a broadcast system. The sites are assumed to communicate using a reliable broadcast. This refinement is given in Section 3.6.

L5 In this refinement we introduce the notion of site failures. We address the issue of participating site failures and show that a replicated database remains in a consistent state even in the presence of site failures. This refinement is outlined in Section 3.7.

3.4.2 First Refinement: Introducing the Replicated Databases

The initial part of the first refinement of the abstract model is given in Fig. 3.4. The Event-B specification of events of the refinement is introduced later in this section. The abstract Event-B model of transactions maintains a notion of an abstract central database. The variable database represents a central database in this model. In the refinement, the notion of replicated database is introduced. The abstract variable database is replaced by a concrete variable replica in the refinement. It may be noted that in the abstract model given in Fig. 3.2, an update transaction performs updates on an abstract central database, whereas, in the refined model, an update transaction updates replicas at each site separately. Similarly, a read-only transaction reads the data from the replica at the site of submission of that transaction. A brief description of the refinement is given below.

- The new variables coordinator, replica, activetrans, freeobject and sitetransstatus are introduced in the refinement. The variable coordinator is defined as a total function from trans to SITE. A mapping of form (t→s) ∈ coordinator implies that site s is the coordinator site for transaction t.

- Each site maintains a replica of the database. The variable replica is initialized to have the same value of each data object at each site. A mapping (s→(o→v)) ∈ replica indicates that site s currently has value v for object o.
Chapter 3 Distributed Transactions

REFINEMENT Replica2
REFINES Replica1

SETS SITE : $\{\text{commit, abort, precommit, pending}\}$

VARIABLES trans, transstatus, activetrans, coordinator, sitetransstatus, transeffect, transobject, freeobject, replica

INVARIANT activetrans $\subseteq$ SITE $\leftrightarrow$ trans
$\land$ coordinator $\subseteq$ trans $\rightarrow$ SITE
$\land$ sitetransstatus $\subseteq$ trans $\rightarrow$ (SITE $\rightarrow$ SITETRANSSTATUS)
$\land$ replica $\subseteq$ SITE $\rightarrow$ (OBJECT $\rightarrow$ VALUE)
$\land$ freeobject $\subseteq$ SITE $\leftrightarrow$ OBJECT

INITIALISATION trans $:$= $\emptyset$
$\parallel$ transstatus $:$= $\emptyset$
$\parallel$ activetrans $:$= $\emptyset$
$\parallel$ coordinator $:$= $\emptyset$
$\parallel$ sitetransstatus $:$= $\emptyset$
$\parallel$ transeffect $:$= $\{\}$
$\parallel$ transobject $:$= $\{\}$
$\parallel$ freeobject $:$= SITE $\times$ OBJECT
$\parallel$ ANY data WHERE data $\in$ OBJECT $\rightarrow$ VALUE
THEN replica $:$= SITE $\times$ $\{data\}$ END

Figure 3.4: Initial part of Refinement

- Variable activetrans keeps a record of transactions running at various sites, i.e., it is in the state precommit or pending. A mapping ($s \mapsto t$) $\in$ activetrans indicates that site $s$ is running transaction $t$. The variable freeobject keeps a record of objects at various sites which are free, i.e., those objects which are not locked by any active transaction.

- The variable sitetransstatus maintains the status of all started transactions at various sites. A mapping of form ($t \mapsto (s \mapsto \text{commit})$) $\in$ sitetransstatus indicates that $t$ has committed at site $s$.

- The new events such as IssueWriteTran, BeginSubTran, SiteAbortTx, SiteCommitTx, ExecAbortDecision and ExecCommitDecision are introduced in operations.

3.4.3 Events of Update Transaction

In this refinement, various events of an update transaction are triggered within the framework of two phase commit protocol. An informal logical ordering of the occurrence of various events of the refinement for an update transaction is outlined in Fig. 3.5.

- The events StartTran(tt) and IssueWriteTran(tt) occur at the coordinating site of tt. Once a transaction is started at the coordinator, the coordinator sends update messages to the participating sites. The update messages are delivered
to all sites, including the coordinator, in an arbitrary order. Upon delivery of an update message at the coordinator site, the coordinator site issues a transaction at the coordinator. The event IssueWriteTran(tt) models the issuance of an update transaction at the coordinator.

Upon delivery of an update message, a participating site starts a sub-transaction at that site. The event BeginSubTran(tt,ss) models starting a sub-transaction of tt at site ss. The site may independently decide to either commit or abort tt. The events SiteCommitTx(tt,ss) and SiteAbortTx(tt,ss) are events of the commitment or abortion of an update transaction tt at the participating site ss. Participating sites communicate their decision to the coordinator of tt by sending either a Vote-Commit or Vote-Abort message.

Upon receipt of Vote-Commit/Abort messages, the coordinator site triggers either the event AbortWriteTran(tt) or CommitWriteTran(tt). The event CommitWriteTran(tt) occurs when the coordinator site receives Vote-Commit message from all participating sites, whereas, the delivery of just one Vote-Abort message from any participating site triggers the AbortWriteTran(tt) event. The coordinating site communicates its decision by broadcasting a commit/abort decision message through a global commit or global abort message. Upon receipt of a global commit/abort decision message from the coordinator, a participating site ss decides to abort or commit tt by triggering either ExeAbortDecision(ss,tt) or ExeCommitDecision(ss,tt) event.

### 3.4.4 Starting and Issuing a Transaction

Submission of a transaction tt is modelled by the event StartTran(tt). The event IssueWriteTran(tt) models the issuing of an update transaction at the coordinator from a set of started transactions, which are not in conflict with other issued transactions.
at the coordinator site. The guard of \( \text{IssueWriteTran}(tt) \) ensures that a transaction \( tt \) is issued by the coordinator when all active transactions \( tz \), running at the coordinator site of \( tt \), are not in conflict with \( tt \), i.e.,

\[
tz \in \text{trans} \land (\text{coordinator}(tt) \mapsto tz) \in \text{activetrans} \\
\Rightarrow \text{transobject}(tt) \cap \text{transobject}(tz) = \emptyset
\]

The Event-B specification for the events \( \text{StartTran}(tt) \) and \( \text{IssueWriteTran}(tt) \) of the refinement are given in Fig 3.6.

### 3.4.5 Commitment and Abortion of Update Transactions

Refined specifications for the commit and abort events of update transaction \( tt \) are given in Fig. 3.7 and Fig 3.8. An update transaction \( tt \) globally commits only if all participating sites are ready to commit it, i.e., it has status \( \text{pre-committed} \) at all sites. As a

\[
\text{StartTran}(tt) \equiv
\]

\[
\text{ANY} \quad ss, \text{updates}, \text{objects} \\
\text{WHERE} \quad ss \in \text{SITE} \\
\quad tt \notin \text{trans} \\
\quad \text{updates} \in \text{UPDATE} \\
\quad \text{objects} \in \text{P}_1(\text{OBJECT}) \\
\quad \text{ValidUpdate (updates,objects)} \\
\text{THEN} \\
\quad \text{trans} := \text{trans} \cup \{tt\} \\
\quad \text{transstatus}(tt) := \text{PENDING} \\
\quad \text{transobject}(tt) := \text{objects} \\
\quad \text{transeffect}(tt) := \text{updates} \\
\quad \text{coordinator}(tt) := ss \\
\quad \text{sitetransstatus}(tt)(\text{coordinator}(tt)) := \{\text{coordinator}(tt) \mapsto \text{pending}\} \\
\text{END};
\]

\[
\text{IssueWriteTran}(tt) \equiv
\]

\[
\text{WHEN} \quad tt \in \text{trans} \\
\quad (\text{coordinator}(tt) \mapsto tt) \notin \text{activetrans} \\
\quad \text{sitetransstatus}(tt)(\text{coordinator}(tt)) = \text{pending} \\
\quad \text{ran}((\text{transeffect}(tt))) \neq \{\emptyset\} \\
\quad \text{transobject}(tt) \subseteq \text{freeobject}([\text{coordinator}(tt)]) \\
\quad \forall tz.(tz \in \text{trans} \land (\text{coordinator}(tt) \mapsto tz) \in \text{activetrans} \\
\quad \Rightarrow \text{transobject}(tt) \cap \text{transobject}(tz) = \emptyset) \\
\text{THEN} \\
\quad \text{activetrans} := \text{activetrans} \cup \{\text{coordinator}(tt) \mapsto tt\} \\
\quad \text{sitetransstatus}(tt)(\text{coordinator}(tt)) := \text{precommit} \\
\quad \text{freeobject} := \text{freeobject} - \{\text{coordinator}(tt)\} \times \text{transobject}(tt) \\
\text{END};
\]

\textbf{Figure 3.6:} Refinement : Coordinator Site Events-I
consequence of the occurrence of the commit event at the coordinator, the replica maintained at the coordinator site is updated with the transaction effects, data objects held for transaction $tt$ are declared free and the status of the transaction at the coordinator site is set to commit. The $\text{AbortWriteTran}(tt)$ event given in Fig. 3.8 ensures that an update will abort if it has aborted at some participating site.

$\text{CommitWriteTran}(tt) \equiv$

\begin{align*}
\text{ANY} \quad & \quad pdb \\
\text{WHERE} \quad & \quad tt \in \text{trans} \\
\wedge & \quad pdb = \text{transobject}(tt) \triangleleft \text{replica}(\text{coordinator}(tt)) \\
\wedge & \quad \text{ran}(\text{transeffect}(tt)) \neq \{\emptyset\} \\
\wedge & \quad (\text{coordinator}(tt) \mapsto tt) \in \text{activetrans} \\
\wedge & \quad \text{transstatus}(tt) = \text{PENDING} \\
\wedge & \quad \forall s.( s \in \text{SITE} \Rightarrow \text{sitetransstatus}(tt)(s) = \text{precommit} ) \\
\wedge & \quad \forall (s,o). (s \in \text{SITE} \wedge o \in \text{OBJECT} \wedge o \in \text{transobject}(tt) \Rightarrow (s \mapsto o) \notin \text{freeobject}) \\
\wedge & \quad \forall s.(s \in \text{SITE} \Rightarrow (s \mapsto tt) \in \text{activetrans}) \\
\text{THEN} \\
& \quad \text{transstatus}(tt) := \text{COMMIT} \\
& \quad \| \quad \text{activetrans} := \text{activetrans} \setminus \{\text{coordinator}(tt) \mapsto tt\} \\
& \quad \| \quad \text{sitetransstatus}(tt)(\text{coordinator}(tt)) := \text{commit} \\
& \quad \| \quad \text{freeobject} := \text{freeobject} \cup \{\text{coordinator}(tt)\} \times \text{transobject}(tt) \\
& \quad \| \quad \text{replica}(\text{coordinator}(tt)) := \text{replica}(\text{coordinator}(tt)) \triangleleft \text{transeffect}(tt)(pdb) \\
\text{END};
\end{align*}

\text{Figure 3.7: Refinement : Coordinator Site Events - II}

Further Refinement of Commit Event

The event $\text{CommitWriteTran}(tt)$ can be further refined under the following observations.

\begin{itemize}
  \item $o \in \text{transobject}(t) \wedge \text{sitetransstatus}(t)(s) = \text{precommit} \Rightarrow (s \mapsto o) \notin \text{freeobject}$
  \item $\text{sitetransstatus}(t)(s) = \text{precommit} \Rightarrow (s \mapsto t) \in \text{activetrans}$
  \item $o \in \text{transobject}(t) \wedge (s \mapsto t) \in \text{activetrans} \Rightarrow (s \mapsto o) \notin \text{freeobject}$
\end{itemize}

These observations can be included as invariants in a further refinement allowing the guards of the $\text{CommitWriteTran}(tt)$ event to be simplified. The simplified guards for the refined $\text{CommitWriteTran}(tt)$ are given below.

\begin{align*}
[ \\
\quad tt \in \text{trans} \\
\wedge & \quad \text{ran}(\text{transeffect}(tt)) \neq \{\emptyset\} \\
\wedge & \quad \text{transstatus}(tt) = \text{PENDING} \\
\wedge & \quad \forall s.( s \in \text{SITE} \wedge \text{sitetransstatus}(tt)(s) = \text{precommit} ) ]
\end{align*}
3.4.6 Read-Only Transactions

The specifications of executing a read-only transaction is given in Fig. 3.8. A pending read-only transaction \( tt \) returns the value of objects in the set \( \text{transobject}(tt) \) from the replica at its coordinator. The necessary conditions for occurrence of this event are as follows.

\[
\begin{align*}
\text{transstatus}(tt) &= \text{PENDING} \\
\land \quad \text{ran}(\text{transeffect}(tt)) &= \{\emptyset\} \\
\land \quad \text{transobject}(tt) &\subseteq \text{freeobject}[\{ss\}]
\end{align*}
\]

As a consequence of the occurrence of this event, transaction \( tt \) reads the objects from the replica at site \( ss \) as,

\[
\text{val} := \text{transobject}(tt) \triangleleft \text{replica}(ss)
\]

It may be noted that in the abstract model given in Fig 3.3, a read-only transaction reads the objects from abstract database as,

\[
\text{val} := \text{transobject}(tt) \triangleleft \text{database}
\]

\[
\text{AbortWriteTran}(tt) \equiv
\begin{align*}
\text{WHEN} & \quad t\in \text{trans} \\
& \quad \text{ran}(\text{transeffect}(tt)) \neq \{\emptyset\} \\
& \quad (\text{coordinator}(tt) \mapsto tt) \in \text{activetrans} \\
& \quad \text{transstatus}(tt) = \text{PENDING} \\
& \quad \exists s. (s \in \text{SITE} \land \text{sitetransstatus}(tt)(s) = \text{abort})
\end{align*}
\]

\[
\text{THEN} \\
\quad \text{activetrans} := \text{activetrans} - \{\text{coordinator}(tt) \mapsto tt\} \\
\quad \text{sitetransstatus}(tt)(\text{coordinator}(tt)) := \text{abort} \\
\quad \text{freeobject} := \text{freeobject} \cup \{\text{coordinator}(tt)\} \times \text{transobject}(tt)
\]

\[
\text{END;}
\]

\[
\text{val} \leftarrow \text{ReadTran}(tt, ss) \equiv
\begin{align*}
\text{WHEN} & \quad t\in \text{trans} \\
& \quad \text{transstatus}(tt) = \text{PENDING} \\
& \quad \text{transobject}(tt) \subseteq \text{freeobject}[\{ss\}] \\
& \quad ss = \text{coordinator}(tt) \\
& \quad \text{ran}(\text{transeffect}(tt)) = \{\emptyset\}
\end{align*}
\]

\[
\text{THEN} \\
\quad \text{val} := \text{transobject}(tt) \triangleleft \text{replica}(ss) \\
\quad \text{sitetransstatus}(tt)(ss) := \text{commit} \\
\quad \text{transstatus}(tt) := \text{COMMIT}
\]

\[
\text{END;}
\]

\textbf{Figure 3.8: Refinement : Coordinator Site Events - III}
In refinement checking, we need the following invariant to show that the refinement is valid.

$$(ss \mapsto oo) \in freeobject \Rightarrow database(oo) = replica(ss)(oo)$$

This is explained further in section 3.5.

### 3.4.7 Starting a Sub-Transaction

In our model we assume full replication, i.e., each data object is replicated at all sites. A global update transaction can be submitted to any one site, called the coordinator site for that transaction. However, it accesses and updates the data at other sites, called participating sites. Upon submission of an update transaction, the coordinating site of the transaction broadcasts all operations of the transaction to the participating sites by an *update* message. Upon receiving the update message at a participating site, the transaction manager at that site creates a sub-transaction. The activity of a global update transaction at a given site is referred as a sub-transaction.

The *BeginSubTran(tt,ss)* event models starting a sub-transaction of *tt* at participating site *ss*. The specification of this event is given in Fig. 3.9. The following guard of *BeginSubTran(tt)* ensures that a sub-transaction of *tt* is started at site *ss* when no active transaction *tz* running at *ss* is in conflict with *tt*:

$$(ss \mapsto tz) \in activetrans \Rightarrow transobject(tt) \cap transobject(tz) = \emptyset$$

A sub-transaction at a participating site is started when it has *precommitted* at the coordinator site of *tt*. Also, a sub-transaction at a participating site may be started if the coordinator has already decided to globally abort it. The coordinator may decide to globally abort an update transaction if it has received any one *vote-abort* message from any participating site. In such cases, the rest of the sites go ahead with starting a sub-transaction when they deliver an update message and the abort of sub-transaction at that site will take place when it delivers *global-abort* message from the coordinator. Therefore, we add following as a guard to the event *BeginSubTran*.

**sitetransstatus(tt)(coordinator(tt)) \in \{precommit, abort\}**

The guard $ss \notin \text{dom(sitetransstatus(tt))}$ prevents starting a sub-transaction again at the site *ss*. As a consequence of the occurrence of this event, transaction *tt* becomes *active* at site *ss* and the *sitetransstatus* of *tt* at *ss* is set to pending.
3.4.8 Pre-Commitment and Abortion of Sub-transaction

A participating site $ss$ can independently decide to either pre-commit or abort a sub-transaction. The events $\text{SiteCommitTx}(tt,ss)$ and $\text{SiteAbortTx}(tt,ss)$, given in Fig. 3.10, model pre-committing or aborting a sub-transaction of $tt$ at $ss$. Pre-committing a transaction at a participating site is considered as a commit guarantee given to the coordinator by a participating site. In the case of abort, a site sets all objects of transaction $tt$ free and a related sub-transaction is removed from list of active transactions at that site.
3.4.9 Completing the Global Commit/Abort

We have already seen how the refined \texttt{CommitWriteTran(tt)} and \texttt{AbortWriteTran(tt)} events model the global commit or abort decision. The events \texttt{ExeCommitDecision(tt,ss)} and \texttt{ExeAbortDecision(tt,ss)} given in Fig. 3.11 model the commit and abort of \textit{tt} at participating site \textit{ss} once a global abort or commit decision has been taken by the coordinating site. In the case of global commit, each site updates its replica separately.

\texttt{ExeAbortDecision(ss,tt) \equiv}
\begin{verbatim}
WHEN \hspace{1cm} tt \in \text{trans} \\
\hspace{1cm} (ss \rightarrow tt) \in \text{activetrans} \\
\hspace{1cm} ss \neq \text{coordinator(tt)} \\
\hspace{1cm} \text{ran(transeffect(tt))} \neq \{\emptyset\} \\
\hspace{1cm} \text{sitetransstatus(tt)(coordinator(tt))} = \text{abort} \\
\hspace{1cm} \text{sitetransstatus(tt)(ss)} = \text{precommit}
\hspace{1cm} \text{THEN}
\hspace{1cm} \text{sitetransstatus(tt)(ss)} := \text{abort}
\hspace{1cm} \text{activetrans := activetrans -\{ss \rightarrow tt\}}
\hspace{1cm} \text{freeobject := freeobject \cup \{ss\} \times transeffect(tt)}
\hspace{1cm} \text{END;}
\end{verbatim}

\texttt{ExeCommitDecision(ss,tt) \equiv}
\begin{verbatim}
ANY \hspace{1cm} pdb \\
WHERE \hspace{1cm} tt \in \text{trans} \\
\hspace{1cm} (ss \rightarrow tt) \in \text{activetrans} \\
\hspace{1cm} ss \neq \text{coordinator(tt)} \\
\hspace{1cm} \text{ran(transeffect(tt))} \neq \{\emptyset\} \\
\hspace{1cm} pdb = \text{transeffect(tt)} \leftarrow \text{replica(ss)} \\
\hspace{1cm} \text{sitetransstatus(tt)(coordinator(tt))} = \text{commit} \\
\hspace{1cm} \text{sitetransstatus(tt)(ss)} = \text{precommit}
\hspace{1cm} \text{THEN}
\hspace{1cm} \text{sitetransstatus(tt)(ss)} := \text{commit}
\hspace{1cm} \text{activetrans := activetrans -\{ss \rightarrow tt\}}
\hspace{1cm} \text{freeobject := freeobject \cup \{ss\} \times transeffect(tt)}
\hspace{1cm} \text{replica(ss) := replica(ss) \leftarrow transeffect(tt)(pdb)}
\hspace{1cm} \text{END;}
\end{verbatim}

\textsc{Figure 3.11:} Refinement : Participating Site Events -III

3.5 Gluing Invariants

The \textit{one-copy equivalence} consistency criterion requires us to prove that our refinement (replicated database) is a valid refinement of the abstract transaction model (abstract central database). We have replaced the abstract variable \textit{database} in the abstract model by the variable \textit{replica} in the refinement.
Initially, the only proof obligation that could not be proved using the prover involves the relationship between database and replica. This proof obligation associated with the event ReadTran is given below.

\[
\begin{align*}
\text{ReadTran}(PO1) & \quad \text{transstatus}(tt) = \text{PENDING} \land \\
& \quad \text{ran}(\text{transeffect}(tt)) = \{\phi\} \land \\
& \quad \text{oo} \in \text{transobject}(tt) \land \\
& \quad \text{coordinator}(tt) \mapsto \text{oo} \in \text{freeobject} \land \\
& \quad \Rightarrow \\
& \quad \text{replica}(\text{coordinator}(tt))(\text{oo}) = \text{database}(\text{oo})
\end{align*}
\]

This proof obligation states that for a given read-only transaction whose transaction objects are free at its coordinator site then the value of those objects at the replica at the coordinator site is same as that in the abstract database. This observation is generalized in to order to construct a gluing invariant, such that, if any data object is in the free list at any site then it represents the value of that data object in the abstract database. Therefore, we added the gluing invariant given as Inv-1 in Fig. 3.12.

The name of various events of our model and their corresponding event code are given in Table 3.1.

<table>
<thead>
<tr>
<th>Invariants</th>
<th>Required By</th>
</tr>
</thead>
<tbody>
<tr>
<td>/<em>Inv-1</em>/</td>
<td>(ss ↦ oo) ∈ freeobject =&gt; database(oo) = replica(ss)(oo)</td>
</tr>
</tbody>
</table>

Figure 3.12: Gluing Invariants-I

The invariant Inv-1 means that a free object oo at site ss represents the value of oo in the abstract database. We have omitted the quantification over all identifiers (ss, oo, tt etc.) to avoid clutter. When invariant Inv-1 is added to the refined machine, the B tool generates further proof obligations associated with several other events. One of the important proof obligations associated with the AbortWriteTran event is outlined below.
AbortWriteTran(PO2)

\[
\begin{align*}
\text{transstatus}(tt) &= \text{PENDING} \land \\
\text{ran}(\text{transeffect}(tt)) &\neq \{\phi\} \\
\text{coordinator}(tt) &\mapsto tt \in \text{activetrans} \land \\
\text{oo} &\in \text{transobject}(tt) \land \\
\Rightarrow &\\
\text{replica}(\text{coordinator}(tt))(oo) &= \text{database}(oo)
\end{align*}
\]

This proof obligation states that if a pending transaction is active at its coordinator site then all objects of the transaction in the abstract database have same value in the replica at the coordinator. Thus, in order to discharge this proof obligation we construct an invariant given as Inv-2 in the Fig. 3.13.

In order to discharge the proof obligations generated due to the addition of Inv-1 we add a set of invariants given in Fig. 3.13. A brief description of these invariants is given in the following:

- Inv-2: If a transaction \( t \) is active at its coordinator then all transaction objects \( o \in \text{transobject}(t) \) in the abstract database have the same value in the replica at the coordinator.

- Inv-3: If two conflicting transactions \( t_1 \) and \( t_2 \) are active at a site \( s \), they must represent the same transaction, i.e., \( t_1 = t_2 \). This also implies that two different conflicting transactions cannot be active at the same time at the same site \( s \).

<table>
<thead>
<tr>
<th>Invariants</th>
<th>Required By</th>
</tr>
</thead>
<tbody>
<tr>
<td>/<em>Inv-2</em>/</td>
<td>AWT,CWT,EAD,ECD</td>
</tr>
<tr>
<td>/<em>Inv-3</em>/</td>
<td>ST,IWT,BST</td>
</tr>
</tbody>
</table>

Figure 3.13: Gluing Invariants -II

After addition of Inv-2, a new proof obligation associated with the events CommitWriteTran and SiteCommitTx is generated. This proof obligation requires us to prove that if a committed update transaction is still active at a participating site then the value of all updateable objects in the abstract database is equal to the values given by transeffect.
function of that transaction. A simplified form of the proof obligation is outlined below.

\[
\text{ExeCommitDecision}(PO3) \begin{cases} 
\text{sitetransstatus}(tt)(\text{coordinator}(tt)) = \text{commit} \land \\
ss \neq \text{coordinator}(tt) \land \\
\text{ran}(\text{transeffect}(tt)) \neq \{\phi\} \land \\
\text{oo} \in \text{transobject}(tt) \land \\
ss \mapsto tt \in \text{activetrans} \land \\
\text{oo} \in \text{dom}(\text{transeffect}(tt)(\text{transobject}(tt) \triangleleft \text{replica}(ss))) \\
\Rightarrow \\
\text{transeffect}(tt)(\text{transobject}(tt) \triangleleft \text{replica}(ss))(\text{oo}) = \text{database}(\text{oo}) 
\end{cases}
\]

In order to discharge the proof obligation \(PO3\), we construct an invariant given as \(\text{Inv-4}\). A brief description of the invariant \(\text{Inv-4}\) is outlined below.

- \(\text{Inv-4}\) : For a committed transaction \(t\) which is active at one of the site \(s\), the new values of objects defined by \(\text{transeffect}(t)\) reflects the value of those objects in the abstract database.

\[
\begin{array}{l}
\text{Invariants} \\
\text{Required By}
\end{array}
\]

\[
\begin{array}{l}
/*\text{Inv-4}*\]
\text{transstatus}(tt) = \text{COMMIT} \\
\land (s \mapsto t) \in \text{activetrans} \\
\land \text{oo} \in \text{dom}(\text{transeffect}(tt)(\text{transobject}(tt) \triangleleft \text{replica}(ss))) \\
\Rightarrow \text{database}(\text{oo}) = \text{transeffect}(tt)(\text{transobject}(tt) \triangleleft \text{replica}(ss))(\text{oo})
\end{array}
\]

\text{Figure 3.14: Gluing Invariants -III}

Further, due to the addition of the invariant \(\text{Inv-4}\) a new proof obligation is generated. The simplified form of this proof obligation is outlined below.

\[
\text{CommitWriteTrran}(PO4) \begin{cases} 
\text{transstatus}(tt) = \text{COMMIT} \land \\
ss \neq \text{coordinator}(tt) \land \\
\text{ran}(\text{transeffect}(tt)) \neq \{\phi\} \land \\
\text{oo} \in \text{transobject}(tt) \land \\
ss \mapsto tt \in \text{activetrans} \land \\
\text{oo} \notin \text{dom}(\text{transeffect}(tt)(\text{transobject}(tt) \triangleleft \text{replica}(ss))) \\
\Rightarrow \\
\text{replica}(ss)(\text{oo}) = \text{database}(\text{oo})
\end{cases}
\]
This proof obligation associated with the event \textit{CommitWriteTran} requires us to prove that for a committed update transaction, which is still active at a participating site, the value of all non-updateable objects of that transaction at that site is equal to that in the abstract database. In order to discharge the proof obligation \textit{PO4}, we construct and add the \textit{Inv-5} to the refined model.

Following a similar approach, in order to preserve the invariants in Fig. 3.14, we have to prove another set of invariants given in Fig. 3.15. The brief description of invariants in Fig. 3.15 are given below.

- \textit{Inv-5}: For a committed transaction \( t \) which is still active at a participating site \( s \), the value of any read-only objects of \( t \) is the same in replica(s) as in the database.

- \textit{Inv-6,7}: If a transaction \( t \) commits or aborts globally, it must have either committed or aborted locally at its coordinator.

<table>
<thead>
<tr>
<th>Invariants</th>
<th>Required By</th>
</tr>
</thead>
<tbody>
<tr>
<td>/<em>Inv-5</em>/</td>
<td></td>
</tr>
<tr>
<td>\textit{transstatus}(t) = \textit{COMMIT} &amp; o \in \textit{transobject}(t) &amp; (s \mapsto t) \in \textit{activetrans} &amp; o \notin \textit{dom}(/\textit{transeffect}(tt)(\textit{transobject}(tt) \cup \textit{replica}(s))) \Rightarrow \textit{database}(o) = \textit{replica}(s)(o)</td>
<td>CWT, AWT, BST, ECD, SAT, SCT</td>
</tr>
<tr>
<td>/<em>Inv-6</em>/</td>
<td></td>
</tr>
<tr>
<td>\textit{transstatus}(t) = \textit{ABORT} \Rightarrow \textit{sitetransstatus}(t)(\text{coordinator}(t)) = \textit{abort}</td>
<td>AWT, EAD, ECD, ST</td>
</tr>
<tr>
<td>/<em>Inv-7</em>/</td>
<td></td>
</tr>
<tr>
<td>\textit{transstatus}(t) = \textit{COMMIT} \Rightarrow \textit{sitetransstatus}(t)(\text{coordinator}(t)) = \textit{commit}</td>
<td>CWT, AWT, EAD, ECD, ST</td>
</tr>
</tbody>
</table>

\textit{Figure 3.15: Gluing Invariants -IV}

Another important proof obligation associated with \textit{ExeCommitDesicion} and \textit{ExeAbortDecision} generated due to the addition of \textit{Inv-5} requires us to prove that if a transaction that is either \textit{pending} or \textit{aborted} state and still active at a site \( ss \), then all transaction objects \( oo \) in the abstract database have the same value in the replica at that site. A simplified form of this proof obligation is given below.

\textit{ExeCommitDecision(PO5)}

\[
\begin{align*}
\textit{sitetransstatus}(tt)(\text{coordinator}(tt)) &= \text{pending} \land \\
ss \mapsto oo &\notin \textit{freeobject} \land \\
\text{run}(\textit{transeffect}(tt)) &\neq \text{\{\phi\}} \land \\
 oo &\in \textit{transobject}(tt) \land \\
ss \mapsto tt &\in \textit{activetrans} \land \\
\Rightarrow &\text{\,} \\
\textit{replica}(ss)(oo) &= \textit{database}(oo)
\end{align*}
\]
In order to discharge the proof obligations PO5 and PO6 we construct and add Inv-8 to our model and discharge the proof obligations.

Similarly, due to the addition of Inv-8 new proof obligations associated with the event IssueWriteTran are generated. A simplified form of these proof obligations is outlined below.

**IssueWriteTran (PO7)**

\[
\begin{align*}
\text{transstatus}(tt) &= \text{COMMIT} \land \\
ss &= \text{coordinator}(tt) \land \\
ss \mapsto oo &\in \text{freeobject} \land \\
\text{ran} (\text{transeffect}(tt)) &\neq \{\phi\} \land \\
oo &\in \text{transobject}(tt) \land \\
\Rightarrow \\
\text{replica}(ss)(oo) &= \text{database}(oo)
\end{align*}
\]

PO7 states that if an update transaction that has committed and all transaction objects at the coordinator are in the free object list then it is not active at the coordinator site.

**IssueWriteTran (PO8)**

\[
\begin{align*}
\text{transstatus}(tt) &= \text{ABORT} \land \\
ss &= \text{coordinator}(tt) \land \\
ss \mapsto oo &\in \text{freeobject} \land \\
\text{ran} (\text{transeffect}(tt)) &\neq \{\phi\} \land \\
oo &\in \text{transobject}(tt) \land \\
\Rightarrow \\
\text{replica}(ss)(tt) &\notin \text{activetrans}
\end{align*}
\]

Similarly, PO8 states that if an update transaction that has aborted and all transaction objects at the coordinator are in the free object list then it is not active at the coordinator.

In order to discharge proof obligations PO7 and PO8 we add invariant Inv-9 to the refined model. This invariant states that an update transaction not pending at the
coordinator site, is also not active at the coordinator site. Recall that a transaction which is not pending implies that either it has committed or aborted. Finally the B tool generates more proof obligations to preserve Gluing Invariant-IV which in turn requires Gluing Invariants-V in Fig. 3.16. The brief description of Gluing Invariants-V is given below.

- **Inv-8**: For a transaction $t$ which has not globally committed and is still active at some site $s$, then for all objects $o \in \text{transobject}(t)$, the value of object $o$ at $\text{replica}(s)$ is the same as its value in abstract database. Since this refers to the situations where a transaction is not committed, it also involves the situations where the transaction global status is either PENDING or ABORT.

- **Inv-9**: An update transaction whose global status is not PENDING must not be active at its coordinator site. This refers to situations where the global status of an update transaction is either COMMIT or ABORT.

We observe that at every stage new proof obligations are generated by the B tool due to the addition of new invariants. In this process, at every stage, we also discover further invariants to be expressed in our model. After five iterations of invariant strengthening, we arrive at an invariant that is sufficient to discharge all proof obligations. By discharging the proof obligations we ensure that our refinement is a valid refinement of the abstract specification.

### 3.6 Processing Transactions over a Reliable Broadcast

As outlined in the previous sections, our abstract model of a transaction maintains a notion of the central database. In the refinement we introduce the notion of a replicated database by replacing the abstract variable database by a concrete variable replica.
In this section, a further refinement of this model, given as replica4, is outlined which explicitly models messaging among the sites illustrating the integration of the transaction model with a reliable broadcast.

### 3.6.1 Introducing Messaging in the Transactional Model

In this section, we outline how various messages of the protocol are represented in the refinement replica4. The new variables update, voteabort, votecommit, globalabort and globalcommit are introduced in this refinement to represent the respective messages. These variables are typed as follows:

- \( \text{update} \subseteq \text{MESSAGE} \land \text{update} \in \text{dom}(\text{sender}) \)
- \( \text{voteabort} \subseteq \text{MESSAGE} \land \text{voteabort} \in \text{dom}(\text{sender}) \)
- \( \text{votecommit} \subseteq \text{MESSAGE} \land \text{votecommit} \in \text{dom}(\text{sender}) \)
- \( \text{globalabort} \subseteq \text{MESSAGE} \land \text{globalabort} \in \text{dom}(\text{sender}) \)
- \( \text{globalcommit} \subseteq \text{MESSAGE} \land \text{globalcommit} \in \text{dom}(\text{sender}) \)

A message \( m \in \text{update} \) indicates that \( m \) is an update message. Similarly, a message in the set \( \text{voteabort}, \text{votecommit}, \text{globalabort} \) or \( \text{globalcommit} \), respectively, indicates that it is either a vote abort/commit or global abort/commit message. We also introduce following new variables to relate a message to the transaction as follows:

- \( \text{tranupdate} \in \text{update} \rightarrow \text{trans} \)
- \( \text{tranvoteabort} \in \text{voteabort} \rightarrow \text{trans} \)
- \( \text{tranvotecommit} \in \text{votecommit} \rightarrow \text{trans} \)
- \( \text{tranglobalabort} \in \text{globalabort} \rightarrow \text{trans} \)
- \( \text{tranglobalcommit} \in \text{globalcommit} \rightarrow \text{trans} \)

A mapping of the form \( (m \rightarrow t) \in \text{tranupdate} \) indicates that a message \( m \) is an update message for an update transaction \( t \). A \text{tranupdate} is a total injective function which indicates that there is only one update message for each update transaction and vice-versa. \text{tranvoteabort} is defined as a total function which indicates that each message \( m \in \text{voteabort} \) is related to exactly one update transaction. However, for an update transaction there will be several \text{votecommit} or \text{voteabort} messages. The variables \text{tranglobalabort} and \text{tranglobalcommit} are defined as a total injective function indicating that a message is related to exactly one transaction and each update transaction is related to exactly one \text{globalabort} or \text{globalcommit} message. The reason for modelling variables \text{tranupdate}, \text{tranglobalabort} and \text{tranglobalcommit} as total injective functions is that the respective messages are sent by the coordinating site only once for a given transaction.

The variables \text{sender}, \text{deliver} and \text{completed} are also introduced in this refinement to model sending a message, the delivery of a message and the completion of an update.
transaction as follows:

\begin{align*}
\text{sender} &\in MESSAGE \rightarrow SITE \\
\text{deliver} &\in SITE \rightarrow MESSAGE \\
\text{completed} &\in \text{trans} \rightarrow SITE
\end{align*}

A mapping of the form \((mm \mapsto ss) \in \text{sender}\) indicates that the site \(ss\) is the sender of message \(mm\). Similarly, a mapping \((ss \mapsto mm) \in \text{deliver}\) indicates that a site \(ss\) has delivered \(mm\). The completion of a transaction is modelled by a variable \(\text{completed}\), where a mapping \((tt \mapsto ss) \in \text{completed}\) indicates that a transaction \(tt\) completed its execution at site \(ss\).

### 3.6.2 The Events of Message Send and Delivery

In this refinement we introduce two new events given as \(\text{SendUpdate}\) and \(\text{Deliver}\). The event \(\text{SendUpdate}\) models the broadcast of an update message for an update transaction. The event \(\text{Deliver}\) models the delivery of a message at a site. The specifications of these events are outlined in the Fig. 3.17.

\begin{itemize}
  \item \textbf{SendUpdate} \((ss \in \text{SITE}, mm \in \text{MESSAGE}, tt \in \text{TRANSACTION}) \equiv\)
  \item \textbf{Deliver} \((ss \in \text{SITE}, mm \in \text{MESSAGE}) \equiv\)
\end{itemize}

\begin{align*}
\text{SendUpdate} &\in \text{SITE} \rightarrow \text{MESSAGE} \rightarrow \text{TRANSACTION} \\
\text{Deliver} &\in \text{SITE} \rightarrow \text{MESSAGE}
\end{align*}

As shown in the specifications, the coordinator site \(ss\) of an update transaction \(tt\) broadcasts an update message \(mm\) after the submission of the transaction \(tt\) at the site \(ss\). The guard \(tt \in \text{trans}\) indicates that a transaction \(tt\) has started. Similarly, the guard \(tt \notin \text{ran} (\text{tranupdate})\) indicates that an update message corresponding to the transaction \(tt\) has not been sent. The variable \(update\) and \(\text{tranupdate}\) are updated accordingly to
indicate that $mm$ is an update message and that update message $mm$ is also related to the transaction $tt$. The event Deliver models the delivery of a message $mm$ to the site $ss$. The guard of this event ensures that a message is delivered to a site only once. Since delivery of message does not have any other conditions specified in the guard, as required for the delivery of ordered broadcasts, the Deliver event models delivery of a message using a reliable broadcast.

**IssueWriteTran**($tt \in TRANSACTION$) ≡

\[
\begin{align*}
\text{ANY} & \quad mm \\
\text{WHERE} & \quad mm \in \text{update} \\
& \quad tt \in \text{trans} \\
& \quad (mm \rightarrow tt) \in \text{tranupdate} \\
& \quad (\text{coordinator}(tt) \rightarrow mm) \in \text{deliver} \\
& \quad (\text{coordinator}(tt) \rightarrow tt) \notin \text{activetrans} \\
& \quad \text{sitetransstatus}(tt)(\text{coordinator}(tt)) = \text{pending} \\
& \quad \text{ran}(\text{tranffect}(tt)) \neq \emptyset \\
& \quad \text{transobject}(tt) \subset \text{freeobject}[\{\text{coordinator}(tt)\}] \\
& \quad \forall tz. (tz \in \text{trans} \land (\text{coordinator}(tt) \rightarrow tz) \in \text{activetrans}) \\
& \quad \Rightarrow \text{transobject}(tt) \cap \text{transobject}(tz) = \emptyset \\
\text{THEN} & \quad \text{activetrans} := \text{activetrans} \cup \{\text{coordinator}(tt) \rightarrow tt\} \\
& \quad \Rightarrow \text{sitetransstatus}(tt)(\text{coordinator}(tt)) := \text{precommit} \\
& \quad \text{freeobject} := \text{freeobject} - \{\text{coordinator}(tt)\} \times \text{transobject}(tt) \\
\text{END};
\end{align*}
\]

**BeginSubTran**($tt \in TRANSACTION, ss \in SITE$) ≡

\[
\begin{align*}
\text{ANY} & \quad mm \\
\text{WHERE} & \quad mm \in \text{update} \\
& \quad tt \in \text{trans} \\
& \quad (mm \rightarrow tt) \in \text{tranupdate} \\
& \quad (ss \rightarrow mm) \in \text{deliver} \\
& \quad (ss \rightarrow tt) \notin \text{activetrans} \\
& \quad ss \notin \text{dom}(\text{sitetransstatus}(tt)) \\
& \quad ss \neq \text{coordinator}(tt) \\
& \quad \text{ran}(\text{tranffect}(tt)) \neq \emptyset \\
& \quad \text{transobject}(tt) \subset \text{freeobject}[\{ss\}] \\
& \quad \forall tz. (tz \in \text{trans} \land (ss \rightarrow tz) \in \text{activetrans}) \\
& \quad \Rightarrow \text{transobject}(tt) \cap \text{transobject}(tz) = \emptyset \\
\text{THEN} & \quad \text{activetrans} := \text{activetrans} \cup \{ss \rightarrow tt\} \\
& \quad \Rightarrow \text{sitetransstatus}(tt)(ss) := \text{pending} \\
& \quad \text{freeobject} := \text{freeobject} - \{ss\} \times \text{transobject}(tt) \\
\text{END};
\end{align*}
\]

**Figure 3.18:** Events IssueWriteTran and BeginSubTran: A Reliable Broadcast
3.6.3 Starting a Sub-transaction

The specifications of IssueWriteTran and BeginSubTran events in this refinement are given in the Fig. 3.18. The event IssueWriteTran models issuing a started transaction upon delivery of an update message at the coordinating site if it is not in conflict with other active transactions at the coordinator. The guards of this refinement assume that a started transaction is issued only when an update message is delivered to the coordinator site. It may be noted that in our model of reliable broadcast, a message is eventually delivered to all sites, including the sender. Also, as outlined in the specifications of BeginSubTran event, a sub-transaction of \(tt\) starts at a site \(ss\) upon delivery of an update message \(mm\) corresponding to the transaction \(tt\).

It can be noticed in the specifications of the event BeginSubTran that the following guard is removed.

\[
sitetransstatus(tt)(\text{coordinator}(tt)) \in \{\text{precommit, pending}\}
\]

The reason is that it is not possible for a participating site to determine the transaction state at the coordinating site when an update message is delivered to a participating site. The removal of the guard generates a new proof obligation PO9 shown below.

\[
\begin{align*}
\text{BeginSubTran}(PO9) & \quad \begin{cases} 
ss \in \text{SITE} \\
\tt \in \text{trans} \\
mm \in \text{update} \\
mm \rightarrow \tt \in \text{tranupdate} \\
\ss \rightarrow \mm \in \text{deliver} \\
\ss \rightarrow \tt \notin \text{activetrans} \\
\ss \notin \text{dom(sitetransstatus}(tt)) \\
\ss \neq \text{coordinator}(tt) \\
\text{transobject}(tt) \subseteq \text{freeobject}\{\ss\} \\
\Rightarrow \\
\text{sitetransstatus}(tt)(\text{coordinator}(tt)) \in \{\text{precommit, abort}\}
\end{cases}
\end{align*}
\]

In order to discharge the proof obligation PO9, we construct an invariant Inv-10 given in Fig. 3.19\(^1\) and add it to the refinement. This invariant is sufficient to discharge the proof obligation PO9.

Inv-10 states that when an update message \(m\) related to the transaction \(t\) is delivered to a participating site \(s\) and if site \(s\) has not already started a sub-transaction then the status of transaction \(t\) at the coordinator is either precommit or abort.

---

\(^1\)For the explanation of codes, see Table 3.1 in Section 3.5
3.6.4 Local Commit/Abort

The events of commit/abort of a sub-transaction at a participating site in the refinement are given in the Fig. 3.20 as SiteCommitTx and SiteAbortTx. As outlined in the specifications of SiteCommitTx event, a participating site ss may decide to pre-commit a transaction tt if it is active at ss. At the time of pre-commit of tt, the participating site also sends a votecommit message mm. The variable tranvotecommit is also updated to indicate that the mm is a votecommit message related to the transaction tt. It may be recalled that both the events SiteCommitTx and SiteAbortTx occur as a consequence of occurrence of event BeginSubTran. Also, as shown in the specifications of the SiteAbortTx event that a site ss may decide to abort a transaction tt if it is active at ss. It does so by sending a voteabort message mm. The variable tranvoteabort is also updated to indicate that mm is a voteabort message related to the transaction tt.

Instead of presenting all events in similar detail we will briefly outline other events of this refinement. A global abort/commit event occurs at the coordinator site when a coordinator delivers voteabort/votecommit messages from the participating sites. The coordinator then decides to commit or abort a transaction globally and informs the participating sites by sending globalabort or globalcommit messages. Upon delivery of either of these message, a participating site either aborts or commits a transaction at that site. The events ExeAbortDecision and ExeCommitDecision model the abort and commit of an active transaction tt at a participating site ss. The detailed specifications of this refinement is given third refinement in Appendix-A.

In this model ordering on the messages is not dealt with explicitly. A transaction may deadlock due to race conditions in a replicated database. It is our assumption that ordered delivery of messages may be used to prevent deadlock arising due to two simultaneous update requests on the same objects from two different sites. A formal development of ordering of messages for fault tolerant transactions and their implementation with logical clock is developed in later chapters.
SiteCommitTx(tt ∈ TRANSACTION, ss ∈ SITE) ≜

**ANY**  
**WHERE**  
`mm` ∈ MESSAGE  
∧ `mm` ∈ dom(sender)  
∧ `tt` ∈ trans  
∧ (ss → `tt`) ∈ activetrans  
∧ sitetransstatus(`tt`(ss)) = pending  
∧ ss ≠ coordinator(`tt`)  
∧ ran(transeffect(`tt`)) ≠ {∅}  
**THEN**  
`sitetransstatus(`tt`(ss)) := precommit`

|| `votecommit := votecommit ∪ {mm}`
|| `tranvotecommit := tranvotecommit ∪ {mm → tt}`
|| `sender := sender ∪ {mm → ss}`

**END:**

SiteAbortTx(tt ∈ TRANSACTION, ss ∈ SITE) ≜

**ANY**  
**WHERE**  
`mm` ∈ MESSAGE  
∧ `mm` ∈ dom(sender)  
∧ `tt` ∈ trans  
∧ (ss → `tt`) ∈ activetrans  
∧ `sitetransstatus(`tt`(ss)) = pending  
∧ ss ≠ coordinator(`tt`)  
∧ ran(transeffect(`tt`)) ≠ {∅}  
**THEN**  
`sitetransstatus(`tt`(ss)) := abort`

|| `freeobject := freeobject ∪ {ss} × transobject(`tt`)`
|| `activetrans := activetrans -{ss → `tt`}`
|| `voteabort := voteabort ∪ {mm}`
|| `tranvoteabort := tranvoteabort ∪ {mm → `tt`}`
|| `sender := sender ∪ {mm → ss}`
|| `completed := completed ∪ {tt → ss}`

**END:**

Figure 3.20: Refined Local Commit and Local Abort events: A Reliable Broadcast

### 3.7 Site Failures and Abortion by Time-Outs

Our model of a distributed transaction ensures global atomicity despite transaction failures and preserves the *one-copy equivalence* consistency criterion. In this section, we address the issue of participating site failures and show that a replicated database remains in a consistent state even in the presence of site failures\(^2\).

A simple refinement is outlined to illustrate that this model preserves the consistency of the database when transactions are aborted due to timeouts and site failures. In this

\(^2\)We assume that a site fails by crash and does not resume operation.
refinement, we explicitly model site failures and assume that a failed participating site does not communicate with the coordinating site. If the coordinator does not receive a communication from a participating site then the coordinator aborts a global transaction by timeout and sends a global abort message to the participating sites. To model the site failures we introduce new variables $oks$ and $fails$ typed as follows:

$$
oks \subseteq SITE \\
 fails \subseteq SITE \\
oks \cap fails = \emptyset
$$

The variables $oks$ and $fails$ are initialized as follows.

$$
oks := SITE, fails := \emptyset
$$

A new event $SiteFailure(ss)$ is introduced in the refinement to model failure of a site. The specification of this event is given in Fig. 3.21. As shown in the specifications, an $oks$ may fail and becomes unavailable.

$$
SiteFailure(ss \in SITE) \equiv \\
\text{WHEN } ss \in oks \\
\text{THEN } fails := fails \cup \{ss\} \\
\quad || oks := oks - \{ss\} \\
\text{END}
$$

**Figure 3.21: Event Site Failure**

Since we assume the failure of a site by crash, the non-availability of a failed site during the rest of computation is also assumed. Also, we do not consider site failures due to omission, malicious or Byzantine faults$^3$. In our model of a transaction, we also assume that the coordinating site of a transaction does not fail during a transaction execution. The failure of sites is restricted to participating site failures due to crash. Since, in the present work we do not deal with the database recovery, we assume that a coordinator will recover successfully from the failure when it will resume operations. However, we plan to address the issue of recovery of the coordinator in the future work.

Before failing, a participating site may be in any one of the following states.

1. It has not yet sent out $votecommit$ or $voteabort$ message to the coordinating site.

2. It has sent $votecommit$ or $voteabort$ message to the coordinating site but did not deliver $global abort$ or $global commit$ message from the coordinating site.

$^3$It has been argued that the distributed systems with unreliable communication, i.e., loss of messages, generation of messages or garbling of messages do not admit solutions to Non-Blocking Atomic Commitment problem [49, 107]. The problem known as ‘Generals Paradox’ is outlined in [49]
3. It has delivered global abort or global commit message from the coordinating site.

In the first case, if a participating site has not sent out votecommit or voteabort message to the coordinating site, the coordinating site waits for a random amount of time and aborts the transaction by trigging an timeout event. We have already outlined that aborting a transaction by its coordinating site still preserves the consistency. In the second case, if a participating site fails before delivering a global abort/commit message, it delivers these message when it recovers. If a participating site has already delivered a global abort/commit message then it does not affect the computation.

To model the abortion of a global transaction at the coordinating site, we introduce a new event TimeOut to our model. The specification of this event for this refinement is given in the Fig. 3.22. As shown in the specifications, a coordinating site sends global abort messages to participating sites and a transaction is globally aborted. Also, a coordinator is in oksite when event TimeOut is activated.

It can be noted that the effects of the TimeOut event are similar to AbortWriteTran event. The event AbortWriteTran is activated when a coordinating site delivers a vote-abort message from a participating site, whilst the event TimeOut may be activated if the coordinator does not receive any communication from a participating site. In order to add this event to this refinement, we have to add this event to each level of the refinement chain. We observe that the addition of this event at each level of the refinement chain preserves the invariants. The detailed specifications of the event TimeOut for each level of refinement chain are given in Appendix-B.

\[
\text{TimeOut}(tt \in \text{TRANSACTION}) \equiv \\
\text{ANY} \quad mm \\
\text{WHERE} \quad mm \in \text{MESSAGE} \land mm \notin \text{dom(sender)} \\
\quad \land \quad tt \in \text{trans} \\
\quad \land \quad \text{ran(\text{transeffect}(tt))} \neq \emptyset \\
\quad \land \quad (\text{coordinator}(tt) \rightarrow tt) \in \text{activetrans} \\
\quad \land \quad \text{transstatus}(tt) = \text{PENDING} \\
\quad \land \quad \text{coordinator}(tt) \in \text{oksite} \\
\text{THEN} \\
\quad \text{transstatus}(tt) := \text{ABORT} \\
\quad \mid \quad \text{activetrans} := \text{activetrans} - \{\text{coordinator}(tt) \rightarrow tt\} \\
\quad \mid \quad \text{sitetransstatus}(tt)(\text{coordinator}(tt)) := \text{abort} \\
\quad \mid \quad \text{freeobject} := \text{freeobject} \cup \{\text{coordinator}(tt)\} \times \text{transobject}(tt) \\
\quad \mid \quad \text{globalabort} := \text{globalabort} \cup \{mm\} \\
\quad \mid \quad \text{tranglobalabort} := \text{tranglobalabort} \cup \{mm \rightarrow tt\} \\
\quad \mid \quad \text{sender} := \text{sender} \cup \{mm \rightarrow \text{coordinator}(tt)\} \\
\quad \mid \quad \text{completed} := \text{completed} \cup \{tt \rightarrow \text{coordinator}(tt)\} \\
\text{END};
\]

\text{Figure 3.22: Event TimeOut}
3.8 Conclusions

In this chapter, we have presented a formal approach to modelling and analyzing a distributed transaction mechanism for replicated databases using Event-B. The abstract model of transactions is based on the notion of a single copy database. In the first refinement of the abstract model, we introduced the notion of a replicated database. The replica control mechanism presented in this refinement allows an update transaction to be submitted at any site. An update transaction commits atomically updating all copies at commit or none when it aborts. A read-only transaction may perform read operations on any single replica. The various events given in the refinement are triggered within the framework of commit protocols which ensure global atomicity of update transactions despite site or transaction failures. The system allows the sites to abort a transaction independently and keeps the replicated database in a consistent state.

The second refinement simplifies the global commit event. In the third refinement, we explicitly model the messaging among the sites and show how the various messages of the protocols are sent by various sites. The fourth refinement model introduced the failure of sites. We have also outlined a timeout event given as TimeOut, which may be activated by a coordinator site to abort a transaction globally if it does not receive a communication from a participating site. The preservation of the invariants of the first refinement ensures that aborting a transaction by the TimeOut event preserves the consistency of a database.

The system development approach considered is based on Event-B, which facilitates incremental development of dependable systems. The work was carried out using the Click’n’Prove B tool. The tool generates the proof obligations for refinement and consistency checking. The majority of proofs were discharged using the automatic prover of the tool, however one third of the complex proofs required use of the interactive prover. Proof statistics of this development are outlined in the Table 3.2.

<table>
<thead>
<tr>
<th>Machine</th>
<th>Total POs</th>
<th>Completely Automatic</th>
<th>Required Interaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract Model</td>
<td>20</td>
<td>20</td>
<td>00</td>
</tr>
<tr>
<td>First Refinement</td>
<td>189</td>
<td>103</td>
<td>86</td>
</tr>
<tr>
<td>Second Refinement</td>
<td>36</td>
<td>22</td>
<td>14</td>
</tr>
<tr>
<td>Third Refinement</td>
<td>41</td>
<td>32</td>
<td>09</td>
</tr>
<tr>
<td>Fourth Refinement</td>
<td>21</td>
<td>14</td>
<td>07</td>
</tr>
<tr>
<td>Overall</td>
<td>307</td>
<td>191</td>
<td>116</td>
</tr>
</tbody>
</table>

Table 3.2: Proof Statistics- Distributed Transactions

In this chapter we have also outlined how we construct an invariant after observing the proof obligations. Due to the large number of proof obligations, it is not possible to accommodate all proof obligations in this thesis. However, the important and significant proof obligations and, the invariants we construct after observing them, are
outlined. Also, in many cases the B tool initially generates very large and complex proof obligations. These proof obligations may be simplified with interaction with the tool by adding new hypothesis or instantiating the hypothesis containing the quantification. Our understanding with this development is that a single proof obligation is not always helpful constructing a right invariant. In most of the cases we have to consider a set of proof obligations to construct a correct invariant.
Chapter 4

Causal Order Broadcast

4.1 Introduction

The notion of causal order broadcast of messages was introduced in [20] to reduce the asynchrony of communication channels perceived by the application processes. The global causal ordering of messages deals with the notion of maintaining the same causal relationship that holds between message send and message receive events. It states that the order of the delivery of messages to the processes can not violate the causal order of corresponding broadcast events in the respective sender processes. If the broadcast of any two messages is concurrent, then the processes are free to deliver them in any order. The concept of causal order in distributed system was introduced and formalized in [75], extended in [76] and it was developed further in ISIS [20] to introduce causal order broadcast. Its vector clock based implementation is proposed in [21, 108].

For some applications it is not sufficient to deliver the messages in the same order (total order) at participating sites but it is also important to the deliver the messages in a predetermined order [20]. For example, consider the network news application [52] where users distribute their articles and reviews by a broadcast. For a user in the system, a review is meaningful only if he has been delivered the main article. Since a broadcaster of the review delivers the main article before he broadcasts the review, the application requires that each user delivers the main article before a review is delivered. Similarly, in an another example, consider a distributed computation in a banking environment where the accounts of employees are to be updated first by paying salary then by paying interest on the account balance. This is done by broadcasting a salary message before broadcasting an interest message. In this case, it is not only important to deliver the messages in the same total order to all participating sites but also each site must deliver a salary message before delivering an interest message. The group communication primitive causal order broadcast alleviates this problem by providing higher guarantees that
messages are delivered to the sites/processes respecting the causality of their broadcast events.

There exists a vast literature [38] on ordering of messages which shows the complexity of the problem. It is further reported in [38] that there exist too few algorithms which provide clear specification of the problem and provide proof of correctness. Some significant applications of formal methods include using I/O automata to provide formal specifications of a broadcast system [42]. The notion of meta-properties to specify and analyze a protocol which switches between two broadcast algorithms is discussed in [82]. The formal results that define cases where total order satisfies causal relations between messages is discussed in [130].

In this chapter we formally develop a system of a causal order broadcast using an incremental development approach in Event-B. We begin with an abstract model of a reliable broadcast, and in the first refinement, we outline the specification of an abstract causal order. In further refinements we show how abstract causal order can correctly be implemented by a system of vector clocks. The gluing invariants discovered in the process define the relationship between the abstract causal order and vector clock mechanism.

4.2 Incremental Development of Causal Order Broadcast

In this section we outline an incremental development of a system of causal order broadcast consisting of five levels of refinement chain.

4.2.1 Outline of the refinement steps

In this development we begin with an abstract model of a reliable broadcast and successively refine it to a model with vector clocks. A brief outline of each level is given below.

L1 This consists of an abstract model of a reliable broadcast. In this model processes communicate by broadcast and messages are delivered to each process only once, including the sender. This model is outlined in Section 4.2.2.

L2 In this refinement, we outline how an abstract causal order is constructed by the sender. An abstract causal order is constructing by combining FIFO and local ordering properties. This refinement is outlined in Section 4.3.

L3 In this refinement, we introduce the notion of vector clocks. The abstract causal order is replaced by the vector clocks rules. We also discover gluing invariants which define the relationship of abstract causal order and vector rules. This refinement is given in Section 4.4.
In this refinement, we present a simplification of the vector rules for updating the vector clock of recipient processes. This refinement is outlined in Section 4.5.

This is another refinement further simplifying the vector rules for updating vector clocks. This refinement also is outlined in Section 4.5.

### 4.2.2 Abstract Model of a Reliable Broadcast

The abstract model of a reliable broadcast system is presented as an Event-B machine in Fig. 4.1. PROCESS and MESSAGE are defined as sets. The brief description of this machine is given as follows.

**MACHINE** Broadcast

**SETS** PROCESS; MESSAGE

**VARIABLES** sender, cdeliver

**INVARIANT**

/* I-1*/ sender \in MESSAGE \rightarrow PROCESS

/* I-2*/ \land cdeliver \in PROCESS \leftrightarrow MESSAGE

/* I-3*/ \land \text{ran}(cdeliver) \subseteq \text{dom}(sender)

**INITIALISATION**

\text{sender} := \emptyset \parallel \text{cdeliver} := \emptyset

**EVENTS**

Broadcast ( pp \in PROCESS , mm \in MESSAGE ) \equiv

\text{WHEN} \quad \text{mm} \notin \text{dom}(\text{sender})

\text{THEN} \quad \text{sender} := \text{sender} \cup \{ \text{mm} \mapsto pp \}

\parallel \text{cdeliver} := \text{cdeliver} \cup \{ pp \mapsto \text{mm} \}

\text{END};

Deliver ( pp \in PROCESS , mm \in MESSAGE ) \equiv

\text{WHEN} \quad \text{mm} \in \text{dom}(\text{sender})

\land ( pp \mapsto \text{mm} ) \notin \text{cdeliver}

\text{THEN} \quad \text{cdeliver} := \text{cdeliver} \cup \{ pp \mapsto \text{mm} \}

\text{END} ;

**END**

*Figure 4.1: Abstract Model of Broadcast*

- *sender* is a partial function from MESSAGE to PROCESS defined in invariant I-1. It contains mappings from MESSAGE to PROCESS. The mapping (m \mapsto p) \in \text{sender} indicates that message m was sent by process p. The partial function ensures that a message is sent by only one process.

- *cdeliver* is a relation between PROCESS and MESSAGE defined in invariant I-2. A mapping of form (p \mapsto m) \in \text{cdeliver} indicates that a process p has delivered a message m. The *sender* and *cdeliver* are initialized as empty sets.
In our model of a broadcast system, a sent message is also delivered to its sender. It may be noticed that all delivered messages must be messages whose Message Sent event is also recorded. This property is defined as invariant I-3.

The events of sending and delivery of messages are shown as the parameterized operations Broadcast(pp,mm) and Deliver(pp,mm). It can be noted that the messages are not yet ordered in the abstract model. When a Broadcast event is invoked, variable sender is updated by adding a mapping of a process and a corresponding message. A sender process also delivers the message at the time of broadcast. It is shown by updating variable deliver. The Deliver event is guarded by predicates. These predicates ensure that a process can only deliver a message whose message sent event is recorded and the message has not been delivered before. Therefore, on activation of this event a message is delivered to a process other than sender. A message is delivered to a process if both conditions are satisfied.

### 4.3 First Refinement : Introducing Ordering on Messages

The refinement of the abstract model of broadcast is given in Fig. 4.2 and Fig. 4.3. A brief description of the refinement steps is given below.

| REFINEMENT | CausalOrder |
| REFINES | Broadcast |
| VARIABLES | sender, cdeliver, corder, delorder |
| INVARIANT |
| /* I-4*/ | corder ∈ MESSAGE ↔ MESSAGE |
| /* I-5*/ | delorder ∈ PROCESS → (MESSAGE ↔ MESSAGE) |
| /* I-6*/ | dom(corder) ⊆ dom(sender) |
| /* I-7*/ | ran(corder) ⊆ dom(sender) |

**INITIALISATION**

\[
\text{sender} := \emptyset \quad \parallel \quad \text{cdeliver} := \emptyset \\
\quad \parallel \quad \text{corder} := \emptyset \quad \parallel \quad \text{delorder} := \emptyset
\]

**Figure 4.2: Causal Order Broadcast : Initialization**

- The abstract causal order is represented by a variable corder. A mapping of the form \((m1 \mapsto m2) \in \text{corder}\) indicates that message \(m1\) causally precedes \(m2\). (Inv I-4)

- In order to represent the delivery order of messages at a process, variable delorder is used. A mapping \((m1 \mapsto m2) \in \text{delorder}(p)\) indicates that process \(p\) has delivered \(m1\) before \(m2\). (Inv I-5)

- Causal order on the messages can be defined only on those messages whose message sent event is recorded. (Inv I-6, I-7)
The events $\text{Broadcast}(pp, mm)$ and $\text{Deliver}(pp, mm)$ respectively model the events of broadcasting a message and the causally ordered delivery of a message. As shown in the operation of the $\text{Broadcast}$ event, a causal order is built by the sender process following a FIFO order and a local order. When a process $pp$ broadcasts a message $mm$, the variable $\text{corder}$ is updated by the mappings in $(sender^{-1}[\{pp\}] \times \{mm\})$. This indicates that all messages sent by $pp$ before broadcasting $mm$ causally precede $mm$ conforming to FIFO order. Similarly, the mappings in $(\text{cdeliver}[\{pp\}] \times \{mm\})$ indicate that the messages causally delivered to the process $pp$ before broadcasting $mm$ also causally precedes $mm$ conforming to a local order.

On the occurrence of the $\text{Broadcast}$ event, variable $\text{sender}$ is updated with corresponding entries of the sender process and the message. The guard $mm \not\in \text{dom}(\text{sender})$ ensures that each time a fresh message is broadcasted. The delivery order at the sender process is updated at the time of broadcast. In the $\text{Deliver}$ event, a process $pp$ delivers a message $mm$ only when all messages which causally precedes $mm$ are delivered. The guards of this event also ensure that a message is delivered only once.

In order to prove that this is a valid refinement of abstract model of a reliable broadcast, we need to prove that the invariants in the Fig. 4.2 are preserved by the activation of the events. For refinement checking, the B tool generates the proof obligations with respect
to these invariants and corresponding events. These proof obligations are discharged automatically by the prover.

### 4.3.1 Invariant Properties of Causal Order

After building the model of the abstract causal order our goal was to formally verify that this model preserves the *causal order* properties *informally* defined in the section 2.3. It states that the delivery order of the messages at a given process must conform to the abstract causal order among them. In order to construct an invariant that states the causal ordering properties are preserved by the model, we consider following two cases generated by *ProB* [80], an animator and model checker for B.

![Figure 4.4: Causal Order : CASE-I](image1)

![Figure 4.5: Causal Order : CASE-II](image2)

As shown in Fig. 4.4, messages $M_1$ and $M_2$ have the same delivery order at processes $P_1$ and $P_2$ but have different delivery order as shown in Fig. 4.5. This is possible when $M_1$ and $M_2$ do not have any causal ordering among them. The case-I tells us that having a same delivery order at the processes does not imply that messages are causally ordered. Similarly, from case-II we conclude that if the delivery order of the messages at the processes are different then the messages are not causally ordered. Also [52] reports that if the broadcast of two messages are not related by causal precedence, a causal broadcast does not impose any requirement on the order they are delivered and the
delivery order of any two messages may be different at various processes. Therefore, we add following invariant to our model as a primary invariant:

\[
\begin{align*}
    m_1 \rightarrow m_2 \in \text{corder} & \land \\
    p \rightarrow m_2 \in \text{cdeliver} & \\
    \Rightarrow & \\
    m_1 \rightarrow m_2 \in \text{delorder}(p)
\end{align*}
\]

This invariant states that if two messages are causally ordered then their delivery order will be same as their causal order only if a process has delivered the later message. In order to verify that our model also preserves the transitivity property on the messages, we also add following invariant to our model as a primary invariant:

\[
\begin{align*}
    m_1 \rightarrow m_2 \in \text{corder} & \land \\
    m_2 \rightarrow m_3 \in \text{corder} & \\
    \Rightarrow & \\
    m_1 \rightarrow m_3 \in \text{corder}
\end{align*}
\]

### 4.3.2 Proof Obligations and Invariant Discovery

In this section we outline how we verify that the model CausalOrder given in Fig. 4.2 and Fig. 4.3 preserves the causal ordering on the messages. We also outline how the proof obligations generated by the B tool and the interactive prover guide us constructing new invariants. The primary invariant properties of the model of causal order broadcast system are given in Fig. 4.6 as predicates which also include transitivity. We have omitted the quantifications over all identifiers \((m_1,m_2,p)\) etc to avoid clutter. We first add the invariant Inv-1 to our model. After addition of this invariant to the model, the B tool generated two proof obligations associated with events Broadcast and Deliver. These proof obligations were discharged using the interactive prover without having to add new invariants.

<table>
<thead>
<tr>
<th>Invariants</th>
<th>Required By</th>
</tr>
</thead>
<tbody>
<tr>
<td>/<em>Inv-1</em>/</td>
<td>((m_1 \rightarrow m_2) \in \text{corder} \land (p \rightarrow m_2) \in \text{cdeliver}) \Rightarrow (m_1 \rightarrow m_2) \in \text{delorder}(p)</td>
</tr>
<tr>
<td>/<em>Inv-2</em>/</td>
<td>((m_1 \rightarrow m_2) \in \text{corder} \land (m_2 \rightarrow m_3) \in \text{corder}) \Rightarrow (m_1 \rightarrow m_3) \in \text{corder}</td>
</tr>
</tbody>
</table>

Figure 4.6: Invariants-I
In the next step, we add invariant Inv-2 to our model. This invariant states that our model of Causal Broadcast preserves transitivity relationship on the messages. When this invariant is added to the model, the B tool generates the following complex proof obligation associated with the Broadcast event.

\[
\text{Broadcast}(pp, mm)\ PO1
\begin{align*}
& \text{Inv2} \land \\
& mm \notin \text{dom}(\text{sender}) \land \\
& m1 \mapsto m2 \in (\text{corder} \cup (\text{sender}^{-1}\{pp\} \times \{mm\}) \cup (\text{deliver}\{pp\} \times \{mm\})) \land \\
& m2 \mapsto m3 \in (\text{corder} \cup (\text{sender}^{-1}\{pp\} \times \{mm\}) \cup (\text{deliver}\{pp\} \times \{mm\})) \\
\Rightarrow \\
& m1 \mapsto m3 \in (\text{corder} \cup (\text{sender}^{-1}\{pp\} \times \{mm\}) \cup (\text{deliver}\{pp\} \times \{mm\}))
\end{align*}
\]

This proof obligation is reduced to following two simple proof obligations using the interactive prover:

\[
\text{Broadcast}(pp, mm)\ PO2
\begin{align*}
& m1 \mapsto m2 \in \text{corder} \land \\
& m2 \in (\text{sender}^{-1}\{pp\}) \land \\
& m1 \notin (\text{sender}^{-1}\{pp\}) \land \\
\Rightarrow \\
& m1 \in (\text{deliver}\{pp\})
\end{align*}
\]

and

\[
\text{Broadcast}(pp, mm)\ PO3
\begin{align*}
& m1 \mapsto m2 \in \text{corder} \land \\
& m2 \in (\text{sender}^{-1}\{pp\}) \land \\
& m1 \notin (\text{deliver}\{pp\}) \land \\
\Rightarrow \\
& m1 \in (\text{sender}^{-1}\{pp\})
\end{align*}
\]

The proof obligation PO2 generated by the Broadcast event states that if a message \(m_1\) causally precedes \(m_2\) i.e., \((m_1 \mapsto m_2) \in \text{order}\), and that \(pp\) is sender of \(m_2\) and \(m_1\) was not sent by process \(pp\) then process \(pp\) must have delivered \(m_1\). This corresponds to the property of local order. Similarly, the proof obligation PO3 states that if \(m_1\) causally precedes \(m_2\) and \(pp\) is the sender of \(m_2\) and \(pp\) has not delivered \(m_1\) then \(pp\) is sender of \(m_1\). It can be noticed that this property corresponds to the FIFO order. Therefore, to discharge these proof obligations, we add following invariant to the model.
This invariant is given as Inv-3 in the Fig. 4.7. After adding invariant Inv-3 to the model we discharge the proof obligations PO2 and PO3 associated with the Broadcast event. However, due to the addition of Inv-3, additional proof obligations associated with Broadcast and Deliver events are generated. The proof obligation associated with the Broadcast event is discharged using the interactive prover. The following proof obligation associated with the Deliver event can not be discharged interactively.

Deliver\((pp, mm)\) PO4

\[
\begin{align*}
\text{Inv} & \ 3 \ \land \\
\text{m1} & \rightarrow \text{m2} \ \epsilon \ \text{corder} \ \land \\
\text{m2} & \epsilon \ (\text{sender}^{-1}[[p]]) \ \land \\
\Rightarrow \\
\text{m1} & \epsilon \ (\text{sender}^{-1}[[p]]) \ \lor \ \text{m1} \ \epsilon \ \text{cdeliver}[[p]]
\end{align*}
\]

PO4 states that for messages \(m1\) and \(m2\) where \(m1\) causally precedes \(m2\) and a process \(pp\) has delivered \(m2\) then \(pp\) has either delivered \(m1\) or broadcasted \(m1\). On simplifying PO4, it requires us to prove following.

\[
\begin{align*}
m1 & \rightarrow m2 \ \epsilon \ \text{corder} \ \land \\
p & \rightarrow m2 \ \epsilon \ \text{cdeliver} \\
\Rightarrow \\
p & \rightarrow m1 \ \epsilon \ \text{cdeliver}
\end{align*}
\]

In order to prove the above, we add an invariant to our model given as Inv-4 in the Fig. 4.7. It states that if \(m1\) causally precedes \(m2\) then any process \(p\) that has delivered
Chapter 4 Causal Order Broadcast

We observe that after three iterations of invariant strengthening we arrive at an invariant that is sufficient to discharge all proof obligations. By discharging all proof obligations we ensure that this model preserves the causal precedence relationship on the messages.

4.4 Second Refinement : Introducing Vector Clocks

In this section, we outline how an abstract causal order can be refined by a system of vector clocks. The goals of this refinement are given below.

- To replace the abstract global variable \( \text{corder} \) with vector clock rules.
- To refine the \( \text{Broadcast} \) event to generate the vector timestamp of messages which realizes the global causal order.
- To refine the \( \text{Deliver} \) event to include a mechanism by which an early reception of a message violating the global causal order may be detected at the recipient process.

In our model, we use Birman, Schiper and Stephenson’s protocol [21] to update the vector clock of a process broadcasting or delivering a message and to timestamp a message.

I. While sending a message \( M \) from process \( P_i \) to \( P_j \), sender process \( P_i \) updates its own time( \( i^{th} \) entry of vector) by updating \( VT_{P_i}(i) \) as \( VT_{P_i}(i) := VT_{P_i}(i) + 1 \).

The message timestamp \( VT_M \) of message \( M \) is generated as \( VT_M(k) := VT_{P_i}(k) \), \( \forall k \in (1..N) \), where \( N \) is number of processes in system. Since a process \( P_i \) increments its own value only at the time of sending a message, \( VT_{P_i}(i) \) indicates number of messages sent out by process \( P_i \).

II. The recipient process \( P_j \) (\( P_j \neq P_i \)) delays the delivery of message \( M \) until following conditions are satisfied.

\[
\begin{align*}
\text{i} & \quad VT_{P_j}(i) = VT_M(i) - 1 \\
\text{ii} & \quad VT_{P_j}(k) \geq VT_M(k), \forall k \in (1..N) \land (k \neq i).
\end{align*}
\]

The first condition ensures that process \( P_j \) has received all but one message sent by process \( P_i \). The second condition ensures that process \( P_j \) has received all
messages received by sender $P_i$ before sending the message $M$. A sender process need not delay the delivery of a message. These conditions ensures global ordering on messages.

III. The recipient process $P_j$ updates its vector clock $VTP_j$ at message receive event of message $M$ as $VTP_j(k) := \text{Max}(VTP_j(k), VTM(k))$. Therefore in the vector clock of process $P_j$, $VTP_j(i)$ indicates the number of messages delivered to process $P_j$ sent by process $P_i$.

This refinement (second refinement) consists of four state variables $sender, cdeliver, VTP$ and $VTM$. The new state variables $VTP$ and $VTM$ respectively represents the vector time of a process and the vector timestamp of a message. These variables are typed as follows.

**Broadcast** ($pp \in \text{PROCESS}, mm \in \text{MESSAGE} \equiv$

WHEN $mn \notin \text{dom}(sender)$ THEN LET $nVTP$ BE $nVTP = VTP(pp) \subset \{ pp \rightarrow VTP(pp)(pp)+1 \}$

IN $VTM(mm) := nVTP$

$\|$ $VTP(pp) := nVTP$

$\|$ $sender := sender \cup \{ mm \rightarrow pp \}$

$\|$ $cdeliver := cdeliver \cup \{ pp \rightarrow mm \}$

END ;

**Deliver** ($pp \in \text{PROCESS}, mm \in \text{MESSAGE} \equiv$

WHEN $mm \in \text{dom}(sender)$

$\wedge (pp \rightarrow mm) \notin cdeliver$

$\wedge \forall p. (p \in \text{PROCESS} \wedge p \notin sender(mm) \Rightarrow VTP(pp)(p) \geq VTM(mm)(p))$

$\wedge VTP(pp)(sender(mm)) = VTM(mm)(sender(mm)) - 1$

THEN $cdeliver := cdeliver \cup \{ pp \rightarrow mm \}$

$\|$ $VTP(pp) := VTP(pp) \subset$

$\{(q \mid q \in \text{PROCESS} \wedge VTP(pp)(q) < VTM(mm)(q)) \subset VTM(mm)\}$

END;

**Figure 4.8:** Second Refinement : Refinement with Vector Clocks

$VTP \in \text{PROCESS} \rightarrow (\text{PROCESS} \rightarrow \text{NATURAL})$

$VTM \in \text{MESSAGE} \rightarrow (\text{PROCESS} \rightarrow \text{NATURAL})$

These variables are initialized as follows,

$VTP := \text{PROCESS} \times \{ \text{PROCESS} \times 0 \}$

$VTM := \text{MESSAGE} \times \{ \text{PROCESS} \times 0 \}$

As shown above, the variables $VTP$ and $VTM$ are initialized by assigning a array of vectors initialized with zero to each process and messages.
The refined specifications of the Broadcast and Deliver events are given in Fig 4.8. A brief description of the refinement is given in following steps.

- As shown in the Broadcast specifications, operations involving the abstract variable corder are replaced by the vector rules. It can be noticed that at the time of broadcasting a message $mm$, process $pp$ increments its own clock value $VTP(pp)(pp)$ by one. $VTP(pp)(pp)$ represents the number of messages sent by process $pp$. The modified vector timestamp of the process is assigned to message $mm$ giving the vector timestamp of message $mm$.

- As shown in the event Deliver, messages are delivered at a process only if it has delivered all but one message from the sender of that message. Vector timestamps of recipient processes and messages are also compared to ensure that all messages delivered by the sender of the message before sending it, are also delivered at the recipient process. These conditions are included as a guard in Deliver operation. It can be noticed that the guard involving the variable corder in the abstract model is replaced by the guards involving comparison of the vector timestamps of messages and processes in the refinement.\(^1\), \(^2\)

### 4.4.1 Gluing invariants relating Causal Order and Vector Rules

The replacement of the operations and guards involving variable corder in the abstract model with operations and guards involving vector clock rules in refinement generates proof obligations. These proof obligations can be discharged interactively using the B tool after three rounds of invariant strengthening. A full set of gluing invariants involving the abstract causal order and the vector clock rules is given in Fig. 4.9. A brief description of these properties is given below.

- If the vector time of process $P$ is equal or more than the vector timestamp of any sent message $M$ then $P$ must have delivered message $M$. (Inv-5)

- For any two messages $m1$ and $m2$ where $m1$ causally precedes $m2$, the vector timestamp of $m1$ is always less than vector timestamp of $m2$. (Inv-6)

- Since $VTP(p)(p)$ represents the total number of messages sent by process $p$ and $VTM(m)(p)$ represents the number of messages received by sender of $m$ from process $p$ before sending $m$, the number of messages sent by process $p$ will be greater than or equal to the number of messages received by sender($m$) from $p$. (Inv-7)

\(^1\) $(f \leftarrow g)$ represents function $f$ overridden by $g$.
\(^2\) $(s \circ f)$ represents function $f$ is domain restricted by $s$. 

Chapter 4 Causal Order Broadcast

<table>
<thead>
<tr>
<th>Invariants</th>
<th>Required By</th>
</tr>
</thead>
<tbody>
<tr>
<td>/<em>Inv-5</em>/ $m \in \text{dom}(\text{sender}) \land \text{VTP}(p_1)(p_2) \geq \text{VTM}(m)(p_2)$</td>
<td>Broadcast, Deliver</td>
</tr>
<tr>
<td>$(p_1 \rightarrow m) \in \text{cdeliver}$</td>
<td></td>
</tr>
<tr>
<td>/<em>Inv-6</em>/ $(m_1 \rightarrow m_2) \in \text{corder}$</td>
<td>Broadcast, Deliver</td>
</tr>
<tr>
<td>$\Rightarrow \text{VTM}(m_1)(p) \leq \text{VTM}(m_2)(p)$</td>
<td></td>
</tr>
<tr>
<td>/<em>Inv-7</em>/ $m \in \text{dom}(\text{sender})$</td>
<td>Broadcast, Deliver</td>
</tr>
<tr>
<td>$\Rightarrow \text{VTM}(m)(p) \leq \text{VTP}(p)(p)$</td>
<td></td>
</tr>
<tr>
<td>/<em>Inv-8</em>/ $\text{VTM}(m)(p) = 0$</td>
<td>Broadcast</td>
</tr>
<tr>
<td>$\Rightarrow m \notin (\text{dom(corder)} \cup \text{ran(corder)})$</td>
<td></td>
</tr>
<tr>
<td>/<em>Inv-9</em>/ $p_1 \neq p_2 \Rightarrow \text{VTP}(p_1)(p_2) \leq \text{VTP}(p_2)(p_2)$</td>
<td>Broadcast</td>
</tr>
</tbody>
</table>

Figure 4.9: Invariants-III

- A message whose timestamp is a vector of zero’s implies that it is not causally ordered. (Inv-8)
- For any two separate processes $p_1$ and $p_2$, knowledge of $p_2$ at $p_1$ can not be greater than the knowledge at $p_2$ itself. (Inv-9)

4.5 Further Refinements of Deliver Event

As outlined in the Rule II of original protocol [21], a recipient process $P_j$ delays the delivery of message $M$ until following conditions are satisfied.

i $\text{VT}_{P_j}(i) = \text{VT}_M(i) - 1$

ii $\text{VT}_{P_j}(k) \geq \text{VT}_M(k), \forall k \in (1..N) \land (k \neq i)$.

Also, Rule III of the protocol [21] states that in the event of causally ordered delivery of a message $M$, the recipient process $P_j$ updates its vector clock $\text{VT}_{P_j}$ as, $\text{VT}_{P_j}(k) := \text{Max}(\text{VT}_{P_j}(k), \text{VT}_M(k))$. The protocol requires updating the whole vector of the recipient process.

Further refinements of Deliver event are outlined here stating that instead of updating whole vector of the recipient process as outlined in the original protocol, updating only one value in the vector clock of recipient process corresponding to the sender process is sufficient to realize causally ordered delivery of the messages.
In the second refinement, the vector clock of the recipient process \( pp \) is updated as:

\[
VTP(pp) := VTP(pp) - \{ (q | q \in PROCESS \land VTP(pp)(q) < VTM(mm)(q) \} \triangleright VTM(mm)
\]

under the following guards:

\[
\forall p \cdot (p \in PROCESS \land p \neq sender(mm) \Rightarrow VTP(pp)(p) \geq VTM(mm)(p))
\]

\[
VTP(pp)(sender(mm)) = VTM(mm)(sender(mm)) - 1
\]

It can be noticed that the vector clock of \( pp \) is updated by the values wherever the values in the message vector are greater, while the guard of the event indicates that except the sender of message, all values of the message vector must be smaller than recipient process vector. This eventually results in updating only one value of the vector of the recipient process which corresponds to the sender of the message. Therefore, we replace the above operation by the following simplified operation in the third refinement which states that only one value in the vector clock of the recipient process \( pp \) corresponding to the sender process of message is updated.

\[
VTP(pp) := VTP(pp) - \{ \text{sender}(mm) \mapsto VTM(mm)(\text{sender}(mm)) \}
\]

**Broadcast** \((pp \in PROCESS, mm \in MESSAGE) \equiv \)

**WHEN** \( mm \notin \text{dom}(sender) \)

**THEN** \( \text{LET} \)

\[
nVTP := VTP(pp) - \{ pp \mapsto VTP(pp)(pp)+1 \}
\]

**IN**

\[
VTM(mm) := nVTP
\]

\[
\parallel VTP(pp) := nVTP
\]

**END**

\[
\parallel \text{sender} := \text{sender} \cup \{ mm \mapsto pp \}
\]

\[
\parallel \text{cdeliver} := \text{cdeliver} \cup \{ pp \mapsto mm \}
\]

**END**;

**Deliver** \((pp \in PROCESS, mm \in MESSAGE) \equiv \)

**WHEN** \( mm \in \text{dom}(sender) \)

\[
\land (pp \mapsto mm) \notin \text{cdeliver}
\land \forall p \cdot (p \in PROCESS \land p \neq \text{sender}(mm) \Rightarrow VTP(pp)(p) \geq VTM(mm)(p))
\land \quad VTP(pp)(\text{sender}(mm)) = VTM(mm)(\text{sender}(mm)) - 1
\]

**THEN**

\[
\text{cdeliver} := \text{cdeliver} \cup \{ pp \mapsto mm \}
\]

\[
\parallel VTP(pp)(\text{sender}(mm)) := VTM(mm)(\text{sender}(mm))
\]

**END**;

**Figure 4.10:** Fourth Refinement
The replacement of the operation generates new proof obligations which are discharged by the automatic prover. This operation is further refined in the fourth refinement, which precisely states that only one value in the vector clock of the recipient process corresponding to the sender of message is updated.

\[ VTP(pp)(sender(mm)) := VTM(mm)(sender(mm)) \]

The fourth refinement is outlined in the Fig. 4.10. In this refinement step we observe that proof obligations are generated due to the replacement of the operations of the event \textit{Deliver}. These proof obligations are automatically discharged by the B prover. A full chain of refinement with complete set of invariants is given in the Appendix-C.

### 4.6 Conclusions

In this chapter we have presented Event-B specifications for global causal ordering of messages. In the abstract specifications of causal order we outlined how a causal order is constructed by combining both FIFO and local order. In the refinement steps we outline how an abstract causal order can correctly be implemented by a system of vector clocks. This is done by replacing the abstract variable \textit{corder} by vector clock rules. We have considered the Birman, Schiper and Stephenson protocol [21] for implementing global causal ordering using vector clocks. In the third and fourth refinements we found that instead of updating whole vector of a recipient process as outlined in the original protocol, updating only \textit{one value} in the vector clock of recipient process corresponding to the sender process is sufficient to realize causally ordered delivery of the message. In this approach we have also discovered several invariants which help us to understand why a causal order broadcast can be implemented using vector clocks. The overall proof statistics are given in Table 4.1. Approximately sixty eight percent of the proofs were discharged by the automatic prover, the rest were discharged by using interactive prover of B tool.

<table>
<thead>
<tr>
<th>Machine</th>
<th>Total POs</th>
<th>Completely Automatic</th>
<th>Required Interaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract Model</td>
<td>14</td>
<td>14</td>
<td>00</td>
</tr>
<tr>
<td>Refinement1</td>
<td>43</td>
<td>21</td>
<td>22</td>
</tr>
<tr>
<td>Refinement2</td>
<td>47</td>
<td>28</td>
<td>19</td>
</tr>
<tr>
<td>Refinement3</td>
<td>06</td>
<td>06</td>
<td>00</td>
</tr>
<tr>
<td>Refinement4</td>
<td>02</td>
<td>02</td>
<td>00</td>
</tr>
<tr>
<td>Overall</td>
<td>112</td>
<td>71</td>
<td>41</td>
</tr>
</tbody>
</table>

\textbf{Table 4.1:} Proof Statistics- Causal Order Broadcast
Chapter 5

Total Order Broadcast

5.1 Introduction

As outlined in the previous chapter, if the broadcast of two messages is not related by the causal precedence (parallel message) relationship, the causal order broadcast does not impose any requirement on the order they are delivered to the other processes [52]. For example, consider a case of replicated databases where the bank accounts of users are replicated across several sites. Suppose a user deposits amount \( x \) to account \( A \), it does so by broadcasting \( \text{add } x \text{ to } A \) to all sites. Suppose at the same time, at another site, the bank decides to pay interest at the rate \( y \) by initiating a broadcast \( \text{add } y \text{ percent to } A \). As the broadcast of both messages are not causally related, the causal order broadcast allows delivery of these messages to participating sites in different orders. It may result in two copies of account \( A \) at different sites having different values, thus transforming the database into an inconsistent state. To prevent this situation, it is required that these two messages must be delivered to all sites in the same order. The group communication primitive called total order broadcast\(^1\) alleviates this problem by providing guarantees that messages sent to a set of sites/processes are delivered in the same order.

The total order broadcast has been proposed for implementing active replication (state machine approach) [74, 116, 103]. The state machine approach is a general method for implementing fault-tolerant services in distributed systems. It has also been proposed to improve the performance of replicated databases [9, 102], transactional systems [65, 114], clock synchronization [132] and crash recovery [111] etc.

The total order broadcast can be defined in terms of two primitives \( \text{TOBroadcast}(m) \) and \( \text{TODeliver}(m) \) where \( m \in M \) and \( M \) is a set of possible messages [38]. It is assumed that

\(^1\)The total order broadcast is also known as atomic broadcast. Both of the terms are used interchangeably, however there is a slight dispute with respect to using one over the other [38]. The term atomic suggest the agreement property rather than total order.
each message is uniquely identified and carries the identity of the sender. The total order broadcast is defined as a reliable broadcast which satisfies the following requirement.

If processes $p$ and $q$ both deliver messages $m_1$ and $m_2$, then $q$ delivers $m_1$ before $m_2$ if and only if $p$ delivers $m_1$ before $m_2$.

The agreement property of a reliable broadcast and total order requirements imply that all correct processes eventually deliver the same sequence of messages [52].

### 5.2 Mechanism for Total Order Implementations

The key issues with respect to the total order broadcast algorithms are how to build a total order and what information is necessary for defining a total order. The algorithms for building a total order can broadly be classified [38] as sequencer based algorithms, token based algorithms, communication history based algorithms and the destination agreement algorithms [21, 36, 37, 38]. In sequencer based algorithms, a specific process takes the role of a sequencer and becomes responsible for building a total order. In token based algorithms (also known as privilege based algorithms), a sender can broadcast a message only when it is granted the privilege (token) to do so. The order is defined by a group of senders and the privilege to broadcast (and order) is granted to one process at a time. The communication history based algorithms use logical timestamps. In these algorithms, as in token based algorithms, the delivery order is determined by the senders. However, processes are free to broadcast messages at any time. Most of these algorithms ensure total order by delaying the delivery of a message at the destination process. In destination agreement algorithms, a delivery order is determined by reaching an agreement between the destination processes. Our model of total order broadcast is based on the sequencer based algorithm.

In sequencer based algorithms, a specific process is elected as a sequencer and becomes responsible for building a total order. It is assumed that each process may broadcast a message at any time and a message will eventually be delivered to all processes in the system inclusive of the sender. A sequencer process also takes the role of a sender and destination in addition to the role of sequencer. There are two class of sequencer based algorithms called fixed sequencer algorithms and moving sequencer algorithms. In a fixed sequencer approach [21, 59], to broadcast a message $m$, a sender sends $m$ to the sequencer. Upon receiving $m$, the sequencer assigns it a sequence number and sends its sequence number to all destinations. Each process delivers the message according the sequence number assigned by the sequencer process. Moving sequencer algorithms [35, 131] are similar to fixed sequencer algorithms except that they allow the role of sequencer to be moved from one process to another for load balancing.
There exist three variants of fixed sequencer algorithms. These are called UB (Unicast Broadcast), BB (Broadcast-Broadcast) and UUB (Unicast-Unicast Broadcast). In the unicast broadcast (UB) variant of the fixed sequencer algorithm, in order to broadcast a message \( m \), a process first unicasts \( m \) to the sequencer. Upon receiving the message, the sequencer assigns a sequence number to it and again broadcasts \( m \) with the sequence number. The protocol steps of the UB variant are illustrated in the Fig. 5.1.

In the broadcast broadcast (BB) variant [21] of the fixed sequencer algorithm, the protocol consists of first broadcasting \( m \) to all destinations including the sequencer, followed by another broadcast of its sequence number by the sequencer. All destination processes deliver messages according to their sequence numbers assigned by the sequencer process. As shown in the Fig. 5.2 process \( P2 \) broadcasts a computation message \( m \). Upon delivery of \( m \) to a sequencer process, the sequencer assigns a sequence number and broadcasts its sequence number by a control message \( (m') \). Upon receipt of the control messages, a destination process delivers its computation message according to the sequence numbers.

In the third variant UUB, the protocol consists of three steps. As shown in the Fig. 5.3, firstly a sender process unicasts \( \text{request}_{\text{-seqno}}(m) \) requesting a sequence number from the sequencer for message \( (m) \). The sequencer unicasts the sequence number of the message \( (\text{seqno}(m)) \) to the sender. In the third step, the sender broadcasts the computation message \( m \) alongside its sequence number \( (\text{seqno}(m)) \).

\[ \text{We use the notion of computation Message to represent the messages to be delivered in a total order. The control messages are generated by the system to implement ordering on the computation messages.} \]
Since our model of transactions for the replicated databases is based on a broadcast system, we focus on the BB variant only. In [139, 141] we have outlined how to process update transactions in a replicated database system using a total order broadcast. In the next section we present a formal analysis of total order broadcast with respect to the broadcast broadcast (BB) variant of the fixed sequencer approach.

### 5.3 Abstract Model of Total Order Broadcast

In this section abstract specifications of a total order broadcast are presented. Later in the refinements we show how a total order on the messages can be implemented by assigning sequence numbers. The abstract model of a total order broadcast system is given in Fig. 5.4 and Fig. 5.5. The initial part of the machine is given in Fig. 5.4 and the specifications of events are given in Fig. 5.5. Types PROCESS and MESSAGE are used to represent a set of processes and messages. The specification consists of four variables sender, totalorder, tdeliver and delorder.

**Figure 5.4: TotalOrder Abstract Model: Initial Part**

A brief description of the machine is given in the following steps.

- **sender** is defined as a partial function from MESSAGE to PROCESS. The mapping \((m \mapsto p) \in \text{sender}\) indicates that message \(m\) was sent by a process \(p\).
The variable `totalorder` is defined as a relation among the messages. A mapping of the form \((m_1 \mapsto m_2) \in \text{totalorder}\) indicates that message \(m_1\) is totally ordered before \(m_2\).

In order to represent the delivery order of messages at a process, variable `delorder` is used. A mapping \((m_1 \mapsto m_2) \in \text{delorder}(p)\) indicates that process \(p\) has delivered \(m_1\) before \(m_2\).

The variable `tdeliver` represents the messages delivered following a total order. A mapping of form \((p \mapsto m) \in \text{tdeliver}\) represents that a process \(p\) has delivered \(m\) following a total order.

The event `Broadcast` given in the Fig. 5.5 models the broadcast of a message. Similarly, the event `Order` models the construction of a total order on a message when it is delivered to a process in the system for the first time, i.e., an abstract global total order is constructed on a message at the first ever delivery of it to any process in the system. Later in the refinement we show that it is a role of a sequencer process. The `TODeliver` models the delivery of the messages to a process when a total order on the message has been constructed.

**Broadcast** \((pp \in \text{PROCESS}, mm \in \text{MESSAGE})\) \(\equiv\)

WHEN \(mm \in \text{dom(sender)}\)
THEN \(\text{sender} := \text{sender} \cup \{mm \mapsto pp\}\)
END;

**Order** \((pp \in \text{PROCESS}, mm \in \text{MESSAGE})\) \(\equiv\)

WHEN \(mm \in \text{dom(sender)}\)
\(\land\) \(mm \notin \text{ran(tdeliver)}\)
\(\land\) \(\text{ran(tdeliver)} \subseteq \text{tdeliver}\{pp\}\)
THEN \(\text{tdeliver} := \text{tdeliver} \cup \{pp \mapsto mm\}\)
\(\mathrel{\|}\) \(\text{totalorder} := \text{totalorder} \cup \{\text{ran(tdeliver)} \times \{mm\}\}\)
\(\mathrel{\|}\) \(\text{delorder}(pp) := \text{delorder}(pp) \cup \{\text{tdeliver}\{pp\} \times \{mm\}\}\)
END;

**TODeliver** \((pp \in \text{PROCESS}, mm \in \text{MESSAGE})\) \(\equiv\)

WHEN \(mm \in \text{dom(sender)}\)
\(\land\) \(mm \notin \text{ran(tdeliver)}\)
\(\land\) \(pp \mapsto mm \notin \text{tdeliver}\)
\(\land\) \(\forall m. (m \in \text{MESSAGE} \land (m \mapsto mm) \in \text{totalorder} \Rightarrow (pp \mapsto m) \in \text{tdeliver})\)
THEN \(\text{tdeliver} := \text{tdeliver} \cup \{pp \mapsto mm\}\)
\(\mathrel{\|}\) \(\text{delorder}(pp) := \text{delorder}(pp) \cup \{\text{tdeliver}\{pp\} \times \{mm\}\}\)
END

**Figure 5.5**: TotalOrder Abstract Model: Events
Constructing a Total Order

The event *Order* models the delivery of a message *mm* at a process *pp* when it is delivered for the *first* time. The following guards of this event ensure that the message *mm* has not been delivered elsewhere and that each message delivered at any other process has also been delivered to this process(*pp*).

\[ \text{mm} \notin \text{ran(tdeliver)} \]
\[ \text{ran(tdeliver)} \subseteq \text{tdeliver}[\{\text{pp}\}] \]

Later in the refinement we show that this is a function of a designated process called *sequencer*. As a consequence of the occurrence of the *Order* event, the message *mm* is delivered to the process *pp* and variable *totalorder* is updated by mappings in (\(\text{ran(tdeliver)} \times \text{mm}\)). This indicates that all messages delivered at any process in the system are ordered before *mm*. Similarly, the delivery order at the process is also updated such that all messages delivered at any process precede *mm*. It can be noticed that the total order for a message is built when it is delivered to a process for the *first* time.

The event \(TODeliver(pp,mm)\) models the delivery of a message *mm* to a process *pp* respecting the total order. The guard \(mm \in \text{ran(tdeliver)}\) implies that *mm* has already been delivered to at least one process. The guards of this event also ensure that all messages, which precede *mm* in the abstract total order, have also been delivered to *pp*.

### 5.4 Invariant Properties of Total Order

After building the model of a total order broadcast, our goal was to formally verify that our model preserves the total ordering properties defined in the section 2.3. The agreement and total order requirements imply that all correct processes eventually deliver all messages in the same order [52]. Thus, we add following invariant as a primary invariant to our model.

\[(m1 \mapsto m2) \in \text{delorder}(p) \Rightarrow (m1 \mapsto m2) \in \text{totalorder}\]

This invariant states that if a process delivers any two messages then their delivery order at that process corresponds to their abstract total order. Also, to prove that the total order also preserves transitivity, we add the following as a primary invariant to our model.

\[(m1 \mapsto m2) \in \text{totalorder} \land (m2 \mapsto m3) \in \text{totalorder} \Rightarrow (m1 \mapsto m3) \in \text{totalorder}\]
Due to the addition of these invariants to our model as primary invariants, the B tool generates several proof obligations associated with various events. In the next section, we outline how the proof obligations generated by the interactive prover guide us in discovering new invariants. Due to the large number of proof obligations generated by the prover only a few important proof obligations are outlined. A complete list of a set of primary and secondary invariants is outlined in Fig. 5.6 and 5.7.

5.4.1 Proving Total Ordering Property

In order to prove the total ordering property we add following primary invariant to our model.

$$\forall (m_1, m_2, p). ((m_1 \leftrightarrow m_2) \in delorder(p) \Rightarrow (m_1 \leftrightarrow m_2) \in totalorder)$$

When we added this invariant to our model two proof obligations were generated associated with the events Order and TODeliver. The proof obligation associated with the event Order was discharged using interactive prover, however the proof obligation associated with TODeliver could not be discharged. Following is the simplified form of this proof obligation generated by the interactive prover.

$$\text{TODeliver(PO1)}$$

\[
\begin{align*}
& p \mapsto m_1 \in tdeliver \land \\
& p \mapsto m_2 \notin tdeliver \land \\
& m_2 \in \text{ran}(tdeliver) \\
& \Rightarrow \\
& m_1 \mapsto m_2 \in totalorder
\end{align*}
\]

This states that if process p has delivered m1 but m2 has been delivered elsewhere then m1 precedes m2 in total order. In order to discharge this proof obligation, we add an invariant to our model given as Inv-2 in Fig. 5.6. The addition of Inv-2 was sufficient to discharge PO1, however a new proof obligation associated with TODeliver was generated due to the addition of Inv-2. Following is the simplified form of the proof obligation.

$$\text{TODeliver(PO2)}$$

\[
\begin{align*}
& m_1 \in \text{ran}(tdeliver) \land \\
& m_2 \in \text{ran}(tdeliver) \land \\
& m_2 \mapsto m_1 \notin totalorder \\
& \Rightarrow \\
& m_1 \mapsto m_2 \in totalorder
\end{align*}
\]
This proof obligation required us to prove that if two messages \( m_1 \) and \( m_2 \) are delivered to any process(es) in the system then a total order exists among them, i.e., either \( m_1 \) precedes \( m_2 \) or \( m_2 \) precedes \( m_1 \) in the abstract total order. In order to discharge the proof obligation we add another invariant \( \text{Inv-3} \) to our model. Addition of this invariant to the model generate further proof obligations.

After four rounds of invariant strengthening we arrive at the set of invariants given in Fig. 5.6 which were sufficient to discharge all proof obligations. It may be noted that the invariant \( \text{Inv-1} \) is a primary invariant which states the total ordering property while the other invariants are discovered when the proof obligations with respect to \( \text{Inv-1} \) are discharged. A brief description of the properties is given below.

- The total ordering property is given as \( \text{Inv-1} \). It states that all processes deliver the messages in the same abstract total order.
- If a process \( p \) has delivered \( m_1 \), but not \( m_2 \), and if \( m_2 \) was delivered to at least one process elsewhere in the system then \( m_1 \) precedes \( m_2 \) in total order (\( \text{Inv-2} \)).

<table>
<thead>
<tr>
<th>Invariants</th>
<th>Required By</th>
</tr>
</thead>
<tbody>
<tr>
<td>/<em>Inv-1</em>/</td>
<td>((m_1 \rightarrow m_2) \in \text{delorder}(p)) (\Rightarrow (m_1 \rightarrow m_2) \in \text{totalorder})</td>
</tr>
<tr>
<td>/<em>Inv-2</em>/</td>
<td>((p \rightarrow m_1) \in \text{tdeliver} \land (p \rightarrow m_2) \notin \text{tdeliver} \land m_2 \in \text{ran(tdeliver)}) (\Rightarrow (m_1 \rightarrow m_2) \in \text{totalorder})</td>
</tr>
<tr>
<td>/<em>Inv-3</em>/</td>
<td>(m_1 \in \text{ran(tdeliver)} \land m_2 \in \text{ran(tdeliver)} \land (m_2 \rightarrow m_1) \notin \text{totalorder}) (\Rightarrow (m_1 \rightarrow m_2) \in \text{totalorder})</td>
</tr>
<tr>
<td>/<em>Inv-4</em>/</td>
<td>((p \rightarrow m_1) \in \text{tdeliver} \land (p \rightarrow m_2) \in \text{tdeliver} \land (m_2 \rightarrow m_1) \notin \text{totalorder}) (\Rightarrow (m_1 \rightarrow m_2) \in \text{totalorder})</td>
</tr>
<tr>
<td>/<em>Inv-5</em>/</td>
<td>((p_1 \rightarrow m_1) \in \text{tdeliver} \land (p_1 \rightarrow m_2) \notin \text{tdeliver} \land (p_2 \rightarrow m_1) \in \text{tdeliver} \land (p_2 \rightarrow m_2) \notin \text{tdeliver}) (\Rightarrow (m_1 \rightarrow m_2) \in \text{totalorder})</td>
</tr>
<tr>
<td>/<em>Inv-6</em>/</td>
<td>(m \in \text{MESSAGE} \Rightarrow m \rightarrow m \in \text{totalorder})</td>
</tr>
</tbody>
</table>

Figure 5.6: Invariants-I
– If two messages \( m_1 \) and \( m_2 \) have been delivered anywhere in the system then a total order exists among them, such that, either \( m_1 \) precedes \( m_2 \) or \( m_2 \) precedes \( m_1 \) in total order. \((\text{Inv-3})\)

– If a process \( p \) has delivered two message \( m_1 \) and \( m_2 \) then either \( m_1 \) precedes \( m_2 \) or \( m_2 \) precedes \( m_1 \) in total order \((\text{Inv-4})\).

– Given two processes \( p_1 \) and \( p_2 \), then for any two messages \( m_1 \) and \( m_2 \) if the process \( p_2 \) has delivered both messages and \( p_1 \) has delivered \( m_1 \) but not \( m_2 \) then \( m_1 \) precedes \( m_2 \) in total order \((\text{Inv-5})\).

– A total order is irreflexive \((\text{Inv-6})\).

### 5.4.2 Proving Transitivity Property

Our next step was to verify that our model of the total order broadcast also preserves transitive properties on the abstract total order. In order to verify that total order is transitive, we add following to the list of the invariants.

\[
\begin{align*}
(m_1 \rightarrow m_2) & \in \text{totalorder} \land \\
(m_2 \rightarrow m_3) & \in \text{totalorder} \\
\Rightarrow \\
(m_1 \rightarrow m_3) & \in \text{totalorder}
\end{align*}
\]

Addition of this invariant generates proof obligations associated with the events Broadcast, Order and TODEliver. We are able to discharge proofs related to the Broadcast event using the interactive prover. However, the following Proof Obligation associated with Order event could not be discharged by the automatic prover.

\[\text{Order}(pp, mm)PO3\]

\[
\begin{align*}
(m_1 \rightarrow m_2) & \in \text{totalorder} \land \\
(p \rightarrow m_2) & \in \text{tdeliver} \\
\Rightarrow \\
(p \rightarrow m_1) & \in \text{tdeliver}
\end{align*}
\]

This property on the messages states that for two message \( m_1 \) and \( m_2 \) if \( m_1 \) is totally ordered before \( m_2 \) then for any process \( p \) which has delivered \( m_2 \) implies that it has also delivered \( m_1 \). In order to discharge this proof obligations, we add Inv-8 given in Fig. 5.7.

When we add this invariant to our model it generates further proof obligations associated with the events Broadcast, Order and TODEliver. The proof obligation associated with TODEliver is discharged using the automatic prover. The simplified form of proof obligation associated with the events Broadcast which cannot be discharged automatically...
is given below.

\[
\text{BroadCast}(pp, mm)\ PO4
\begin{align*}
\text{Inv-8} \land \\
mm \notin \text{dom(sender)} \land \\
(pp \mapsto m2) \in \text{tdeliver} \land \\
(mm \mapsto m2) \in \text{totalorder} \land \\
m1 = mm \land \\
m2 \neq mm \\
\Rightarrow \\
(pp \mapsto mm) \in \text{tdeliver}
\end{align*}
\]

It can be noticed that there is a contradiction in the hypotheses of this proof obligation, i.e., the hypothesis \(mm \notin \text{dom(sender)}\) and \((mm \mapsto m2) \in \text{totalorder}\) can not be true simultaneously because of our assumption that a total order is built only when a message has been sent out. Similarly, the goal \((pp \mapsto mm) \in \text{tdeliver}\) cannot be proved under the hypothesis \(mm \notin \text{dom(sender)}\). Thus, we add the following invariant(s) to our model given as Inv-9,10 in Fig. 5.7.

\[
\forall m . ( m \in ( \text{dom(totalorder}) \cup \text{ran(totalorder}) ) \Rightarrow m \in \text{ran(tdeliver)})
\]

\[
\forall (m). (m \notin \text{dom(sender)} \Rightarrow m \notin \text{ran(totalorder)})
\]

\[
\forall (m). (m \notin \text{dom(sender)} \Rightarrow m \notin \text{dom(totalorder)})
\]

\[
\text{ran(tdeliver)} \subseteq \text{dom(sender)}
\]

Addition of these invariants was sufficient to discharge all proof obligations. Therefore after four iterations of invariant strengthening we arrive at a set of invariant that is sufficient to discharge all proof obligations generated due the addition of Inv-7. The full set of invariants is given in Fig. 5.7. A brief description of these properties are outlined below.

- A total order is transitive (Inv-7).

- For any two messages \(m1\) and \(m2\) where \(m1\) is totally ordered before \(m2\) then a process \(p\) which delivered \(m2\) has also delivered \(m1\) (Inv-8).

- The total order is built for those messages which have been delivered to at least one process (Inv-9).

- A total order cannot be build for messages which have not been sent and each message delivered at any process must be a sent message (Inv-10).
### Invariants

<table>
<thead>
<tr>
<th>Invariants</th>
<th>Required By</th>
</tr>
</thead>
<tbody>
<tr>
<td>/<em>Inv-7</em>/ ((ml \rightarrow m2) \in \text{totalorder} \land (m2 \rightarrow m3) \in \text{totalorder}) \rightarrow (ml \rightarrow m3) \in \text{totalorder}</td>
<td>Primary Invariant</td>
</tr>
<tr>
<td>/<em>Inv-8</em>/ ((m1 \rightarrow m2) \in \text{totalorder} \land (p \rightarrow m2) \in \text{tdeliver}) \rightarrow (p \rightarrow m1) \in \text{tdeliver}</td>
<td>Broadcast, Order, TODeliver</td>
</tr>
<tr>
<td>/<em>Inv-9</em>/ (m \in (\text{dom (totalorder)} \cup \text{ran (totalorder)})) \rightarrow m \in \text{ran (tdeliver)}</td>
<td>Order</td>
</tr>
<tr>
<td>/<em>Inv-10</em>/ (m \notin \text{dom (sender)} \Rightarrow m \notin \text{dom (totalorder)}) \land (m \notin \text{dom (sender)} \Rightarrow m \notin \text{ran (totalorder)}) \land \text{ran (tdeliver)} \subset \text{dom (sender)}</td>
<td>Broadcast, Order, TODeliver</td>
</tr>
</tbody>
</table>

**Figure 5.7: Invariants-II**

## 5.5 Total Order Refinements

In the previous section we have given an overview of the abstract model of total order broadcast. In this section we present a overview of our refinement chain consisting of six levels. A brief outline of each refinement step is given below.

**L1** This consists of an abstract model of total order broadcast. In this model, abstract total order is constructed when a message is delivered to a process for the first time. At all other processes a message is delivered in the total order. We have already outlined this level in section 5.3.

**L2** This is a refinement of the abstract model which introduces the notion of the *sequencer*. In this refinement we outline how a total order on the messages is constructed by the *sequencer*.

**L3** This is a simple refinement giving a more concrete specification of the *Order* event. Through this refinement we illustrate that a total order can be built using the messages delivered to the sequencer rather than all sites.

**L4** In this refinement we introduce the notion of *computation* messages and *sequence numbers*. Global sequence numbers of the computation messages are generated by the sequencer. The delivery of messages is done based on the sequence numbers.

**L5** In this refinement we introduce the notion of *control* messages. We also introduce the relationship of each *computation* message to the *control* messages.
A new event *Receive Control* is introduced. We illustrate that a process other than sequencer can deliver a *computation* message only if it has received a *control* message for it.

### 5.5.1 First Refinement: Introducing the Sequencer

In the first refinement, given in Fig. 5.8, we introduce the notion of a sequencer. The sequencer is defined as a constant for this model as \( \text{sequencer} \in \text{PROCESS} \).

\[
\text{Broadcast}(pp \in \text{PROCESS}, mm \in \text{MESSAGE}) \equiv \\
\text{WHEN } mm \notin \text{dom}(sender) \\
\text{THEN } \text{sender} := \text{sender} \cup \{mm \mapsto pp\} \\
\text{END};
\]

\[
\text{Order}(pp \in \text{PROCESS}, mm \in \text{MESSAGE}) \equiv \\
\text{WHEN } pp = \text{sequencer} \\
\quad \land \ mm \in \text{dom}(sender) \\
\quad \land \ (\text{sequencer} \mapsto mm) \notin \text{tdeliver} \\
\text{THEN } \text{tdeliver} := \text{tdeliver} \cup \{pp \mapsto mm\} \\
\quad \parallel \ \text{totalorder} := \text{totalorder} \cup (\text{ran(tdeliver}) \times \{mm\}) \\
\text{END};
\]

\[
\text{TODeliver}(pp \in \text{PROCESS}, mm \in \text{MESSAGE}) \equiv \\
\text{WHEN } pp \neq \text{sequencer} \\
\quad \land \ mm \in \text{dom}(sender) \\
\quad \land \ mm \in \text{ran(tdeliver}) \\
\quad \land \ pp \mapsto mm \notin \text{tdeliver} \\
\quad \land \ \forall m.(m \in \text{MESSAGE} \land (m \mapsto mm) \in \text{totalorder} \Rightarrow (pp \mapsto m) \notin \text{tdeliver}) \\
\text{THEN } \text{tdeliver} := \text{tdeliver} \cup \{pp \mapsto mm\} \\
\text{END}
\]

**Figure 5.8: TotalOrder Refinement-I**

As shown in the refined specification of *Order* event given in Fig. 5.8, a message is first delivered to the sequencer process. It can be noticed that the following guards in the abstract specification

\[
mm \notin \text{ran(tdeliver)} \\
\text{ran(tdeliver)} \subseteq tdeliver[[\{pp\}]]
\]

are replaced by following.

\[
pp = \text{sequencer} \\
(\text{sequencer} \mapsto mm) \notin \text{tdeliver}
\]
The replacement of the guards in the Order event generates new proof obligations. Using the same approach of invariant discovery as outlined in section 5.4, we arrived at a set of invariants that was sufficient to discharge all proof obligations. These invariants are given in Fig. 5.9.

<table>
<thead>
<tr>
<th>Invariants</th>
<th>Required By</th>
</tr>
</thead>
<tbody>
<tr>
<td>/<em>Inv-11</em>/ (sequencer $\mapsto$ m) $\in$ tdeliver $\Rightarrow$ m $\notin$ ran(tdeliver)</td>
<td>Order, TODeliver</td>
</tr>
<tr>
<td>/<em>Inv-12</em>/ m $\in$ dom(totalorder) $\Rightarrow$ (sequencer $\mapsto$ m) $\in$ tdeliver</td>
<td>Order</td>
</tr>
<tr>
<td>/<em>Inv-13</em>/ m $\in$ ran(totalorder) $\Rightarrow$ (sequencer $\mapsto$ m) $\in$ tdeliver</td>
<td>Order</td>
</tr>
</tbody>
</table>

**Figure 5.9:** TotalOrder Refinement-I: Invariants

A brief description of these invariants is given in the following steps.

- A message not delivered to the sequencer has not been delivered elsewhere. *(Inv-11)*
- If a total order on any message m has been constructed then it must have been delivered to the sequencer. *(Inv-12,13)*

Similarly, it can be noticed that a guard $pp \neq$ sequencer is added in the specifications of TODeliver event. Thus, on occurrence of the event TODeliver, a message mm is delivered to a process other than the sequencer.

### 5.5.2 Second Refinement: Refinement of Order event

This refinement outlines a more concrete specification of the Order event. Through this refinement we illustrate that a total order can be built using the messages delivered to the sequencer. As shown in the Fig. 5.8, a total order is generated as follows:

\[
\text{totalorder} := \text{totalorder} \cup (\text{ran}(\text{tdeliver}) \times \{mm\})
\]

It states that all messages delivered at any process are ordered before the new message mm. In the refined Order event the totalorder is constructed as follows:

\[
\text{totalorder} := \text{totalorder} \cup (\text{tdeliver}[\{\text{sequencer}\}] \times \{mm\})
\]

This states that all messages delivered to the sequencer are ordered before the new message mm. The specifications of this refinement are given in the Fig. 5.10.
Chapter 5 Total Order Broadcast

**Broadcast** \( (pp \in \text{PROCESS}, mm \in \text{MESSAGE}) \)

\[
\text{WHEN} \quad mm \not\in \text{dom}(sender) \\
\text{THEN} \quad \text{sender} := \text{sender} \cup \{mm \mapsto pp\} \\
\text{END;}
\]

**Order** \( (pp \in \text{PROCESS}, mm \in \text{MESSAGE}) \)

\[
\text{WHEN} \quad pp = \text{sequencer} \\
\quad \land \quad mm \in \text{dom}(sender) \\
\quad \land \quad (\text{sequencer} \mapsto mm) \not\in tdeliver \\
\text{THEN} \quad tdeliver := tdeliver \cup \{pp \mapsto mm\} \\
\quad \parallel \quad \text{totalorder} := \text{totalorder} \cup (tdeliver[\{\text{sequencer}\}] \times \{mm\}) \\
\text{END;}
\]

**TODeliver** \( (pp \in \text{PROCESS}, mm \in \text{MESSAGE}) \)

\[
\text{WHEN} \quad pp \not= \text{sequencer} \\
\quad \land \quad mm \in \text{dom}(sender) \\
\quad \land \quad mm \in \text{ran}(tdeliver) \\
\quad \land \quad pp \mapsto mm \not\in tdeliver \\
\quad \land \quad \forall m. (m \in \text{MESSAGE} \land (m \mapsto mm) \in \text{totalorder}) \\
\quad \Rightarrow (pp \mapsto m) \in tdeliver \\
\text{THEN} \quad tdeliver := tdeliver \cup \{pp \mapsto mm\} \\
\text{END}
\]

**Figure 5.10:** TotalOrder Refinement-II : Refined Order Event

The replacement of the operations in the event *Order* generates proof obligations which require us to prove that the message delivered elsewhere in the system has also been delivered to the sequencer. In order to discharge the proof obligations we add the invariant *Inv-14* given in the Fig. 5.11. This invariant was sufficient to discharge the proof obligations.

<table>
<thead>
<tr>
<th>Invariants</th>
<th>Required By</th>
</tr>
</thead>
<tbody>
<tr>
<td>/<em>Inv-14</em>/ ran(tdeliver) \subseteq tdeliver[{sequencer}]</td>
<td>Order, TODeliver</td>
</tr>
</tbody>
</table>

**Figure 5.11:** TotalOrder Refinement-II : Invariants

### 5.5.3 Third Refinement : Introducing Sequence Numbers

In the third refinement, given in Fig. 5.12, we introduce the notion of a computation message and the sequence numbers. The messages broadcast by the processes which
need to be delivered in the total order are called computation messages. In this intermediate refinement step, the sequence number of a computation message is assigned by the sequencer. This refinement introduces the following new variables.

\[
\begin{align*}
\text{computation} & \subseteq \text{MESSAGE} \\
\text{seqno} & \in \text{computation} \rightarrow \text{Natural} \\
\text{counter} & \in \text{Natural}
\end{align*}
\]

The variable \textit{seqno} is used to assign the sequence number to the computation messages. The \textit{counter}, initialized to \textit{zero}, is maintained by the sequencer process and incremented by \textit{one} each time a control message is sent out by the \textit{sequencer} process. It can be noted in the specification of the \textit{TODeliver} event that these messages are delivered to the processes other than the sequencer in their sequence numbers.

\[
\begin{align*}
\text{Broadcast} \ (pp \in \text{PROCESS}, \ mm \in \text{MESSAGE}) & \equiv \\
\text{WHEN} \ mm \notin \text{dom}(\text{sender}) & \\
\text{THEN} \ & \text{sender} := \text{sender} \cup \{mm \mapsto pp\} \\
& \quad \text{computation} := \text{computation} \cup \{mm\}
\end{align*}
\]

\[
\begin{align*}
\text{Order} \ (pp \in \text{PROCESS}, mm \in \text{MESSAGE}) & \equiv \\
\text{WHEN} \ pp = \text{sequencer} & \quad \wedge \ mm \in \text{dom}(\text{sender}) \\
& \quad \wedge \ mm \in \text{computation} \\
& \quad \wedge \ (\text{sequencer} \mapsto mm) \notin \text{tdeliver} \\
\text{THEN} \ & \text{totalorder} := \text{totalorder} \cup (\text{tdeliver}[\{\text{sequencer}\}] \times \{mm\}) \\
& \quad \text{tdeliver} := \text{tdeliver} \cup \{pp \mapsto mm\} \\
& \quad \text{seqno} := \text{seqno} \cup \{mm \mapsto \text{counter}\} \\
& \quad \text{counter} := \text{counter} + 1
\end{align*}
\]

\[
\begin{align*}
\text{TODeliver} \ (pp \in \text{PROCESS}, mm \in \text{MESSAGE}) & \equiv \\
\text{WHEN} \ pp \neq \text{sequencer} & \quad \wedge \ mm \in \text{dom}(\text{sender}) \\
& \quad \wedge \ mm \in \text{ran}(\text{tdeliver}) \\
& \quad \wedge \ pp \mapsto mm \notin \text{tdeliver} \\
& \quad \forall m. (m \in \text{computation} \wedge (\text{seqno}(m) < \text{seqno}(mm)) \\
& \quad \Rightarrow (pp \mapsto m) \in \text{tdeliver}) \\
\text{THEN} \ & \text{tdeliver} := \text{tdeliver} \cup \{pp \mapsto mm\}
\end{align*}
\]

\textit{Figure 5.12: TotalOrder Refinement-III}
It can be noticed that following guard in the abstract TODeliver

\[(m \mapsto mm) \in \text{totalorder} \Rightarrow (pp \mapsto m) \in \text{tdeliver}\]

is replaced by

\[\text{seqno}(m) < \text{seqno}(mm) \Rightarrow (pp \mapsto m) \in \text{tdeliver}\]

The change of the guards in the TODeliver event generates new proof obligations. These proof obligations are discharged by adding new invariants given in Fig. 5.13 to the model. Invariant Inv-15 states that if \(m_1\) precedes \(m_2\) in the abstract total order then the sequence number assigned to \(m_1\) is less than the sequence number assigned to \(m_2\). The invariant Inv-16 states that if a computation message has been assigned a sequence number then the sequencer must have delivered it.

<table>
<thead>
<tr>
<th>Invariants</th>
<th>Required By</th>
</tr>
</thead>
<tbody>
<tr>
<td>/<em>Inv-15</em>/ (m_1 \rightarrow m_2 \in \text{totalorder}) (\Rightarrow \text{seqno}(m_1) &lt; \text{seqno}(m_2))</td>
<td>Order,TODeliver</td>
</tr>
<tr>
<td>/<em>Inv-16</em>/ (m \in \text{computation} \land m \in \text{dom(seqno)}) (\Rightarrow \text{sequencer} \mapsto m \in \text{tdeliver})</td>
<td>Order,TODeliver</td>
</tr>
</tbody>
</table>

**Figure 5.13**: TotalOrder Refinement-III : Invariants

### 5.5.4 Fourth Refinement : Introducing Control Messages

In this refinement we introduce the notion of control messages. A control message is broadcast by the sequencer process for each computation message. In this refinement, a process broadcasts a computation message \(mm\) to all processes including the sequencer. Upon delivery of this message, the sequencer assigns it a sequence number and broadcast its control message. All processes except the sequencer deliver the corresponding computation messages in the order of the sequence numbers. This refinement consists of following new state variables typed as follows:

\[\text{control} \subseteq \text{MESSAGE}\]

\[\text{messcontrol} \in \text{control} \mapsto \text{computation}\]

The variables control and computation are used to cast a message as either a computation or a control message. The set control contains the control messages sent by the sequencer. The variable messcontrol is a partial injective function which defines the relationship between a control message and its computation message. A mapping \((m_1 \mapsto m_2) \in \text{messcontrol}\) indicates that message \(m_1\) is the control message related to the computation
message \( m_2 \). Since \( \text{messcontrol} \) is defined as a partial injective function, it also implies that there can only be one control message for each computation message and vice-versa. The set \( \text{ran}(\text{messcontrol}) \) contains the computation messages for which control messages have been sent by the sequencer. The refined model is given in the Fig. 5.14.

The guard \( \text{mm} \in \text{ran}((\text{tdeliver})) \) of the \( \text{TODeliver} \) event is replaced by the guard \( \text{mm} \in \text{ran}(\text{messcontrol}) \) in the Refinement-IV. This indicates that a computation message is delivered to a process other than a sequencer only if its control message has been sent out by the sequencer. Later in the refinement we replace this guard stating that a computation message is delivered to a process other than the sequencer only if its control message has been received by this process. The change in the guards of \( \text{Order} \) and \( \text{TODeliver} \) events generates proof obligations which are discharged by adding a set of invariants given in Fig. 5.15 to the model.

\[
\begin{align*}
\text{Broadcast} & ( pp \in \text{PROCESS} , \text{mm} \in \text{MESSAGE} ) \equiv \\
& \text{WHEN} \quad \text{mm} \notin \text{dom}(\text{sender}) \\
& \quad \text{THEN} \quad \text{sender} := \text{sender} \cup \{ \text{mm} \mapsto pp \} \\
& \quad \quad \| \text{computation} := \text{computation} \cup \{ \text{mm} \} \\
& \text{END} ;
\end{align*}
\]

\[
\begin{align*}
\text{Order} & ( pp \in \text{PROCESS} , \text{mm} \in \text{MESSAGE}, \text{mc} \in \text{MESSAGE} ) \equiv \\
& \text{WHEN} \quad pp = \text{sequencer} \\
& \quad \land \quad \text{mm} \in \text{dom}(\text{sender}) \\
& \quad \land \quad \text{mm} \in \text{computation} \\
& \quad \land \quad (\text{sequencer} \mapsto \text{mm}) \notin \text{tdeliver} \\
& \quad \land \quad \text{mc} \notin \text{dom}(\text{messcontrol}) \\
& \quad \land \quad \text{mm} \notin \text{ran}(\text{messcontrol}) \\
& \quad \text{THEN} \quad \text{totalorder} := \text{totalorder} \cup (\text{tdeliver}[\{\text{sequencer}\}] \times \{\text{mm}\}) \\
& \quad \quad \| \text{tdeliver} := \text{tdeliver} \cup \{ pp \mapsto \text{mm} \} \\
& \quad \quad \| \text{control} := \text{control} \cup \{ \text{mc} \} \\
& \quad \quad \| \text{messcontrol} := \text{messcontrol} \cup \{ \text{mc} \mapsto \text{mm} \} \\
& \quad \quad \| \text{seqno} := \text{seqno} \cup \{ \text{mm} \mapsto \text{counter} \} \\
& \quad \quad \| \text{counter} := \text{counter} + 1 \\
& \text{END} ;
\end{align*}
\]

\[
\begin{align*}
\text{TODeliver} & ( pp \in \text{PROCESS} , \text{mm} \in \text{MESSAGE} ) \equiv \\
& \text{WHEN} \quad pp \neq \text{sequencer} \\
& \quad \land \quad \text{mm} \in \text{dom}(\text{sender}) \\
& \quad \land \quad \text{mm} \in \text{ran}(\text{messcontrol}) \\
& \quad \land \quad pp \mapsto \text{mm} \notin \text{tdeliver} \\
& \quad \land \quad \forall m. ( m \in \text{computation} \land ( \text{seqno}(m) < \text{seqno}(\text{mm}) ) ) \\
& \quad \quad \Rightarrow ( pp \mapsto m ) \in \text{tdeliver} \\
& \quad \text{THEN} \quad \text{tdeliver} := \text{tdeliver} \cup \{ pp \mapsto \text{mm} \} \\
& \text{END}
\end{align*}
\]

Figure 5.14: TotalOrder Refinement-IV
Invariants | Required By
--- | ---
/*Inv-17*/ ran(messcontrol) ⊆ ran(tdeliver) | Order,TODeliver
/*Inv-18*/ ran(messcontrol) ⊆ computation | Order,TODeliver

Figure 5.15: TotalOrder Refinement-IV : Invariants

5.5.5 Fifth Refinement : Introducing Receive Control Event

A new event *ReceiveControl* is introduced in this refinement. This event models receiving a control message at a process. A new variable *receive* is also introduced in this refinement typed as follows:

receive ∈ PROCESS ↔ control

A mapping *p* ↦ *m* ∈ *receive* indicates that process *p* has received a control message *m*. The specifications of the refined events are given in 5.16.

**ReceiveControl** (*pp* ∈ PROCESS, *mc* ∈ MESSAGE) ≡

WHEN *mc* ∈ control

∧ (pp ↦ mc) ∉ *receive*

THEN receive := receive ∪ {pp ↦ mc}

END

**TODeliver** (*pp* ∈ PROCESS, *mm* ∈ MESSAGE) ≡

WHEN *pp* ≠ sequencer

∧ *mm* ∈ computation

∧ (pp ↦ mm) ∉ tdeliver

∧ (pp ↦ messcontrol⁻¹ (mm)) ∈ receive

∧ ∀m.( m ∈ computation ∧ (seqno(m) < seqno(mm))

⇒ (pp ↦ m) ∈ tdeliver)

THEN tdeliver := tdeliver ∪ {pp ↦ mm}

END

Figure 5.16: Refinement-V : Receive Control

As shown in the specifications, variable *receive* is updated when a control message is received at a process. The event *TODeliver* models the delivery of a *computation* message to a process. Also, as shown in the *TODeliver* event given in Fig. 5.16, the guard

mm ∈ ran(messcontrol)
is replaced by the following:

\[(pp \mapsto \text{messcontrol}^{-1}(mm)) \in \text{receive}\]

This guard of the \text{TODeliver} event ensures that a process \(pp\) delivers a computation message \(mm\) only when its corresponding control message has been received by the process \(pp\). The change in the guards generates proof obligations associated with the event \text{TODeliver}. In order to discharge these proof obligations we add the following to the list of invariants.

<table>
<thead>
<tr>
<th>Invariants</th>
<th>Required By</th>
</tr>
</thead>
<tbody>
<tr>
<td>/<em>Inv-19</em>/ (m \in \text{computation} \land \text{messcontrol}^{-1}(m) \in \text{receive}) (\Rightarrow m \in \text{ran(messcontrol)})</td>
<td>\text{Order,TOdeliver}</td>
</tr>
</tbody>
</table>

\textbf{Figure 5.17: TotalOrder Refinement-V: Invariants}

5.6 Conclusions

In this chapter we presented the abstract specifications of a total order broadcast. The \textit{Broadcast} variant of a fixed sequencer protocol is used for the development of a system of total order broadcast. In the abstract model, we outline how an abstract total order is constructed on the messages. Precisely, an abstract total order is constructed at the first ever delivery of a message to any process in the system. All other processes deliver that message in the abstract total order. We have also outlined invariant properties of the abstract total order broadcast and outline how new invariants are discovered while discharging the proof obligations.

In the first refinement we introduce the notion of the sequencer process and show that a message is first delivered to the sequencer and a total order is built by the sequencer. In the second refinement, we precisely outline that an abstract total order is constructed using the messages delivered to the sequencer rather than all processes. In the third refinement, we provide further detail of the protocols steps and introduce the notion of sequence numbers. A process delivers a message based on sequence numbers rather than using abstract total order. In the fourth refinement, a notion of control messages is introduced. We also introduce the relationship of computation and control messages. A control message is sent by a sequencer process for each computation message after the delivery of a computation message to the sequencer process. In the fifth refinement, we illustrate how a computation message is delivered to a process using its sequence number after delivering a control message to that process.
This case study illustrates how an incremental approach to system development can be used to obtain more concrete specifications. Powerful tool support helped us to discover several new invariants that help to understand why a total order broadcast can correctly be implemented using sequence numbers. A clear relationship between computation and control messages is outlined to indicate that our system generates exactly one control message for each computation message. The full refinement chain is outlined in the Appendix-D. The overall proof statistics are given in Table 5.1. Approximately seventy five percent of the proofs were discharged by the automatic prover, the rest were discharged by using the interactive prover of the B tool.

<table>
<thead>
<tr>
<th>Machine</th>
<th>Total POs</th>
<th>Completely Automatic</th>
<th>Required Interaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract Model</td>
<td>48</td>
<td>29</td>
<td>19</td>
</tr>
<tr>
<td>Refinement1</td>
<td>19</td>
<td>16</td>
<td>03</td>
</tr>
<tr>
<td>Refinement2</td>
<td>2</td>
<td>2</td>
<td>00</td>
</tr>
<tr>
<td>Refinement3</td>
<td>18</td>
<td>14</td>
<td>04</td>
</tr>
<tr>
<td>Refinement4</td>
<td>15</td>
<td>14</td>
<td>01</td>
</tr>
<tr>
<td>Refinement5</td>
<td>04</td>
<td>04</td>
<td>00</td>
</tr>
<tr>
<td>Overall</td>
<td>106</td>
<td>79</td>
<td>27</td>
</tr>
</tbody>
</table>

Table 5.1: Proof Statistics- Total Order Broadcast
Chapter 6

Causally and Totally Ordered Broadcast

6.1 Introduction

In this chapter we extend the system of a causal order broadcast to a system of total causal order broadcast\(^1\) such that the delivery of the messages also satisfies a total order on the messages in addition to a causal order. A total causal order broadcast not only preserves the causality among the messages but also delivers them in a total order. Our model is based on the Broadcast Broadcast variant of a fixed sequencer algorithm and it uses a notion of the sequencer that builds a total order on the messages. The advantage of processing update transactions over a total causal order broadcast is that the database always remains in a consistent state due to the guarantees of providing a total order on the delivery of update messages. Also, this broadcast preserves the causality among the update messages.

6.2 Mechanism for building a Total Causal Order

In this section we outline the mechanism for building a total causal order on computation messages. In our model, a computation message is first delivered to a process in a causal order followed by another delivery in a total order. A process is said to codeliver a message when it is delivered following a causal order. Similarly, a process is said to todeliver a message when it is delivered following a total order. It may be noted that in our model, the todelivery of a message also corresponds to the delivery in a total causal order. The mechanism for implementing a total causal order is outlined through an example in the Fig. 6.1.

\(^1\)A reliable broadcast that satisfies both causal and total order is also known as Causal Atomic Broadcast.
Consider a broadcast of messages $M_1$ and $M_2$ by the processes $P_2$ and $P_3$ respectively. As shown in Fig. 6.1, broadcasts of $M_1$ and $M_2$ are related by a causal precedence relationship as given below:

$$broadcast(M_1) \rightarrow broadcast(M_2)$$

Since the broadcast of $M_1$ and $M_2$ is related with the causal precedence relationship, they are codelivered at other processes inclusive of the sequencer respecting their causal order as given below:

$$codeliver(M_1) \rightarrow codeliver(M_2)$$

Upon codelivery of the computation messages at the sequencer process, the sequencer assigns computation messages a sequence number and further broadcasts its sequence number through the control messages. It may be noted that the sequencer broadcasts the control messages of computation messages in the order they were codelivered at sequencer. Therefore, upon codelivery of $M_1$ and $M_2$ at the sequencer, the sequencer broadcasts control messages of $M_1$ and $M_2$ such that:

$$broadcast(controlmessage(M_1)) \rightarrow broadcast(controlmessage(M_2))$$

As all broadcasts in our model are done through a causal order broadcast, the control messages are also codelivered at all processes. Therefore, for any recipient of the control messages of $M_1$ and $M_2$ the following also holds:

$$codeliver(controlmessage(M_1)) \rightarrow codeliver(controlmessage(M_2))$$

\(^2\) "\(\rightarrow\)" denote precedes relation.
When a process code delivers a control message, it also to delivers the corresponding computation message. Thus the following also holds:

\[ \text{todeliver}(M_1) \rightarrow \text{todeliver}(M_2) \]

It can be noted that to delivery of a computation message represents a delivery following a total causal order. Since a sequencer assigns sequence numbers to the computation messages in the order they were code delivered to the sequencer, each computation message is to delivered to a process respecting the causality of their respective computation message.

Therefore, for any two computation messages \( M_1 \) and \( M_2 \) related by a causal precedence relationship, the following relationship holds which states that if the broadcast of \( M_1 \) precedes the broadcast of \( M_2 \) then the to delivery of \( M_1 \) also precedes to delivery of \( M_2 \).

\[ \text{broadcast}(M_1) \rightarrow \text{broadcast}(M_2) \Rightarrow \text{todeliver}(M_1) \rightarrow \text{todeliver}(M_2) \]

If the broadcast of any two computation messages is not related by a causal precedence relation (parallel messages), causal order broadcast is free to code deliver them in any order at the sequencer. However, the sequencer will assign them the separate sequence numbers guaranteeing that they are delivered to all processes in a total order. Therefore parallel messages are also delivered to a process in a total order in the total causal order broadcast.

In the following sections we present the incremental development of a system of a total causal order broadcast.

### 6.2.1 Overview of the Refinement Chain

The refinement chain for this development consists of five levels. An overview of the refinement steps is outlined below.

L1 In the abstract model we outline the construction of abstract total and causal order on the computation messages. This model is outlined in the Section 6.3.

L2 In this refinement we introduce the notion of vector clocks and sequence numbers. The abstract causal order and abstract total order are replaced with the vector clock rules and sequence numbers respectively. This refinement is outlined in Section 6.4.

L3 In the second refinement, we outline how the need for the generation of separate sequence numbers can correctly be implemented by the vector clock rules. This refinement is given in Section 6.5.
In this refinement, we present a simplification of the vector rules for updating the vector clock of recipient processes. This refinement is outlined in Section 6.6.

This is another refinement further simplifying the vector rules for updating vector clocks. This refinement also is outlined in Section 6.6.

### 6.3 Abstract Model of Total Causal Order Broadcast

In the abstract model we outline how an abstract causal order and abstract total order on the computation messages are constructed. We also outline how they are delivered in a total causal order.

#### 6.3.1 Abstract Variables

The initial part of the abstract model of total causal order broadcast is in Fig. 6.2 as a B machine. The specifications of the events of the machine are given in Fig. 6.3 and Fig. 6.4. As shown in Fig. 6.2, sequencer is defined as a constant, where a process is assigned as the sequencer non-deterministically. The variable sender is used to represent messages broadcast by a process.

The variable cdeliver represents the messages codelivered to the processes following a causal order. Similarly, the variable tdeliver represents the messages todelivered to the processes following a total order. This machine also consists of the following state variables typed as follows:

\[
\begin{align*}
\text{computation} & \subseteq \text{MESSAGE} \\
\text{control} & \subseteq \text{MESSAGE} \\
\text{messcontrol} & \in \text{control} \mapsto \text{computation}
\end{align*}
\]

The variables control and computation are used to cast a message as either a computation or a control message. The variable messcontrol is a partial injective function which defines the relationship between a computation message and its control message. A mapping \((m1 \mapsto m2) \in \text{messcontrol}\) indicates that the message \(m1\) is the control message related to the computation message \(m2\). Since messcontrol is defined as a partial injective function, it also implies that there can be only one control message for each computation message and vice-versa. The set \(\text{ran(messcontrol)}\) contains the computation messages for which control messages have been sent by the sequencer.

In order to represent the causally ordered delivery of the messages at a process, variable cdelorder is used. A mapping of the form \((m1 \mapsto m2) \in \text{cdelorder}(p)\) indicates that the process \(p\) has codelivered \(m1\) before \(m2\). Similarly, a mapping \((m1 \mapsto m2) \in \text{tdelorder}(p)\) indicates that the process \(p\) has todelivered \(m1\) before \(m2\). It may be noted
Chapter 6 Causally and Totally Ordered Broadcast

MACHINE TotalCausalOrder
CONSTANTS sequencer
PROPERTIES sequencer ∈ PROCESS

SETS PROCESS; MESSAGE;

VARIABLES sender, cdeliver, tdeliver, computation, control, messcontrol, causalorder, totalorder, cdelorder, tdelorder

INVARIANT sender ∈ MESSAGE → PROCESS
∧ cdeliver ∈ PROCESS ↔ MESSAGE
∧ tdeliver ∈ PROCESS ↔ MESSAGE
∧ computation ∈ MESSAGE
∧ control ∈ MESSAGE
∧ messcontrol ∈ control → computation
∧ causalorder ∈ MESSAGE ↔ MESSAGE
∧ totalorder ∈ MESSAGE ↔ MESSAGE
∧ cdelorder ∈ PROCESS→ (MESSAGE ↔ MESSAGE)
∧ tdelorder ∈ PROCESS→ (MESSAGE ↔ MESSAGE)

INITIALISATION
sender := ∅ || cdeliver := ∅ || tdeliver := ∅ ||
computation := ∅ || control := ∅ || messcontrol := ∅ ||
causalorder :=∅ || totalorder :=∅ ||
cdelorder := PROCESS × {∅} ||
tdelorder := PROCESS × {∅}

Figure 6.2: TotalCausalOrder: Initial Part

that a message may have been codelivered at a process but is still waiting for it to be todelivered.

6.3.2 Events in the abstract model

The Broadcast event given in the Fig. 6.3 models the broadcast of a computation message. It can be noticed that a causal order is built by the sender process while broadcasting a computation message. A message is codelivered to the sender at the time of broadcast. The event CausalDeliver models the event of causally ordered delivery of a message to a process. The guards of the CausalDeliver event also ensure that a message is codelivered only once. The following guards of the CausalDeliver event ensure that a process pp causally codelivers a message mm only if it has codelivered all messages which causally precedes mm.

∀m.((m → mm) ∈ causalorder ⇒ (pp → m) ∈ cdeliver)
Upon delivery of a message \( mm \) in causal order the variable \( cdelorder \) is also updated so that all messages \( codelivered \) to process \( pp \) are ordered before \( mm \).

\[
\text{Broadcast} \ (pp \in \text{PROCESS}, mm \in \text{MESSAGE}) \implies \\
\text{WHEN} \ mm \notin \text{dom}(sender) \\
\text{THEN} \ 
\begin{align*}
\text{sender} &: = \text{sender} \cup \{mm \mapsto pp\} \\
\text{causalorder} &: = \text{causalorder} \cup (\text{sender}^{-1}\{pp\} \times \{mm\}) \\
& \quad \cup (\text{cdeliver}\{pp\} \times \{mm\}) \\
\text{cdeliver} &: = \text{cdeliver} \cup \{pp \mapsto mm\} \\
\text{cdelorder}(pp) &: = \text{cdelorder}(pp) \cup (\text{cdeliver}\{pp\} \times \{mm\}) \\
\text{computation} &: = \text{computation} \cup \{mm\}
\end{align*}
\]
\text{END};

\[
\text{CausalDeliver} \ (pp \in \text{PROCESS}, mm \in \text{MESSAGE}) \equiv \\
\text{WHEN} \ mm \notin \text{dom}(sender) \\
\land \ (pp \mapsto mm) \notin \text{cdeliver} \\
\land \ \forall m.( m \in \text{MESSAGE} \land (m \mapsto mm) \in \text{causalorder} \implies (pp \mapsto m) \in \text{cdeliver}) \\
\text{THEN} \ 
\begin{align*}
\text{cdeliver} &: = \text{cdeliver} \cup \{pp \mapsto mm\} \\
\text{cdelorder}(pp) &: = \text{cdelorder}(pp) \cup (\text{cdeliver}\{pp\} \times \{mm\})
\end{align*}
\]
\text{END};

\textbf{Figure 6.3: TotalCausalOrder: Events-I}

The specifications of the events \textit{SendControl} and \textit{TODeliver} are given in Fig. 6.4. The \textit{SendControl} is an event of sending a control message once a computation message is \textit{codelivered} at the sequencer. The following guard of this event ensures that a control message(\(mc\)) for a computation message(\(mm\)) is broadcasted only when it has already broadcasted control messages for the computation messages which \textit{causally precedes} \( mm \).

\[
\forall m.((m \mapsto mm) \in \text{causalorder} \implies m \in \text{ran}(\text{messcontrol}))
\]

The set \( \text{ran}(\text{messcontrol}) \) contains the computation messages for which control messages have been sent by the sequencer. In the operations of event \textit{SendControl}, it can be noticed that the sequencer also builds the causal order on the control messages and the variable \textit{messcontrol} is updated by adding a corresponding mapping. A total order for the computation messages \( mm \) is also built by the sequencer by updating the abstract variable \textit{totalorder} as :

\[
\text{totalorder} := \text{totalorder} \cup (m \times \{mm\})
\]

where \( m = \text{ran}(\text{messcontrol}) \). This implies that all computation messages, for which the sequencer has already sent out control messages, are now totally ordered before \( mm \).
SendControl \((pp \in \text{PROCESS}, mm \in \text{MESSAGE}, mc \in \text{MESSAGE})\) \(\Xi\)

\[
\text{WHEN} \quad pp = \text{sequencer} \\
\quad \land \quad mc \in \text{dom}(\text{sender}) \\
\quad \land \quad mm \in \text{ran}(\text{messcontrol}) \\
\quad \land \quad mm \in \text{computation} \\
\quad \land \quad (pp \mapsto mm) \in \text{cdeliver} \\
\quad \land \quad \forall m. \ (m \in \text{MESSAGE} \land m \in \text{computation} \\
\quad \quad \quad \land \quad (m \mapsto mm) \in \text{causalorder} \Rightarrow m \in \text{ran}(\text{messcontrol}) \\
\text{THEN} \quad \text{causalorder} := \text{causalorder} \cup ( (\text{sender} \mapsto [\text{sequencer}]) \times \{mc\}) \\
\quad \quad \cup ( \text{cdeliver}[\text{sequencer}] \times \{mc\})) \\
\quad \quad \| \quad \text{sender} := \text{sender} \cup \{mc \mapsto \text{sequencer}\} \\
\quad \quad \| \quad \text{control} := \text{control} \cup \{mc\} \\
\quad \quad \| \quad \text{messcontrol} := \text{messcontrol} \cup \{mc \mapsto mm\} \\
\quad \quad \| \quad \text{LET} \ m \ \text{BE} \ m = \text{ran}(\text{messcontrol}) \\
\quad \quad \| \quad \text{IN} \ \text{totalorder} := \text{totalorder} \cup (m \times \{mm\}) \quad \text{END}
\]

\[
\text{END};
\]

TODeliver \((pp \in \text{PROCESS}, mc \in \text{MESSAGE})\) \(\Xi\)

\[
\text{WHEN} \quad mc \in \text{dom}(\text{sender}) \\
\quad \land \quad mc \in \text{control} \\
\quad \land \quad (pp \mapsto mc) \in \text{cdeliver} \\
\quad \land \quad (pp \mapsto \text{messcontrol}(mc)) \in \text{cdeliver} \\
\quad \land \quad (pp \mapsto \text{messcontrol}(mc)) \in \text{tdeliver} \\
\quad \land \quad \forall m. \ (m \in \text{MESSAGE} \land m \in \text{computation} \\
\quad \quad \quad \land \quad (m \mapsto \text{messcontrol}(mc) \in \text{totalorder}) \Rightarrow (pp \mapsto m) \in \text{tdeliver} \) \\
\text{THEN} \quad \text{tdeliver} := \text{tdeliver} \cup \{pp \mapsto \text{messcontrol}(mc)\} \\
\quad \| \quad \text{tdelorder}(pp) := \text{tdelorder}(pp) \cup (\text{tdeliver}[\{pp\}] \times \{\text{messcontrol}(mc)\}) \\
\text{END}
\]

\text{Figure 6.4: TotalCausalOrder: Event-II}

The TODeliver event models a totally ordered delivery of a computation message to a process. This event is activated when a process \(pp\) codelivers a control message \(mc\). The guard of the event ensures that at the codelivery of a control message \(mc\) by a process \(pp\), it also delivers a computation message in a total order corresponding to the control message \(mc\) if it has already delivered all computation messages which are totally ordered before a computation message defined as \(\text{messcontrol}(mc)\). The \(\text{messcontrol}(mc)\) represents a computation message corresponding to the control message \(mc\).

Upon todelivery of a message \(mm\), the variable \(\text{tdelorder}\) is also updated so that all messages \(\text{tdelivered}\) to the process \(pp\) are ordered before \(mm\).
6.3.3 Verification of Ordering Properties

In order to verify that our model of total causal order broadcast preserves the abstract causal order when the messages are *todelivered* to the processes, we need to prove that if the broadcast of any two messages is related by a causal precedence relationship then they are *todelivered* to all processes in a total order respecting their causal precedence relationship. Therefore, we add the following to the list of invariants as a primary invariant.

\[
\begin{align*}
   & m_1 \in \text{ran}(\text{messcontrol}) \land \\
   & m_2 \in \text{ran}(\text{messcontrol}) \land \\
   & m_1 \hookrightarrow m_2 \in \text{causalorder} \\
   \Rightarrow \\
   & m_1 \hookrightarrow m_2 \in \text{totalorder}
\end{align*}
\]

This invariant states that for any two computation messages \(m_1\) and \(m_2\), whose control messages have been sent out by the sequencer, and if \(m_1\) causally precedes \(m_2\) i.e., \((m_1 \hookrightarrow m_2) \in \text{causalorder}\) then \(m_1\) also precedes \(m_2\) in the abstract total order i.e., \((m_1 \hookrightarrow m_2) \in \text{totalorder}\). The reasons for adding the clauses \(m_1 \in \text{ran}(\text{messcontrol})\) and \(m_2 \in \text{ran}(\text{messcontrol})\) is that an abstract total order on the messages is constructed by the sequencer only when their control messages are sent out. This invariant also shows that the causality is preserved while building an abstract total order by the sequencer. Therefore, if the broadcast of any two messages is related by a causal precedence relationship then they are *todelivered* to all processes in a total order respecting their causal precedence relationship.

The addition of this invariant (Inv-1) as a primary invariant generates several other proof obligations. In order to discharge these proof obligations we need to add new invariants to the model. After two rounds of invariant strengthening, we arrive at a set of invariants that is sufficient to discharge all proof obligations. These invariants are outlined in the Fig. 6.5. The codes for the events are given in the Table 6.1.

A brief description of these invariants is given below.

- For any two computation messages \(m_1\) and \(m_2\) whose control message has been sent out i.e., \(m_1, m_2 \in \text{ran}(\text{messcontrol})\), if \(m_1\) causally precedes \(m_2\) then a total order also exists among them i.e., \(m_1\) is totally ordered before \(m_2\). (Inv-1)

- A message is *codelivered* to a process before it is *todelivered* to that process. This invariant also states that a message delivered in a total order has also been delivered in a causal order.(Inv-2)

- For any two computation messages \(m_1\) and \(m_2\) where \(m_1\) causally precedes \(m_2\) and the control messages for \(m_2\) have been sent out implies that the control message for \(m_1\) have also been sent. This invariant also indicates that the sequencer
 broadcasts the control messages for the computation messages in their causal order.  
(Inv-3)

- Each message whose control message has been sent should also have been \textit{codelivered} at the sequencer. (Inv-4)

In order to verify that the TotalCausalOrder model also preserves both total order and causal ordering properties, we add a set of invariants given as Invariant-II in Fig. 6.6 as primary invariants. Addition of these invariants to the model generates proof obligations. Following a similar approach given in the Chapter 4 and Chapter 5, we discharge the proof obligations associated with these invariants.

A brief description of these invariants is given below.

- Given two messages \( m_1 \) and \( m_2 \), if message \( m_1 \) causally precedes \( m_2 \) and a process \( p \) has \textit{codelivered} \( m_2 \) then the delivery order at process \( p \) must have been \( m_1 \).
followed by $m_2$. This invariant states the required property for the causal order. *(Inv-5)*

- For two messages $m_1$ and $m_2$ where $m_1$ is *todelivered* before $m_2$ at a process $p$ ($m_1 \rightarrow m_2 \in \text{delorder}(p)$) then $m_1$ precedes $m_2$ in the abstract total order. This invariant states the required property for the total order. *(Inv-6)*

<table>
<thead>
<tr>
<th>Invariants</th>
<th>Required By</th>
</tr>
</thead>
<tbody>
<tr>
<td>/<em>Inv-7</em>/ $\ (m_1 \rightarrow m_2) \in \text{causalorder} \wedge (m_2 \rightarrow m_3) \in \text{causalorder} \Rightarrow (m_1 \rightarrow m_3) \in \text{causalorder}$</td>
<td>Primary Invariant</td>
</tr>
<tr>
<td>/<em>Inv-8</em>/ $\ (m_1 \rightarrow m_2) \in \text{causalorder} \wedge (p \rightarrow m_2) \in \text{cdeliver} \Rightarrow (p \rightarrow m_1) \in \text{cdeliver}$</td>
<td>BC,CD, SC,TOD</td>
</tr>
<tr>
<td>/<em>Inv-9</em>/ $\ (m_1 \rightarrow m_2) \in \text{totalorder} \wedge (m_2 \rightarrow m_3) \in \text{totalorder} \Rightarrow (m_1 \rightarrow m_3) \in \text{totalorder}$</td>
<td>Primary Invariant</td>
</tr>
<tr>
<td>/<em>Inv-10</em>/ $\ (m_1 \rightarrow m_2) \in \text{totalorder} \wedge (p \rightarrow m_2) \in \text{tdeliver} \Rightarrow (p \rightarrow m_1) \in \text{tdeliver}$</td>
<td>SC,TOD</td>
</tr>
</tbody>
</table>

**Figure 6.7: Invariants-III : Abstract Model**

The invariant properties of the model of total causal order showing the transitivity on the abstract causal and total order are given in the Fig. 6.7. A brief description of these properties is given below.

- An abstract causal order is transitive. *(Inv-7)*

- For two messages $m_1$ and $m_2$, if $m_1$ causally precedes $m_2$ and process $p$ has *codelivered* the message $m_2$ then $p$ has also *codelivered* the message $m_1$. *(Inv-8)*

- An abstract total order is transitive. *(Inv-9)*

- For two messages $m_1$ and $m_2$, if $m_1$ precedes $m_2$ in *total order* and process $p$ has *todelivered* the message $m_2$ then $p$ has also *todelivered* $m_1$. *(Inv-10)*

The proof obligations associated with these invariants are discharged using the process outlined in sections 4.3 and 5.4. Discharging these proof obligations was relatively easy, because we already knew the invariants needed to discharge these proof obligations. A complete set of invariants for this model is given in the Appendix-E.
6.4 First Refinement of Total Causal Order

In this section, we outline the first refinement of the abstract model of total causal order broadcast. In this refinement we introduce the notion of vector clocks and sequence numbers. The abstract variables `causalorder` and `totalorder` in this refinement are replaced with the vector clock rules and sequence numbers respectively. In this refinement we introduce two new variables `VTP` and `VTM` to implement causal ordering. The variables `VTP` and `VTM` respectively represent the vector time of a process and the vector timestamp of a message. Similarly, in order to implement total ordering we also introduce variables `seqno` and `counter`.

6.4.1 Events in the First Refinement

The events of the first refinement of the machine `TotalCausalOrder` using vector clocks and sequence numbers are shown in the Fig. 6.8 and Fig. 6.9. It can be noticed that the operations of events (`Broadcast`, `CausalDeliver` and `SendControl`) involving the abstract variable `causalorder` are replaced by the vector rules. Similarly, the operations of events `SendControl` and `TODeliver` involving the abstract variable `totalorder` are replaced by the sequence numbers(`seqno`).

```
Broadcast(pp \in PROCESS, mm \in MESSAGE) \equiv
  \text{WHEN} \quad \text{mm} \in \text{dom(sender)}
  \text{THEN} \quad \text{LET} \quad nVTP \quad \text{BE} \quad nVTP = VTP(pp) \gets \{ pp \rightarrow VTP(pp)(pp)+1 \}
  \quad \text{IN} \quad \text{VTM(mm)} := nVTP
  \quad || \quad VTP(pp) := nVTP \quad \text{END}
  || \quad \text{sender} := \text{sender} \cup \{ \text{mm} \rightarrow pp \}
  || \quad \text{cdeliver} := \text{cdeliver} \cup \{ pp \rightarrow mm \}
  || \quad \text{computation} := \text{computation} \cup \{ mm \}
  \text{END} ;
```

```
CausalDeliver (pp \in PROCESS, mm \in MESSAGE) \equiv
  \text{WHEN} \quad \text{mm} \in \text{dom(sender)}
  \quad \land \quad (pp \rightarrow mm) \notin \text{cdeliver}
  \quad \land \quad \forall p.( p \in \text{PROCESS} \land p \neq \text{sender(mm)} \Rightarrow VTP(pp)(p) \geq VTP(mm)(p))
  \quad \land \quad VTP(pp)(\text{sender(mm)}) = VTP(mm)(\text{sender(mm)})-1
  \text{THEN}
  \quad \text{cdeliver} := \text{cdeliver} \cup \{ pp \rightarrow mm \}
  \quad || \quad VTP(pp) := VTP(pp) \gets \{ (q \in \text{PROCESS} \land VTP(pp)(q) < VTP(mm)(q)) \land VTM(mm) \}
  \text{END};
```

Figure 6.8: First Refinement- Part I
SendControl \((pp \in \text{PROCESS}, \ mm \in \text{MESSAGE}, \ mc \in \text{MESSAGE})\) ≜

**WHEN**
\(pp = \text{sequencer}\)
\(\land \ mc \notin \text{dom}(\text{sender})\)
\(\land \ mm \notin \text{ran}(\text{messcontrol})\)
\(\land \ mm \in \text{computation}\)
\(\land \ pp \rightarrow m \in \text{cdeliver}\)
\(\land \ \forall (m,p) \cdot (p \in \text{PROCESS} \land m \in \text{MESSAGE} \land m \in \text{computation} \land VTP(m)(p) \leq VTP(mm)(p) \Rightarrow m \in \text{ran}(\text{messcontrol}))\)

**THEN**
control := control \cup \{mc\}
\(\| \ \text{messcontrol} := \text{messcontrol} \cup \{\text{mc} \rightarrow \text{mm}\}\)
\(\| \ \text{LET} \ nVTP \ BE \ nVTP = VTP(pp) \triangleq \{ pp \rightarrow VTP(pp)(pp)+1\}\)
\(\| \ \text{IN} \ VTP(mc) := nVTP\)
\(\| \ VTP(pp) := nVTP\)
\(\| \ \text{END}\)

\(\| \ \text{sender} := \text{sender} \cup \{mc \rightarrow pp\}\)
\(\| \ \text{LET} \ ncount \ BE \ ncount = \text{counter} + 1\)
\(\| \ \text{IN} \ \text{counter} := ncount\)
\(\| \ \text{seqno}(\text{mm}) := ncount\)
\(\| \ \text{END}\)

\(\| \ \text{END}\);

TODeliver \((pp \in \text{PROCESS}, \ mc \in \text{MESSAGE})\) ≜

**WHEN**
mc \in \text{dom}(\text{sender})
\(\land \ mc \in \text{control}\)
\(\land \ (pp \rightarrow mc) \in \text{cdeliver}\)
\(\land \ (pp \rightarrow \text{messcontrol}(mc)) \in \text{cdeliver}\)
\(\land \ (pp \rightarrow \text{messcontrol}(mc)) \notin \text{ideliver}\)
\(\land \ \forall m. (m \in \text{MESSAGE} \land m \in \text{computation} \land (\text{seqno}(m) < \text{seqno}(\text{messcontrol}(mc)) \Rightarrow (pp \rightarrow m) \in \text{ideliver})\)

**THEN**
\text{ideliver} := \text{ideliver} \cup \{pp \rightarrow \text{messcontrol}(mc)\}
\(\| \ \text{END}\)

**Figure 6.9:** First Refinement - Part II

The events Broadcast and SendControl are the events of sending a message. The event Broadcast models the broadcast of computation messages and event SendControl models the broadcast of control messages. In both of the events, the sender process pp increments its own clock value \(VTP(pp)(pp)\) by one. Recall that \(VTP(pp)(pp)\) represents the number of messages sent by the process pp. The modified vector timestamp of the process is also assigned to message mm giving vector timestamp of message mm.

The CausalDeliver event models causally ordered delivery of a message mm at process pp. Consider the following guard of this event involving abstract causal order.

\(\forall m. ((m \rightarrow mm) \in \text{causalorder} \Rightarrow (pp \rightarrow m \in \text{cdeliver})\)
This guard is replaced by the following guards involving vector clock rules in the refinement.

\[ (1) \forall p. (p \in \text{PROCESS} \land p \neq \text{sender}(mm)) \Rightarrow \text{VTP}(pp)(p) \geq \text{VTM}(mm)(p) \]

\[ (2) \text{VTP}(pp)(\text{sender}(mm)) = \text{VTM}(mm)(\text{sender}(mm)) - 1 \]

The first condition states that the vector timestamp of a recipient process \( pp \) and message \( mm \) are compared to ensure that all messages received by the sender of a message before sending it, are also received at the recipient process. The second condition states that process \( pp \) has received all but one message from the sender of the message \( mm \).

An operation updating the vector clock of recipient process \( pp \) is also shown in the specification of \( \text{CausalDeliver} \) event.

The variable \( \text{seqno} \) is used for building a total order on the computation messages. In the refined specification of event \( \text{SendControl} \), it can be noticed that the operation involving abstract \( \text{totalorder} \) is replaced by an operation containing variables \( \text{seqno} \) and \( \text{counter} \).

The counter is incremented each time a control message is sent and it is assigned to the control messages.

The guards of the event \( \text{TODeliver} \) are strengthened in this refinement. It can be noticed that the following guard of the event \( \text{TODeliver} \) involving abstract \( \text{totalorder} \)

\[ \forall m. (m \in \text{computation} \land (m \mapsto m \text{esscontrol}(mc) \in \text{totalorder}) \Rightarrow (pp \mapsto m) \in t\text{deliver}) \]

is replaced by the following guard involving sequence numbers.

\[ \forall m. (m \in \text{computation} \land (\text{seqno}(m) < \text{seqno}(m \text{esscontrol}(mc))) \Rightarrow (pp \mapsto m) \in t\text{deliver}) \]

The above states that the process has \( t\text{odelivered} \) all computation messages whose sequence number is less than the sequence number of the computation message corresponding to the control message \( mc \).

### 6.4.2 Constructing Gluing Invariants

In this section we briefly outline how the proof obligations generated due to the replacement of the guards and operations containing abstract variables \( \text{causalorder} \) and \( \text{totalorder} \) by the vector clock rules and sequence numbers respectively help us discover gluing invariants. A few important gluing invariants are given in the Fig 6.10. A complete list of invariants is given in Appendix-E.
6.4.2.1 Relationship of abstract causal order and vector clock rules

The replacement of the guards and operations involving the variable \textit{causalorder} in the abstract model by the equivalent rules of \textit{vector clocks} generate several proof obligations due to refinement checking. Initially, the only proof obligation that can not be proved is given below in simplified form. It involves the relationship between \textit{causalorder} and the vector timestamp of a message generated by the event \textit{CausalDeliver}.

\begin{align*}
\textit{CausalDeliver}(pp, mm) & \text{PO1} \\
mm & \in \text{dom(sender)} \\
(pp \mapsto mm) & \notin \text{cdeliver} \\
\forall p. (p \in \text{PROCESS} \land p \neq \text{sender(mm)} \Rightarrow VTP(pp)(p) \geq VTM(mm)(p)) \\
VTP(pp)(\text{sender(mm)}) & = VTM(mm)(\text{sender(mm)}) - 1 \\
m & \in \text{MESSAGE} \\
m \mapsto mm & \in \text{causalorder} \\
\Rightarrow \\
(pp \mapsto m) & \in \text{cdeliver}
\end{align*}

In this proof obligation it can be noticed that a message \textit{m} causally precedes \textit{mm} i.e., \((m \mapsto mm) \in \text{causalorder}\) and process \textit{pp} has not \textit{cdelivered} \textit{mm}. According to the vector clock rules, \textit{pp} can \textit{cdeliver} \textit{mm} only when it has \textit{cdelivered} all messages which causally precede \textit{mm}. If a process \textit{pp} has \textit{cdelivered} all but one message from the sender of \textit{mm} then the following must be hold:

\[ VTP(pp)(\text{sender(mm)}) = VTM(mm)(\text{sender(mm)}) - 1 \]

Similarly, if a process \textit{pp} has \textit{cdelivered} all messages sent by the sender of \textit{mm} before sending \textit{mm} and it has also \textit{cdelivered} \textit{mm} then the following must hold:

\[ VTP(pp)(\text{sender(mm)}) \geq VTM(mm)(\text{sender(mm)}) \]

Thus we add an invariant given at \textit{Inv 11} in Fig. 6.10 which states that if the vector time of process \textit{p1} is equal or greater than the vector timestamp of any sent message \textit{m} then \textit{p1} must have \textit{cdelivered} the message \textit{m}. Adding \textit{Inv 11} to the model generates proof obligations associated with other events. Discharging these proof obligations required other invariants given as \textit{Inv 12,13} and \textit{14}. After three iterations of invariant strengthening we arrive at a set of invariants which is sufficient to discharge all proof obligations relating abstract \textit{causalorder} and \textit{vector clock rules}. 
6.4.2.2 Relationship of abstract total order and sequence number

Replacing the abstract variable \( \text{totalorder} \) by the sequence number in the operations of \( \text{SendControl} \) and the guards of the \( \text{TODeliver} \) event generates proof obligations. The first proof obligation which can not be discharged automatically requires us to prove the following for the \( \text{TODeliver} \) event.

\[
\text{TODeliver}(pp, mc) PO 2
\begin{align*}
mc & \in \text{dom}(sender) \\
mc & \in \text{control} \\
pp & \mapsto \text{messcontrol}(mc) \in \text{cdeliver} \\
(pp & \mapsto \text{messcontrol}(mc)) \notin \text{tdeliver} \\
\forall m. (m \in \text{computation} \land (\text{seqno}(m) < \text{seqno}(\text{messcontrol}(mc))) \Rightarrow (pp \mapsto m) \in \text{tdeliver}) \\
m & \mapsto \text{messcontrol}(mc) \in \text{totalorder} \\
\Rightarrow (pp & \mapsto m) \in \text{tdeliver}
\end{align*}
\]

It may also be noted that this proof obligation appears due to the replacement of the following guard of \( \text{TODeliver} \) involving the abstract variable \( \text{totalorder} \):

\[
\forall m. ((m \mapsto \text{messcontrol}(mc) \in \text{totalorder}) \Rightarrow (pp \mapsto m) \in \text{tdeliver})
\]

by the guard involving variable \( \text{seqno} \):

\[
\forall m. ((\text{seqno}(m) < \text{seqno}(\text{messcontrol}(mc))) \Rightarrow (pp \mapsto m) \in \text{tdeliver})
\]

Therefore, in order to discharge this proof obligation we add the invariant \( \text{Inv-15} \) to our model which relates the abstract variable \( \text{totalorder} \) to the concrete \( \text{seqno} \). This invariant states that if two computation messages \( m1 \) and \( m2 \) are in \( \text{totalorder} \) then the sequence number of \( m1 \) is less than sequence number of \( m2 \). We notice that this invariant is sufficient to discharge all proof obligations generated by the \( \text{SendControl} \) and the \( \text{TODeliver} \) events.

6.4.2.3 Gluing Invariants

The invariant showing the relationship of the abstract \( \text{causorder} \) and \( \text{totalorder} \) with the \textit{vector rules} and \textit{sequence numbers} is given in the Fig. 6.10. The codes for the events are given in Table 6.1. A brief description of these properties is given below.

- If the vector time of process \( P \) is equal to or greater than the vector timestamp of any sent message \( M \) then \( P \) must have \textit{codelivered} the message \( M \) (\text{Inv-11}).
6.5 Second Refinement: Replacing Sequence Number by the Vector Clocks

In the second refinement, we outline how the need for generating separate sequence numbers can correctly be implemented by the vector clock rules. It can be noticed that the total order on the messages in the first refinement is realized with the sequence numbers. The specifications of the Broadcast and CausalDeliver events of the first refinement remain unaltered as none of these events make use of sequence numbers. The events SendControl and TODeliver in the first refinement are further refined to eliminate the need for the sequence number generated by the sequencer. In the second refinement, the variables seqno and counter are replaced by the vector clock rules. The specifications of the refined SendControl and TODeliver events are given in Fig. 6.11, 6.12.

As shown in Fig. 6.11, the operation assigning the sequence number to the computation message is removed in the refined SendControl event. We use the fact that the vector
timestamp of the control message contains the information required for todelivery of the messages. Also, as shown in the Fig. 6.12, the guard of the event TODeliver which contains sequence numbers in the abstract model is replaced by the vector rules. We use the fact that the sequence numbers for the computation message are generated by the sequencer each time it sends a control message. Thus, for a given control message \( M \) and the corresponding computation message \( (M') \), the following holds:

\[
\text{seqno}(M') = \text{VTM}(M)(\text{sequencer})
\]

This replacement in the refinement generates proof obligations involving seqno and the vector timestamp of messages. To prove these proof obligations we add Inv-16, shown in Fig. 6.13 to our refined model. Adding Inv-16 to the refinement requires us to add new invariants Inv-17,18 to the refinement. A brief description of these invariants is given below.

- For a control message \( m \) sent by the sequencer, the value \( \text{VTM}(m)(\text{sequencer}) \) of the vector timestamp of \( m \) represents the sequence number of the computation message corresponding to control message \( m \). In other words, the sequence number assigned to a computation message is the same as the sequencer’s own logical time at the time of sending its control message (Inv-16).

- For two control messages \( m1 \) and \( m2 \), if the vector timestamp of \( m1 \) is less than the vector timestamp of \( m2 \) then the sequence number given to the corresponding computation message of \( m1 \) is also less than sequence number of the computation message of \( m2 \) (Inv-17).
**TODeliver** \((pp \in \text{PROCESS}, mc \in \text{MESSAGE}) \equiv\)

**WHEN** \(mc \in \text{dom}(\text{sender})\)

\(\land (pp \rightarrow mc) \in \text{cdeliver}\)

\(\land (pp \rightarrow \text{messcontrol}(mc)) \in \text{cdeliver}\)

\(\land (pp \rightarrow \text{messcontrol}(mc)) \in \text{ideliver}\)

\(\land \forall m. (m \in \text{MESSAGE} \land m \in \text{computation}) \land (\text{VTM}(\text{messcontrol}(mc))(\text{sequencer}) \prec \text{VTM}(mc)(\text{sequencer}) ) \Rightarrow (pp \rightarrow m) \in \text{ideliver})\)

**THEN** \(\text{ideliver} := \text{ideliver} \cup \{pp \rightarrow \text{messcontrol}(mc)\}\)

**END**

**Figure 6.12:** Second Refinement : TODeliver

- For two computation messages \(m1\) and \(m2\), if the sequence number given to \(m1\) is less than the sequence number of \(m2\) then the vector timestamp of the corresponding control message \(m1\) is also less than the vector timestamp of corresponding control message of \(m2\) (Inv-18).

After discharging the proof obligations generated due to the addition of these invariants associated with the events **Broadcast**, **SendControl** and **TODeliver**, we ensure that the events in Fig. 6.11, 6.12 are valid refinements of events in Fig. 6.9.

### 6.6 Further Refinements

In the third and fourth refinements we simplify the operations of the **CausalDeliver** event given in the Fig. 6.8. In the second refinement the vector clock of the recipient process

<table>
<thead>
<tr>
<th>Invariants</th>
<th>Required By</th>
</tr>
</thead>
<tbody>
<tr>
<td>/<em>Inv-16</em>/ (m \in \text{control} \land (m \rightarrow \text{sequencer}) \in \text{sender}) (\Rightarrow \text{seqno}((\text{messcontrol}^{-1}(m)) = \text{VTM}(m)(\text{sequencer})))</td>
<td><strong>SC,TOD</strong></td>
</tr>
<tr>
<td>/<em>Inv-17</em>/ (m1 \in \text{control} \land m2 \in \text{control}) (\land \text{VTM}(m1)(p) \leq \text{VTM}(m2)(p)) (\Rightarrow \text{seqno}((\text{messcontrol}^{-1}(m1)) \leq \text{seqno}((\text{messcontrol}^{-1}(m2)))</td>
<td><strong>BC,SC,TOD</strong></td>
</tr>
<tr>
<td>/<em>Inv-18</em>/ (m1 \in \text{computation} \land m2 \in \text{computation}) (\land \text{seqno}(m1) \leq \text{seqno}(m2)) (\Rightarrow \text{VTM}((\text{messcontrol}(m1))(p) \leq \text{VTM}((\text{messcontrol}(m2))(p))</td>
<td><strong>SC,TOD</strong></td>
</tr>
</tbody>
</table>

**Figure 6.13:** Second Refinement : Gluing Invariant
is updated as:

\[ VTP(pp) := VTP(pp) \triangleq \{ (q \mid q \in \text{PROCESS} \wedge VTP(pp)(q) < VTM(mm)(q) \} \triangleleft VTM(mm) \]  

The above operation is replaced by the following simplified operation in the third refinement which states that only one value in the vector clock of the recipient process \( pp \), corresponding to the sender process of the message, is updated.

\[ VTP(pp) := VTP(pp) \triangleq \{ \text{sender}(mm) \mapsto VTM(mm)(\text{sender}(mm)) \} \]

This operation is further refined to the following in the fourth refinement which precisely states that only one value in the vector clock of the recipient process is updated.

\[ VTP(pp)(\text{sender}(mm)) := VTM(mm)(\text{sender}(mm)) \]

The refined \textit{CausalDeliver} event in the fourth refinement is given in Fig 6.14. In each refinement step we observed that proof obligations are generated due to the replacement of the operations of the event \textit{CausalDeliver}. These proof obligations are automatically discharged by the B prover. A full chain of refinement with a complete set of invariants is given in the Appendix-E.

\begin{verbatim}
CausalDeliver (pp \in \text{PROCESS}, mm \in \text{MESSAGE}) \equiv
    WHEN  mm \in \text{dom}(\text{sender})
        \wedge  (pp \mapsto mm) \notin cdeliver
        \wedge  \forall p. p \in \text{PROCESS} \wedge p \neq \text{sender}(mm) \Rightarrow VTP(pp)(p) \geq VTM(mm)(p))
        \wedge  VTP(pp)(\text{sender}(mm)) = VTM(mm)(\text{sender}(mm)) - 1
    THEN
        cdeliver := cdeliver \cup \{ pp \mapsto mm \}
        \| VTP(pp)(\text{sender}(mm)) := VTM(mm)(\text{sender}(mm))
    END;
\end{verbatim}

\textsc{Figure 6.14: Causal Deliver Event}

\section{6.7 Conclusions}

In this chapter, we have outlined the development of a system of total causal order broadcast. A total causal order broadcast not only preserves the causal precedence relationship among the messages but also delivers them in a total order. In our model of a total causal order broadcast, the computation messages are broadcast using a causal
order broadcast. These computation messages are first delivered to all processes including the sequencer respecting their causal order. Upon causally ordered delivery of a computation message at the sequencer, the sequencer generates a sequence number for that computation message and broadcasts its sequence number by a control message. Similar to our model of a total order broadcast, a process other than the sequencer delivers a computation message in the order of sequence numbers. It may be noted that all computation messages are delivered at the sequencer in causal order and their control messages are sent in the order they were delivered at the sequencer. Therefore, the sequence number generated by the sequencer also captures the causality among the computation messages. Later in the refinement, we outlined that the generation of explicit sequence numbers is redundant and it can be implemented using the vector timestamp of the messages.

In the abstract model of this broadcast we outlined how the abstract causal order and a total order on the computation messages are constructed. In the first refinement of the abstract model we replace abstract causal order by the vector clock rules. Similarly, we also replace the abstract total order with sequence numbers to outline how an abstract total order can correctly be implemented by sequence numbers. In the second refinement we show that an abstract system can be implemented by a vector clock system. We also outline in this refinement, why the generation of sequence numbers is redundant and how it is related to the vector timestamps. The last two refinements show simplification of the event of delivery of a message in a causal order.

We notice that proof obligations are generated due to the addition of the primary invariants to the model and refinement checking. These proof obligations help us construct and discover the new invariants required to discharge the proof obligations. In this case study, the discovery of new invariants was relatively easy as we already knew the invariants relating abstract causal order, total order, vector clocks and sequence numbers. These invariants were discovered while the discharging proof obligations for the case studies given in the Chapter 4 and the Chapter 5. Since most invariants added to the model are predicates with quantification, the average of number of steps involved with each proof is estimated at about twelve to fifteen. The proof statistics for the development of a system of a total causal order are given in Table 6.2.

<table>
<thead>
<tr>
<th>Machine</th>
<th>Total POs</th>
<th>Completely Automatic</th>
<th>Required Interaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract Model</td>
<td>92</td>
<td>57</td>
<td>35</td>
</tr>
<tr>
<td>Refinement 1</td>
<td>50</td>
<td>31</td>
<td>19</td>
</tr>
<tr>
<td>Refinement 2</td>
<td>14</td>
<td>04</td>
<td>10</td>
</tr>
<tr>
<td>Refinement 3</td>
<td>06</td>
<td>06</td>
<td>00</td>
</tr>
<tr>
<td>Refinement 4</td>
<td>04</td>
<td>04</td>
<td>00</td>
</tr>
<tr>
<td>Overall</td>
<td>166</td>
<td>102</td>
<td>64</td>
</tr>
</tbody>
</table>

Table 6.2: Proof Statistics- Total Causal Order Broadcast
Chapter 7

Liveness Properties and Modelling Guidelines

7.1 Introduction

In this chapter, we outline liveness properties that need to be preserved by the B models of distributed systems. We also outline how enabledness preservation and non-divergence are related to the liveness properties of the B models of distributed systems. We address the liveness issues related to our model of distributed transactions. Finally, we present some general modelling guidelines for the development of the models of distributed systems in Event-B.

7.2 Liveness in the Event-B Models

Safety and liveness are two important issues in the development of distributed systems [73]. The distinction between safety and liveness properties was motivated by the different tools and techniques for proving them and various interpretations of these properties are discussed in [66]. Informally, as described in [73], a safety property expresses that something (bad) will not happen during a system execution. A liveness property expresses that something (good) will eventually happen during the execution.

With regard to safety, the most important property which we want to prove about models of distributed systems is invariant preservation. The invariant is a condition which must hold permanently on the state variables. By invariant preservation we mean proving that the actions of the events do not violate the invariants. With regards to the safety properties, the existing tools generate proof obligations for consistency checking and refinement checking. Discharging the proof obligations generated due to consistency checking means that the actions of the events do not violate the invariants. Discharging
the proof obligations for refinement checking also implies that each reachable concrete state in the refinement is also reachable in the abstraction.

Despite providing strong proof support to aid reasoning about the safety properties, the existing tools provide weak support for other complex forms of reasoning about liveness properties, such as enabledness preservation or non-divergence, and feasibility checking. By enabledness preservation, we mean whenever some events in the abstraction are enabled then the corresponding events or new events in the refinement are also enabled. Similarly, non-divergence requires us to prove that the new events in the refinement do not take control forever. The issues relating to the liveness properties are currently being addressed in the new generation of Event-B tools being developed [92, 44]. In the remaining sections we outline these issues and present guidelines to address these issues in the Event-B development of distributed systems.

7.2.1 Feasibility

With respect to the safety properties of distributed systems, in addition to consistency and refinement checking, feasibility checking is also an important issue. It is our understanding that verifying the feasibility of a valid initial state of a distributed system is an important step in the development of a distributed system. Consider the following example B machine given in the Fig. 7.1.

As shown in the initialization clause of the machine, the variable \texttt{sender} is initialized to a null set only if the guard is true. Since the guards is always false for all values of \texttt{x}, the initialization of variable \texttt{sender} is not feasible. Therefore, the initialization of the machine in a consistent state is never possible. Similarly, since the guard of the event \texttt{Broadcast} is always false for all values assumed by the variable \texttt{y}, the event will never be enabled. The current B tools generate two trivial proof obligations due to invariant
preservation, one each associated with the initialization and the event broadcast. These proof obligations are automatically discharged by the existing B tools.

Since the existing tools are not able to generate the proof obligations relating to feasibility, it is not possible to determine whether a valid initialization of the machine is feasible or whether an event will ever be enabled. In order to check the feasibility of the initialization, the use of ProB is highly recommended. To check feasibility, the tools must generate proof obligations to determine if there exist any contradictions in the guards of the events. We believe that the new generation B tool e.g., Rodin [44] addresses this issue and generates proof obligations to ensure the feasibility of of a consistent initial state and the possibility of activation of events.

7.2.2 Non-Divergence

New events and variables can be introduced in refinement. Each new event of a refinement refines a skip event and defines a computation on new variables. In such cases, it is useful to prove that the new events do not together diverge, i.e., run forever. If a new event is allowed to run forever then the abstract event possibly may not occur. For example, as outlined in the first refinement of the abstract model of transactions given in Chapter 3, if the new events such as BeginSubTran, SiteAbortTx or SiteCommitTx take the control forever then the events of global commit/abort are never activated and a global commit decision may never be achieved.

In order to prove that the new events do not diverge, we use a VARIANT construct. A variant \( V \) is a variable such that \( V \in \mathbb{N} \), where \( \mathbb{N} \) is a set of natural numbers. For each new event in the refinement we should be able to demonstrate that the execution of each new event decreases the variant and the variant never goes below zero. This allows us prove that a new event can not take control forever, since a variant can not be decreased indefinitely. In order to achieve this, the most challenging task is to construct a variant expression and prove that it is preserved by the activation of the events. The process of the construction of a variant expression for the first refinement of the model of transactions is as below.

In the refinement of our model of transactions, the notion of sites and the status of a transaction at a site is introduced. The new events in the refinement change the concrete state of the transactions. A transaction state at each participating site is first set to pending by the activation of BeginSubTran. The activation of the event SiteCommitTx changes the status from pending to precommit while the activation of SiteAbortTx sets the status from pending to abort at that site. A transaction in the precommit state at a site changes the state to either commit or abort by the activation of event ExeCommitDecision or ExeAbortDecision respectively. The state diagram for a concrete transaction state transitions at a site is shown in the Fig. 7.2. As shown in
the figure each state is represented by a rank. The initial state represents a state of a transaction \( tt \) at a participating site \( ss \) when it is not active, i.e., the \( \text{sitetranssstatus} \) of \( tt \) at \( ss \) is not defined (\( ss \notin \text{dom(sitetranssstatus(tt))} \)). After submission of a transaction, a transaction first becomes active at the coordinator site. Subsequently, due to the activation of the event \( \text{BeginSubTran}(tt,ss) \), sub-transactions are started separately at different sites, i.e., at each activation of this event, \( tt \) becomes active at participating sites \( ss \). As shown in the figure, new events in the refinement change the state of a transaction at a site such that each time the rank is decreased.

![Figure 7.2: Concrete Transaction States in the Refinement](image)

A variant in the refinement is defined as a variable \( \text{variant} \):

\[
\text{variant} \in \text{trans} \rightarrow \text{Natural}
\]

and initialized as \( \text{variant} := \emptyset \).

As shown in the Fig. 7.3, when a fresh transaction \( tt \) is submitted by the activation of the event \( \text{StartTran}(tt) \), the initial value of variant is set as \( \text{variant}(tt) := 3 \times N \), where \( N \) is total number of the sites in the system. Instead of showing all new events that decrease the variant, the events \( \text{SiteCommitTx} \) and \( \text{SiteAbortTx} \) that decrease the variant are shown in the Fig. 7.4. It can be noticed that both events decrease the variant and change the status of a transaction from a pending state to precommit or abort state. Since activation of the new events in the refinement decrease the variant, the rank of
state is changed from three to zero, such that, $\text{variant}(tt)$ will always be greater than or equal to zero.

\[ \text{StartTran}(tt) \equiv \]
\[ \text{ANY } ss, \text{updates, objects} \]
\[ \text{WHERE } ss \in \text{SITE} \]
\[ \wedge tt \notin \text{trans} \]
\[ \wedge \text{updates} \in \text{UPDATE} \]
\[ \wedge \text{objects} \in \mathcal{P}_1(\text{OBJECT}) \]
\[ \wedge \text{ValidUpdate}(\text{updates, objects}) \]
\[ \text{THEN} \]
\[ \text{trans} := \text{trans} \cup \{tt\} \]
\[ \text{transstatus}(tt) := \text{PENDING} \]
\[ \text{transobject}(tt) := \text{objects} \]
\[ \text{transeffect}(tt) := \text{updates} \]
\[ \text{coordinator}(tt) := ss \]
\[ \text{sitetransstatus}(tt) := \{\text{coordinator}(tt) \rightarrow \text{pending}\} \]
\[ \text{variant}(tt) := 3 \times N \]
\[ \text{END}; \]

\[ \text{Figure 7.3: Variant} \]

\[ \text{SiteCommitTx}(tt, ss) \equiv \]
\[ \text{WHEN} \]
\[ (ss \rightarrow tt) \in \text{activetrans} \]
\[ \wedge \text{sitetransstatus}(tt)(ss) = \text{pending} \]
\[ \wedge ss \neq \text{coordinator}(tt) \]
\[ \wedge \text{ran(\text{transeffect}(tt))} \neq \emptyset \]
\[ \text{THEN} \]
\[ \text{sitetransstatus}(tt)(ss) := \text{precommit} \]
\[ \text{variant}(tt) := \text{variant}(tt) - 1 \]
\[ \text{END}; \]

\[ \text{SiteAbortTx}(tt, ss) \equiv \]
\[ \text{WHEN} \]
\[ (ss \rightarrow tt) \in \text{activetrans} \]
\[ \wedge \text{sitetransstatus}(tt)(ss) = \text{pending} \]
\[ \wedge ss \neq \text{coordinator}(tt) \]
\[ \wedge \text{ran(\text{transeffect}(tt))} \neq \emptyset \]
\[ \text{THEN} \]
\[ \text{sitetransstatus}(tt)(ss) := \text{abort} \]
\[ \text{freeobject} := \text{freeobject} \cup \{ss\} \times \text{transobject}(tt) \]
\[ \text{activetrans} := \text{activetrans} - \{ss \rightarrow tt\} \]
\[ \text{variant}(tt) := \text{variant}(tt) - 2 \]
\[ \text{END}; \]

\[ \text{Figure 7.4: Events Decreasing a Variant} \]

In order to prove that the activation of the new events given in the Fig. 7.2 does not diverge, we need to prove that the changes in the state of a transaction at a site corresponds to the decrement in the rank from three to zero. The variable $\text{variant}$ is decreased each time a new event in the refinement is activated. Thus, we construct the invariant
∀ \( t \in \text{trans} \Rightarrow \)
\[
\text{variant}(t) \geq (3 \ast \text{card}(\text{SITE} - \text{activetrans}^{-1}[\{t\}])
+ 2 \ast \text{card}(\text{sitetransstatus}(t)^{-1}[\{\text{pending}\}])
+ 1 \ast \text{card}(\text{sitetransstatus}(t)^{-1}[\{\text{precommit}\}])
+ 0 \ast \text{card}(\text{sitetransstatus}(t)^{-1}[\{\text{commit, abort}\}])
) \]

Figure 7.5: Invariant used in variant Proofs

property involving the variable \( \text{variant} \) that need to be satisfied by the action of the events in the refinement. This property is given in Fig 7.5.

In this expression, \( \text{activetrans}^{-1}[\{t\}] \) returns a set of sites where transaction \( t \) is in active state. Similarly, \( \text{sitetransstatus}(t)^{-1}[\{\text{pending}\}] \) returns a set of site where a transaction \( t \) is in pending state. In order to prove that the new events in the refinement do not diverge, we have to show that the above invariant property on a variable \( \text{variant} \) holds on the activation of the events in the refinement. In order to prove this invariant property we need to add invariants 7.1 and 7.2 to the model. The invariant 7.1 states that if a transaction \( t \) is not active at a site \( s \) then the variable \( \text{variant} \) is greater than or equal to zero. The invariant 7.2 states that the variable \( \text{variant} \) is greater than or equal to zero if the status of a transaction \( t \) at site \( s \) either precommit, pending, abort or commit.

\[
\forall (s, t) \cdot (t \in \text{trans} \land s \in \text{SITE} \land (s \not\rightarrow t) \notin \text{activetrans} \\
\Rightarrow \text{variant}(t) \geq 0) \tag{7.1}
\]
\[
\forall (s, t) \cdot (t \in \text{trans} \land s \in \text{SITE} \land \text{sitetransstatus}(t)(s) \in \{\text{pending, precommit, abort, commit}\}) \\
\Rightarrow \text{variant}(t) \geq 0) \tag{7.2}
\]

Therefore, in order to prove that the new events in the refinement do not diverge, we need to construct an invariant on \( \text{variant} \) that holds on the activation of the events in the refinement such that each new event in the refinement decreases the variant and variant never goes below zero. Also, to prove an invariant property that includes a variant, we need to construct new invariants that are sufficient to discharge the proof obligations.

7.2.3 Enabledness Preservation

With respect to liveness, freedom from deadlock is an important property in a distributed database system. Our model of transactions requires us to prove that each transaction eventually completes execution, i.e., either it commits or aborts. With respect to Event-B models, it requires us to prove that if a transaction completes execution in the abstract model of a system, then it must also complete in the concrete model. We ensure this property by enabledness preservation.
Enabledness preservation requires us to prove that the guards of the one or more events in the refinement are enabled under the hypothesis that the guard of one or more events in the abstraction are also enabled. Precisely, let there exist events $E_a^1, E_a^2, ..., E_a^n$ in the abstraction and a corresponding event $E_r^i$ in the refinement refines the abstraction event $E_a^i$. The events $H_r^1, ..., H_r^k$ are the new events in the refinement. A weakest notion of enabledness preservation can be defined as follows:

$$Grd(E_a^1) \lor Grd(E_a^2) \lor ... \lor Grd(E_a^n) \Rightarrow Grd(E_r^1) \lor Grd(E_r^2) \lor ... \lor Grd(H_r^1) \lor Grd(H_r^2) \lor ... \lor Grd(H_r^k) \quad (7.3)$$

The weakest notion of enabledness preservation given at 7.3 states that if one or more events in the abstraction is enabled then one or more events in the refinement are also enabled. The strongest notion of the enabledness can be defined as below:

$$Grd(E_a^i) \Rightarrow Grd(E_r^i) \lor Grd(H_r^1) \lor Grd(H_r^2) \lor ... \lor Grd(H_r^k) \quad (7.4)$$

The notion of enabledness preservation defined in 7.4 states that if the event $E_a^i$ in the abstraction is enabled then either the refining event $E_r^i$ is enabled or one of the new events are enabled.

We have also outlined in Chapter 3 that a concrete model may be deadlocked due to race conditions. To ensure that all updates are delivered to all sites in the same order, we need to order update transactions such that all sites deliver updates in the same order. This may achieved if a site broadcasts an update using a total order broadcast. In the presence of abstract ordering on the update transactions, all updates are delivered to all sites in a same order, thus the concrete model does not deadlock. In the next section, we outline how enabledness preservation properties relates to our model of transactions in the presence of abstract ordering on the update transactions.

### Abstract Transaction States

In our model of transactions, an update transaction, once started, updates the abstract database atomically when the transaction commits, or makes no changes in the database, when it aborts. We have represented the global state of update transactions by a variable $transstatus$ in the abstract model of the transactions. The variable $transstatus$ is defined as $transstatus \in trans \rightarrow TRANSSTATUS$, where $TRANSSTATUS = \{COMMIT, ABORT, PENDING\}$. The $transstatus$ maps each transaction to its global state. With respect to an update transaction, activation of the following events change the global transaction states.
- \textit{StartTran}(tt) : The activation of this event \textit{starts} a fresh transaction and the state of the transaction is set to \textit{pending}.

- \textit{CommitWriteTran}(tt) : A \textit{pending} update transaction commits by atomically updating the abstract database and its status is set to \textit{commit}.

- \textit{AbortWriteTran}(tt) : A \textit{pending} update transaction aborts by making no change in the abstract database and its status is set to \textit{abort}.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{transaction_states.png}
\caption{Transaction States in the Abstract Model}
\end{figure}

The transitions in the transaction states due to the activation of events in the abstract model of the transactions are outlined in the Fig. 7.6. \textit{CommitWriteTran}(tt) and \textit{AbortWriteTran}(tt) together are represented as \textit{Complete}(tt), as both of these events model the completion of a update transaction. As outlined in the figure, our abstract model of transactions is free from deadlock, since an update transaction in the abstract model commits atomically by updating the database or aborts by doing nothing. However, in the refinement, update transaction consists of collections of interleaved events updating each replica separately. Due to the interleaved execution of transactions at several sites, we need to show that the concrete model does not deadlock in the presence of a total order broadcast.

\textbf{Abstract Ordering on the transactions}

As outlined in the refinement of the model of transactions given as \textit{Replica2} in Chapter 3, in addition to a notion of a replicated database and the sites, new events are also introduced. It may be recalled that in this refinement conflicting transactions may be blocked. In order to ensure that our concrete model of transactions does not block and makes progress, we introduce a new event \textit{Order} in the refinement. The very purpose of introducing new event \textit{Order}(tt) is to ensure that the transactions are executed at all sites in a predefined abstract order on the transactions. The event \textit{Order} models generation of abstract ordering on the update transactions. For the purpose of simplicity,
the event $\text{IssueWriteTran}(tt)$ is also merged with $\text{BeginSubTran}(tt)$ such that the event $\text{BeginSubTran}(tt)$ models starting a sub-transaction at a site including the coordinator. In the refined model, the sub-transactions at the participating sites are started in the order of the abstract ordering on the transactions. This abstract ordering on the transactions can be realized by introducing explicit total ordering on the messages in further refinements.

To model an abstract order on the transactions we introduce new variables $\text{tranorder}$ and $\text{ordered}$ typed as follows:

\[
\text{tranorder} \subseteq \text{trans} \leftrightarrow \text{trans} \\
\text{ordered} \subseteq \text{trans}
\]

A mapping of the form $t_1 \mapsto t_2 \in \text{tranorder}$ indicates that a transaction $t_1$ is ordered before $t_2$, i.e., at all sites $t_1$ will be processed before $t_2$. It may be noted that the abstract transaction ordering can be achieved by implementing total ordering on all update messages. In order to represent the state of a transaction at a site, we use a variable $\text{sitetransstatus}$. The variable $\text{sitetransstatus}$ maps each transaction, at a site, to transaction states given by a set $\text{SITETRANSTATUS}$, where $\text{SITETRANSTATUS} = \{\text{pending, commit, abort, precommit}\}$. The $\text{Order}$ event models building an abstract transaction order on the started transactions. The event $\text{BeginSubTran}$ models starting a sub-transaction in the order defined by the abstract variable $\text{tranorder}$. The specifications of the events $\text{Order}$ and $\text{BeginSubTran}$ are given in the Fig. 7.7.

Instead of giving the specifications of all events of the refinement in the similar detail, brief descriptions of the new events in this refinement are outlined below.

- $\text{Order}(tt)$: This event builds an abstract order on the transactions.
- $\text{BeginSubTran}(tt)$: This event models starting a sub-transaction at a site including the coordinator. The sub-transactions are started in the order of abstract transaction order. The status of the transaction $tt$ at site $ss$ is set to $\text{pending}$.
- $\text{SiteAbortTx}(ss,tt)$: This event models a local abort of a transaction at a site. The transaction is said to complete execution at the site. The status of the transaction $tt$ at site $ss$ is set to $\text{abort}$.
- $\text{SiteCommitTx}(ss,tt)$: This event models precommit of a transaction at a site. The status of the transaction $tt$ at site $ss$ is set to $\text{precommit}$.
- $\text{ExeAbortDecision}(ss,tt)$: This event models abort of a precommitted transaction at a site. This event is activated once the transaction has globally aborted. The status of the transaction $tt$ at site $ss$ is set to $\text{abort}$. The transaction is said to complete execution at the site.
Fig. 7.7: Order and BeginSubTran events

- ExeCommitDecision(ss, tt): This event models commit of a precommitted transaction at a site. This event is activated once the transaction has globally committed. The status of the transaction tt at site ss is set to precommit. The replica at the site is updated with the transaction effects and the transaction is said to complete execution at this site.

The transaction states in the refinement is outlined in the Fig. 7.8 and 7.9. As shown in the figure, initially the status of a fresh transaction is set to pending by the activation of StartTran event. A pending transaction is ordered by the activation of the Order event, before it starts a sub-transaction at a participating site. A site starts a sub-transaction in transaction order and independently decides to either abort or precommit the sub-transaction by activation of either SiteAbortTx or SiteCommitTx. These new events of the refinement set the status of transactions to abort or precommit. The coordinating site takes a decision of global commit by the activation of either CommitWriteTran or AbortWriteTran events. It can be noticed that both CommitWriteTran(tt) and AbortWriteTran(tt) events together are represented as Complete(tt), as both of these events model the completion of a update transaction. An update transaction then reaches the final state of a global commit or abort. A site implements the
global commit decision to update the replica at that site by the activation of \textit{ExeCommitDecision} at participating sites. This event takes place only after the activation of \textit{CommitWriteTran}. Similarly, a site implements a global abort decision by the activation of \textit{ExeAbortDecision}. This event occurs after the activation of \textit{AbortWriteTran} at the coordinator. These events set the transaction status at that site to \textit{abort} or \textit{commit}.

![Figure 7.8: Transaction States in the Refinement-I](image-url)
Proof Obligations for Enabledness Preservation

In this section, we outline the proof obligations to verify that the refinement is enabledness preserving. Our objective is to prove that if a transaction completes in the abstraction then it also completes in the refinement. The weakest notion of enabledness preservation\(^1\) given at 7.3 requires us to prove following:

\[
Grd(StartTran(t) \lor CommitWriteTran(t) \lor Grd(AbortWriteTran(t))) \\
\Rightarrow Grd(StartTran^*(t)) \\
\lor Grd(Order(t)) \\
\lor Grd(BeginSubTran(t,s)) \\
\lor Grd(SiteCommitTx(t,s)) \\
\lor Grd(SiteAbortTx(t,s)) \\
\lor Grd(CommitWriteTran^*(t)) \\
\lor Grd(AbortWriteTran^*(t))
\] (7.5)

The property given at 7.5 is not sufficient as it states that if one or more events in StartTran, AbortWriteTran or CommitWriteTran is enabled in the abstraction then one of the refined events or the new events is enabled in the refinement. It does not guarantee that if a transaction \(t\) completes in the abstraction then it also completes in the refinement. What we need to prove is that if either AbortWriteTran or CommitWriteTran in

\(^1\)An event \(E\) in the abstract model is defined as \(E^*\) in the refinement.
the abstraction is enabled then one of the refined events or new events in the refinement is enabled. According to the strongest notion of enabledness preservation given at 7.4, it requires us to prove 7.6, 7.9 and 7.10.

\[
Grd(StartTran(t)) \implies Grd(StartTran^*(t)) \\
\lor Grd(Order(t)) \\
\lor Grd(BeginSubTran(t,s)) \\
\lor Grd(SiteCommitTx(t,s)) \\
\lor Grd(SiteAbortTx(t,s)) \tag{7.6}
\]

The property at 7.6 states that if the guard of the \textit{StartTran} event is enabled then the guard of refined \textit{StartTran} or the guards of new events are enabled. This property is provable due to following observations.

\[
Grd(StartTran(t)) \implies Grd(StartTran^*(t)) \tag{7.7}
\]

In order to prove this property, the following proof obligation needs to be discharged. This proof obligation is trivial and can be discharged by the automatic prover of the tool.

\[
\forall t (t \in TRANSACTION \land t \notin \text{trans} \implies t \notin \text{trans}) \tag{7.8}
\]

\[
Grd(CommitWriteTran(t)) \implies Grd(Order(t)) \\
\lor Grd(BeginSubTran(t,s)) \\
\lor Grd(SiteCommitTx(t,s)) \\
\lor Grd(SiteAbortTx(t,s)) \\
\lor Grd(CommitWriteTran^*(t)) \tag{7.9}
\]

The property at 7.9 states that if the guard of the \textit{CommitWriteTran} event is enabled then the guards of refined \textit{CommitWriteTran} or the guards of new events are enabled. This property is too strong to prove due to following reasons. A transaction may not commit in the refinement until some other transaction either commits or aborts. Therefore, the guards of the \textit{AbortWriteTran} may be enabled, as commit of a transaction depends on the abort of other transaction. Also, for the same reasons the property at 7.10 is not provable either.
Therefore, we need to reconstruct the properties given at 7.9 and 7.10 given as 7.11 and 7.12 respectively. It can be noticed that we need to prove that if the guards of the events \( \text{AbortWriteTran} \) or \( \text{CommitWriteTran} \) are enabled in the abstract model then either the guards of new events or the guards of refined \( \text{AbortWriteTran} \) or \( \text{CommitWriteTran} \) events are enabled in the refinement.

\[
\begin{align*}
\text{Grd}(\text{AbortWriteTran}(t)) & \Rightarrow \text{Grd}(\text{Order}(t)) \\
\lor \text{Grd}(\text{BeginSubTran}(t,s)) & \\
\lor \text{Grd}(\text{SiteCommitTx}(t,s)) & \\
\lor \text{Grd}(\text{SiteAbortTx}(t,s)) & \\
\lor \text{Grd}(\text{AbortWriteTran}^*(t)) & \quad (7.10)
\end{align*}
\]

We observe that the proof obligations constructed due to the weakest notion of the enabledness preservation are not sufficient to prove that if a transaction completes in abstraction then it also completes in the refinement. Also, we observe that the strongest notion of enabledness preservation is too strong to prove.

What we really need is a notion of enabledness preservation that is stronger than the
weakest notion (see property 7.3) and weaker than the strongest notion (see property 7.4). This can be defined as below.

1. If the event \( \text{StartTran} \) is enabled in the abstraction then it is also enabled in the refinement.

2. If the completion event, i.e., either \( \text{CommitWriteTran} \) or \( \text{AbortWriteTran} \) event is enabled in the abstract model then these completion events are also enabled in the refinement.

We have already outlined that the first property is preserved by our model of transactions given at 7.7. For the second property, we further construct the property given at 7.13.

\[
\text{Grd}(\text{CommitWriteTran}(t)) \lor \text{Grd}(\text{AbortWriteTran}(t)) \\
\Rightarrow \text{Grd}(\text{Order}(t)) \\
\lor \text{Grd}(\text{BeginSubTran}(t, s)) \\
\lor \text{Grd}(\text{SiteCommitTx}(t, s)) \\
\lor \text{Grd}(\text{SiteAbortTx}(t, s)) \\
\lor \text{Grd}(\text{CommitWriteTran}^*(t)) \\
\lor \text{Grd}(\text{AbortWriteTran}^*(t))
\] (7.13)

We observe that property 7.13 is also not provable because a transaction \( t \) cannot complete its execution until some other transaction completes. Therefore, we finally construct the property 7.14.

\[
\text{Grd}(\text{CommitWriteTran}(t_x)) \lor \text{Grd}(\text{AbortWriteTran}(t_x)) \\
\Rightarrow \exists t_y \cdot \text{Grd}(\text{Order}(t_y)) \\
\lor \exists t_y \cdot \text{Grd}(\text{BeginSubTran}(t_y, s)) \\
\lor \exists t_y \cdot \text{Grd}(\text{SiteCommitTx}(t_y, s)) \\
\lor \exists t_y \cdot \text{Grd}(\text{SiteAbortTx}(t_y, s)) \\
\lor \text{Grd}(\text{CommitWriteTran}^*(t_x)) \\
\lor \text{Grd}(\text{AbortWriteTran}^*(t_x))
\] (7.14)

As shown in 7.14, if the events corresponding to a completion of a transaction \( t_x \) in the abstraction are enabled then the new events \( \text{Order}, \text{BeginSubtran}, \text{SiteCommitTx}, \text{SiteAbortTx} \) are enabled for other transactions \( t_y \) or the refined \( \text{Complete} \) events are also enabled for \( t_x \). Since we allow the interleaving of the transactions in the refinement, if a transaction \( t_x \) completes by a commit then another transaction \( t_y \) may also complete by an abort.
The proof obligations for the property 7.14 can be simplified as follows. For a given transaction \( t \) that has \textit{started} but not \textit{ordered} then the event \textit{Order} activates before the activation of other events in the refinement. Therefore, if one or both of the abstract events \textit{CommitWriteTran} or \textit{AbortWriteTran} is enabled and the transaction is not \textit{ordered} then the guard of event \textit{Order} in the refinement must be enabled. The proof obligation corresponding to this property is given below:

\[
\text{Grd}(\text{Complete}(t)) \land t \notin \text{ordered} \Rightarrow \text{Grd}(\text{Order}(t)) \quad (7.15)
\]

The proof obligation 7.15 can be simplified by replacing \textit{Complete}(t) by the transaction completion events shown as below.

\[
\text{Grd}(\text{CommitWriteTran}(t)) \lor \text{Grd}(\text{AbortWriteTran}(t)) \\
\land t \notin \text{ordered} \Rightarrow \text{Grd}(\text{Order}(t)) \quad (7.16)
\]

Similarly, if a transaction \( t \) is \textit{ordered} then the guard of the events \textit{BeginSubTran}, \textit{SiteCommitTx}, \textit{SiteAbortTxC} or the refined \textit{Complete} must be enabled. The proof obligation corresponding to this property is given below.

\[
\text{Grd}(\text{Complete}(t)) \land t \in \text{ordered} \\
\Rightarrow \text{Grd}(\text{BeginSubTran}(t,s)) \\
\lor \exists t_y \cdot \text{Grd}(\text{SiteCommitTxC}(t_y,s)) \\
\lor \exists t_y \cdot \text{Grd}(\text{SiteCommitTxC}(t_y,s)) \\
\lor \text{Grd}(\text{Complete}^*(t)) \quad (7.17)
\]

The proof obligation 7.17 may further be simplified under following observations. Consider a transaction \( t \) that has \textit{started}, \textit{ordered} but it is not active at a site \( s \) then the guard of \textit{BeginSubTran}(t,s) must be enabled. In order to prove this property, the following proof obligation needs to be discharged.

\[
\text{Grd}(\text{Complete}(t)) \\
\land t \in \text{ordered} \\
\land (s \mapsto t) \notin \text{activetrans} \\
\land \forall tx \cdot (tx \in \text{trans} \land (tx \mapsto t) \in \text{tranorder} \Rightarrow (t \mapsto s) \in \text{completed} \\
\Rightarrow \text{Grd}(\text{BeginSubTran}(t,s)) \quad (7.18)
\]

Further, if a transaction \( t \) that has \textit{started}, \textit{ordered} and is active at a site \( s \) then the guard of \textit{SiteCommitTxC}, \textit{SiteAbortTxC} or the refined \textit{Complete}(t) must be enabled. This
proof obligation is given below.

\[
Grd(Complete(t)) \land t \in ordered \land (s \mapsto t) \in activetrans \\
\Rightarrow \exists t_y \cdot Grd(SiteCommitT_x(t_y, s)) \\
\lor \exists t_y \cdot Grd(SiteAbortT_x(t_y, s)) \\
\lor Grd(Complete^*(t)) \tag{7.19}
\]

The existing B tools do not generate proof obligations for enabledness preservation. However, the issue of enabledness preservation and non-divergence is being addressed in the new generation of B tools, e.g., Rodin [44]. The proof obligations outlined above are specific to our model of transactions. However, using the same strategy, the proof obligations for other models of distributed systems may be generated. Discharging these proof obligations ensures that the model is enabledness preserving. The same strategy needs to be used to formulate the proof obligations for each level of refinement.

7.3 Guidelines for an Event-B Development

In this section, we briefly outline the guidelines for the incremental construction of a model of a distributed system in Event-B. It is our understanding that most distributed algorithms are deceptive and they may allow unanticipated behavior during execution. There exists a vast variety of problems related to distributed systems. There also exists several solutions to each of these problems. A formal verification is required to understand that these algorithms achieve what they are supposed to do. The guidelines presented here are particularly helpful if the main purpose of the construction of a model of a distributed system is to specify the abstract problem and to verify the correctness of a proposed solution or a design decision in the refinement steps.

Firstly, we present the general methodological guidelines for modelling in Event-B. Subsequently, the guidelines for an effective management of the B tools to discharge the proof obligations, are presented. These guidelines emerged from the experience of our case studies [139, 140, 141, 142] presented in this thesis and the Mondex case study [31].

7.3.1 General Methodological Guidelines for Modelling in Event-B

In this section, general methodological guidelines for the construction of models of distributed systems in Event-B, are presented.
Guideline 1:

*Sketch the informal requirements and the safety properties of a system*

Before undertaking the development of a large distributed system, it is necessary to formulate informal definitions and the requirements of a system. Formulation of these requirements varies with the system. However, each system requires a clear description of the *functional* and *safety* requirements. The functional requirement usually deals with the main function of the system. The safety requirements underline the critical properties that a system must meet. Formulation of the informal requirements should be an iterative process, which should go on together with the development of the formal models.

In the development of the formal models for the case studies outlined in this thesis, we considered the protocol steps as functional requirements for the construction of formal models. For example, we considered the *read anywhere write everywhere* replica control protocol for the management of replicas in Chapter 3, vector clocks for implementing causal ordering in Chapter 4, sequencer based protocol for total order in Chapter 5 and vector clocks for implementing total causal order in Chapter 6. After the construction of the formal models, at each refinement step proof obligations are generated by the B tool for refinement and consistency checking. By discharging these proof obligations, we ensure that a refinement meets safety requirements. Additionally, in order to prove that a model also preserves critical properties of the protocol, we further construct and add primary invariants to the model that represent critical properties of the system. The addition of these primary invariants to the model generates additional proof obligations. While discharging the proof obligations, we also discover a set of new invariants called secondary invariants. These proof obligations and invariants provide a deeper insight into the system and help us understand why a protocol meets the critical properties.

Guideline 2:

*Use the refinement approach to gain insight*

An abstract model is generally regarded as an bird’s eye view of the system. It is important to make sure that the abstract model appropriately reflects the overall view of the system under development. The first attempt in the formal devolvement of the system should be on the modelling of the problem in the abstract model rather than proposing the solution. Once the abstract problem is defined in the abstract model, the detailed solutions of the problem should gradually be introduced in the refinement steps. The modelling assumptions made in the construction of an abstract model are crucial. The informal process of reconciling the requirements at each level of refinement
help discovering invalid modelling assumptions. A critical view of the proof obligations
generated by the tools also helps discovering invalid modelling assumptions. It should
be remembered that the state of the refined system is largely constrained by the choice
of variables and the events in the abstraction.

Consider our model of transactions in Chapter 3. In the abstract model, an update
transaction performs update on a one-copy database. In the refinement, we introduce
the notion of replicas. Replicas in the refinement are updated within the framework of a
read anywhere write everywhere protocol. The proof obligations and the invariants dis-
covered at this stage provide insight into why a one-copy database is a valid abstraction
of replicated databases, i.e., why a replicated database preserves the one-copy equiva-
lence consistency criterion. Similarly, in Chapter 4, in the abstract model we define
abstract causal ordering on the messages and in a refinement we introduce vector clocks
to implement causal ordering. Both proof obligations and the discovered invariants help
understand why a vector clock mechanism implements abstract causal ordering on mes-
sages. Also, using the same technique in Chapter 5, in the abstract model we outline
how a total order is constructed on the messages and in the refinements we introduce
the details of control messages and sequence numbers. Through the proof and new in-
variants, we understand how the mechanism of generating sequence numbers delivers
the messages correctly in a total order.

Guideline 3 :

*Keep abstraction gap as small as possible*

During the development of a refinement chain, keep the abstraction gap as small as
possible. Precisely, while adding new events and variables in the refinement, it is good
to keep the state space representation as abstract as possible. Allowing a very detailed
state space in a single refinement step may require discharging lengthy and complex
proof obligations. Keeping the abstraction gap smaller means discharging less complex
proof obligations. Discharging a proof obligation may also require addition of the new
invariants to the model. A large and complex proof obligation may require a huge
amount of work which otherwise could be split into easier and smaller units of work.
Since an invariant for the abstract models is available for free for the refined model,
smaller abstraction gaps in each refinement step help splitting otherwise complex proof
steps into the simpler ones. Also, keeping smaller abstraction gaps may lead to a higher
degree of automatic proofs, since a relatively simple proof obligation is more likely to
be discharged by the automatic prover.

For example, consider the first refinement given in the Chapter 3. Due to the intro-
duction of the replica control mechanism and a large number of concrete variables, we
observe a vary large concrete state space. Therefore, we end up discharging a relatively
large number of complex proof obligations compared to the other refinement steps, as outlined in the Table 3.2. Also, due to large abstraction gaps in the first two refinements of the refinement chain in the Chapter 6, a relatively large number of complex proof obligations is generated, as shown in the Table 6.2. However, discharging these proof obligations was relatively easy as most of the invariant properties were already discovered as a part of development of the model of a causal order in Chapter 4 and a total order in Chapter 5.

Guideline 4:

**Tools are critical in managing proofs and the refinement chain**

The B tools are central to Event-B modelling. They greatly ease the burden of modelling efforts by the generation of proof obligations, remembering the proof steps, discharging the proof obligations and maintaining the refinement chain. The complexity of the proof obligations generated by the tool are also dependent on the way the B constructs are used in the modelling. For example, use of the relational override operator may generate more complex proof obligations than using set union. Similarly, the tool may generate simple proof obligations if the state of a variable is represented by a set variable construct rather than using enumerated sets. For example, one way of modelling computation and control messages is by using a variable $mtype \in MESSAGE \rightarrow MTYPE$ where $MTYPE = \{\text{computation, control}\}$ and assigning the type of a message as $mtype(mm) := \text{computation}$. An easier step could be to declare variable computation, control as computation $\subseteq MESSAGE$ and assigning the type of a message as computation $:= \text{computation} \cup \{ mm \}$. An invariant computation $\cap$ control $= \emptyset$ ensures that both messages are distinct. The prover generates relatively easier proof obligations for the later and discharges the proof automatically.

Guideline 5:

**Let the proof obligations guide construction of the gluing invariants**

In our case studies we have outlined the construction of the gluing invariants by inspection of the proof obligations. The proof obligations generated by the B tool contain sufficient information to construct new invariants. However, in the first instance an attempt should be made to discharge a proof obligation through interaction with the tool by inspecting available hypotheses and the invariants. In many cases, there may not be a need to add a new invariant, rather an interaction with the tool e.g., simplifying the hypotheses and goals or by providing a good instantiation will suffice. The addition of a new invariant to the model must be seen as a last solution and must be constructed after a very careful examination of the proof obligations, available hypotheses and the existing
invariants. It should be remembered that each newly constructed invariant needs to be proved to be consistent for each event in the model. It is also necessary to convince yourself informally that a newly discovered invariant is expected to be an invariant. In some cases, a new invariant may also provide a clue that either previously constructed invariants or the model itself need to be fixed. A blind construction of a new invariant may result in a growth in the number of proof obligations or may lead to invalid changes in the model which may result in a situation of proving a wrong invariant for an invalid model. In the case of an addition of an invariant, efforts should be made to construct a new invariant which is close to the form of the proof obligations. By adding an invariant which is close to the proof obligation, the proof efforts are usually eased.

For example, as outlined in the Chapter 4, consider the following two proof obligations generated by the B tool. The first proof obligation requires us to prove that if a message \( m_1 \) causally precedes \( m_2 \) and that \( pp \) is sender of \( m_2 \) and \( m_1 \) was not sent by process \( pp \) then process \( pp \) must have delivered \( m_1 \).

\[
m_1 \rightarrow m_2 \in corder \land m_2 \in (sender^{-1}\{pp\}) \land m_1 \notin (sender^{-1}\{pp\}) \\
\Rightarrow m_1 \in (cdeliver\{pp\})
\]  

(7.20)

Similarly, the second proof obligation states that if \( m_1 \) causally precedes \( m_2 \) and \( pp \) is the sender of \( m_2 \) and \( pp \) has not delivered \( m_1 \) then \( pp \) is sender of \( m_1 \).

\[
m_1 \rightarrow m_2 \in corder \land m_2 \in (sender^{-1}\{pp\}) \land m_1 \notin (cdeliver\{pp\}) \\
\Rightarrow m_1 \in (sender^{-1}\{pp\})
\]  

(7.21)

Therefore, in order to discharge these proof obligations, we add the following invariant to the model that is close to the form of these proof obligations. This invariant states that if \( m_1 \) precedes \( m_2 \) in causal order, \( p \) is sender of \( m_2 \), then \( p \) has either delivered \( m_1 \) or it is a sender of \( m_1 \). We observe that this invariant is sufficient to discharge these proof obligations.

\[
m_1 \rightarrow m_2 \in corder \land m_2 \in (sender^{-1}\{p\}) \\
\Rightarrow m_1 \in (sender^{-1}\{p\}) \lor m_1 \in (cdeliver\{p\})
\]

We also outlined the construction of invariants in the chapters 3-6 guided by the proof obligations.
Guideline 6:

**Frequently use model checker to understand the prover failure**

Discharging complex proof obligations using the interactive prover is quite a tricky affair. In the event of a prover’s failure to discharge a proof obligation, it is not always possible to determine if the prover could not prove the goal due to the inappropriate selection of the hypotheses or the goal can not be proved at all under the available hypotheses. One of the main limitations [6] of the predicate prover of the existing tools is its sensitivity towards useless hypotheses. The predicate prover may prove a certain statement with a selection of right hypotheses but may not prove or takes much longer time to prove the same statement under the selection of a large number of useless hypotheses.

This situation is more of importance if the proof obligation was generated due to the addition of a new invariant. In such cases it is necessary to determine informally if the newly constructed invariant is a valid invariant and the model needs to be fixed or the invariant is violated due to activation of an certain event. The use of a model checker (ProB) is strongly recommended to precisely understand how the state variables are changed due to the activation of events and what invariants are violated. The model checker can also be used to find counter examples which may lead to fixing the model or invariants.

For example, as outlined in the Chapter 4, a **causal order broadcast** is a reliable broadcast that satisfies the **causal order** requirement, i.e., a causal order broadcast delivers messages respecting their causal precedence relationship. In order to verify that our model of causal order, given as first refinement, preserves causal order properties, we considered the following two properties relating abstract causal order and delivery order.

\[
m_1 \rightarrow m_2 \in \text{corder} \Rightarrow m_1 \rightarrow m_2 \in \text{delorder}(p) \quad (7.22)
\]

\[
m_1 \rightarrow m_2 \in \text{delorder}(p) \Rightarrow m_1 \rightarrow m_2 \in \text{corder} \quad (7.23)
\]

The property at 7.22 states that for any two messages \(m_1\) and \(m_2\) such that \(m_1\) precedes \(m_2\) in causal order then the delivery order at a process \(p\) is also \(m_1\) followed by \(m_2\). The property at 7.23 states that given two messages \(m_1\) and \(m_2\), and \(m_1\) is delivered before \(m_2\) to a process \(p\) then \(m_1\) precedes \(m_2\) in causal order. We use model checking to precisely understand why and when both of the above are not the invariant properties. The property at 7.22 is not an invariant property as the causal order is constructed at the time of sending a message and the messages are delivered after arbitrary time. Similarly, the property at 7.23, is not an invariant property due to parallel messages, i.e., parallel messages may be delivered to all processes in same delivery order. After frequent use of
the model checker and animator (ProB), we arrive at the following invariant property.

\[
\begin{align*}
m1 \mapsto m2 \in corder & \land p \mapsto m2 \in cdeliver \\
\Rightarrow m1 \mapsto m2 \in delorder(p)
\end{align*}
\]

Guideline 7:

\textit{Redundancy may be useful}

A redundant variable is one whose value may be extracted from other variables of the model. Using redundant variables may be helpful in constructing the gluing invariants and the generation of relatively simple proof obligations. These variables may gradually be removed in the subsequent refinement steps. While removing the redundant variables in the refinement, the proofs may be easier due to the existing invariants. However, introducing too much redundancy in the model may lead to increased effort in managing it. For example, variables \textit{activetrans}, \textit{sitetransstatus}, \textit{freeobject} used in the first refinement in Chapter 3 help discharge several complex proof obligations. However, there exists a strong relationship among them as outlined in the second refinement.

Guideline 8:

\textit{Be aware that Refinement chains are not always top down}

Contrary to the general belief, refinement chains are not always top down. Due to the detection of modelling errors or lack of understanding in the design decisions, the abstract model may need fixing. In the case of a change in the model, a new refinement chain may evolve. The detection of errors or omissions at a later stage in the refinement and fixing them in the abstract model is an integral part of the evolution of a valid refinement chain. For example, in the third refinement of the model of transactions, given in Chapter 3, we introduce explicit messaging among the sites that corresponds to a reliable broadcast. In this refinement, the update transactions may be blocked due to the race conditions.

In order to deal with blocked transactions, we introduce the \textit{timeout} strategy that aborts an update transaction at the coordinating site. We already outlined in the first refinement that our replica control mechanism preserves the consistency of a database in the event of an abort of an update transaction. The effect of an timeout is similar to globally aborting a transaction by a coordinating site. While adding this event to the third refinement, we realized that this event must also exist in the abstract model, as the activation of this event sets the global status of a transaction to \textit{ABORT} in the abstract model. Also, similar to the effects of \textit{AbortWriteTran} event, activation of the
TimeOut event removes a transaction from a list of active transactions at coordinator and adds the transaction object to a list of free objects at the coordinator. However, the differences between these events are visible in the third refinement where activation of AbortWriteTran requires that the coordinator has delivered at least one vote-abort message from the coordinator. Therefore, we have to modify the refinement chain and we introduce TimeOut at each refinement level. The specifications of TimeOut events for each refinement level of the model of transactions are given in Appendix-B.

7.3.2 Guidelines for Discharging Proof Obligations using B Tools

As outlined above, the tools are critical in managing the proof efforts and the development of the refinement chain. Guidelines are presented below outlining effective strategies to discharge proof obligations.

G1. While constructing a gluing invariant after the inspection of the proof obligations, always try to construct an invariant which is close to the form of the proof obligation.

G2. Where possible avoid using complex structures in the invariants such as quantifications, a relational override operator or an inverse function. For example, consider the following invariant.

\[ \forall (p, m). (p \in process \land m \in message \land (p \mapsto m) \in deliver \Rightarrow m \in dom(sender)) \]

Instead of writing this invariant using the quantification, it can be simply be expressed as \( \text{ran(deliver)} \subseteq \text{dom(sender)} \). The proof obligations generated due to the use of quantifications in the invariant are more complex than using simple set theoretic constructs.

G3. There exist three predicate provers in the tools pr, po and p1. p1 is considered to be the most powerful prover. However, in certain cases, p1 fails to prove a goal while pr or po are able to prove the same goal. Also, it is much quicker to replay the proofs discharged using either pr or po than those discharged using the prover p1.

G4. It is sometimes useful to prove a lemma first (using ah button), which when proved becomes a new hypothesis. A lemma should be constructed in such a way that it is close to the goal of proof obligation. For example, consider the following proof obligation generated during development of a model of total order broadcast.

\[
 m1 \in \text{dom(sender)} \land m2 \in \text{dom(sender)} \land \\
 (m1 \mapsto m2) \in \text{totalorder} \land (pp \mapsto m2) \in \text{tdeliver} \\
 \Rightarrow (m1 \mapsto m2) \in \text{delorder}(pp)
\]
In order to discharge this proof obligation, the following lemma should be proved using *add hypothesis*.

\[ pp \mapsto m1 \in tdeliver \]

**G5.** While interacting with a proof obligation generated due to the addition of an invariant containing universal quantification, propose a valid instantiation for that quantification.

**G6.** While discharging a proof obligation containing existential quantification, always propose a valid witness to this quantification.

**G7.** In certain cases the tool allows you to discharge the proof obligations case by case (using *ov* button). Performing the proof steps case by case allows you to interact with simpler proof obligations.

**G8.** While inspecting the available hypotheses, if any one is found in contradiction, try to prove the negation of that hypothesis. For example, consider the following proof obligation generated by the prover due to addition of primary invariants in the development of a model of total order broadcast.

\[
mm \notin \text{dom}(sender) \land (pp \mapsto m2) \in tdeliver \land (mm \mapsto m2) \in totalorder \land m1 = mm \land m2 \neq mm \\
\Rightarrow (pp \mapsto mm) \in tdeliver
\]

It can be noticed that there is a contradiction in the hypotheses of this proof obligation, i.e., the hypothesis \( mm \notin \text{dom}(sender) \) and \( (mm \mapsto m2) \in totalorder \) can not be true simultaneously, since a total order is built only on the sent messages. Also, the goal \( (pp \mapsto mm) \in tdeliver \) cannot be proved under the hypothesis \( mm \notin \text{dom}(sender) \). Therefore, we add an hypothesis \( mm \in \text{dom}(sender) \) and discharge it after adding an invariant \( \text{ran}(tdeliver) \subseteq \text{dom}(sender) \).

**G9.** In the case of prover failures, inspect the available hypotheses and remove the useless hypotheses from the list of available hypotheses. If the model of the system is fairly large, the most likely cause of the failure of the prover is the presence of useless hypotheses in the selection.

**G10.** In most cases it is useful to simplify the goals and available hypotheses by interaction, before attempting to prove a goal (using *ov,rm,rr*\(^2\)). The provers are good at proving the simpler goals. For example, consider following goal in a proof obligation:

\[
m1 \mapsto m2 \in totalorder \cup \text{ran}(tdeliver) \times \{mm\}
\]

\(^2\text{For explanation of these clause, see [6].}
This goal on starting \textit{remove membership} (rm) can be simplified to following:

\[(m_1 \mapsto m_2 \in \text{totalorder}) \lor (m_1 \mapsto m_2 \in \text{ran}(tdeliver) \times \{mm\})\]

This goal can be reduced to the following by using the \textit{remove disjunction} (rd) clause of the tool.

\[
m_1 \mapsto m_2 \in \text{totalorder} \\
m_1 \mapsto m_2 \in \text{ran}(tdeliver) \times \{mm\}
\]

However, it can be noticed that each time a goal is modified, the available hypotheses displayed by the tool also changes. The simpler goals are easily proved by the automatic prover of the tool.

### 7.4 Conclusions

In this chapter, we addressed the issue of liveness in the B models of distributed transactions. Safety and liveness are two important issues in the design and development of distributed systems \cite{73}. Safety properties express that something \textit{bad} will not happen during system execution. A liveness property expresses that something \textit{(good)} will eventually happen during the execution. With regards to safety properties, the existing tools generate proof obligations for consistency and refinement checking. Discharging the proof obligations generated due to consistency checking mean that the activation of the events does not violate the invariants. Discharging the proof obligations due to the refinement checking implies that the gluing invariants that relate abstract and concrete variables are preserved by the activation of the events in the refinement. With regard to the liveness, it is useful to state that the model of the system under development is \textit{non-divergent} and \textit{enabledness} preserving. By enabledness preservation, we mean that whenever some event (or group of events) is enabled at the abstract level then the corresponding event (or group of events) is eventually enabled at the concrete level. Similarly, non-divergence requires us to prove that the new events in the refinement do not take control forever. The issues relating to the liveness properties are currently being addressed in the new generation of Event-B tools being developed \cite{44, 92}.

We outlined how enabledness preservation and non-divergence are related to the liveness of the B models of distributed transactions. To ensure that a concrete model also makes progress and does not deadlock more often than its abstraction, it is necessary to prove that if an abstract model makes \textit{progress} due to the activation of events then the concrete model also makes progress due to the activation of the events in the refinement. We ensure this property by enabledness preservation. In order to prove that a concrete machine also makes a progress, we need to prove that the guards of one or more events in the refinement are enabled under the hypothesis that the guard of one or more events
in the abstraction are also enabled. We specified the necessary conditions for enabled-
ness preservation for the model of transactions that need to be preserved. In order to
show that the new events in the refinement do not take control forever we outlined a
construction of an invariant property on a variant. For each new event in the refinement
we should be able to demonstrate that the execution of each new event decreases the
variant and variant never goes below zero. This allows us to prove that a new event can
not take control forever, since a variant can not be decreased indefinitely.

In the later part of the chapter, we presented the guidelines for the development of a
distributed system using Event-B. Since the use of a tool is critical in managing the proof
obligations and the management of the refinement chain, guidelines for discharging the
proof obligations using B tool are also discussed.
Chapter 8

Conclusions

In this chapter, we outline what we achieved in terms of applying Event-B for the incremental construction of the formal models of distributed transactions and broadcast protocols for replicated databases. In Section 8.1, we first summarize the research carried out within different chapters and explain how we meet the research objectives outlined in Chapter 1. Subsequently, in Section 8.2, we compare our approach of development of formal models of distributed systems and reasoning about them, with other approaches. Lastly, in Section 8.3, we explore areas of future research where the knowledge gained in the thesis can be used to further enhance the understanding of replicated databases.

8.1 Summary

Distributed algorithms are hard to understand and verify. Several approaches exist for the verification of these algorithms which include model checking, proving theorems by hand or proving invariant properties on the trace behavior. However, the application of proof based formal methods for the automated systematic design and development of such distributed systems and verification of the critical properties is rare. Often distributed algorithms are deceptive and an algorithm that looks simple may have complex execution paths and allow unanticipated behavior. There exists a vast variety of problems related to distributed systems. Also there exist several solutions to each of the problems. Rigorous reasoning about the algorithms is required to ensure that an algorithm achieves what it is supposed to do. Event-B is a formal technique that consists of describing rigorously the problem in the abstract model, introducing solutions or design details in refinement steps to obtain more concrete specifications, and verifying that the proposed solutions are correct. The B tools provide significant automated proof support for generating the proof obligations and discharging them. This technique requires the discharge of proof obligations for consistency checking and refinement checking. These proofs help us to understand the complexity of the problem and the correctness of the
solutions. They also help us to discover new system invariants providing a clear insight into the system and enhance our understanding of why a design decision should work.

The aim of the thesis is to demonstrate the application of Event-B to the incremental construction of formal models of distributed transactions and broadcast protocols for replicated databases, and to reason about them. A brief note of the work presented in the thesis is outlined below.

**Rigorous Design of Distributed Transactions**

In Chapter 3, we have presented a formal approach to modelling and analyzing a distributed transaction mechanism for a replicated database using Event-B. The abstract model of transactions is based on the notion of a one-copy database. In the refinement of the abstract model, we introduced the notion of a replicated database. This formal approach carries the development of the system from an initial abstract specification of transactional updates on a one-copy database to a detailed design containing replicated databases in the refinement. The replica control mechanism considered in the refinement allows both update and read-only transactions to be submitted at any site. In the case of a commit of the transaction, each site updates its replica separately. An update transaction commits atomically, updating all copies at commit or none when it aborts. A read-only transaction may perform read operations on any one replica. The various events given in the Event-B refinement are triggered within the framework of commit protocols that ensure global atomicity of update transactions despite transaction failures. The system allows the sites to abort a transaction independently and keeps the replicated database in a consistent state. The transaction mechanism on the replicated database is designed to provide the illusion of atomic update of a one-copy database. By verifying the refinement, we verify that the design of the replicated database conforms to the one-copy equivalence consistency criterion. In the further refinement, we also outlined how these transactions can be processed in the presence of a reliable broadcast.

**Implementing Causal Ordering on Messages by Vector Clocks**

Capturing the causal precedence relation among the different events occurring in a distributed system is key to the success of many distributed computations. Vector clocks have been proposed as a mechanism to capture the causality among the messages and provides a framework to deliver the messages to the sites in their respective causal order. In Chapter 4, we have presented Event-B specifications for the global causal ordering of the messages in a broadcast system. In the specifications we have outlined how an abstract causal order is constructed on the messages. In the refinement steps, we outlined how an abstract causal order can correctly be implemented by a system of vector
clocks. This is done by replacing the abstract variable \textit{corder} in the abstract specifications by vector clock rules in the concrete refinement. Due to refinement checking, several proof obligations are generated by the B tool. These proof obligations help us discover the invariants which define the relationship between abstract causal order and vector clock rules. We have also outlined how the gluing invariants are constructed after the inspection of the proof obligations. Our model is based on the Birman, Schiper and Stephenson’s protocol [21] for implementing global causal ordering using vector clocks. In the refinement, we found that instead of updating the whole vector of a recipient process as outlined in the original protocol, updating only one value in the vector clock of a recipient process corresponding to the sender process is sufficient to realize causally ordered delivery of the message.

**Implementing Total Ordering on Messages by Sequence Numbers**

In Chapter 5, we outlined an incremental development of a system of total order broadcast. A total order broadcast delivers messages to all sites in the same order. The advantage of processing update transactions over a total order broadcast is that a total order broadcast delivers updates to all participating sites in the same order. Unnecessary aborts of update transactions due to blocking can be avoided using a total order broadcast which delivers and executes the conflicting operations at all sites in the same order.

We have outlined the key issues with respect to the total order broadcast algorithms, such as, how to build a total order on messages and what information is required to deliver the messages in a total order. The \textit{Broadcast Broadcast} variant of a fixed sequencer protocol is used for the development of a system of a total order broadcast. In the abstract model, we outline how an abstract total order is constructed at the first ever delivery of a message to any process in the system. All other processes deliver that message in the abstract total order. We also identify the invariant properties for total order and add them to the model as \textit{primary invariants}. We further discover a set of \textit{secondary invariants} that are required to discharge the proof obligations generated by addition of primary invariants to the model. Later in the refinement, we introduce the notion of sequencer and control messages and show how sequence numbers are generated by the sequencer. The gluing invariants discovered in the refinement checking relate the abstract total order with the sequence numbers. Both gluing invariants and proof obligations provide a clear insight into the system and the reasons why the delivery based on the sequence numbers preserves a total order on the messages.
Implementing a Total Causal Order on Messages by Vector Clocks

In chapter 6, we have given the incremental development of a system of total causal order broadcast. A total causal order broadcast not only preserves the causal precedence relationship among the messages but also delivers them in a total order. The main advantage of processing update transactions over a total causal order broadcast is that the database always remains in a consistent state due to the guarantees of providing a total order on update messages. Another advantage of this broadcast is that the causality of the update messages is also preserved.

In this chapter, the Event-B specifications of an execution model of a total causal order broadcast system are presented. In the abstract model of this broadcast we outlined how the abstract causal order and a total order on the computation messages are constructed. In this model, a message is delivered to each process twice, first in a causal order followed by another delivery in a total order. The second delivery of a computation message in a total order corresponds to the delivery in a total causal order. In order to verify that this model satisfies the required ordering properties we add the invariants corresponding to the causal order and total order to our model as a primary invariants and discharge the proof obligations generated by the B tool due to the addition of these invariants. This system is based on a vector clock model and there also exists a specific process sequencer which sequences the computation messages to implement total ordering on the messages. In the refinement we outline how both causal and total order can be implemented using vector clocks.

Liveness Properties and Modelling Guidelines

In Chapter 7, the issue of liveness in a distributed system is addressed. After exploring the enabledness preservation and non-divergence properties for Event-B development, we outline how these liveness properties relate to the model of transactions. We also outlined how the strong variants of the broadcast protocol given in Chapter 4, 5 and 6 can be used to define abstract ordering on the transactions, thus ensuring the delivery of conflicting operations of update transactions to all participating sites in the same serial order.

The existing tools currently do not generate proof obligations for ensuring enabledness preservation and non-divergence. We outlined construction of the proof obligations to show that the model of transactions is both enabledness preserving and non-divergent. Lastly, general methodological modelling guidelines for the incremental development of a distributed system are also presented. We have also presented guidelines for discharging the proof obligations generated by the B tool.
8.2 Comparison with other Related Work

Though there exists vast literature on distributed algorithms and protocols covering several aspects of transactions, group communication and distributed databases, the application of proof based formal methods for precise definition of problems and verification of the correctness of their solutions is rare. In this section, we compare our approach with the other significant work on the application of formal methods.

The input/output (I/O) automaton model [83, 84], developed by Lynch and Tuttle, is a labelled transition system model for components in asynchronous concurrent systems. In [41], I/O Automata are used for formal modelling and verification of a sequentially consistent shared object system in a distributed network. In order to keep the replicated data in a consistent state, a combination of total order multicast and point to point communication is used. In [40], I/O automata are used to express lazy database replication. The authors present an algorithm for lazy database replication and prove the correctness properties relating performance and fault-tolerance. In [42, 104] the specifications for group communication primitives are presented using I/O automata under different conditions such as partitioning among the group and dynamic view oriented group communication. The proof method used in this method for reasoning about the system involves invariant assertions. An invariant assertion is defined as a property of the state of a system that is true in all execution. A series of invariants relating state variables and reachable states is proved using the method of induction. The work done so far using I/O Automata has been carried out by hand [47, 42].

In [12, 11], Z is used to specify formally a transaction decomposition in a database system. The authors present a mechanism to decompose the transactions to increase the concurrency without sacrificing the consistency of a database. They introduce the notion of semantic histories to formulate and prove necessary properties, and reason about interleaving with other transactions. The authors have used the Z specification language for expressing various models and all analysis is done by hand. The authors also highlighted the need for powerful tool support to discharge proof obligations.

In [130], a formal method is proposed to prove the total and causal order of multicast protocols. The formal results provided in the paper can be used to prove whether an existing system has the required property or not. Their solutions are based on the assumption that a total order is built using the service provided by a causal order protocol. The proof of correctness of the results is done by hand.

Instead of model checking, proving theorems by hand or proving correctness of the trace behavior, our approach is based on defining properties in the abstract model and proving that our model of the algorithm is a correct refinement of abstract model. The formal approach considered in our work is based on Event-B which facilitates incremental development of systems. We have used the Click’n’Prove B tool for the proof management.
This tool generates the proof obligations due to refinement and consistency checking and helps discharge proof obligations by the use of automatic and interactive provers. The majority of the proofs are discharged by the automatic prover. However, some complex proofs require use of the interactive prover. During the process of discharging proof obligations, new invariants are also discovered. We have outlined the process of discovering new invariants by the inspection of the proof obligations. The proofs and the invariants help us to understand the complexity of the problem, providing a clear insight into the system.

The overall proof statistics for various developments are given below.

<table>
<thead>
<tr>
<th>Model</th>
<th>POs</th>
<th>Automatic POs</th>
<th>Interactive POs</th>
<th>% Automatic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model of Transactions</td>
<td>307</td>
<td>191</td>
<td>116</td>
<td>63 %</td>
</tr>
<tr>
<td>Causal Order Broadcast</td>
<td>112</td>
<td>71</td>
<td>41</td>
<td>64 %</td>
</tr>
<tr>
<td>Total Order Broadcast</td>
<td>106</td>
<td>79</td>
<td>27</td>
<td>75 %</td>
</tr>
<tr>
<td>Total Causal Order Broadcast</td>
<td>166</td>
<td>102</td>
<td>64</td>
<td>62 %</td>
</tr>
<tr>
<td>Overall</td>
<td>691</td>
<td>443</td>
<td>248</td>
<td>64 %</td>
</tr>
</tbody>
</table>

Table 8.1: Proof Statistics- Overall

Our experience with the case studies presented in this thesis strengthens our belief that abstraction and refinement are valuable techniques for the modelling and verification of complex distributed systems.

8.3 Future Work

In this section, we outline the possible extensions to our work presented in this thesis.

1. Replica control mechanisms can broadly be classified as *eager* or *lazy* data replication. In eager database replication, all replicas are updated within the transaction boundary. The coordination of all sites takes place before a transaction commits and conflicts among the transactions are also detected before they commit. Eager database replication comes with a significant cost in the case of a site or communication link failure. An update transaction cannot commit until all sites are reachable. An alternative solution is lazy replication where the updates are propagated after an update transaction commits. Lazy replication allows the different copies of the replica to exhibit different values, therefore sacrificing the data consistency for a period of time. The other copies of the replicas at other sites are progressively updated after committing a transaction. Lazy replication can be used in situations where the availability of the data is considered critical. We plan to extend our model such that an update transaction commits by updating the replica at its coordinating site and the new values are communicated to the other
sites by a total order broadcast. The sites update the replica when they receive the update message. This approach is efficient but allows data inconsistencies to take place. We require rigorous reasoning about this approach to show that the database is in a consistent state when the reconciliation take place.

2. Our model of distributed transactions is based on the notion of full replication and the transactions are executed within the framework of the read one and write all (ROWA) replica control protocol. In this technique, a transaction can read a local copy but, to update an object, it must update all copies. This technique is suitable when the transaction workload is predominantly read only. The performance of this mechanism tends to degrade in a system where updates are as frequent as reads. In a separate study in [60], it is also shown that the interleaving of more conflicting transactions leads to more abortions due to the timeouts. One extension to the present work is to use a voting technique [97] instead of ROWA, where a transaction must write to a majority of the replicas instead of all. The updates are then propagated to the rest of the replicas. Similarly, a read-only transaction reads at least enough copies to make sure that one of the copies is up-to-date. Each copy of replica may have a version number representing the number of updates it has had. A rigorous reasoning is required to understand, how a voting technique preserves the data consistency.

3. We also plan to extend the existing model to model explicitly the failure and recovery of the sites. This requires an extension of the ROWA replica control mechanism to ROWA-A. ROWA-A (Read One Write All-Available) allows an update transaction to commit after updating the replicas at all available sites. Since we use ordering on messages, upon recovery, a failed site executes all updates in the order they were received. The commit protocol, based on presumed commit, is proposed for the commitment of an update transaction. This model of replicated databases brings higher performance for the updates because updates will not be blocked at a failed site. The explicit modelling of the coordinator and participating site failure is required to understand precisely how they restore the data consistency after the recovery.

4. The work presented in this thesis assumes that the data is fully replicated. Full replication, however, is not the most efficient strategy for all applications. Many applications require that the data is replicated at only a few sites. In practice, many applications may require both data fragmentation and partial replication for the purpose of efficiency. Also, full replication suffers from storage problems. One of the extensions of the existing model of replicated data is to allow partial replication. The communication among the sites must allow the combination of a total order broadcast and a point-to-point communication. The use of point-to-point communication reduces the communication overhead caused by the broadcast.
The work presented in this thesis focusses on processing transactional updates in replicated databases using ordered broadcasts. We believe that the methodology and the models presented in the thesis may be extended to enhance our understanding of other related techniques used in replicated databases such as lazy data replication, voting techniques, failure and recovery of a site, partial replication and fragmentation.
Appendix A

Distributed Transactions

A.1 Abstract Model

MACHINE Replica

DEFINITIONS PartialDB == ( OBJECT → VALUE );
UPDATE == (PartialDB → PartialDB );
ValidUpdate (update,readset) == ( dom(update)= readset → VALUE 
∧ ran(update) ⊆ readset → VALUE )

SETS TRANSACTION; OBJECT; VALUE;
TRANSSTATUS={COMMIT,ABORT,PENDING}

VARIABLES trans, transstatus, database, transeffect, transobject

INVARIANT 
trans ∈ P(TRANSACTION)
∧ transstatus ∈ trans → TRANSSTATUS
∧ database ∈ OBJECT → VALUE
∧ transeffect ∈ trans → UPDATE
∧ transobject ∈ trans → P₁ (OBJECT)
∧ ∀t.(t ∈ trans ⇒ ValidUpdate (transeffect(t), transobject(t)) )

INITIALISATION trans :=∅ || transstatus :=∅
|| transeffect := {} || transobject := { }
|| database := OBJECT → VALUE

EVENTS

StartTran( tt ∈ TRANSACTION ) ≡

ANY updates , objects
WHERE tt ∈ trans
∧ updates ∈ UPDATE
∧ objects ∈ P₁ (OBJECT)
∧ ValidUpdate (updates,objects)
THEN
trans := trans \ {tt}
|| transstatus(tt) := PENDING
|| transobject(tt) := objects
|| transeffect(tt) := updates
END ;
Appendix A Distributed Transactions

CommitWriteTran( tt ∈ TRANSACTION) ≡
ANY pdb
WHERE tt ∈ trans
∧\ transstatus(tt) = PENDING
∧\ ran(transeffect(tt)) ≠ \{\}
∧\ pdb = transobject(tt) ▷ database
THEN \transstatus(tt) := COMMIT
|| database := database ▷ transeffect(tt)(pdb)
END;

AbortWriteTran( tt ∈ TRANSACTION) ≡
WHEN tt ∈ trans
∧\ transstatus(tt) = PENDING
∧\ ran(transeffect(tt)) ≠ \{\}
THEN \transstatus(tt) := ABORT
END;

val ← ReadTran (tt∈ TRANSACTION ) ≡
WHEN tt ∈ trans
∧\ transstatus(tt) = PENDING
∧\ ran(transeffect(tt)) = \{\}
THEN val := transobject(tt) ▷ database
|| \transstatus(tt) := COMMIT
END;

A.2 First Refinement

REFINEMENT Replica2
REFINES Replica1

SETS SITE;
SITETRANSSTATUS = \{commit,abort,precommit,pending\}

VARIABLES trans, transstatus, activetrans, coordinator, sitetransstatus, transeffect, transobject, freeobject, replica

INVARIANT activetrans ∈ SITE ←→ trans
∧\ coordinator ∈ trans → SITE
∧\ sitetransstatus ∈ trans → (SITE → SITETRANSSTATUS)
∧\ replica ∈ SITE → (OBJECT → VALUE)
∧\ freeobject ∈ SITE ←→ OBJECT
∧\ ran(activetrans) ⊆ trans
∧\ \forall ( t , o ) . ( t ∈ trans ∧ o ∈ OBJECT ∧ s ∈ SITE ∧ ( s → o ) ∈ freeobject
⇒ database(o) = replica(s)(o) )
∧\ \forall ( o , s ) . ( o ∈ OBJECT ∧ s ∈ SITE ∧ ( s → o ) ∈ freeobject
⇒ database(o) = replica(s)(o) )
∧\ \forall ( t , o ) . ( t ∈ trans ∧ o ∈ OBJECT
∧\ (coordinator(t) → t) ∈ activetrans ∧ o ∈ transobject(t)
⇒ database(o) = replica(coordinator(t))(o) )
∧\ \forall ( s , t1 , t2) . ( s ∈ SITE ∧ t1 ∈ trans ∧ t2 ∈ trans
∧\ (s → t1) ∈ activetrans ∧ (s → t2) ∈ activetrans
∧\ transobject(t1) ∧ transobject(t2) ≠ \{\}
⇒ t1 = t2 )
\[ \forall (t, s, o). (t \in \text{trans} \land s \in \text{SITE} \land o \in \text{OBJECT} \\
\land \text{transstatus}(t) = \text{COMMIT} \land (s \mapsto t) \in \text{activetrans} \\
\land o \in \text{dom}(\text{transeffect}(t)(\text{transobject}(t) \cup \text{replica}(s))) \\
\Rightarrow \text{database}(o) = \text{transeffect}(t)(\text{transobject}(t) \cup \text{replica}(s))(o) ) \]

\[ \forall (t, s, o). (t \in \text{trans} \land s \in \text{SITE} \land o \in \text{OBJECT} \\
\land \text{transstatus}(t) = \text{COMMIT} \land o \in \text{transobject}(t) \\
\land (s \mapsto t) \in \text{activetrans} \\
\land o \notin \text{dom}(\text{transeffect}(t)(\text{transobject}(t) \cup \text{replica}(s))) \\
\Rightarrow \text{database}(o) = \text{replica}(s)(o) \]

\[ \forall (t). (t \in \text{trans} \land \text{transstatus}(t) = \text{ABORT} \\
\Rightarrow \text{sitetransstatus}(t)(\text{coordinator}(t)) = \text{abort} ) \]

\[ \forall (t). (t \in \text{trans} \land \text{transstatus}(t) = \text{COMMIT} \\
\Rightarrow \text{sitetransstatus}(t)(\text{coordinator}(t)) = \text{commit} ) \]

\[ \forall (t, s, o). (t \in \text{trans} \land s \in \text{SITE} \land o \in \text{OBJECT} \\
\land \text{transstatus}(t) \neq \text{COMMIT} \land (s \mapsto t) \in \text{activetrans} \\
\land o \in \text{transobject}(t) \\
\Rightarrow \text{database}(o) = \text{replica}(s)(o) \]

\[ \forall (t). (t \in \text{trans} \land \text{transstatus}(t) \neq \text{PENDING} \\
\land \text{ran}(\text{transeffect}(t)) \neq \emptyset \]
\[ \Rightarrow (\text{coordinator}(t) \mapsto t) \in \text{activetrans} \]

**ASSERTIONS**

\[ \forall (t_1, t_2). (s \in \text{SITE} \land t_1 \in \text{trans} \land t_2 \in \text{trans} \\
\land (\text{coordinator}(t_1) \mapsto t_1) \in \text{activetrans} \\
\land (\text{coordinator}(t_1) \mapsto t_2) \in \text{activetrans} \\
\land \text{transobject}(t_1) \cap \text{transobject}(t_2) \neq \emptyset \\
\Rightarrow t_1 = t_2 ) \]

\[ \forall (t, s, o). (t \in \text{trans} \land s \in \text{SITE} \land o \in \text{OBJECT} \\
\land \text{transstatus}(t) = \text{ABORT} \land (s \mapsto t) \in \text{activetrans} \\
\land o \in \text{transobject}(t) \\
\Rightarrow \text{database}(o) = \text{replica}(s)(o) \]

\[ \forall (t, s, o). (t \in \text{trans} \land s \in \text{SITE} \land o \in \text{OBJECT} \\
\land \text{transstatus}(t) = \text{PENDING} \land (s \mapsto t) \in \text{activetrans} \\
\land o \in \text{transobject}(t) \\
\Rightarrow \text{database}(o) = \text{replica}(s)(o) \]

**INITIALISATION**

\begin{align*}
\text{trans} &:= \emptyset \\
\text{transstatus} &:= \emptyset \\
\text{activetrans} &:= \emptyset \\
\text{coordinator} &:= \emptyset \\
\text{sitetransstatus} &:= \emptyset \\
\text{transeffect} &:= \{\} \\
\text{transobject} &:= \{\} \\
\text{freeobject} &:= \text{SITE} \times \text{OBJECT} \\
\text{ANY data WHERE} \text{ data} &\in \text{OBJECT} \rightarrow \text{VALUE} \\
\text{THEN} \text{replica} &:= \text{SITE} \times \{\text{data}\} \text{ END}
\end{align*}
Appendix A Distributed Transactions

\[
\text{StartTran}(tt) \equiv \\
\text{ANY} \quad ss, \text{updates, objects} \\
\text{WHERE} \quad ss \in \text{SITE} \\
\quad tt \notin \text{trans} \\
\quad \text{updates} \in \text{UPDATE} \\
\quad \text{objects} \in P_1(\text{OBJECT}) \\
\quad \text{ValidUpdate}(\text{updates,objects}) \\
\text{THEN} \\
\quad \text{trans} \leftarrow \text{trans} \cup \{tt\} \\
\quad \text{transstatus}(tt) \leftarrow \text{PENDING} \\
\quad \text{transobject}(tt) \leftarrow \text{objects} \\
\quad \text{transeffect}(tt) \leftarrow \text{updates} \\
\quad \text{coordinator}(tt) \leftarrow ss \\
\quad \text{sitetransstatus}(tt) \leftarrow \{\text{coordinator}(tt) \rightarrow \text{pending}\} \\
\text{END:} \\
\]

\[
\text{IssueWriteTran}(tt) \equiv \\
\text{WHEN} \quad tt \in \text{trans} \\
\quad (\text{coordinator}(tt) \leftrightarrow tt) \notin \text{activetrans} \\
\quad \text{sitetransstatus}(tt)(\text{coordinator}(tt)) = \text{pending} \\
\quad \text{ran}(\text{transeffect}(tt)) \neq \{\emptyset\} \\
\quad \text{transobject}(tt) \notin \text{freeobject}(\{\text{coordinator}(tt)\}) \\
\quad \forall tz. (tz \in \text{trans} \land (\text{coordinator}(tt) \leftrightarrow tz) \in \text{activetrans} \\
\quad \Rightarrow \text{transobject}(tt) \cap \text{transobject}(tz) = \emptyset) \\
\text{THEN} \\
\quad \text{activetrans} \leftarrow \text{activetrans} \cup \{\text{coordinator}(tt) \rightarrow tt\} \\
\quad \text{sitetransstatus}(tt)(\text{coordinator}(tt)) \leftarrow \text{precommit} \\
\quad \text{freeobject} \leftarrow \text{freeobject} - \{\text{coordinator}(tt)\} \times \text{transobject}(tt) \\
\text{END:} \\
\]

\[
\text{CommitWriteTran}(tt) \equiv \\
\text{ANY} \quad pdb \\
\text{WHERE} \quad tt \in \text{trans} \\
\quad pdb = \text{transobject}(tt) \land \text{replica}(\text{coordinator}(tt)) \\
\quad \text{ran}(\text{transeffect}(tt)) \neq \{\emptyset\} \\
\quad (\text{coordinator}(tt) \leftrightarrow tt) \in \text{activetrans} \\
\quad \text{transstatus}(tt) = \text{PENDING} \\
\quad \forall s. (s \in \text{SITE} \Rightarrow \text{sitetransstatus}(tt)(s) = \text{precommit}) \\
\quad \forall (s,o). (s \in \text{SITE} \land o \in \text{OBJECT} \land o \in \text{transobject}(tt) \\
\quad \Rightarrow (s \rightarrow o) \notin \text{freeobject}) \\
\quad \forall s. (s \in \text{SITE} \Rightarrow (s \rightarrow tt) \in \text{activetrans}) \\
\text{THEN} \\
\quad \text{transstatus}(tt) \leftarrow \text{COMMIT} \\
\quad \text{activetrans} \leftarrow \text{activetrans} - \{\text{coordinator}(tt) \rightarrow tt\} \\
\quad \text{sitetransstatus}(tt)(\text{coordinator}(tt)) \leftarrow \text{commit} \\
\quad \text{freeobject} \leftarrow \text{freeobject} \cup \{\text{coordinator}(tt)\} \times \text{transobject}(tt) \\
\quad \text{replica}(\text{coordinator}(tt)) \leftarrow \text{replica}(\text{coordinator}(tt)) \cup \text{transeffect}(tt)(pdb) \\
\text{END:} \\
\]

\[
\text{AbortWriteTran}(tt) \equiv \\
\text{WHEN} \quad tt \in \text{trans} \\
\quad \text{ran}(\text{transeffect}(tt)) \neq \{\emptyset\} \\
\quad (\text{coordinator}(tt) \leftrightarrow tt) \in \text{activetrans} \\
\quad \text{transstatus}(tt) = \text{PENDING} \\
\quad \exists s. (s \in \text{SITE} \land \text{sitetransstatus}(tt)(s) = \text{abort}) \\
\text{THEN} \\
\quad \text{transstatus}(tt) \leftarrow \text{ABORT} \\
\quad \text{activetrans} \leftarrow \text{activetrans} - \{\text{coordinator}(tt) \rightarrow tt\} \\
\quad \text{sitetransstatus}(tt)(\text{coordinator}(tt)) \leftarrow \text{abort} \\
\quad \text{freeobject} \leftarrow \text{freeobject} \cup \{\text{coordinator}(tt)\} \times \text{transobject}(tt) \\
\text{END:} \\
\]
\text{val} \leftarrow \text{ReadTran}(tt, ss) \equiv
\begin{align*}
\text{WHEN} & & t t \in \text{trans} \\
& & \land \text{transstatus}(tt) = \text{PENDING} \\
& & \land \text{transobject}(tt) \subseteq \text{freeobject}[\{ss\}] \\
& & \land ss = \text{coordinator}(tt) \\
& & \land \text{ran}(\text{transeffect}(tt)) = \{\emptyset\} \\
\text{THEN} & & \text{val} := \text{transobject}(tt) \land \text{replica}(ss) \\
& & \land \text{sitetransstatus}(tt)(ss) := \text{commit} \\
& & \land \text{transstatus}(tt) := \text{COMMIT} \\
\text{END};
\end{align*}

\begin{align*}
\text{BeginSubTran}(tt, ss) \equiv
\begin{align*}
\text{WHEN} & & tt \in \text{trans} \\
& & \land \text{sitetransstatus}(tt)(\text{coordinator}(tt)) \in \{\text{precommit, abort}\} \\
& & \land (ss \rightarrow tt) \notin \text{activetrans} \\
& & \land ss \neq \text{coordinator}(tt) \\
& & \land \text{ran}(\text{transeffect}(tt)) \neq \{\emptyset\} \\
& & \land \text{transobject}(tt) \subseteq \text{freeobject}[\{ss\}] \\
& & \land ss \notin \text{dom}(\text{sitetransstatus}(tt)) \\
& & \land \forall tz. (tz \in \text{trans} \land (ss \rightarrow tz) \in \text{activetrans} \\
& & \Rightarrow \text{transobject}(tt) \cap \text{transobject}(tz) = \emptyset) \\
\text{THEN} & & \text{activetrans} := \text{activetrans} \cup \{ss \rightarrow tt\} \\
& & \land \text{sitetransstatus}(tt)(ss) := \text{pending} \\
& & \land \text{freeobject} := \text{freeobject} \setminus \{ss\} \times \text{transobject}(tt) \\
\text{END};
\end{align*}
\end{align*}

\begin{align*}
\text{SiteCommitTx}(tt, ss) \equiv
\begin{align*}
\text{WHEN} & & (ss \rightarrow tt) \in \text{activetrans} \\
& & \land \text{sitetransstatus}(tt)(ss) = \text{pending} \\
& & \land ss \neq \text{coordinator}(tt) \\
& & \land \text{ran}(\text{transeffect}(tt)) \neq \{\emptyset\} \\
\text{THEN} & & \text{sitetransstatus}(tt)(ss) := \text{precommit} \\
\text{END};
\end{align*}
\end{align*}

\begin{align*}
\text{SiteAbortTx}(tt, ss) \equiv
\begin{align*}
\text{WHEN} & & (ss \rightarrow tt) \in \text{activetrans} \\
& & \land \text{sitetransstatus}(tt)(ss) = \text{pending} \\
& & \land ss \neq \text{coordinator}(tt) \\
& & \land \text{ran}(\text{transeffect}(tt)) \neq \{\emptyset\} \\
\text{THEN} & & \text{sitetransstatus}(tt)(ss) := \text{abort} \\
& & \land \text{freeobject} := \text{freeobject} \cup \{ss\} \times \text{transobject}(tt) \\
& & \land \text{activetrans} := \text{activetrans} \setminus \{ss \rightarrow tt\} \\
\text{END};
\end{align*}
\end{align*}

\begin{align*}
\text{ExeAbortDecision}(ss, tt) \equiv
\begin{align*}
\text{WHEN} & & tt \subseteq \text{trans} \\
& & \land (ss \rightarrow tt) \in \text{activetrans} \\
& & \land ss \neq \text{coordinator}(tt) \\
& & \land \text{ran}(\text{transeffect}(tt)) \neq \{\emptyset\} \\
& & \land \text{sitetransstatus}(tt)(\text{coordinator}(tt)) = \text{abort} \\
& & \land \text{sitetransstatus}(tt)(ss) = \text{precommit} \\
\text{THEN} & & \text{sitetransstatus}(tt)(ss) := \text{abort} \\
& & \land \text{activetrans} := \text{activetrans} \setminus \{ss \rightarrow tt\} \\
& & \land \text{freeobject} := \text{freeobject} \cup \{ss\} \times \text{transobject}(tt) \\
\text{END};
\end{align*}
\end{align*}
Appendix A Distributed Transactions

\[ \text{ExeCommitDecision}(ss,tt) \equiv \]
\[ \begin{align*}
\text{ANY} & \quad \text{pdb} \\
\text{WHERE} & \quad t \in \text{trans} \\
\wedge & \quad (ss \rightarrow tt) \in \text{activetrans} \\
\wedge & \quad ss \neq \text{coordinator}(tt) \\
\wedge & \quad \text{ran}(\text{transeffect}(tt)) \neq \{\emptyset\} \\
\wedge & \quad \text{pdb} = \text{transobject}(tt) \land \text{replica}(ss) \\
\wedge & \quad \text{sitetransstatus}(tt)(\text{coordinator}(tt)) = \text{commit} \\
\wedge & \quad \text{sitetransstatus}(tt)(ss) = \text{precommit} \\
\text{THEN} & \quad \text{activetrans} := \text{activetrans} \setminus \{ss \rightarrow tt\} \\
\| & \quad \text{sitetransstatus}(tt)(ss) := \text{commit} \\
\| & \quad \text{freeobject} := \text{freeobject} \cup \{ss\} \times \text{transobject}(tt) \\
\| & \quad \text{replica}(ss) := \text{replica}(ss) \land \text{transeffect}(tt)(\text{pdb}) \\
\end{align*} \]
\[ \text{END}; \]

A.3 Second Refinement

REFINEMENT Replica3
REFINES Replica2

VARIABLES \text{trans}, \text{transstatus}, \text{activetrans}, \text{coordinator}, \text{sitetransstatus}, \text{transeffect}, \text{transobject}, \text{freeobject}, \text{replica}

IN Variant
\[ \forall (t,s,o). \left( t \in \text{trans} \land s \in \text{SITE} \land o \in \text{OBJECT} \land o \in \text{transobject}(t) \land \text{sitetransstatus}(t)(s) = \text{precommit} \Rightarrow s \rightarrow o \notin \text{freeobject} \right) \]
\[ \wedge \forall (t,s,o). \left( t \in \text{trans} \land s \in \text{SITE} \land \text{sitetransstatus}(t)(s) = \text{precommit} \Rightarrow s \rightarrow t \in \text{activetrans} \right) \]
\[ \wedge \forall (t,s,o). \left( t \in \text{trans} \land s \in \text{SITE} \land o \in \text{OBJECT} \land o \in \text{transobject}(t) \land s \rightarrow t \in \text{activetrans} \Rightarrow s \rightarrow o \notin \text{freeobject} \right) \]

INITIALIZATION
\[ \text{trans} := \emptyset \quad \| \quad \text{transstatus} := \emptyset \quad \| \quad \text{activetrans} := \emptyset \]
\[ \| \quad \text{coordinator} := \emptyset \quad \| \quad \text{sitetransstatus} := \emptyset \quad \| \quad \text{transeffect} := \{\} \]
\[ \| \quad \text{transobject} := \{\} \quad \| \quad \text{freeobject} := \text{SITE} \times \text{OBJECT} \]
\[ \| \quad \text{ANY} \text{ data WHERE} \text{ data} \in \text{OBJECT} \rightarrow \text{VALUE} \]
\[ \text{THEN} \quad \text{replica} := \text{SITE} \times \{\text{data}\} \text{ END} \]

\[ \text{StartTran}(tt) \equiv \]
\[ \begin{align*}
\text{ANY} & \quad ss, \text{updates}, \text{objects} \\
\text{WHERE} & \quad ss \in \text{SITE} \\
\wedge & \quad t \in \text{trans} \\
\wedge & \quad \text{updates} \in \text{UPDATE} \\
\wedge & \quad \text{objects} \in \text{P}_1(\text{OBJECT}) \\
\wedge & \quad \text{ValidUpdate}(\text{updates},\text{objects}) \\
\text{THEN} & \quad \text{trans} := \text{trans} \cup \{tt\} \\
\| & \quad \text{transstatus}(tt) := \text{PENDING} \\
\| & \quad \text{transobject}(tt) := \text{objects} \\
\| & \quad \text{transeffect}(tt) := \text{updates} \\
\| & \quad \text{coordinator}(tt) := ss \\
\| & \quad \text{sitetransstatus}(tt) := \{\text{coordinator}(tt) \mapsto \text{pending}\} \\
\text{END}; \]
Appendix A Distributed Transactions

IssueWriteTran$(tt) \equiv$

WHEN  $tt \in \mathit{trans}$
\hspace{1em}$\wedge (\mathit{coordinator}(tt) \rightarrow tt) \notin \mathit{activetrans}$
\hspace{1em}$\wedge \mathit{sitetransstatus}(tt)(\mathit{coordinator}(tt)) = \mathit{pending}$
\hspace{1em}$\wedge \mathit{ran}(\mathit{transeffect}(tt)) \neq \{\emptyset\}$
\hspace{1em}$\wedge \mathit{transobject}(tt) \subseteq \mathit{freeobject}[\{\mathit{coordinator}(tt)\}]$
\hspace{1em}$\forall tz.(tz \in \mathit{trans} \wedge (\mathit{coordinator}(tt) \rightarrow tz) \in \mathit{activetrans}$
\hspace{1em}$\Rightarrow \mathit{transobject}(tt) \cap \mathit{transobject}(tz) = \emptyset)$
THEN
\hspace{2em} activetrans := activetrans \cup \{\mathit{coordinator}(tt) \rightarrow tt\}$
\hspace{2em} $\mathit{sitetransstatus}(tt)(\mathit{coordinator}(tt)) := \mathit{precommit}$
\hspace{2em} $\mathit{freeobject} := \mathit{freeobject} - \{\mathit{coordinator}(tt)\}$
END;

CommitWriteTran$(tt) \equiv$

ANY $pdb$
WHERE  $tt \in \mathit{trans}$
\hspace{1em}$\wedge \mathit{pdb} = \mathit{transobject}(tt) \cup \mathit{replica}(\mathit{coordinator}(tt))$
\hspace{1em}$\wedge \mathit{ran}(\mathit{transeffect}(tt)) \neq \{\emptyset\}$
\hspace{1em}$\wedge (\mathit{coordinator}(tt) \rightarrow tt) \in \mathit{activetrans}$
\hspace{1em}$\wedge \mathit{transstatus}(tt) = \mathit{PENDING}$
\hspace{1em}$\forall s.(s \in \text{SITE} \Rightarrow \mathit{sitetransstatus}(tt)(s) = \mathit{precommit})$
THEN
\hspace{2em} $\mathit{transstatus}(tt) := \mathit{COMMIT}$
\hspace{2em} activetrans := activetrans -$\{\mathit{coordinator}(tt) \rightarrow tt\}$
\hspace{2em} $\mathit{sitetransstatus}(tt)(\mathit{coordinator}(tt)) := \mathit{commit}$
\hspace{2em} $\mathit{freeobject} := \mathit{freeobject} \cup \{\mathit{coordinator}(tt)\} \times \mathit{transobject}(tt)$
\hspace{2em} $\mathit{replica}(\mathit{coordinator}(tt)) := \mathit{replica}(\mathit{coordinator}(tt)) \cup \mathit{transeffect}(tt)(\mathit{pdb})$
END;

AbortWriteTran$(tt) \equiv$

WHEN  $tt \in \mathit{trans}$
\hspace{1em}$\wedge \mathit{ran}(\mathit{transeffect}(tt)) \neq \{\emptyset\}$
\hspace{1em}$\wedge (\mathit{coordinator}(tt) \rightarrow tt) \in \mathit{activetrans}$
\hspace{1em}$\wedge \mathit{transstatus}(tt) = \mathit{PENDING}$
\hspace{1em}$\exists s.(s \in \text{SITE} \land \mathit{sitetransstatus}(tt)(s) = \mathit{abort})$
THEN
\hspace{2em} $\mathit{transstatus}(tt) := \mathit{ABORT}$
\hspace{2em} activetrans := activetrans -$\{\mathit{coordinator}(tt) \rightarrow tt\}$
\hspace{2em} $\mathit{sitetransstatus}(tt)(\mathit{coordinator}(tt)) := \mathit{abort}$
\hspace{2em} $\mathit{freeobject} := \mathit{freeobject} \cup \{\mathit{coordinator}(tt)\} \times \mathit{transobject}(tt)$
END;

val $\leftarrow$ ReadTran$(tt,ss) \equiv$

WHEN  $tt \in \mathit{trans}$
\hspace{1em}$\wedge \mathit{transstatus}(tt)=\mathit{PENDING}$
\hspace{1em}$\wedge \mathit{transobject}(tt) \subseteq \mathit{freeobject}[\{ss\}]$
\hspace{1em}$\wedge \mathit{ss} = \mathit{coordinator}(tt)$
\hspace{1em}$\wedge \mathit{ran}(\mathit{transeffect}(tt)) = \{\emptyset\}$
THEN
\hspace{2em} val := $\mathit{transobject}(tt) \cup \mathit{replica}(ss)$
\hspace{2em} $\mathit{sitetransstatus}(tt)(ss) := \mathit{commit}$
\hspace{2em} $\mathit{transstatus}(tt) := \mathit{COMMIT}$
END;
Appendix A Distributed Transactions

BeginSubTran\((tt, ss)\) ≡

\[\text{WHEN} \quad \begin{align*}
& tt \in \text{trans} \\
& \land \quad \text{sitetransstatus}(tt)(\text{coordinator}(tt)) \in \{\text{precommit, abort}\} \\
& \land \quad (ss \rightarrow tt) \not\in \text{activetrans} \\
& \land \quad ss \not= \text{coordinator}(tt) \\
& \land \quad \text{ran}(\text{transeffect}(tt)) \not= \{\emptyset\} \\
& \land \quad \text{transobject}(tt) \not\subset \text{freeobject} \{ss\} \\
& \land \quad ss \notin \text{dom}(\text{sitetransstatus}(tt)) \\
& \land \quad \forall tz, (tz \in \text{trans} \land (ss \rightarrow tz) \in \text{activetrans}) \\
& \quad \Rightarrow \text{transobject}(tt) \cap \text{transobject}(tz) = \emptyset
\end{align*} \]

\[\text{THEN} \quad \begin{align*}
& \quad \text{activetrans} := \text{activetrans} \cup \{ss \rightarrow tt\} \\
& \quad \text{sitetransstatus}(tt)(ss) := \text{pending} \\
& \quad \text{freeobject} := \text{freeobject} \cup \{ss\} \times \text{transobject}(tt)
\end{align*} \]

\[\text{END};\]

SiteCommitTx\((tt, ss)\) ≡

\[\text{WHEN} \quad (ss \rightarrow tt) \in \text{activetrans} \land \text{sitetransstatus}(tt)(ss) = \text{pending} \land ss \neq \text{coordinator}(tt) \land \text{ran}(\text{transeffect}(tt)) \neq \{\emptyset\} \]

\[\text{THEN} \quad \text{sitetransstatus}(tt)(ss) := \text{precommit}\]

\[\text{END};\]

SiteAbortTx\((tt, ss)\) ≡

\[\text{WHEN} \quad (ss \rightarrow tt) \in \text{activetrans} \land \text{sitetransstatus}(tt)(ss) = \text{pending} \land ss \neq \text{coordinator}(tt) \land \text{ran}(\text{transeffect}(tt)) \neq \{\emptyset\} \]

\[\text{THEN} \quad \text{sitetransstatus}(tt)(ss) := \text{abort} \land \text{freeobject} := \text{freeobject} - \{ss\} \times \text{transobject}(tt) \land \text{activetrans} := \text{activetrans} - \{ss \rightarrow tt\}\]

\[\text{END};\]

ExeAbortDecision\((ss, tt)\) ≡

\[\text{WHEN} \quad tt \in \text{trans} \land (ss \rightarrow tt) \in \text{activetrans} \land ss \neq \text{coordinator}(tt) \land \text{ran}(\text{transeffect}(tt)) \neq \{\emptyset\} \land \text{sitetransstatus}(tt)(\text{coordinator}(tt)) = \text{abort} \land \text{sitetransstatus}(tt)(ss) = \text{precommit} \]

\[\text{THEN} \quad \text{sitetransstatus}(tt)(ss) := \text{abort} \land \text{activetrans} := \text{activetrans} - \{ss \rightarrow tt\} \land \text{freeobject} := \text{freeobject} - \{ss\} \times \text{transobject}(tt)\]

\[\text{END};\]

ExeCommitDecision\((ss, tt)\) ≡

\[\text{ANY} \quad pdb \quad \text{WHERE} \quad tt \in \text{trans} \land (ss \rightarrow tt) \in \text{activetrans} \land ss \neq \text{coordinator}(tt) \land \text{ran}(\text{transeffect}(tt)) \neq \{\emptyset\} \land pdb = \text{transobject}(tt) \land \text{replica}(ss) \land \text{sitetransstatus}(tt)(\text{coordinator}(tt)) = \text{commit} \land \text{sitetransstatus}(tt)(ss) = \text{precommit} \land \text{activetrans} := \text{activetrans} - \{ss \rightarrow tt\} \land \text{replica}(ss) := \text{replica}(ss) \land \text{transeffect}(tt)(pdb) \]

\[\text{END};\]
A.4 Third Refinement

REFINEMENT  

Replica4

REFINES  

Replica3

SETS  

MESSAGE

VARIABLES  

trans, transstatus, activetrans, coordinator, sitetransstatus, transeffect, transobject, freeobject, replica, sender, deliver, update, voteabort, votecommit, globalabort, globalcommit, tranupdate, tranvoteabort, tranvotecommit, tranglobalabort, tranglobalcommit, completed

INVARIANT  

\[\begin{align*}
\text{sender} &\in \text{MESSAGE} \rightarrow \text{SITE} \land \text{deliver} \in \text{SITE} \leftrightarrow \text{MESSAGE} \\
\land & \text{update} \subseteq \text{MESSAGE} \land \text{update} \subseteq \text{dom(sender)} \\
\land & \text{voteabort} \subseteq \text{MESSAGE} \land \text{voteabort} \subseteq \text{dom(sender)} \\
\land & \text{votecommit} \subseteq \text{MESSAGE} \land \text{votecommit} \subseteq \text{dom(sender)} \\
\land & \text{globalabort} \subseteq \text{MESSAGE} \land \text{globalabort} \subseteq \text{dom(sender)} \\
\land & \text{globalcommit} \subseteq \text{MESSAGE} \land \text{globalcommit} \subseteq \text{dom(sender)} \\
\land & \text{tranupdate} \in \text{update} \leftrightarrow \text{trans} \\
\land & \text{tranvoteabort} \in \text{voteabort} \leftrightarrow \text{trans} \\
\land & \text{tranvotecommit} \in \text{votecommit} \leftrightarrow \text{trans} \\
\land & \text{tranglobalabort} \in \text{globalabort} \leftrightarrow \text{trans} \\
\land & \text{tranglobalcommit} \in \text{globalcommit} \leftrightarrow \text{trans} \\
\land & \text{completed} \in \text{trans} \leftrightarrow \text{SITE} \\
\land & \forall (t, s, m) : (t \in \text{trans} \land s \in \text{SITE} \land m \in \text{update} \land (m \mapsto t) \in \text{tranupdate} \\
\land & (s \mapsto m) \in \text{deliver} \land s \not\in \text{dom(sitetransstatus(t))} \land s \not\in \text{coordinator(t)} \\
\Rightarrow & \text{sitetransstatus(t)(coordinator(t))} \in \{\text{precommit,abort}\}
\end{align*}\]

INITIALISATION  

\[\begin{align*}
\text{trans} &:= \emptyset \land \text{transstatus} := \emptyset \land \text{activetrans} := \emptyset \\
\land & \text{coordinator} := \emptyset \land \text{sitetransstatus} := \emptyset \land \text{transeffect} := \emptyset \\
\land & \text{transobject} := \{\} \land \text{freeobject} := \text{SITE} \times \text{OBJECT} \\
\land & \text{ANY} \text{ data} \text{ WHERE} \text{ data} \in \text{OBJECT} \rightarrow \text{VALUE} \\
\text{THEN} & \text{replica} := \text{SITE} \times \{\text{data}\} \text{ END} \\
\land & \text{update} := \emptyset \land \text{voteabort} := \emptyset \land \text{votecommit} := \emptyset \\
\land & \text{globalabort} := \emptyset \land \text{globalcommit} := \emptyset \land \text{tranupdate} := \emptyset \\
\land & \text{tranvoteabort} := \emptyset \land \text{tranvotecommit} := \emptyset \land \text{tranglobalabort} := \emptyset \\
\land & \text{tranglobalcommit} := \emptyset
\end{align*}\]

EVENTS

\[\begin{align*}
\text{StartTran}(tt) \equiv \\
\text{ANY} & \text{ ss, updates, objects} \\
\text{WHERE} & \text{ ss} \in \text{SITE} \\
\land & \text{ tt} \not\in \text{trans} \\
\land & \text{ updates} \in \text{UPDATE} \\
\land & \text{ objects} \in P_1(\text{OBJECT}) \\
\land & \text{ ValidUupdate (updates,objects)} \\
\text{THEN} & \text{trans} := \text{trans} \cup \{tt\} \\
\land & \text{transstatus(tt)} := \text{PENDING} \\
\land & \text{transobject(tt)} := \text{objects} \\
\land & \text{transeffect(tt)} := \text{updates} \\
\land & \text{coordinator(tt)} := \text{ss} \\
\land & \text{sitetransstatus(tt)} := \{\text{coordinator(tt)} \mapsto \text{pending}\}
\end{align*}\]

END:
Appendix A Distributed Transactions

SendUpdate(ss ∈ SITE, mm ∈ MESSAGE, tt ∈ TRANSACTION) ≡
    WHEN mm ∈ dom(sender) ∧ tt ∈ trans ∧ sitetransstatus(tt)(coordinator(tt)) = pending ∧ ss = coordinator(tt) ∧ tt /∈ ran(tranupdate) ∧ ran(transeffect(tt)) ≠ {[]} THEN
        sender := sender ∪ {mm → ss} ∪ {mm} ∪ tranupdate := tranupdate ∪ {mm → tt}
    END;

Deliver(ss ∈ SITE, mm ∈ MESSAGE) ≡
    WHEN mm ∈ dom(sender) ∧ (ss → mm) ∈ deliver THEN
        deliver := deliver ∪ {ss → mm}
    END;

IssueWriteTran(tt ∈ TRANSACTION) ≡
    ANY mm WHERE mm ∈ update ∧ tt ∈ trans ∧ (mm → tt) ∈ tranupdate ∧ (coordinator(tt) → mm) ∈ deliver ∧ (coordinator(tt) → tt) /∈ activetrans ∧ sitetransstatus(tt)(coordinator(tt)) = pending ∧ ran(transeffect(tt)) ≠ {[]} ∧ tranobject(tt) ⊆ freeobject[{coordinator(tt)}] ∧ ∀tz. (tz ∈ trans ∧ (coordinator(tt) → tz) ∈ activetrans ⇒ tranobject(tt) ∩ tranobject(tz) = []) ∧ sitetransstatus(tt)(coordinator(tt)) = precommit ∧ freeobject := freeobject - {coordinator(tt)} × tranobject(tt)
    THEN
        activetrans := activetrans ∪ {coordinator(tt) → tt} ∪ sitetransstatus(tt)(coordinator(tt)) := precommit ∪ freeobject := freeobject - {coordinator(tt)} × tranobject(tt)
    END;

AbortWriteTran(tt) ≡
    ANY m1,m2 WHERE m1 ∈ voteabort ∧ m1 → tt ∈ tranvoteabort ∧ coordinator(tt) → m1 ∈ deliver ∧ m2 ∈ MESSAGE ∧ m2 /∈ dom(sender) ∧ tt ∈ trans ∧ ran(transeffect(tt)) ≠ {[]} ∧ (coordinator(tt) → tt) ∈ activetrans ∧ transstatus(tt) = PENDING ∧ ∃s. (s ∈ SITE ∧ sitetransstatus(tt)(s) = abort) ∧ transstatus(tt) := ABORT
    THEN
        activetrans := activetrans - {coordinator(tt) → tt} ∪ sitetransstatus(tt)(coordinator(tt)) := abort ∪ freeobject := freeobject ∪ {coordinator(tt)} × tranobject(tt) ∪ globalabort := globalabort ∪ {m2} ∪ tranglobalabort := tranglobalabort ∪ {m2 → tt} ∪ sender := sender ∪ {m2 → coordinator(tt)} ∪ completed := completed ∪ {tt → coordinator(tt)}
    END;
\[\text{CommitWriteTran}(tt) \equiv \]
\[
\begin{align*}
\text{ANY} & \quad pdb, mm \\
\text{WHERE} & \quad mm \in \text{MESSAGE} \land mm \notin \text{dom}(\text{sender}) \\
& \quad tt \in \text{trans} \\
& \quad pdb = \text{transobject}(tt) \land \text{replica}(\text{coordinator}(tt)) \\
& \quad \text{ran}(\text{transobject}(tt)) \neq \{\emptyset\} \\
& \quad (\text{coordinator}(tt) \rightarrow tt) \in \text{activetrans} \\
& \quad \text{transstatus}(tt) = \text{PENDING} \\
& \quad \forall s. (s \in \text{SITE} \Rightarrow \text{sitetransstatus}(tt)(s) = \text{precommit}) \\
& \quad \forall m. (m \in \text{votecommit} \land m \rightarrow tt \in \text{tranvotecommit} \\
& \quad \Rightarrow \text{coordinator}(tt) \rightarrow m \in \text{deliver}) \\
\text{THEN} & \quad \text{transstatus}(tt) := \text{COMMIT} \\
& \quad \text{activetrans} := \text{activetrans} - \{\text{coordinator}(tt) \rightarrow tt\} \\
& \quad \text{sitetransstatus}(tt)(\text{coordinator}(tt)) := \text{commit} \\
& \quad \text{freeobject} := \text{freeobject} \cup \{\text{coordinator}(tt) \times \text{transobject}(tt)\} \\
& \quad \text{repli}ca(\text{coordinator}(tt)) := \text{replica}(\text{coordinator}(tt)) \land \text{transobject}(tt)(pdb) \\
& \quad \text{globalcommit} := \text{globalcommit} \cup \{mm\} \\
& \quad \text{tranglobalcommit} := \text{tranglobalcommit} \cup \{mm \rightarrow tt\} \\
& \quad \text{sender} := \text{sender} \cup \{mm \rightarrow \text{coordinator}(tt)\} \\
& \quad \text{completed} := \text{completed} \cup \{tt \rightarrow \text{coordinator}(tt)\} \\
\end{align*}
\]
\[\text{END;}\]

\[\text{val} \leftarrow \text{ReadTran}(tt, ss) \equiv \]
\[
\begin{align*}
\text{WHEN} & \quad tt \in \text{trans} \\
& \quad \text{transstatus}(tt) = \text{PENDING} \\
& \quad \text{transobject}(tt) \in \text{freeobject}[\{ss\}] \\
& \quad ss = \text{coordinator}(tt) \\
& \quad \text{ran}(\text{transobject}(tt)) = \{\emptyset\} \\
\text{THEN} & \quad \text{val} := \text{transobject}(tt) \land \text{replica}(ss) \\
& \quad \text{sitetransstatus}(tt)(ss) := \text{commit} \\
& \quad \text{transstatus}(tt) := \text{COMMIT} \\
& \quad \text{completed} := \text{completed} \cup \{tt \rightarrow ss\} \\
\end{align*}
\]
\[\text{END;}\]

\[\text{BeginSubTran} (tt \in \text{TRANSACTION}, ss \in \text{SITE}) \equiv \]
\[
\begin{align*}
\text{ANY} & \quad mm \\
\text{WHERE} & \quad mm \in \text{update} \\
& \quad tt \in \text{trans} \\
& \quad (mm \rightarrow tt) \in \text{tranupdate} \\
& \quad (ss \rightarrow mm) \in \text{deliver} \\
& \quad (ss \rightarrow tt) \in \text{activetrans} \\
& \quad ss \notin \text{dom}(\text{sitetransstatus}(tt)) \\
& \quad ss \neq \text{coordinator}(tt) \\
& \quad \text{ran}(\text{transobject}(tt)) \neq \{\emptyset\} \\
& \quad \text{transobject}(tt) \in \text{freeobject}[\{ss\}] \\
& \quad \forall tt. (tt \in \text{trans} \land (ss \rightarrow tt) \in \text{activetrans} \\
& \quad \Rightarrow \text{transobject}(tt) \cap \text{transobject}(tt) = \emptyset) \\
\text{THEN} & \quad \text{activetrans} := \text{activetrans} \cup \{ss \rightarrow tt\} \\
& \quad \text{sitetransstatus}(tt)(ss) := \text{pending} \\
& \quad \text{freeobject} := \text{freeobject} - \{ss \times \text{transobject}(tt)\} \\
\end{align*}
\]
\[\text{END;}\]
Appendix A Distributed Transactions

SiteCommitTx$(tt \in TRANSACTION, ss \in SITE) =$

\begin{align*}
\text{ANY} & \quad mm \\
\text{WHERE} & \quad mm \in MESSAGE \\
& \quad \land mm \notin \text{dom}(sender) \\
& \quad \land tt \in \text{trans} \\
& \quad \land (ss \rightarrow tt) \in \text{activetrans} \\
& \quad \land \text{sitetransstatus}(tt)(ss) = \text{pending} \\
& \quad \land ss \neq \text{coordinator}(tt) \\
& \quad \land \text{ran}(\text{transelect}(tt)) \neq \emptyset \\
\text{THEN} & \quad \text{sitetransstatus}(tt)(ss) := \text{precommit} \\
& \quad \land \text{votecommit} := \text{votecommit} \cup \{ mm \} \\
& \quad \land \text{tranvotecommit} := \text{tranvotecommit} \cup \{ mm \rightarrow tt \} \\
& \quad \land \text{sender} := \text{sender} \cup \{ mm \rightarrow ss \} \\
\end{align*}

END;

SiteAbortTx$(tt \in TRANSACTION, ss \in SITE) =$

\begin{align*}
\text{ANY} & \quad mm \\
\text{WHERE} & \quad mm \in MESSAGE \\
& \quad \land mm \notin \text{dom}(sender) \\
& \quad \land tt \in \text{trans} \\
& \quad \land (ss \rightarrow tt) \in \text{activetrans} \\
& \quad \land \text{sitetransstatus}(tt)(ss) = \text{pending} \\
& \quad \land ss \neq \text{coordinator}(tt) \\
& \quad \land \text{ran}(\text{transelect}(tt)) \neq \emptyset \\
\text{THEN} & \quad \text{sitetransstatus}(tt)(ss) := \text{abort} \\
& \quad \land \text{freeobject} := \text{freeobject} \cup \{ ss \} \times \text{transobject}(tt) \\
& \quad \land \text{activetrans} := \text{activetrans} \setminus \{ ss \rightarrow tt \} \\
& \quad \land \text{voteabort} := \text{voteabort} \cup \{ mm \} \\
& \quad \land \text{tranvoteabort} := \text{tranvoteabort} \cup \{ mm \rightarrow tt \} \\
& \quad \land \text{sender} := \text{sender} \cup \{ mm \rightarrow ss \} \\
& \quad \land \text{completed} := \text{completed} \cup \{ tt \rightarrow ss \} \\
\end{align*}

END;

ExeAbortDecision$(ss, tt) =$

\begin{align*}
\text{ANY} & \quad mm \\
\text{WHERE} & \quad mm \in \text{globalabort} \\
& \quad \land tt \in \text{trans} \\
& \quad \land (mm \rightarrow tt) \in \text{tranglobalabort} \\
& \quad \land (ss \rightarrow mm) \in \text{deliver} \\
& \quad \land (ss \rightarrow tt) \in \text{activetrans} \\
& \quad \land ss \neq \text{coordinator}(tt) \\
& \quad \land \text{ran}(\text{transelect}(tt)) \neq \emptyset \\
& \quad \land \text{sitetransstatus}(tt)(\text{coordinator}(tt)) = \text{abort} \\
& \quad \land \text{sitetransstatus}(tt)(ss) = \text{precommit} \\
\text{THEN} & \quad \text{sitetransstatus}(tt)(ss) := \text{abort} \\
& \quad \land \text{activetrans} := \text{activetrans} \setminus \{ ss \rightarrow tt \} \\
& \quad \land \text{freeobject} := \text{freeobject} \cup \{ ss \} \times \text{transobject}(tt) \\
& \quad \land \text{completed} := \text{completed} \cup \{ tt \rightarrow ss \} \\
\end{align*}

END;
Appendix A Distributed Transactions

\[\text{ExeCommitDecision}(ss,tt) \equiv\]
\[
\begin{align*}
\text{ANY} & \quad pdb, mm \\
\text{WHERE} & \quad tt \in \text{trans} \\
& \quad mm \in \text{globalcommit} \\
& \quad (mm \rightarrow tt) \in \text{tranglobalcommit} \\
& \quad (ss \rightarrow mm) \in \text{deliver} \\
& \quad (ss \rightarrow tt) \in \text{activetrans} \\
& \quad ss \neq \text{coordinator}(tt) \\
& \quad \text{ran}(\text{transeffect}(tt)) \neq \emptyset \\
& \quad pdb = \text{transobject}(tt) \triangle \text{replica}(ss) \\
& \quad \text{sitetransstatus}(tt)(\text{coordinator}(tt)) = \text{commit} \\
& \quad \text{sitetransstatus}(tt)(ss) = \text{precommit} \\
\text{THEN} & \quad \text{activetrans} := \text{activetrans} - \{ss \rightarrow tt\} \\
& \quad \text{sitetransstatus}(tt)(ss) := \text{commit} \\
& \quad \text{freeobject} := \text{freeobject} \cup \{ss\} \times \text{transobject}(tt) \\
& \quad \text{replica}(ss) := \text{replica}(ss) \triangle \text{transeffect}(tt)(pdb) \\
& \quad \text{completed} := \text{completed} \cup \{tt \rightarrow ss\} \\
\end{align*}
\]

END;

A.5 Fourth Refinement

REFINEMENT Replica5
REFINES Replica4
SETS MESSAGE
VARIABLES trans, transstatus, activetrans, coordinator, sitetransstatus, transeffect, transobject, freeobject, replica, sender, deliver, update, voteabort, votecommit, globalabort, globalcommit, tranupdate, transvoteabort, tranvotecommit, tranglobalabort, tranglobalcommit, completed oksite, failedsite

INVARIANT \begin{align*}
\text{sender} & \in \text{MESSAGE} \rightarrow \text{SITE} \quad \text{\&} \quad \text{deliver} \in \text{SITE} \leftrightarrow \text{MESSAGE} \\
\land & \quad \text{update} \subseteq \text{MESSAGE} \quad \land \quad \text{update} \subseteq \text{dom}(\text{sender}) \\
\land & \quad \text{voteabort} \subseteq \text{MESSAGE} \quad \land \quad \text{voteabort} \subseteq \text{dom}(\text{sender}) \\
\land & \quad \text{votecommit} \subseteq \text{MESSAGE} \quad \land \quad \text{votecommit} \subseteq \text{dom}(\text{sender}) \\
\land & \quad \text{globalabort} \subseteq \text{MESSAGE} \quad \land \quad \text{globalabort} \subseteq \text{dom}(\text{sender}) \\
\land & \quad \text{globalcommit} \subseteq \text{MESSAGE} \quad \land \quad \text{globalcommit} \subseteq \text{dom}(\text{sender}) \\
\land & \quad \text{tranupdate} \subseteq \text{update} \rightarrow \text{trans} \\
\land & \quad \text{transvoteabort} \subseteq \text{voteabort} \rightarrow \text{trans} \\
\land & \quad \text{tranvotecommit} \subseteq \text{votecommit} \rightarrow \text{trans} \\
\land & \quad \text{tranglobalabort} \subseteq \text{globalabort} \rightarrow \text{trans} \\
\land & \quad \text{tranglobalcommit} \subseteq \text{globalcommit} \rightarrow \text{trans} \\
\land & \quad \text{completed} \subseteq \text{trans} \leftrightarrow \text{SITE} \\
\land & \quad \text{oksite} \subseteq \text{SITE} \\
\land & \quad \text{failedsite} \subseteq \text{SITE} \\
\land & \quad \text{oksite} \cap \text{failedsite} = \emptyset
\end{align*}
INITIALISATION

\[ \text{trans} := \emptyset \]  \| \text{transstatus} := \emptyset \]  \| \text{activetrans} := \emptyset \]
\[ \| \text{coordinator} := \emptyset \]  \| \text{sitetransstatus} := \emptyset \]  \| \text{transeffect} := \emptyset \]
\[ \| \text{transobject} := \emptyset \]  \| \text{freeobject} := \text{SITE} \times \text{OBJECT} \]
\[ \| \text{ANY} \text{ data WHERE} \text{ data} \in \text{OBJECT} \rightarrow \text{VALUE} \]
\[ \text{THEN} \text{ replica} := \text{SITE} \times \{\text{data}\} \text{ END} \]
\[ \| \text{update} := \emptyset \]  \| \text{voteabort} := \emptyset \]  \| \text{votecommit} := \emptyset \]
\[ \| \text{globalabort} := \emptyset \]  \| \text{globalcommit} := \emptyset \]  \| \text{tranupdate} := \emptyset \]
\[ \| \text{tranglobalabort} := \emptyset \]  \| \text{transvoteabort} := \emptyset \]  \| \text{tranglobalcommit} := \emptyset \]
\[ \| \text{oksite} := \text{SITE} \]  \| \text{failedsite} := \emptyset \]

EVENTS

\[ \text{SiteFailure}(ss \in \text{SITE}) \equiv \]
\[ \text{WHEN} \quad ss \not\in \text{oksite} \]
\[ \text{THEN} \quad \text{failedsite} := \text{failedsite} \cup \{ss\} \]
\[ \| \text{oksite} := \text{oksite} - \{ss\} \]
\[ \text{END}; \]

\[ \text{StartTran}(tt) \equiv \]
\[ \text{ANY} \quad ss, \text{updates}, \text{objects} \]
\[ \text{WHERE} \quad ss \in \text{SITE} \]
\[ \quad tt \not\in \text{trans} \]
\[ \quad \text{updates} \in \text{UPDATE} \]
\[ \quad \text{objects} \in \text{P}_1(\text{OBJECT}) \]
\[ \quad \text{ValidUpdate (updates,objects)} \]
\[ \quad ss \not\in \text{oksite} \]
\[ \text{THEN} \quad \text{trans} := \text{trans} \cup \{tt\} \]
\[ \| \quad \text{transstatus}(tt) := \text{PENDING} \]
\[ \| \quad \text{transobject}(tt) := \text{objects} \]
\[ \| \quad \text{transeffect}(tt) := \text{updates} \]
\[ \| \quad \text{coordinator}(tt) := ss \]
\[ \| \quad \text{sitetransstatus}(tt) := \{\text{coordinator}(tt) \rightarrow \text{pending}\} \]
\[ \text{END}; \]

\[ \text{SendUpdate}(ss \in \text{SITE}, mm \in \text{MESSAGE}, tt \in \text{TRANSACTION}) \equiv \]
\[ \text{WHEN} \quad mm \not\in \text{dom}(\text{sender}) \]
\[ \quad tt \in \text{trans} \]
\[ \quad \text{sitetransstatus}(tt)(\text{coordinator}(tt)) = \text{pending} \]
\[ \quad ss = \text{coordinator}(tt) \]
\[ \quad tt \not\in \text{ran}\{\text{tranupdate}\} \]
\[ \quad \text{ran}\{\text{transeffect}(tt)\} \neq \emptyset \]
\[ \quad \text{coordinator}(tt) \in \text{oksite} \]
\[ \text{THEN} \quad \text{sender} := \text{sender} \cup \{mm \rightarrow ss\} \]
\[ \| \quad \text{update} := \text{update} \cup \{mm\} \]
\[ \| \quad \text{tranupdate} := \text{tranupdate} \cup \{mm \rightarrow tt\} \]
\[ \text{END}; \]

\[ \text{Deliver}(ss \in \text{SITE}, mm \in \text{MESSAGE}) \equiv \]
\[ \text{WHEN} \quad mm \in \text{dom}(\text{sender}) \]
\[ \quad (ss \rightarrow mm) \in \text{deliver} \]
\[ \quad ss \in \text{oksite} \]
\[ \text{THEN} \quad \text{deliver} := \text{deliver} \cup \{ss \rightarrow mm\} \]
\[ \text{END}; \]
IssueWriteTran\(tt\) \(\in\) TRANSACTION \(\equiv\)
\[\forall m, m_{\text{update}}: m \in \text{update} \wedge tt \in \text{trans} \wedge (m \rightarrow tt) \in \text{tranupdate} \wedge (\text{coordinator}(tt) \rightarrow m) \in \text{activetrans} \wedge \text{sitetransstatus}(tt)(\text{coordinator}(tt)) = \text{pending} \wedge \text{ran}(\text{transeffect}(tt)) \neq \emptyset \wedge \text{transobject}(tt) \subseteq \text{freeobject}[^{\text{coordinator}(tt)}] \wedge \forall t, (t \in \text{trans} \land (\text{coordinator}(tt) \rightarrow t) \in \text{activetrans} \Rightarrow \text{transobject}(tt) \cap \text{transobject}(t) = \emptyset) \wedge \text{coordinator}(tt) \in \text{oksite} \]
\[\text{THEN} \]
\[\text{activetrans} := \text{activetrans} \cup ([\text{coordinator}(tt) \rightarrow tt]) \wedge \text{sitetransstatus}(tt)(\text{coordinator}(tt)) = \text{precommit} \wedge \text{freeobject} := \text{freeobject} - [\text{coordinator}(tt)] \times \text{transobject}(tt) \]
\[\text{END};\]

AbortWriteTran\(tt\) \(\equiv\)
\[\forall m_1, m_2: m_1 \in \text{voteabort} \land m_1 \rightarrow tt \in \text{tranvoteabort} \wedge \text{coordinator}(tt) \rightarrow m_1 \in \text{deliver} \wedge m_2 \in \text{MESSAGE} \land m_2 \notin \text{dom}(\text{sender}) \wedge tt \in \text{trans} \wedge \text{ran}(\text{transeffect}(tt)) \neq \emptyset \wedge (\text{coordinator}(tt) \rightarrow tt) \in \text{activetrans} \wedge \text{transstatus}(tt) = \text{PENDING} \wedge \exists s. (s \in \text{SITE} \land \text{sitetransstatus}(tt)(s) = \text{abort}) \wedge \text{coordinator}(tt) \in \text{oksite} \]
\[\text{THEN} \]
\[\text{transstatus}(tt) := \text{ABORT} \wedge \text{activetrans} := \text{activetrans} - [\text{coordinator}(tt) \rightarrow tt] \wedge \text{sitetransstatus}(tt)(\text{coordinator}(tt)) := \text{abort} \wedge \text{freeobject} := \text{freeobject} \cup [\text{coordinator}(tt)] \times \text{transobject}(tt) \wedge \text{globalabort} := \text{globalabort} \cup \{m_2\} \wedge \text{tranglobalabort} := \text{tranglobalabort} \cup \{m_2 \rightarrow tt\} \wedge \text{sender} := \text{sender} \cup \{m_2 \rightarrow \text{coordinator}(tt)\} \wedge \text{completed} := \text{completed} \cup \{tt \rightarrow \text{coordinator}(tt)\} \]
\[\text{END};\]

CommitWriteTran\(tt\) \(\equiv\)
\[\forall pdb, m, m_{\text{MESSAGE}}: m_{\text{MESSAGE}} \in \text{MESSAGE} \wedge m \notin \text{dom}(\text{sender}) \wedge tt \in \text{trans} \wedge pdb = \text{transobject}(tt) \setminus \text{replica}(\text{coordinator}(tt)) \wedge \text{ran}(\text{transeffect}(tt)) \neq \emptyset \wedge (\text{coordinator}(tt) \rightarrow tt) \in \text{activetrans} \wedge \text{transstatus}(tt) = \text{PENDING} \wedge \forall s. (s \in \text{SITE} \Rightarrow \text{sitetransstatus}(tt)(s) = \text{precommit} \land \forall m. (m \in \text{votecommit} \land m \rightarrow tt \in \text{tranvotecommit} \Rightarrow \text{coordinator}(tt) \rightarrow m \in \text{deliver}) \wedge \text{coordinator}(tt) \in \text{oksite} \]
\[\text{THEN} \]
\[\text{transstatus}(tt) := \text{COMMIT} \wedge \text{activetrans} := \text{activetrans} - [\text{coordinator}(tt) \rightarrow tt] \wedge \text{sitetransstatus}(tt)(\text{coordinator}(tt)) := \text{commit} \wedge \text{freeobject} := \text{freeobject} \cup [\text{coordinator}(tt)] \times \text{transobject}(tt) \wedge \text{replica}(\text{coordinator}(tt)) := \text{replica}(\text{coordinator}(tt)) \setminus \text{transeffect}(tt)(pdb)\]
Appendix A Distributed Transactions


globalcommit := globalcommit \cup \{mm\}
tranglobalcommit := tranglobalcommit \cup \{mm\rightarrow tt\}
sender := sender \cup \{mm \rightarrow \text{coordinator}(tt)\}
completed := completed \cup \{tt \rightarrow \text{coordinator}(tt)\}

END;

val \leftarrow \text{ReadTran}(tt,ss) \equiv

WHEN \text{tt} \in \text{trans}
\wedge \text{transstatus}(tt) = \text{PENDING}
\wedge \text{transobject}(tt) \subseteq \text{freeobject}(\{ss\})
\wedge \text{ss} = \text{coordinator}(tt)
\wedge \text{ran}(\text{transaffect}(tt)) = \{\emptyset\}
\wedge \text{coordinator}(tt) \in \text{oksite}

THEN
val := \text{transobject}(tt) \cup \text{replica}(ss)
|| \text{sitetransstatus}(tt)(ss) := \text{commit}
|| \text{transstatus}(tt) := \text{COMMIT}
|| \text{completed} := \text{completed} \cup \{tt \rightarrow ss\}

END;

\text{BeginSubTran}(\text{tt} \in \text{TRANSACTION}, \text{ss} \in \text{SITE}) \equiv

ANY \text{mm}
\text{WHERE} \text{mm} \in \text{update}
\wedge \text{tt} \in \text{trans}
\wedge (\text{mm} \rightarrow \text{tt}) \in \text{transupdate}
\wedge (\text{ss} \rightarrow \text{mm}) \in \text{deliver}
\wedge (\text{ss} \rightarrow \text{tt}) \in \text{activetrans}
\wedge \text{ss} \notin \text{dom} (\text{sitetransstatus}(tt))
\wedge \text{ss} \notin \text{coordinator}(tt)
\wedge \text{ran}(\text{transaffect}(tt)) \neq \{\emptyset\}
\wedge \text{transobject}(tt) \subseteq \text{freeobject}(\{ss\})
\wedge \forall \text{tz}(\text{tz} \in \text{trans} \wedge (\text{ss} \rightarrow \text{tz}) \in \text{activetrans} 
\quad \Rightarrow \text{transobject}(tt) \cap \text{transobject}(\text{tz}) = \emptyset)
\wedge \text{ss} \in \text{oksite}

THEN
\text{activetrans} := \text{activetrans} \cup \{\text{ss} \rightarrow \text{tt}\}
|| \text{sitetransstatus}(tt)(ss) := \text{pending}
|| \text{freeobject} := \text{freeobject} \setminus \{\text{ss}\} \times \text{transobject}(tt)

END;

\text{SiteCommitTx}(tt \in \text{TRANSACTION}, \text{ss} \in \text{SITE}) \equiv

ANY \text{mm}
\text{WHERE} \text{mm} \in \text{MESSAGE}
\wedge \text{mm} \notin \text{dom}(\text{sender})
\wedge \text{tt} \in \text{trans}
\wedge (\text{ss} \rightarrow \text{tt}) \in \text{activetrans}
\wedge \text{sitetransstatus}(tt)(ss) := \text{pending}
\wedge \text{ss} \neq \text{coordinator}(tt)
\wedge \text{ran}(\text{transaffect}(tt)) \neq \{\emptyset\}
\wedge \text{ss} \in \text{oksite}

THEN
\text{sitetransstatus}(tt)(ss) := \text{precommit}
|| \text{votecommit} := \text{votecommit} \cup \{\text{mm}\}
|| \text{tranvotecommit} := \text{tranvotecommit} \cup \{\text{mm} \rightarrow \text{tt}\}
|| \text{sender} := \text{sender} \cup \{\text{mm} \rightarrow \text{ss}\}

END;
Appendix A Distributed Transactions

SiteAbortTx(tt.Transaction, ss.Site) ≡

\[
\begin{align*}
\text{ANY } &\quad mm \\
\text{WHERE } &\quad mm \in \text{MESSAGE} \\
&\quad \wedge mm \notin \text{dom(sender)} \\
&\quad \wedge tt \in \text{trans} \\
&\quad \wedge (ss \rightarrow tt) \in \text{activetrans} \\
&\quad \wedge \text{sitetransstatus}(tt)(ss) = \text{pending} \\
&\quad ss \neq \text{coordinator}(tt) \\
&\quad \text{ran}(\text{transeffect}(tt)) \neq \emptyset \\
&\quad ss \in \text{oksге} \\
\text{THEN } &\quad \text{sitetransstatus}(tt)(ss) := \text{abort} \\
&\quad \text{freeobject} := \text{freeobject} \cup \{ss \rightarrow tt\} \\
&\quad \text{activetrans} := \text{activetrans} -\{ss \rightarrow tt\} \\
&\quad \text{voteabort} := \text{voteabort} \cup \{mm\} \\
&\quad \text{tranvoteabort} := \text{tranvoteabort} \cup \{mm \rightarrow tt\} \\
&\quad \text{sender} := \text{sender} \cup \{mm \rightarrow ss\} \\
&\quad \text{completed} := \text{completed} \cup \{tt \rightarrow ss\} \\
\end{align*}
\]

END;

ExeAbortDecision(ss, tt) ≡

\[
\begin{align*}
\text{ANY } &\quad mm \\
\text{WHERE } &\quad \wedge mm \in \text{globalabort} \\
&\quad \wedge tt \in \text{trans} \\
&\quad \wedge (mm \rightarrow tt) \in \text{tranglobalabort} \\
&\quad \wedge (ss \rightarrow mm) \in \text{deliver} \\
&\quad \wedge (ss \rightarrow tt) \in \text{activetrans} \\
&\quad ss \neq \text{coordinator}(tt) \\
&\quad \text{ran}(\text{transeffect}(tt)) \neq \emptyset \\
&\quad \text{sitetransstatus}(tt)(\text{coordinator}(tt)) = \text{abort} \\
&\quad \text{sitetransstatus}(tt)(ss) = \text{precommit} \\
&\quad ss \in \text{oksге} \\
\text{THEN } &\quad \text{sitetransstatus}(tt)(ss) := \text{abort} \\
&\quad \text{activetrans} := \text{activetrans} -\{ss \rightarrow tt\} \\
&\quad \text{freeobject} := \text{freeobject} \cup \{ss \times \text{transobject}(tt)\} \\
&\quad \text{completed} := \text{completed} \cup \{tt \rightarrow ss\} \\
\end{align*}
\]

END;

ExeCommitDecision(ss, tt) ≡

\[
\begin{align*}
\text{ANY } &\quad pdb, mm \\
\text{WHERE } &\quad tt \in \text{trans} \\
&\quad \wedge mm \in \text{globalcommit} \\
&\quad \wedge (mm \rightarrow tt) \in \text{tranglobalcommit} \\
&\quad \wedge (ss \rightarrow mm) \in \text{deliver} \\
&\quad \wedge (ss \rightarrow tt) \in \text{activetrans} \\
&\quad ss \neq \text{coordinator}(tt) \\
&\quad \text{ran}(\text{transeffect}(tt)) \neq \emptyset \\
&\quad \text{pdb} = \text{transobject}(tt) \land \text{replica}(ss) \\
&\quad \text{sitetransstatus}(tt)(\text{coordinator}(tt)) = \text{commit} \\
&\quad \text{sitetransstatus}(tt)(ss) = \text{precommit} \\
&\quad ss \in \text{oksге} \\
\text{THEN } &\quad \text{activetrans} := \text{activetrans} -\{ss \rightarrow tt\} \\
&\quad \text{sitetransstatus}(tt)(ss) := \text{commit} \\
&\quad \text{freeobject} := \text{freeobject} \cup \{ss \times \text{transobject}(tt)\} \\
&\quad \text{replica}(ss) := \text{replica}(ss) \land \text{transeffect}(tt)(pdb) \\
&\quad \text{completed} := \text{completed} \cup \{tt \rightarrow ss\} \\
\end{align*}
\]

END:
Appendix B

TimeOut

\text{TimeOut}(tt \in \text{TRANSACTION}) \equiv /* Abstract Model */
\begin{align*}
\text{WHEN} & \quad tt \in \text{trans} \\
& \quad \land \, \text{transstatus}(tt) = \text{PENDING} \\
& \quad \land \, \text{ran}(\text{transeffect}(tt)) \neq \emptyset \\
\text{THEN} & \quad \text{transstatus}(tt) := \text{ABORT} \\
\text{END;}
\end{align*}

\text{TimeOut}(tt \in \text{TRANSACTION}) \equiv /* First Refinement */
\begin{align*}
\text{WHEN} & \quad tt \in \text{trans} \\
& \quad \land \, \text{ran}(\text{transeffect}(tt)) \neq \emptyset \\
& \quad \land \, (\text{coordinator}(tt) \rightarrow tt) \in \text{activetrans} \\
& \quad \land \, \text{transstatus}(tt) = \text{PENDING} \\
\text{THEN} & \quad \text{activetrans} := \text{activetrans} - \{\text{coordinator}(tt)\rightarrow tt\} \\
& \quad \text{sitetransstatus}(tt)(\text{coordinator}(tt)) := \text{abort} \\
& \quad \text{freeobject} := \text{freeobject} \cup \{\text{coordinator}(tt)\} \times \text{transobject}(tt) \\
\text{END;}
\end{align*}

\text{TimeOut}(tt \in \text{TRANSACTION}) \equiv /* Second Refinement */
\begin{align*}
\text{WHEN} & \quad tt \in \text{trans} \\
& \quad \land \, \text{ran}(\text{transeffect}(tt)) \neq \emptyset \\
& \quad \land \, (\text{coordinator}(tt) \rightarrow tt) \in \text{activetrans} \\
& \quad \land \, \text{transstatus}(tt) = \text{PENDING} \\
\text{THEN} & \quad \text{activetrans} := \text{activetrans} - \{\text{coordinator}(tt)\rightarrow tt\} \\
& \quad \text{sitetransstatus}(tt)(\text{coordinator}(tt)) := \text{abort} \\
& \quad \text{freeobject} := \text{freeobject} \cup \{\text{coordinator}(tt)\} \times \text{transobject}(tt) \\
\text{END;}
\end{align*}
\textbf{TimeOut}(tt \in \textsc{Transaction}) \equiv \quad /* \text{Third Refinement} */

\begin{align*}
\text{ANY} & \quad \text{mm} \\
\text{WHERE} & \quad \text{mm} \in \text{MESSAGE} \land \text{mm} \notin \text{dom}(\text{sender}) \\
& \quad \land \quad \text{tt} \in \text{trans} \\
& \quad \land \quad \text{ran}(\text{transeffect}(\text{tt})) \neq \{\emptyset\} \\
& \quad \land \quad (\text{coordinator}(\text{tt}) \rightarrow \text{tt}) \in \text{activetrans} \\
& \quad \land \quad \text{transstatus}(\text{tt}) = \text{PENDING} \\
\text{THEN} & \quad \text{transstatus}(\text{tt}) := \text{ABORT} \\
& \quad \land \quad \text{activetrans} := \text{activetrans} - \{\text{coordinator}(\text{tt}) \rightarrow \text{tt}\} \\
& \quad \land \quad \text{sitetransstatus}(\text{tt})(\text{coordinator}(\text{tt})) := \text{abort} \\
& \quad \land \quad \text{freeobject} := \text{freeobject} \cup \{\text{coordinator}(\text{tt})\} \times \text{transobject}(\text{tt}) \\
& \quad \land \quad \text{globalabort} := \text{globalabort} \cup \{\text{mm}\} \\
& \quad \land \quad \text{tranglobalabort} := \text{tranglobalabort} \cup \{\text{mm} \rightarrow \text{tt}\} \\
& \quad \land \quad \text{sender} := \text{sender} \cup \{\text{mm} \rightarrow \text{coordinator}(\text{tt})\} \\
& \quad \land \quad \text{completed} := \text{completed} \cup \{\text{tt} \rightarrow \text{coordinator}(\text{tt})\} \\
\text{END};
\end{align*}

\textbf{TimeOut}(tt \in \textsc{Transaction}) \equiv \quad /* \text{Fourth Refinement} */

\begin{align*}
\text{ANY} & \quad \text{mm} \\
\text{WHERE} & \quad \text{mm} \in \text{MESSAGE} \land \text{mm} \notin \text{dom}(\text{sender}) \\
& \quad \land \quad \text{tt} \in \text{trans} \\
& \quad \land \quad \text{ran}(\text{transeffect}(\text{tt})) \neq \{\emptyset\} \\
& \quad \land \quad (\text{coordinator}(\text{tt}) \rightarrow \text{tt}) \in \text{activetrans} \\
& \quad \land \quad \text{transstatus}(\text{tt}) = \text{PENDING} \\
& \quad \land \quad \text{coordinator}(\text{tt}) \in \text{oksite} \\
\text{THEN} & \quad \text{transstatus}(\text{tt}) := \text{ABORT} \\
& \quad \land \quad \text{activetrans} := \text{activetrans} - \{\text{coordinator}(\text{tt}) \rightarrow \text{tt}\} \\
& \quad \land \quad \text{sitetransstatus}(\text{tt})(\text{coordinator}(\text{tt})) := \text{abort} \\
& \quad \land \quad \text{freeobject} := \text{freeobject} \cup \{\text{coordinator}(\text{tt})\} \times \text{transobject}(\text{tt}) \\
& \quad \land \quad \text{globalabort} := \text{globalabort} \cup \{\text{mm}\} \\
& \quad \land \quad \text{tranglobalabort} := \text{tranglobalabort} \cup \{\text{mm} \rightarrow \text{tt}\} \\
& \quad \land \quad \text{sender} := \text{sender} \cup \{\text{mm} \rightarrow \text{coordinator}(\text{tt})\} \\
& \quad \land \quad \text{completed} := \text{completed} \cup \{\text{tt} \rightarrow \text{coordinator}(\text{tt})\} \\
\text{END};
Appendix C

Causal Order Broadcast

C.1 Abstract Model

\[
\begin{align*}
\text{MACHINE} & \quad C11 \\
\text{SETS} & \quad \text{PROCESS}; \text{MESSAGE} \\
\text{VARIABLES} & \quad \text{sender}, \text{cdeliver} \\
\text{INVARIANT} & \\
& \quad \text{sender} \in \text{MESSAGE} \rightarrow \text{PROCESS} \\
& \quad \land \quad \text{cdeliver} \in \text{PROCESS} \leftrightarrow \text{MESSAGE} \\
& \quad \land \quad \text{ran}(\text{cdeliver}) \subseteq \text{dom}(\text{sender}) \\
\text{INITIALISATION} & \quad \text{sender} := \emptyset || \quad \text{cdeliver} := \emptyset \\
\text{EVENTS} & \\
\text{Broadcast} (pp \in \text{PROCESS}, mm \in \text{MESSAGE}) \triangleq \\
& \quad \text{WHEN} \quad mm \notin \text{dom}(\text{sender}) \\
& \quad \text{THEN} \quad \text{sender} := \text{sender} \cup \{mm \rightarrow pp\} \\
& \quad \| \quad \text{cdeliver} := \text{cdeliver} \cup \{pp \rightarrow mm\} \\
\text{END;} \\
\text{Deliver} (pp \in \text{PROCESS}, mm \in \text{MESSAGE}) \triangleq \\
& \quad \text{WHEN} \quad mm \in \text{dom}(\text{sender}) \\
& \quad \\land \quad (pp \rightarrow mm) \notin \text{cdeliver} \\
& \quad \text{THEN} \quad \text{cdeliver} := \text{cdeliver} \cup \{pp \rightarrow mm\} \\
\text{END;} \\
\text{END}
\end{align*}
\]
C.2 First Refinement

REFINEMENT C22
REFINES C11
VARIABLES sender, cdeliver, corder, delorder
INVARIANT

\[ \text{corder} \in \text{MESSAGE} \leftrightarrow \text{MESSAGE} \]
\[ \land \text{delorder} \in \text{PROCESS} \rightarrow (\text{MESSAGE} \leftrightarrow \text{MESSAGE}) \]
\[ \land \text{dom(corder)} \subseteq \text{dom(sender)} \]
\[ \land \text{ran(corder)} \subseteq \text{dom(sender)} \]
\[ \land \text{ran(cdeliver) \subseteq dom(sender)} \]

\[ \forall (m_1, m_2, p) . \ (m_1 \in \text{MESSAGE} \land m_2 \in \text{MESSAGE} \land p \in \text{PROCESS} \]
\[ \land (m_1 \rightarrow m_2) \in \text{corder} \land (p \rightarrow m_2) \in \text{cdeliver} \]
\[ \Rightarrow (m_1 \rightarrow m_2) \in \text{delorder}(p) \]

\[ \forall (m_1, m_2, m_3) . \ (m_1 \in \text{MESSAGE} \land m_2 \in \text{MESSAGE} \land m_3 \in \text{MESSAGE} \]
\[ \land (m_1 \rightarrow m_2) \in \text{corder} \land (m_2 \rightarrow m_3) \in \text{corder} \]
\[ \Rightarrow (m_1 \rightarrow m_3) \in \text{corder} \]

\[ \forall (m_1, m_2, p) . \ (m_1 \in \text{MESSAGE} \land m_2 \in \text{MESSAGE} \land p \in \text{PROCESS} \]
\[ \land (m_1 \rightarrow m_2) \in \text{corder} \land m_2 \in \text{sender}^{-1}[[p]] \]
\[ \Rightarrow (m_1 \in \text{sender}^{-1}[[p]] \lor m_2 \in \text{cdeliver}[[p]]) \]

\[ \forall (m_1, m_2, p) . \ (m_1 \in \text{MESSAGE} \land m_2 \in \text{MESSAGE} \land p \in \text{PROCESS} \]
\[ \land (m_1 \rightarrow m_2) \in \text{corder} \land m_2 \in \text{sender}^{-1}[[p]] \lor \text{cdeliver}[[p]] \]
\[ \Rightarrow (m_1 \in \text{sender}^{-1}[[p]] \lor \text{cdeliver}[[p]]) \]

INITIALISATION

\[ \text{sender} := \emptyset \ \ \ \text{cdeliver} := \emptyset \]
\[ \| \ \text{corder} := \emptyset \ \ \ \text{delorder} := \emptyset \]

EVENTS

Broadcast (pp \in \text{PROCESS} , mm \in \text{MESSAGE} ) \equiv

WHEN \ mm \in \text{dom}(\text{sender})
THEN \ corder := \text{corder} \cup (\text{sender}^{-1}[[pp]] \times \{m_1\})
\cup (\text{cdeliver}[[pp]] \times \{mm\})
\| \ \text{sender} := \text{sender} \cup \{mm \rightarrow pp\}
\| \ \text{cdeliver} := \text{cdeliver} \cup \{pp \rightarrow mm\}
\| \ \text{delorder}(pp) := \text{delorder}(pp) \cup \{ \text{cdeliver}[[pp]] \times \{mm\} \}
END;
Deliver \((pp \in \text{PROCESS}, mm \in \text{MESSAGE})\) 

\[
\begin{align*}
\text{WHEN} & \quad mm \in \text{dom(sender)} \\
& \quad (pp \mapsto mm) \in \text{cdeliver} \\
& \quad \forall m.(m \in \text{MESSAGE} \land (m \mapsto mm) \in \text{corder}) \\
& \quad \Rightarrow (pp \mapsto m) \in \text{cdeliver} \\
\text{THEN} & \quad \text{cdeliver} := \text{cdeliver} \cup (pp \mapsto mm) \\
& \quad \parallel \text{delorder}(pp) := \text{delorder}(pp) \cup (\text{cdeliver} \{[pp]\} \times [mm])
\end{align*}
\]

END

C.3 Second Refinement

REFINEMENT \ C33
REFINES \ C22
VARIABLES \ VTP, VTM, sender, cdeliver,

INVARIANT

\[
\begin{align*}
\text{VTP} & \in \text{PROCESS} \rightarrow (\text{PROCESS} \rightarrow \mathbb{N}) \\
\land \text{VTM} & \in \text{MESSAGE} \rightarrow (\text{PROCESS} \rightarrow \mathbb{N}) \\
\land \forall (m,p1,p2). (m \in \text{MESSAGE} \land p1 \in \text{PROCESS} \land p2 \in \text{PROCESS} \\
& \land (m \mapsto m1) \in \text{corder} \\
& \Rightarrow \text{VTP}(p1)(p2) \geq \text{VTM}(m)(p2) \\
\land \forall (m1,m2,p). (m1 \in \text{MESSAGE} \land m2 \in \text{MESSAGE} \land p \in \text{PROCESS} \\
& \land (m1 \mapsto m2) \in \text{corder} \\
& \Rightarrow \text{VTM}(m1)(p) \leq \text{VTM}(m2)(p) \\
\land \forall (m,p). (m \in \text{MESSAGE} \land p \in \text{PROCESS} \\
& \land \text{dom(sender)} \Rightarrow \text{VTM}(m)(p) \leq \text{VTP}(p)(p) \\
\land \forall (m,p). (m \in \text{MESSAGE} \land p \in \text{PROCESS} \\
& \land \text{VTM}(m)(p) = 0 \Rightarrow m \notin (\text{dom(corder)} \cup \text{ran(corder)}) \\
\land \forall (p1,p2). (p1 \in \text{PROCESS} \land p2 \in \text{PROCESS} \\
& \land p1 \neq p2 \Rightarrow \text{VTP}(p1)(p2) \leq \text{VTP}(p2)(p2) \\
\land \forall (m1,m2,p). (m1 \in \text{MESSAGE} \land m2 \in \text{MESSAGE} \land p \in \text{PROCESS} \\
& \land \text{VTM}(m1)(p) \leq \text{VTM}(m2)(p) \\
& \Rightarrow ((m1 \mapsto m2) \in (\text{sender}^r[[\text{sender}(m1)] \times \{m2\}] \\
& \quad \cup (\text{cdeliver}[[\text{sender}(m2)] \times \{m2\}])) \\
\end{align*}
\]

INITIALISATION

\[
\begin{align*}
\text{sender} := & \emptyset \\
\text{cdeliver} := & \emptyset \\
\text{VTP} := & \text{PROCESS} \times \{\text{PROCESS} \times \{0\}\} \\
\text{VTM} := & \text{MESSAGE} \times \{\text{PROCESS} \times \{0\}\}
\end{align*}
\]
Appendix C Causal Order Broadcast

EVENTS

\textbf{Broadcast} \((pp \in \text{PROCESS}, mm \in \text{MESSAGE})\) \equiv

\textbf{WHEN} \quad mm \not\in \text{dom(sender)}

\textbf{THEN} \quad \text{LET} \quad nVTP \quad \text{BE} \quad nVTP = VTP(pp) \triangleq \{ pp \rightarrow VTP(pp)(pp)+1 \}

\quad \text{IN} \quad VTM(mm) := nVTP

\quad \text{END}

\quad \text{sender} := \text{sender} \cup \{ mm \rightarrow pp \}

\quad \text{cdeliver} := \text{cdeliver} \cup \{ pp \rightarrow mm \}

\textbf{END} ;

\textbf{Deliver} \((pp \in \text{PROCESS}, mm \in \text{MESSAGE})\) \equiv

\textbf{WHEN} \quad mm \in \text{dom(sender)} \land (pp \rightarrow mm) \not\in \text{cdeliver} \land \forall p. (p \in \text{PROCESS} \land p \not\in \text{sender(mm)} \Rightarrow VTP(pp)(p) \geq VTM(mm)(p)) \land VTP(pp)(\text{sender(mm)}) = VTM(mm)(\text{sender(mm)}) - 1

\textbf{THEN} \quad \text{cdeliver} := \text{cdeliver} \cup \{ pp \rightarrow mm \}

\quad \text{VTP}(pp) := VTP(pp) \triangleq

\quad \{ q \mid q \in \text{PROCESS} \land VTP(pp)(q) < VTM(mm)(q) \} \triangleq VTM(mm)

\textbf{END} ;

\section*{C.4 Third Refinement}

\textbf{REFINEMENT} \quad \text{C44}

\textbf{REFINES} \quad \text{C33}

\textbf{VARIABLES} \quad VTP, VTM, sender, cdeliver

\textbf{INITIALISATION} \quad \begin{align*}
\text{sender} & := \emptyset \\
\text{cdeliver} & := \emptyset \\
\text{VTP} & := \text{PROCESS} \times \{ \text{PROCESS} \times \{0\} \} \\
\text{VTM} & := \text{MESSAGE} \times \{ \text{PROCESS} \times \{0\} \}
\end{align*}

\textbf{EVENTS}

\textbf{Broadcast} \((pp \in \text{PROCESS}, mm \in \text{MESSAGE})\) \equiv

\textbf{WHEN} \quad mm \not\in \text{dom(sender)}

\textbf{THEN} \quad \text{LET} \quad nVTP \quad \text{BE} \quad nVTP = VTP(pp) \triangleq \{ pp \rightarrow VTP(pp)(pp)+1 \}

\quad \text{IN} \quad VTM(mm) := nVTP

\quad \text{END}

\quad \text{sender} := \text{sender} \cup \{ mm \rightarrow pp \}

\quad \text{cdeliver} := \text{cdeliver} \cup \{ pp \rightarrow mm \}

\textbf{END} ;

\textbf{Deliver} \((pp \in \text{PROCESS}, mm \in \text{MESSAGE})\) \equiv

\textbf{WHEN} \quad mm \in \text{dom(sender)} \land (pp \rightarrow mm) \not\in \text{cdeliver} \land \forall p. (p \in \text{PROCESS} \land p \not\in \text{sender(mm)} \Rightarrow VTP(pp)(p) \geq VTM(mm)(p)) \land VTP(pp)(\text{sender(mm)}) = VTM(mm)(\text{sender(mm)}) - 1

\textbf{THEN} \quad \text{cdeliver} := \text{cdeliver} \cup \{ pp \rightarrow mm \}

\quad \text{VTP}(pp) := VTP(pp) \triangleq \{ \text{sender(mm)} \rightarrow VTM(mm)(\text{sender(mm)}) \}

\textbf{END} ;
C.5 Fourth Refinement

REFINEMENT C55
REFINES C44
VARIABLES VTP, VTM, sender, cdeliver
INITIALISATION

sender := ∅
|| cdeliver := ∅
|| VTP := PROCESS × {PROCESS × {0}}
|| VTM := MESSAGE × {PROCESS × {0}}

EVENTS

Broadcast (pp ∈ PROCESS, mm ∈ MESSAGE) ≡
WHEN mm ∈ dom(sender)
THEN LET nVTP BE nVTP = VTP(pp) ⊕ { pp → VTP(pp)(pp)+1 }

IN VTM(mm) := nVTP

|| VTP(pp) := nVTP
END
|| sender := sender ∪ { mm → pp }
|| cdeliver := cdeliver ∪ { pp → mm }

END :

Deliver (pp ∈ PROCESS, mm ∈ MESSAGE) ≡
WHEN mm ∈ dom(sender)

∧ (pp → mm) ∈ cdeliver
∧ ∀p.( p ∈ PROCESS ∧ p ≠ sender(mm) ⇒ VTP(pp)(p) ≥ VTM(mm)(p))
∧ VTP(pp)(sender(mm)) = VTM (mm)(sender(mm)) - 1

THEN cdeliver := cdeliver ∪ { pp → mm }
|| VTP(pp) (sender(mm)) := VTM(mm)(sender(mm))

END:
Appendix D

Total Order Broadcast

D.1 Abstract Model

MACHINE TO11

SETS PROCESS, MESSAGE

VARIABLES sender, totalorder, delorder, tdeliver

INVARIANT sender ∈ MESSAGE → PROCESS

∧ totalorder ∈ MESSAGE ↔ MESSAGE

∧ delorder ∈ PROCESS → (MESSAGE ↔ MESSAGE)

∧ tdeliver ∈ PROCESS ↔ MESSAGE

∧ ran(tdeliver) ⊆ dom(sender)

∧ ∀ (m1,m2,p) . ( m1 ∈ MESSAGE ∧ m2 ∈ MESSAGE ∧ p ∈ PROCESS

∧ (m1→m2) ∈ delorder(p) )

⇒ (m1 → m2) ∈ totalorder )

∧ ∀ (m1,m2,p) . ( m1 ∈ MESSAGE ∧ m2 ∈ MESSAGE ∧ p ∈ PROCESS

∧ (p → m1) ∈ tdeliver ∧ (p → m2) ∈ tdeliver

∧ m2 ∈ ran(tdeliver) )

⇒ (m1 → m2) ∈ totalorder )

∧ ∀ (m1,m2) . ( m1 ∈ MESSAGE ∧ m2 ∈ MESSAGE

∧ m1 ∈ ran(tdeliver) ∧ m2 ∈ ran(tdeliver)

∧ (m2 → m1) ∈ totalorder )

⇒ (m1 → m2) ∈ totalorder )

∧ ∀ (m1,m2,p) . ( m1 ∈ MESSAGE ∧ m2 ∈ MESSAGE ∧ p ∈ PROCESS

∧ (p → m1) ∈ tdeliver ∧ (p → m2) ∈ tdeliver

∧ (m2 → m1) ∈ totalorder )

⇒ (m1 → m2) ∈ totalorder )
\[ \forall (m_1, m_2, p_1, p_2) . \ (m_1 \in \text{MESSAGE} \land m_2 \in \text{MESSAGE} \land p_1 \in \text{PROCESS} \land p_2 \in \text{PROCESS} \land \langle p_1 \mapsto m_1 \rangle \in \text{tdeliver} \land \langle p_1 \mapsto m_2 \rangle \notin \text{tdeliver} \land \langle p_2 \mapsto m_1 \rangle \in \text{tdeliver} \land \langle p_2 \mapsto m_2 \rangle \in \text{tdeliver} \implies \langle m_1 \mapsto m_2 \rangle \in \text{totalorder} \] \\
\[ \forall (m) . \ (m \in \text{MESSAGE} \implies m \mapsto m \in \text{totalorder}) \] \\
\[ \forall (m_1, m_2) . \ (m_1 \in \text{MESSAGE} \land m_2 \in \text{MESSAGE} \land (m_1 \mapsto m_2) \in \text{totalorder} \land (m_2 \mapsto m_3) \in \text{totalorder} \implies (m_1 \mapsto m_3) \in \text{totalorder} \] \\
\[ \forall (m_1, m_2) . \ (m_1 \in \text{MESSAGE} \land m_2 \in \text{MESSAGE} \land (m_1 \mapsto m_2) \in \text{totalorder} \land (p \mapsto m_2) \in \text{tdeliver} \implies (p \mapsto m_1) \in \text{tdeliver} \] \\
\[ \forall (m) . \ (m \in \text{MESSAGE} \land m \in \text{dom} (\text{totalorder} \cup \text{ran} (\text{totalorder})) \implies m \in \text{ran} (\text{tdeliver}) \] \\
\[ \forall (m) . \ (m \in \text{MESSAGE} \land m \in \text{dom} (\text{sender}) \implies m \in \text{dom} (\text{totalorder}) \] \\
\[ \forall (m) . \ (m \in \text{MESSAGE} \land m \in \text{ran} (\text{tdeliver}) \implies m \notin \text{dom} (\text{totalorder} \cup \text{ran} (\text{totalorder})) \] 

**INITIALISATION**

\[
\begin{align*}
\text{sender} & := \emptyset & \| \text{totalorder} & := \emptyset \\
\text{delorder} & := \text{PROCESS} \times \{\emptyset\} & \| \text{tdeliver} & := \emptyset
\end{align*}
\]

**EVENTS**

**Broadcast** \((pp \in \text{PROCESS} , mm \in \text{MESSAGE}) \equiv \)

\[
\begin{align*}
\text{WHEN} & \quad mm \notin \text{dom} (\text{sender}) \\
\text{THEN} & \quad \text{sender} := \text{sender} \cup \{mm \mapsto pp\} \\
\text{END;}
\end{align*}
\]

**Order** \((pp \in \text{PROCESS} , mm \in \text{MESSAGE}) \equiv \)

\[
\begin{align*}
\text{WHEN} & \quad mm \in \text{dom} (\text{sender}) \\
& \quad \land mm \in \text{ran} (\text{tdeliver}) \\
& \quad \land \text{ran} (\text{tdeliver}) \subseteq \text{tdeliver}[\{pp\}] \\
\text{THEN} & \quad \text{tdeliver} := \text{tdeliver} \cup \{pp \mapsto mm\} \\
& \quad \| \text{totalorder} := \text{totalorder} \cup (\text{ran} (\text{tdeliver}) \times \{mm\}) \\
& \quad \| \text{delorder}(pp) := \text{delorder}(pp) \cup (\text{tdeliver}[\{pp\}] \times \{mm\}) \\
\text{END;}
\end{align*}
\]

**TODeliver** \((pp \in \text{PROCESS} , mm \in \text{MESSAGE}) \equiv \)

\[
\begin{align*}
\text{WHEN} & \quad mm \in \text{dom} (\text{sender}) \\
& \quad \land mm \in \text{ran} (\text{tdeliver}) \\
& \quad \land pp \mapsto mm \notin \text{tdeliver} \\
& \quad \land \forall m. (m \in \text{MESSAGE} \land (m \mapsto mm) \in \text{totalorder} \implies (pp \mapsto m) \in \text{tdeliver}) \\
\text{THEN} & \quad \text{tdeliver} := \text{tdeliver} \cup \{pp \mapsto mm\} \\
& \quad \| \text{delorder}(pp) := \text{delorder}(pp) \cup (\text{tdeliver}[\{pp\}] \times \{mm\}) \\
\text{END}
\end{align*}
\]
D.2 First Refinement

REFINEMENT TO22
REFINES TO11

CONSTANTS sequencer
PROPERTIES sequencer ∈ PROCESS

VARIABLES sender, totalorder, tdeliver

INVARIANT
∀ (m) . ( m ∈ MESSAGE ∧ (sequencer → m) ∈ tdeliver \\
⇒ m ∉ ran(tdeliver) )

∧ ∀ (m) . ( m ∈ MESSAGE ∧ m ∈ dom(totalorder) \\
⇒ (sequencer → m) ∈ tdeliver )

∧ ∀ (m) . ( m ∈ MESSAGE ∧ m ∈ ran(totalorder) \\
⇒ (sequencer → m) ∈ tdeliver )

INITIALISATION
sender := ∅ || totalorder := ∅ || tdeliver := ∅

EVENTS

Broadcast (pp ∈ PROCESS , mm ∈ MESSAGE ) ≡
WHEN mm ∈ dom(sender)
THEN sender := sender ∪ {mm → pp}
END;

Order (pp ∈ PROCESS ,mm ∈ MESSAGE ) ≡
WHEN pp = sequencer
∧ mm ∈ dom(sender)
∧ (sequencer → mm) ∈ tdeliver
THEN tdeliver := tdeliver ∪ {pp → mm}
∥ totalorder := totalorder ∪ ( ran(tdeliver) × {mm} )
END;

TODeliver (pp ∈ PROCESS , mm ∈ MESSAGE ) ≡
WHEN pp ≠ sequencer
∧ mm ∈ dom(sender)
∧ mm ∈ ran ( tdeliver )
∧ pp → mm ∈ tdeliver
∧ ∀m.( m ∈ MESSAGE ∧ (m→ mm) ∈ totalorder \\
⇒ (pp → m) ∈ tdeliver )
THEN tdeliver := tdeliver ∪ {pp → mm}
END
D.3  Second Refinement

REFINEMENT  TO33  
REFINES  TO22  
VARIABLES  sender, totalorder, tdeliver  
INVARIANT  ran(tdeliver) \subseteq tdeliver[[\text{sequencer}]]  
INITIALISATION  sender := \emptyset  \parallel  \text{totalorder} := \emptyset  \parallel  \text{tdeliver} := \emptyset  
EVENTS

Broadcast \((pp \in \text{PROCESS}, mm \in \text{MESSAGE}) \equiv \)

WHEN  \( mm \notin \text{dom}(sender) \)
THEN  sender := sender \cup \{mm \mapsto pp\}  
END;

Order \((pp \in \text{PROCESS}, mm \in \text{MESSAGE}) \equiv \)

WHEN  \( pp = \text{sequencer} \)
\( \land \ mm \in \text{dom}(sender) \)
\( \land \ (\text{sequencer} \mapsto mm) \notin \text{tdeliver} \)
THEN  tdeliver := tdeliver \cup \{pp \mapsto mm\}  
\( \parallel \text{totalorder} := \text{totalorder} \cup (\text{tdeliver}[\text{sequencer}] \times \{mm\}) \)
END;

TODeliver \((pp \in \text{PROCESS}, mm \in \text{MESSAGE}) \equiv \)

WHEN  \( pp \neq \text{sequencer} \)
\( \land \ mm \in \text{dom}(sender) \)
\( \land \ mm \in \text{ran}(\text{tdeliver}) \)
\( \land \ pp \mapsto mm \notin \text{tdeliver} \)
\( \land \ \forall m. (m \in \text{MESSAGE} \land (m \mapsto mm) \in \text{totalorder} \Rightarrow (pp \mapsto m) \in \text{tdeliver}) \)
THEN  tdeliver := tdeliver \cup \{pp \mapsto mm\}  
END

D.4  Third Refinement

REFINEMENT  TO44  
REFINES  TO33  
VARIABLES  sender, totalorder, tdeliver, computation, seqno, counter  
INVARIANT  \( \text{computation} \subseteq \text{MESSAGE} \)
\( \land \ \text{seqno} \in \text{computation} \mapsto \mathbb{N} \)
\( \land \ \text{counter} \in \mathbb{N} \)
\( \land \ \forall (m1,m2). (m1 \in \text{MESSAGE} \land m2 \in \text{MESSAGE} \land m1 \mapsto m2 \in \text{totalorder} \Rightarrow \text{seqno}(m1) < \text{seqno}(m2)) \)
\( \land \ \forall (m1,m2). (m1 \in \text{MESSAGE} \land m2 \in \text{MESSAGE} \land m \in \text{computation} \land m \in \text{dom}(\text{seqno}) \Rightarrow \text{sequencer} \mapsto m \in \text{tdeliver} \)
INITIALISATION

\[
\begin{align*}
\text{sender} &:= \emptyset & \text{totalorder} &:= \emptyset \\
\text{tdeliver} &:= \emptyset & \text{computation} &:= \emptyset \\
\text{seqno} &:= \emptyset & \text{counter} &:= 0
\end{align*}
\]

EVENTS

Broadcast \((pp \in \text{PROCESS}, mm \in \text{MESSAGE})\) \(\equiv\)

\[
\begin{align*}
\text{WHEN} & \quad mm \notin \text{dom}(sender) \\
\text{THEN} & \quad \text{sender} := \text{sender} \cup \{mm \mapsto pp\} \\
& \quad \text{computation} := \text{computation} \cup \{mm\}
\end{align*}
\]

END;

Order \((pp \in \text{PROCESS}, mm \in \text{MESSAGE})\) \(\equiv\)

\[
\begin{align*}
\text{WHEN} & \quad pp = \text{sequencer} \\
& \quad mm \in \text{dom}(sender) \\
& \quad mm \in \text{computation} \\
& \quad (\text{sequencer} \mapsto mm) \notin \text{tdeliver} \\
\text{THEN} & \quad \text{totalorder} := \text{totalorder} \cup (\text{tdeliver}(\{\text{sequencer}\}) \times \{mm\}) \\
& \quad \text{tdeliver} := \text{tdeliver} \cup \{pp \mapsto mm\} \\
& \quad \text{seqno} := \text{seqno} \cup \{mm \mapsto \text{counter}\} \\
& \quad \text{counter} := \text{counter} + 1
\end{align*}
\]

END;

TODeliver \((pp \in \text{PROCESS}, mm \in \text{MESSAGE})\) \(\equiv\)

\[
\begin{align*}
\text{WHEN} & \quad pp \neq \text{sequencer} \\
& \quad mm \in \text{dom}(sender) \\
& \quad mm \in \text{ran}(\text{tdeliver}) \\
& \quad pp \mapsto mm \notin \text{tdeliver} \\
& \quad \forall m. (m \in \text{computation} \land (\text{seqno}(m) < \text{seqno}(mm)) \Rightarrow (pp \mapsto m) \in \text{tdeliver}) \\
\text{THEN} & \quad \text{tdeliver} := \text{tdeliver} \cup \{pp \mapsto mm\}
\end{align*}
\]

END

D.5 Fourth Refinement

REFINEMENT TO55 REFINES TO44

VARIABLES sender, totalorder, tdeliver, computation, seqno, counter messcontrol, control

INVARIANT

\[
\begin{align*}
& \text{control} \subseteq \text{MESSAGE} \\
& \text{messcontrol} \subseteq \text{control} \Rightarrow \text{computation} \\
& \text{ran}(\text{messcontrol}) \subseteq \text{ran}(\text{tdeliver}) \\
& \text{ran}(\text{messcontrol}) \subseteq \text{computation}
\end{align*}
\]
INITIALISATION

\[
\begin{align*}
\text{sender} &:= \emptyset \quad & \text{totalorder} := \emptyset \\
\text{tdeliver} &:= \emptyset \quad & \text{computation} := \emptyset \\
\text{seqno} &:= \emptyset \quad & \text{counter} := 0 \\
\text{messcontrol} &:= \emptyset \quad & \text{control} := \emptyset
\end{align*}
\]

EVENTS

**Broadcast** \((pp \in \text{PROCESS}, mm \in \text{MESSAGE}) \equiv\)

\[
\begin{align*}
\text{WHEN} & \quad mm \notin \text{dom(sender)} \\
\text{THEN} & \quad \text{sender} := \text{sender} \cup \{ mm \mapsto pp \} \\
& \quad \text{computation} := \text{computation} \cup \{ mm \}
\end{align*}
\]

**Order** \((pp \in \text{PROCESS}, mm \in \text{MESSAGE}, mc \in \text{MESSAGE}) \equiv\)

\[
\begin{align*}
\text{WHEN} & \quad pp = \text{sequencer} \\
& \quad mm \in \text{computation} \\
& \quad (\text{sequencer} \mapsto mm) \notin \text{tdeliver} \\
& \quad mm \in \text{dom(messcontrol)} \\
& \quad mm \notin \text{ran(messcontrol)} \\
\text{THEN} & \quad \text{totalorder} := \text{totalorder} \cup (\text{tdeliver}[\{\text{sequencer}\}] \times \{mm\}) \\
& \quad \text{tdeliver} := \text{tdeliver} \cup \{ pp \mapsto mm \} \\
& \quad \text{control} := \text{control} \cup \{ mc \} \\
& \quad \text{messcontrol} := \text{messcontrol} \cup \{ mc \mapsto mm \} \\
& \quad \text{counter} := \text{counter} + 1
\end{align*}
\]

**TODeliver** \((pp \in \text{PROCESS}, mm \in \text{MESSAGE}) \equiv\)

\[
\begin{align*}
\text{WHEN} & \quad pp \neq \text{sequencer} \\
& \quad mm \in \text{computation} \\
& \quad pp \mapsto mm \notin \text{tdeliver} \\
& \quad \forall m. ( m \in \text{computation} \land (\text{seqno}(m) < \text{seqno}(mm)) \Rightarrow (pp \mapsto m) \in \text{tdeliver}) \\
\text{THEN} & \quad \text{tdeliver} := \text{tdeliver} \cup \{ pp \mapsto mm \}
\end{align*}
\]

D.6  Fifth Refinement

**REFINEMENT**  TO66  **REFINES**  TO55

**VARIABLES**  \(\text{sender}, \text{totalorder}, \text{tdeliver}, \text{computation}, \text{seqno}, \text{counter}, \text{messcontrol}, \text{control}, \text{receive}\)
Appendix D Total Order Broadcast

INVARIANT 

\[ \forall (m) . (m \in MESSAGE \land m \in \text{computation} \land \text{messcontrol}^t(m) \in \text{receive} \Rightarrow m \in \text{ran(messcontrol)}) \]

INITIALISATION

\[
\begin{align*}
\text{sender} & := \emptyset & \| \text{totalorder} & := \emptyset \\
\| \text{tdeliver} & := \emptyset & \| \text{computation} & := \emptyset \\
\| \text{seqno} & := \emptyset & \| \text{counter} & := 0 \\
\| \text{messcontrol} & := \emptyset & \| \text{control} & := \emptyset \\
\| \text{receive} & := \emptyset
\end{align*}
\]

EVENTS

Broadcast \((pp \in PROCESS, mm \in MESSAGE) \equiv \)

WHEN \(mm \in \text{dom(sender)}\)

THEN \(\text{sender} := \text{sender} \cup \{mm \mapsto pp\}\)

\| \text{computation} := \text{computation} \cup \{mm\}

END;

Order \((pp \in PROCESS, mm \in MESSAGE, mc \in MESSAGE) \equiv \)

WHEN \(pp = \text{sequencer}\)

\& \(mm \in \text{dom(sender)}\)

\& \(mm \in \text{computation}\)

\& \(mm \notin \text{ran(messcontrol)}\)

\& \(mc \notin \text{dom(messcontrol)}\)

\& \((\text{sequencer} \mapsto mm) \notin \text{tdeliver}\)

THEN \(\text{totalorder} := \text{totalorder} \cup \{\text{tdeliver}\{pp\} \times \{mm\}\}\)

\| \text{tdeliver} := \text{tdeliver} \cup \{pp \mapsto mm\}

\| \text{messcontrol} := \text{messcontrol} \cup \{mc \mapsto mm\}

\| \text{control} := \text{control} \cup \{mc\}

\| \text{seqno} := \text{seqno} \cup \{mm \mapsto \text{counter}\}

\| \text{counter} := \text{counter} + 1

END;

ReceiveControl \((pp \in PROCESS, mc \in MESSAGE) \equiv \)

WHEN \(mc \in \text{control}\)

\& \((pp \mapsto mc) \notin \text{receive}\)

THEN \(\text{receive} := \text{receive} \cup \{pp \mapsto mc\}\)

END

TODeliver \((pp \in PROCESS, mm \in MESSAGE) \equiv \)

WHEN \(pp \neq \text{sequencer}\)

\& \(mm \in \text{computation}\)

\& \((pp \mapsto mm) \notin \text{tdeliver}\)

\& \((pp \mapsto \text{messcontrol}^t(mm)) \in \text{receive}\)

\& \(\forall m.(m \in \text{computation} \land (\text{seqno}(m) < \text{seqno}(mm)) \Rightarrow (pp \mapsto m) \in \text{tdeliver})\)

THEN \(\text{tdeliver} := \text{tdeliver} \cup \{pp \mapsto mm\}\)

END
Appendix E

Total Causal Order Broadcast

E.1 Abstract Model

MACHINE tco11
CONSTANTS sequencer
PROPERTIES sequencer ∈ PROCESS
SETS PROCESS, MESSAGE;
VARIABLES sender, cdeliver, tdeliver, computation, control, messcontrol, causalorder, totalorder, cdelorder, tdelorder

INVARIANT sender ∈ MESSAGE → PROCESS
∧ cdeliver ∈ PROCESS ↔ MESSAGE  ∧ tdeliver ∈ PROCESS ↔ MESSAGE
∧ computation ∈ MESSAGE  ∧ control ∈ MESSAGE
∧ messcontrol ∈ control ⊆ computation  ∧ causalorder ∈ MESSAGE ↔ MESSAGE
∧ totalorder ∈ MESSAGE ↔ MESSAGE
∧ cdelorder ∈ PROCESS→ (MESSAGE ↔ MESSAGE)
∧ tdelorder ∈ PROCESS→ (MESSAGE ↔ MESSAGE)
∧ dom(causalorder) ⊆ dom(sender)
∧ ran(causalorder) ⊆ dom(sender)
∧ dom(messcontrol) ⊆ dom(sender)
∧ ran(messcontrol) ⊆ dom(sender)

∧ ∀ (m1,m2). ( m1 ∈ MESSAGE ∧ m2 ∈ MESSAGE ∧ m1 ∈ ran(messcontrol)
∧ m2 ∈ ran(messcontrol) ∧ (m1 → m2) ∈ causalorder
⇒ (m1 → m2) ∈ totalorder

∧ ∀ (m,p). ( m ∈ MESSAGE ∧ p ∈ PROCESS ∧ (p → m) ∈ tdeliver
⇒ (p → m) ∈ cdeliver

∧ ∀ (m1,m2). ( m1 ∈ MESSAGE ∧ m2 ∈ MESSAGE ∧ m1 ∈ computation
∧ m2 ∈ computation ∧ (m1 → m2) ∈ causalorder
∧ m2 ∈ ran(messcontrol)
⇒ m1 ∈ ran(messcontrol)
∀ (m) . ( m ∈ MESSAGE  ∧ m ∈ ran(messcontrol) )
⇒ (sequencer ⇒ m) ∈ cdeliver

∀ (m1,m2,p) . ( m1 ∈ MESSAGE  ∧ m2 ∈ MESSAGE  ∧ p ∈ PROCESS
∧ (m1 → m2) ∈ causalorder  ∧ (p → m2) ∈ cdeliver
⇒ (m1 → m2) ∈ cdeloder(p)

∀ (m1,m2) . ( m1 ∈ MESSAGE  ∧ m2 ∈ MESSAGE
∧ (m1 → m2) ∈ tdelorder(p)
⇒ (m1 → m2) ∈ totalorder

∀ (m1,m2,m3) . ( m1 ∈ MESSAGE  ∧ m2 ∈ MESSAGE  ∧ m3 ∈ MESSAGE
∧ (m1 → m2) ∈ causalorder  ∧ (m2 → m3) ∈ causalorder
⇒ (m1 → m3) ∈ causalorder

∀ (m1,m2,p) . ( m1 ∈ MESSAGE  ∧ m2 ∈ MESSAGE  ∧ p ∈ PROCESS
∧ (m1 → m2) ∈ totalorder  ∧ (p → m2) ∈ cdeliver
⇒ (p → m1) ∈ cdeliver

∀ (m1,m2,m3) . ( m1 ∈ MESSAGE  ∧ m2 ∈ MESSAGE  ∧ m3 ∈ MESSAGE
∧ (m1 → m2) ∈ totalorder  ∧ (m2 → m3) ∈ totalorder
⇒ (m1 → m3) ∈ totalorder

∀ (m1,m2,p) . ( m1 ∈ MESSAGE  ∧ m2 ∈ MESSAGE  ∧ p ∈ PROCESS
∧ (m1 → m2) ∈ totalorder  ∧ (p → m2) ∈ tdeliver
⇒ (p → m1) ∈ tdeliver

∀ (m1,m2,p) . ( m1 ∈ MESSAGE  ∧ m2 ∈ MESSAGE  ∧ p ∈ PROCESS
∧ (p → m1) ∈ tdeliver  ∧ (p → m2) ∈ tdeliver
∧ m2 ∈ ran(tdeliver)
⇒ (m1 → m2) ∈ totalorder )

∀ (m1,m2) . ( m1 ∈ MESSAGE  ∧ m2 ∈ MESSAGE
∧ m1 ∈ ran(tdeliver)  ∧ m2 ∈ ran(tdeliver)
∧ (m2 → m1) ∈ totalorder
⇒ (m1 → m2) ∈ totalorder )

∀ (m1,m2,p) . ( m1 ∈ MESSAGE  ∧ m2 ∈ MESSAGE  ∧ p ∈ PROCESS
∧ (p → m1) ∈ tdeliver  ∧ (p → m2) ∈ tdeliver
∧ (m2 → m1) ∈ totalorder
⇒ (m1 → m2) ∈ totalorder )

∀ (m1,m2,p1,p2) . ( m1 ∈ MESSAGE  ∧ m2 ∈ MESSAGE
∧ p1 ∈ PROCESS  ∧ p2 ∈ PROCESS
∧ (p1 → m1) ∈ tdeliver  ∧ (p1 → m2) ∈ tdeliver
∧ (p2 → m1) ∈ tdeliver  ∧ (p2 → m2) ∈ tdeliver
⇒ (m1 → m2) ∈ totalorder )

∀ (m) . ( m ∈ MESSAGE  ⇒ m → m ∈ totalorder )

∀ (m) . ( m ∈ MESSAGE  ∧ m ∈ ( dom (totalorder) ∪ ran(totalorder) )
⇒ m ∈ ran(tdeliver) )

∀ (m1,m2,p) . ( m1 ∈ MESSAGE  ∧ m2 ∈ MESSAGE  ∧ p ∈ PROCESS
∧ (m1 → m2) ∈ causalorder  ∧ (p → m2) ∈ cdeliver
⇒ (m1 → m2) ∈ cdeloder(p)
Appendix E Total Causal Order Broadcast

\( \forall (m_1, m_2, p) . (m_1 \in \text{MESSAGE} \land m_2 \in \text{MESSAGE} \land p \in \text{PROCESS} \land (m_1 \rightarrow m_2) \in \text{causalorder} \land m_2 \in \text{sender}^{-1}[[p]]) \Rightarrow (m_1 \in \text{sender}^{-1}[[p]] \lor m_1 \in \text{cdeliver}[[p]]) \)

\( \forall (m_1, m_2, p) . (m_1 \in \text{MESSAGE} \land m_2 \in \text{MESSAGE} \land p \in \text{PROCESS} \land (m_1 \rightarrow m_2) \in \text{causalorder} \land m_2 \in \text{sender}^{-1}[[p]]) \Rightarrow m_1 \in \text{sender}^{-1}[[p]] \lor m_2 \in \text{cdeliver}[[p]]) \)

\( \forall (m_1, m_2, p) . (m_1 \in \text{MESSAGE} \land m_2 \in \text{MESSAGE} \land p \in \text{PROCESS} \land (m_1 \rightarrow m_2) \in \text{causalorder} \land m_2 \in \text{sender}^{-1}[[p]]) \Rightarrow m_1 \in \text{sender}^{-1}[[p]] \lor m_2 \in \text{cdeliver}[[p]]) \)

**INITIALISATION**

sender := \O \parallel cdeliver := \O \parallel ideliver := \O \parallel
computation := \O \parallel control := \O \parallel messcontrol := \O \parallel
causalorder := \O \parallel totalorder := \O \parallel
cdelorder := \text{PROCESS} \times \{\O\} \parallel
tdelorder := \text{PROCESS} \times \{\O\}

**Broadcast** \((pp \in \text{PROCESS}, mm \in \text{MESSAGE}) \equiv \)

WHEN mm \not\in \text{dom}(sender)
THEN
sender := sender \cup\{mm \rightarrow pp\}
\parallel cdeliver := cdeliver \cup\{pp \rightarrow mm\}
\parallel cdelorder(pp):=\text{cdelorder}(pp) \cup\{\text{cdeliver}[[pp]] \times \{mm\}\}
\parallel computation := computation \cup\{mm\}
\parallel causalorder := causalorder \cup\{(\text{sender}^{-1}[[pp]] \times \{mm\}\}
\cup\{\text{cdeliver}[[pp]] \times \{mm\}\})

END;

**CausalDeliver** \((pp \in \text{PROCESS}, mm \in \text{MESSAGE}) \equiv \)

WHEN mm \not\in \text{dom}(sender)
\land (pp \rightarrow mm) \not\in \text{cdeliver}
\land \forall m . (m \in \text{MESSAGE} \land (m \rightarrow mm) \in \text{causalorder} \Rightarrow (pp \rightarrow m) \in \text{cdeliver})
THEN
\text{cdeliver} := \text{cdeliver} \cup\{pp \rightarrow mm\}
\parallel \text{cdelorder}(pp):=\text{cdelorder}(pp) \cup\{\text{cdeliver}[[pp]] \times \{mm\}\}

END;

**SendControl** \((pp \in \text{PROCESS}, mm \in \text{MESSAGE}, mc \in \text{MESSAGE}) \equiv \)

WHEN pp = sequencer
\land mc \not\in \text{dom}(sender)
\land mm \in \text{ran}(messcontrol)
\land mm \in \text{computation}
\land (pp \rightarrow mm) \in \text{cdeliver}
\land \forall m . (m \in \text{MESSAGE} \land m \in \text{computation} \land (m \rightarrow mm) \in \text{causalorder} \Rightarrow m \in \text{ran}(messcontrol))
THEN
\text{causalorder} := \text{causalorder} \cup\{(\text{sender}^{-1}[[\text{sequencer}]] \times \{mc\}\}
\cup\{\text{cdeliver}[[\text{sequencer}]] \times \{mc\}\)
\parallel \text{sender} := \text{sender} \cup\{mc \rightarrow \text{sequencer}\}
\parallel \text{control} := \text{control} \cup\{mc\}
\parallel \text{messcontrol} := \text{messcontrol} \cup\{mc \rightarrow mm\}
\parallel \text{LET} m \BE m = \text{ran}(messcontrol)
\text{IN} \text{totalorder} := \text{totalorder} \cup (m \times \{mm\}) \text{ END
TODeliver \((pp \in \text{PROCESS}, mc \in \text{MESSAGE})\) \(\equiv\)

\[\text{WHEN} \quad \begin{align*}
& mc \in \text{dom}(sender) \\
& \land mc \in \text{control} \\
& \land (pp \mapsto mc) \in \text{cdeliver} \\
& \land (pp \mapsto \text{messcontrol}(mc)) \in \text{cdeliver} \\
& \land (pp \mapsto \text{messcontrol}(mc)) \in \text{tdeliver} \\
& \forall m. \ (m \in \text{MESSAGE} \land m \in \text{computation}) \\
& \land (m \mapsto \text{messcontrol}(mc) \in \text{totalorder}) \Rightarrow (pp \mapsto m) \in \text{tdeliver} \\
\end{align*}\]

\[\text{THEN} \quad \begin{align*}
& \text{tdeliver} := \text{tdeliver} \cup \{pp \mapsto \text{messcontrol}(mc)\} \\
& \text{idelorder}(pp) := \text{idelorder}(pp) \cup (\text{ideliver}(pp) \times \text{messcontrol}(mc)) \\
\end{align*}\]

\[\text{END}\]

### E.2 First Refinement

REFINEMENT \(tco22\)

REFINES \(tco11\)

**VARIABLES**

\(\text{sender, cdeliver, tdeliver, computation, control, messcontrol, VTP, VTM, seqno, counter}\)

**INVARIANT**

\(\begin{align*}
& VTP \in \text{PROCESS} \rightarrow (\text{PROCESS} \rightarrow N) \\
& \land VTM \in \text{MESSAGE} \rightarrow (\text{PROCESS} \rightarrow N) \\
& \land \text{seqno} \in \text{computation} \rightarrow N \\
& \land \text{counter} \in N \\
& \land \forall (m,p1,p2). \ (m \in \text{MESSAGE} \land p1 \in \text{PROCESS} \land p2 \in \text{PROCESS} \\
& \land m \in \text{dom}(sender) \land VTP(p1)(p2) \geq VTM(m)(p2) \\
& \Rightarrow (p1 \mapsto m) \in \text{cdeliver} \\
& \land \forall (m1,m2,p). \ (m1 \in \text{MESSAGE} \land m2 \in \text{MESSAGE} \land p \in \text{PROCESS} \\
& \land (m1 \mapsto m2) \in \text{causalorder} \\
& \Rightarrow VTM(m1)(p) \leq VTM(m2)(p) \\
& \land \forall (m,p). \ (m \in \text{MESSAGE} \land p \in \text{PROCESS} \\
& \land \text{dom}(sender) \Rightarrow VTM(m)(p) \leq VTP(p)(p) \\
& \land \forall (m,p). \ (m \in \text{MESSAGE} \land p \in \text{PROCESS} \\
& \land VTM(m)(p) = 0 \Rightarrow m \in (\text{dom(causalorder)} \cup \text{ran(causalorder)}) \\
& \land \forall (p1,p2). \ (p1 \in \text{PROCESS} \land p2 \in \text{PROCESS} \\
& \land p1 \neq p2 \Rightarrow VTP(p1)(p2) \leq VTP(p2)(p2) \\
& \land \forall (m1,m2,p). \ (m1 \in \text{MESSAGE} \land m2 \in \text{MESSAGE} \land p \in \text{PROCESS} \\
& \land VTM(m1)(p) \leq VTM(m2)(p) \\
& \Rightarrow (m1 \mapsto m2) \in (\text{sender}[[\text{sender}(m1)] \times \{m2\}) \\
& \lor (\text{cdeliver}[[\text{sender}(m2)] \times \{m2\}) \\
& \land \forall (m1,m2). \ (m1 \in \text{MESSAGE} \land m2 \in \text{MESSAGE} \\
& \land m1 \mapsto m2 \in \text{totalorder} \\
& \Rightarrow \text{seqno}(m1) < \text{seqno}(m2) \\
& \land \forall (m1,m2). \ (m1 \in \text{MESSAGE} \land m2 \in \text{MESSAGE} \\
& \land m \in \text{computation} \land m \in \text{dom(seqno)} \\
& \Rightarrow \text{sequencer} \mapsto m \in \text{tdeliver} \\
\end{align*}\)

**INITIALISATION**

\(\begin{align*}
\text{sender} := \emptyset & \quad |\quad \text{cdeliver} := \emptyset & \quad |\quad \text{tdeliver} := \emptyset \\
\text{computation} := \emptyset & \quad |\quad \text{control} := \emptyset & \quad |\quad \text{messcontrol} := \emptyset \\
\text{VTP} := \emptyset & \quad |\quad \text{VTM} := \emptyset & \quad |\quad \text{seqno} := \emptyset & \quad |\quad \text{counter} := 0 \\
\end{align*}\)
Broadcast \((pp \in\text{PROCESS}, mm \in\text{MESSAGE}) \equiv\)

WHEN \(mm \not\in \text{dom}(sender)\)
THEN

\[
\begin{align*}
\text{LET} & \quad nVTP \\
\text{BE} & \quad nVTP = VTP(pp) \triangleq \{ pp \mapsto \text{VTP}(pp) + 1 \} \\
\text{IN} & \quad \text{VTM}(mm) := nVTP
\end{align*}
\]

\[
\begin{cases}
\text{sender} := \text{sender} \cup \{ mm \mapsto pp \} \\
\text{cdeliver} := \text{cdeliver} \cup \{ pp \mapsto mm \} \\
\text{computation} := \text{computation} \cup \{ mm \}
\end{cases}
\]

END;

CausalDeliver \((pp \in\text{PROCESS}, mm \in\text{MESSAGE}) \equiv\)

WHEN \(mm \in \text{dom}(sender)\)
\& \((pp \mapsto mm) \not\in \text{cdeliver}\)
\& \(!\exists p (p \in \text{PROCESS} \land p \not\in \text{sender}(mm) \Rightarrow \text{VTP}(pp)(p) \geq \text{VTM}(mm)(p))\)
\& \(\text{VTP}(pp)(\text{sender}(mm)) = \text{VTM}(mm)(\text{sender}(mm)) - 1\)
THEN

\[
\begin{align*}
\text{cdeliver} := \text{cdeliver} \cup \{ pp \mapsto mm \} \\
\text{VTP}(pp) := \text{VTP}(pp) \triangleq \\
\quad (\{ q \mid q \in \text{PROCESS} \land \text{VTP}(pp)(q) < \text{VTM}(mm)(q) \} \triangleq \text{VTM}(mm))
\end{align*}
\]

END;

SendControl \((pp \in\text{PROCESS}, mm \in\text{MESSAGE}, mc \in\text{MESSAGE}) \equiv\)

WHEN \(pp = \text{sequencer}\)
\& \(mc \not\in \text{dom}(sender)\)
\& \(mm \not\in \text{ran}(\text{messcontrol})\)
\& \(mm \in \text{computation}\)
\& \((pp \mapsto mm) \not\in \text{cdeliver}\)
\& \(!\forall (m, p) \cdot (p \in \text{PROCESS} \land m \in \text{MESSAGE} \land m \in \text{computation}
\quad \land \text{VTM}(m)(p) \leq \text{VTM}(mm)(p) \Rightarrow m \in \text{ran}(\text{messcontrol}))\)
THEN

\[
\begin{align*}
\text{control} := \text{control} \cup \{ mc \} \\
\text{messcontrol} := \text{messcontrol} \cup \{ mc \mapsto mm \} \\
\text{LET} & \quad nVTP \text{ BE } nVTP = VTP(pp) \triangleq \{ pp \mapsto VTP(pp) + 1 \} \\
\text{IN} & \quad \text{VTM}(mc) := nVTP \\
\text{VTP}(pp) := nVTP
\end{align*}
\]

END

\[
\begin{cases}
\text{sender} := \text{sender} \cup \{ mc \mapsto pp \} \\
\text{LET} & \quad ncount \text{ BE } ncount = \text{counter} + 1 \\
\text{IN} & \quad \text{counter} := ncount \\
\text{seqno}(mm) := ncount
\end{cases}
\]

END;

TODeliver \((pp \in\text{PROCESS}, mc \in\text{MESSAGE}) \equiv\)

WHEN \(mc \in \text{dom}(sender)\)
\& \(mc \in \text{control}\)
\& \((pp \mapsto mc) \not\in \text{cdeliver}\)
\& \((pp \mapsto \text{messcontrol}(mc)) \not\in \text{cdeliver}\)
\& \((pp \mapsto \text{messcontrol}(mc)) \not\in \text{tdeliver}\)
\& !\forall (m \in \text{MESSAGE} \land m \in \text{computation}
\quad \land (\text{seqno}(m) < \text{seqno}(\text{messcontrol}(mc)) \Rightarrow (pp \mapsto m) \in \text{tdeliver}))
THEN

\[
\begin{align*}
\text{tdeliver} := \text{tdeliver} \cup \{ pp \mapsto \text{messcontrol}(mc) \}
\end{align*}
\]

END
Appendix E Total Causal Order Broadcast

E.3 Second Refinement

REFINEMENT \( tco33 \) REFINES \( tco22 \)

VARIABLES \( sender, cdeliver, tdeliver, computation, control, messcontrol, VTP, VTM \)

INVARIANT
\[
\forall (m) \cdot (m \in \text{MESSAGE} \land m \in \text{control} \\
\land (m \mapsto \text{sequencer}) \in \text{sender} \\
\Rightarrow \text{seqno}(\text{messcontrol}^{-1}(m)) = VTM(m)(\text{sequencer})) \\
\land \forall (m1, m2, p) \cdot (m1 \in \text{MESSAGE} \land m2 \in \text{MESSAGE} \land p \in \text{PROCESS} \\
\land m1 \in \text{control} \land m2 \in \text{control} \\
\land VTM(m1)(p) \leq VTM(m2)(p) \\
\Rightarrow \text{seqno}(\text{messcontrol}^{-1}(m1)) \leq \text{seqno}(\text{messcontrol}^{-1}(m2)) \\
\land \forall (m1, m2, p) \cdot (m1 \in \text{MESSAGE} \land m2 \in \text{MESSAGE} \land p \in \text{PROCESS} \\
\land m1 \in \text{computation} \land m2 \in \text{computation} \\
\land \text{seqno}(m1) \leq \text{seqno}(m2) \\
\Rightarrow VTM(\text{messcontrol}(m1))(p) \leq VTM(\text{messcontrol}(m2))(p)
\]

INITIALISATION
\[
\text{sender} := \emptyset \mid cdeliver := \emptyset \mid \text{tdeliver} := \emptyset \mid \text{computation} := \emptyset \mid \text{control} := \emptyset \mid \text{messcontrol} := \emptyset \mid VTP := \emptyset \mid VTM := \emptyset
\]

\text{Broadcast}(pp \in \text{PROCESS}, mm \in \text{MESSAGE}) \equiv

\text{WHEN } mm \not\in \text{dom}(\text{sender})
\text{THEN } \text{LET } nVTP
\text{BE } nVTP = VTP(pp) \not\in \{ pp \mapsto VTP(pp)(pp)+1 \}
\text{IN } VTM(mm) := nVTP \\
\text{\parallel VTP(pp) := nVTP END}
\text{\parallel sender := sender \cup \{ mm \mapsto pp \}}
\text{\parallel cdeliver := cdeliver \cup \{ pp \mapsto mm \}}
\text{\parallel computation := computation \cup \{ mm \}}
\text{END ;}

\text{CausalDeliver}(pp \in \text{PROCESS}, mm \in \text{MESSAGE}) \equiv

\text{WHEN } mm \in \text{dom}(\text{sender})
\text{\land (pp \mapsto mm) \in \text{cdeliver}}
\text{\land } \forall p, (p \in \text{PROCESS} \land p \neq \text{sender}(mm) \Rightarrow VTP(pp)(p) \geq VTM(mm)(p))
\text{\land VTP(pp)(\text{sender}(mm)) = VTM(mm)(\text{sender}(mm))-1}
\text{THEN}
\text{\parallel cdeliver := cdeliver \cup \{ pp \mapsto mm \}}
\text{\parallel VTP(pp) := VTP(pp) \not\in \{ (q \mid q \in \text{PROCESS} \land VTP(pp)(q) < VTM(mm)(q) \} \cup VTM(mm))}
\text{END ;}
Appendix E Total Causal Order Broadcast

\[
\text{SendControl} \left( pp \in \text{PROCESS} \land mm \in \text{MESSAGE} \land mc \in \text{MESSAGE} \right) \equiv \\
\text{WHEN} \quad pp = \text{sequencer} \\
\wedge \quad mc \in \text{dom}(\text{sender}) \\
\wedge \quad mm \in \text{ran}(\text{messcontrol}) \\
\wedge \quad mm \in \text{computation} \\
\wedge \quad pp \mapsto mm \in cdeliver \\
\wedge \quad \forall (m, p) \cdot (p \in \text{PROCESS} \land m \in \text{MESSAGE} \land m \in \text{computation} \\
\wedge \quad VTM(m)(p) \leq VTM(mm)(p) \Rightarrow m \in \text{ran}(\text{messcontrol})) \\
\text{THEN} \quad \text{control} := \text{control} \cup \{mc\} \\
\quad \text{messcontrol} := \text{messcontrol} \cup \{mc \mapsto mm\} \\
\quad \text{LET} \quad nVTP \quad \text{BE} \quad nVTP = VTP(pp) \triangleq \{pp \mapsto VTP(pp)(pp) + 1\} \\
\quad \text{IN} \quad VTM(mc) := nVTP \quad \text{||} \quad VTP(pp) := nVTP \quad \text{END} \\
\quad \text{sender} := \text{sender} \cup \{mc \mapsto pp\} \\
\text{END;} \\
\]

\[
\text{TODeliver} \left( pp \in \text{PROCESS} \land mc \in \text{MESSAGE} \right) \equiv \\
\text{WHEN} \quad mc \in \text{dom}(\text{sender}) \\
\wedge \quad (pp \mapsto mc) \in cdeliver \\
\wedge \quad (pp \mapsto \text{messcontrol}(mc)) \in cdeliver \\
\wedge \quad (pp \mapsto \text{messcontrol}(mc)) \in tdeliver \\
\wedge \quad \forall m, p \in \text{MESSAGE} \land m \in \text{computation} \\
\wedge \quad (VTM(\text{messcontrol}^1(m))(\text{sequencer}) < VTM(mc)(\text{sequencer})) \\
\quad \Rightarrow (pp \mapsto m) \in tdeliver \\
\text{THEN} \quad tdeliver := tdeliver \cup \{pp \mapsto \text{messcontrol}(mc)\} \\
\text{END} \\
\]

\textbf{E.4 Third Refinement}

\text{REFINEMENT} \quad \text{tco44} \\
\text{REFINES} \quad \text{tco33} \\
\text{VARIABLES} \quad \text{sender}, \text{cdeliver}, \text{tdeliver}, \text{computation}, \text{control}, \text{messcontrol}, \text{VTP}, \text{VTM} \\
\text{INITIALISATION} \\
\quad \text{sender} := \emptyset \quad \text{||} \quad \text{cdeliver} := \emptyset \quad \text{||} \quad \text{tdeliver} := \emptyset \quad \text{||} \\
\quad \text{computation} := \emptyset \quad \text{||} \quad \text{control} := \emptyset \quad \text{||} \quad \text{messcontrol} := \emptyset \quad \text{||} \\
\quad VTP := \emptyset \quad \text{||} \quad VTM := \emptyset \\
\]

\[
\text{Broadcast}(pp \in \text{PROCESS} \land mm \in \text{MESSAGE}) \equiv \\
\text{WHEN} \quad mm \notin \text{dom}(\text{sender}) \\
\text{THEN} \quad \text{LET} \quad nVTP \\
\quad \text{BE} \quad nVTP = VTP(pp) \triangleq \{pp \mapsto VTP(pp)(pp) + 1\} \\
\quad \text{IN} \quad VTM(mm) := nVTP \\
\quad \text{||} \quad VTP(pp) := nVTP \quad \text{END} \\
\quad \text{||} \quad \text{sender} := \text{sender} \cup \{mm \mapsto pp\} \\
\quad \text{||} \quad \text{cdeliver} := \text{cdeliver} \cup \{pp \mapsto mm\} \\
\quad \text{||} \quad \text{computation} := \text{computation} \cup \{mm\} \\
\text{END;} \\
\]

\[
\text{CausalDeliver} \left( pp \in \text{PROCESS} \land mm \in \text{MESSAGE} \right) \equiv \\
\text{WHEN} \quad mm \in \text{dom}(\text{sender}) \\
\wedge \quad (pp \mapsto mm) \in \text{cdeliver} \\
\wedge \quad \forall p, p \in \text{PROCESS} \land p \neq \text{sender}(mm) \Rightarrow VTP(pp)(p) \geq VTM(mm)(p) \\
\wedge \quad VTP(pp)(\text{sender}(mm)) = VTM(mm)(\text{sender}(mm)) - 1 \\
\text{THEN} \quad \text{cdeliver} := \text{cdeliver} \cup \{pp \mapsto mm\} \\
\quad \text{||} \quad VTP(pp) := VTP(pp) \triangleq \{ \text{sender}(mm) \mapsto VTM(mm)(\text{sender}(mm)) \} \\
\text{END;} \\
\]
Appendix E Total Causal Order Broadcast

SendControl \((pp \in \text{PROCESS}, \ mm \in \text{MESSAGE}, \ mc \in \text{MESSAGE})\) \(\equiv\)

\[
\text{WHEN } pp = \text{sequencer} \quad \land \quad mc \notin \text{dom(sender)} \\
\land \quad mm \notin \text{ran}(\text{messcontrol}) \\
\land \quad mm \in \text{computation} \\
\land \quad pp \mapsto mm \in \text{cdeliver} \\
\land \quad \forall (m,p) \ (p \in \text{PROCESS} \land m \in \text{MESSAGE} \land m \in \text{computation} \\
\land \quad \text{VTM}(m)(p) \leq \text{VTM}(mm)(p) \implies m \in \text{ran}(\text{messcontrol}))
\]

\[
\text{THEN} \quad \text{control} := \text{control} \cup \{mc\} \\
\land \quad \text{messcontrol} := \text{messcontrol} \cup \{mc \mapsto mm\} \\
\land \quad \text{LET} \quad nVTP \ BE \ nVTP = \text{VTP}(pp) \triangleq \{ pp \mapsto \text{VTP}(pp)(pp)+1 \} \\
\land \quad \text{IN} \quad \text{VTP}(mc) := \text{nVTP} \quad \land \quad \text{VTP}(pp) := \text{nVTP} \quad \text{END} \\
\land \quad \text{sender} := \text{sender} \cup \{mc \mapsto pp\}
\]

\[
\text{END;}
\]

TODeliver \((pp \in \text{PROCESS}, \ mc \in \text{MESSAGE})\) \(\equiv\)

\[
\text{WHEN} \ mc \in \text{dom(sender)} \\
\land \quad mc \in \text{control} \\
\land \quad (pp \mapsto mc) \in \text{cdeliver} \\
\land \quad (pp \mapsto \text{messcontrol}(mc)) \in \text{cdeliver} \\
\land \quad (pp \mapsto \text{messcontrol}(mc)) \in \text{tdeliver} \\
\land \quad \forall m. (m \in \text{MESSAGE} \land m \in \text{computation} \\
\land \quad (\text{VTM}(\text{messcontrol}^{-1}(m))(\text{sequencer}) < \text{VTM}(mc)(\text{sequencer}) ) \\
\land \quad \implies (pp \mapsto m) \in \text{tdeliver})
\]

\[
\text{THEN} \quad \text{tdeliver} := \text{tdeliver} \cup \{pp \mapsto \text{messcontrol}(mc)\} \\
\text{END}
\]

E.4 Fourth Refinement

REFINEMENT \(\text{tco}55\) \(\text{REFINES} \quad \text{tco}44\)

VARIABLES \(\text{sender, cdeliver, tdeliver, computation, control, messcontrol, VTP, VTM}\)

INITIALISATION

\[
\text{sender} := \emptyset \quad \land \quad \text{cdeliver} := \emptyset \quad \land \quad \text{tdeliver} := \emptyset \\
\land \quad \text{computation} := \emptyset \quad \land \quad \text{control} := \emptyset \quad \land \quad \text{messcontrol} := \emptyset \\
\land \quad \text{VTP} := \emptyset \quad \land \quad \text{VTM} := \emptyset
\]

Broadcast \((pp \in \text{PROCESS}, \ mm \in \text{MESSAGE})\) \(\equiv\)

\[
\text{WHEN} \ mm \notin \text{dom(sender)} \\
\text{THEN} \quad \text{LET} \quad nVTP \\
\land \quad nVTP = \text{VTP}(pp) \triangleq \{ pp \mapsto \text{VTP}(pp)(pp)+1 \} \\
\land \quad \text{IN} \quad \text{VTP}(mm) := \text{nVTP} \\
\land \quad \text{sender} := \text{sender} \cup \{mm \mapsto pp\} \\
\land \quad \text{cdeliver} := \text{cdeliver} \cup \{pp \mapsto mm\} \\
\land \quad \text{computation} := \text{computation} \cup \{mm\}
\]

\[
\text{END;}
\]
CausalDeliver \((pp \in PROCESS, mm \in MESSAGE)\) \text{\(\equiv\)}

\text{WHEN} \quad mm \in \text{dom}(sender)
\quad \land \quad (pp \rightarrow mm) \in cdeliver
\quad \land \quad \forall p.\ p \in PROCESS \land p \neq \text{sender}(mm) \Rightarrow VTP(pp)(p) \geq VTM(mm)(p))
\quad \land \quad VTP(pp)(\text{sender}(mm)) = VTM(mm)(\text{sender}(mm))-1

\text{THEN}
\quad cdeliver := cdeliver \cup \{pp \rightarrow mm\}
\quad \land \quad VTP(pp)(\text{sender}(mm)) := VTM(mm)(\text{sender}(mm))

\text{END;}

SendControl \((pp \in PROCESS, mm \in MESSAGE, mc \in MESSAGE)\) \text{\(\equiv\)}

\text{WHEN} \quad pp = \text{sequencer}
\quad \land \quad mc \in \text{dom}(sender)
\quad \land \quad mm \in \text{ran}(\text{messcontrol})
\quad \land \quad mm \in \text{computation}
\quad \land \quad pp \rightarrow mm \in cdeliver
\quad \land \quad \forall (m,p) \cdot (p \in PROCESS \land m \in MESSAGE \land m \in \text{computation}
\quad \land \quad VTM(m)(p) \leq VTM(mm)(p) \Rightarrow m \in \text{ran}(\text{messcontrol}))

\text{THEN}
\quad control := control \cup \{mc\}
\quad \land \quad \text{messcontrol} := \text{messcontrol} \cup \{mc \rightarrow mm\}
\quad \land \quad \text{LET} \quad nVTP \ BE \ nVTP = VTP(pp) \cup \{pp \rightarrow VTP(pp)(pp)+1\}
\quad \land \quad \text{IN} \quad \text{VTM}(mc) := nVTP \quad \land \quad VTP(pp) := nVTP \text{ END}
\quad \land \quad \text{sender} := \text{sender} \cup \{mc \rightarrow pp\}

\text{END;}

TODeliver \((pp \in PROCESS, mc \in MESSAGE)\) \text{\(\equiv\)}

\text{WHEN} \quad mc \in \text{dom}(sender)
\quad \land \quad mc \in control
\quad \land \quad (pp \rightarrow mc) \in cdeliver
\quad \land \quad (pp \rightarrow \text{messcontrol}(mc)) \in cdeliver
\quad \land \quad (pp \rightarrow \text{messcontrol}(mc)) \in \text{ideliver}
\quad \land \quad \forall m.\ m \in MESSAGE \land m \in \text{computation}
\quad \land \quad (\ \text{VTM(}\text{messcontrol}(mc)(sequence)) < \text{VTM(}\text{mc}(sequence)) \ )
\quad \Rightarrow (pp \rightarrow m) \in \text{ideliver}

\text{THEN}
\quad \text{ideliver} := \text{ideliver} \cup \{pp \rightarrow \text{messcontrol}(mc)\}

\text{END}
Bibliography


[10] Yair Amir, Danny Dolev, Shlomo Kramer, and Dalia Malki. Membership algorithms for multicast communication groups. In Adrian Segall and Shmuel Zaks,


