Formal Development of a Total Order Broadcast for Distributed Transactions using Event-B

Divakar Yadav* and Michael Butler**

School of Electronics and Computer Science
University of Southampton
Southampton SO17 1BJ ,U.K
divakar.yadav@ietlucknow.edu, mjb@ecs.soton.ac.uk

Abstract. In a replicated database system, copies of the database are kept across several sites for fault-tolerance and availability. Data access in such systems is usually done within a transactional framework. A read-only transaction accesses data locally and an update transaction modifies the database at all sites. Total order broadcast primitives have been proposed to support transactions and allow fault-tolerant cooperation between the sites in a distributed system. In this paper, we identify and analyze the problem of formation of deadlocks among conflicting update transactions due to race conditions and outline how a system of total order broadcast prevents deadlocks and transaction failures. Later we outline how a refinement based approach with Event-B can be used for formal development of the models of total order broadcast. In this approach we begin with the abstract model of a total order broadcast and verify that the required ordering properties are preserved by the system. Subsequently, in a series of refinement steps we outline how an abstract total order can correctly be implemented by using a notion of sequence number. This technique requires us to discharge proof obligations due to consistency and refinement checking. To discharge the proof obligations we are required to discover invariants that describes the relationship between the abstract total order and the underlying mechanism.

1 Introduction

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. A distributed transaction may span several sites reading or updating data objects. It is advantageous

* Currently working at Institute of Engineering and Technology, U P Technical University, Lucknow, India. This work was supported by the Commonwealth Scholarship Commission in the United Kingdom.

** Michael Butler's contribution is part of EU projects IST project IST 511599 RODIN (Rigorous Open Development Environment for Complex Systems) and ICT 214158 DEPLOY (Industrial deployment of system engineering methods providing high dependability and productivity www.deploy-project.eu/).

to replicate the data if the transaction workload is predominantly read only. However, during updates, the complexity of keeping the replicas in a consistent state arises due to race conditions among conflicting update transactions. A typical distributed transaction contains a sequence of database operations which must be processed at all of the participating sites or none of the sites to maintain the integrity of the database. The strong consistency criterion in the replicated database requires that the database remains in a consistent state despite transaction failures. In addition to providing fault-tolerance, one of the important issues to be addressed in the design of replica control protocols is consistency. The *one copy equivalence* [9] criteria requires that a replicated database is in a mutually consistent state only if all copies of data objects *logically* have the same identical value.

No common global clock or shared memory exist in a distributed system. The sites communicate by exchange of messages which are delivered to them after arbitrary time delays. In such systems up-to-date knowledge of the system is not known to any process or site. This problem can be dealt by relying on group communication primitives that provide ordering guarantees on the delivery of messages. The group communication primitives have been proposed as a mechanism for the development of reliable fault-tolerant distributed applications [16]. A *total order broadcast* is one such primitive that guarantees the delivery of messages to the sites in the same order. Introduction of the transactions based on group communication primitives represents an important step towards extending the power of group communication in an asynchronous distributed system [34]. These primitives have been proposed for processing transactions and managing replicated databases [21, 35, 22]. In a replicated database that uses a reliable broadcast without ordering guarantees, the operations of the conflicting update transactions may arrive at different sites in different orders. This may lead to the formation of deadlock among conflicting transactions involving several sites. The blocking of the transactions at a site is usually resolved through aborting the transaction by timeouts. 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.

In this paper we present an incremental development of a model of total order broadcast using Event-B [28], which is a variant of B Method [1]. Event-B is a formal technique for the development of models of distributed systems. This technique 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. The B tools provide a 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. The technique is supported by several industrial level B tools such as, Rodin [3] and Click'n'Prove [4] that provide a significant automated proof support for generation of the proof obligations, factorizing complex proof obligations into simpler proofs and discharging them. The majority of the proof obligations are proved by the auto-

matic prover of the tools. However, some complex proof obligations require user guidance through the interactive prover. These proof obligations also help in the discovery of new system invariants. The proof obligations and the invariants help 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 essential features of the modelling and proof guidelines to obtain an high degree of automated proof for an Event-B development are outlined in [15]. We have used the Click'n'Prove [4] B tool for proof obligation generation and to discharge them.

The remainder of this paper is organized as follows: Section 2 outlines the system model, Section 3 identifies the problem of formation of deadlocks among the transactions due to unordered delivery of update messages, Section 4 describes informal specifications of a total order broadcast and mechanism for implementation, Section 5 outlines the abstract model of total order broadcast, and shows how an abstract total order is constructed on the messages. Section 6 present the invariant properties of the system. We also outline how the proof obligations generated by the B tool help us discover new invariants. Section 7 illustrates essential features of the refinement chain. Section 8 present related work on group communication and the application of formal methods to the problem. Finally, Section 9 concludes the paper.

2 Background

We have presented a rigorous design of distributed transactions for a replicated database using Event-B in [38]. Our system model consist of a sets 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* [9, 30] replica control mechanism is considered for updating replicas. In our model, update transactions are processed within the framework of a two phase commit protocol [19] to ensure global atomicity.

2.1 Transaction Model

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. The following types of transactions are considered for this model of replicated database.

- *Read-Only Transactions* : These transactions are submitted locally to the site and *commit* after reading the requested data object locally.
- *Update Transactions* : These transactions update the requested data objects. The effect of update transactions are global, thus when committed, replicas of data objects maintained at all sites must be updated. In the case of abort, none of the sites update the data object.

Let the sequence of read/write operations issued by the transaction T_i be defined by a set of objects $objectset[T_i]$ where $objectset[T_i] \neq \varnothing$. Let the set $writeset[T_i]$ represents the set of object to be *updated* such that $writeset[T_i] \subseteq objectset[T_i]$. A transaction T_i is a read-only transaction if $writeset[T_i] = \varnothing$. Similarly a transaction T_i is an update transaction if its $writeset[T_i] \neq \varnothing$.

2.2 Conflicting 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 set of object $objectset[T_i]$ and $objectset[T_j]$ respectively and $objectset[T_i] \cap objectset[T_j] \neq \varnothing$. To meet the strong consistency requirements, *conflicting* transactions need to be executed in isolation. We ensure this property by not *starting* a transaction at a site if any conflicting update transaction is *active* at that site. In our model the transactions are executed 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 $objectset[T_i]$.
- A global update transaction T_i is executed by broadcasting an update message to the participating sites. On delivery, a participating site S_j initiates a sub-transaction T_{ij} by acquiring locks on $objectset[T_i]$. If the objects are currently locked by another transaction, T_{ij} is blocked. The activity of a global update transaction at a given site is referred as sub-transaction.
- The coordinator site of T_i waits for the vote commit/abort messages from all participating site. A global commit/abort message is broadcast by the coordinator site of T_i only if it receives all local commit message from all participating sites or at-least one vote-abort message from participating sites.

The commit or abort decision of a global transaction T_i is taken at the coordinator site within the framework of a two phase commit protocol as shown in Fig. 1 as follows. A global transaction T_i *commits* if *all* T_{ij} *commit* at S_j. The global transaction T_i *aborts* if *some* T_{ij} *aborts* at S_j.

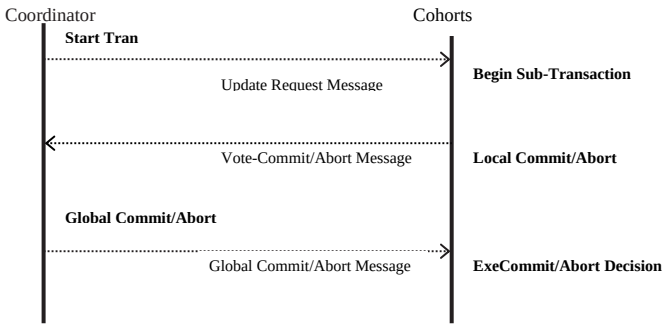

Fig. 1. Events of Update Transaction

3 Blocking and Failures of Conflicting Transactions

This section outlines how conflicting update transactions in our model can be deadlocked. A formal refinement based approach using Event-B to model and analyze distributed transaction is given in [38]. 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 atomic update of a one copy database. Through the refinement proofs, we verify that the design of the replicated database conforms to the one copy database abstraction despite transaction failures at a site. The global atomicity of update transactions is ensured by processing update transactions within the framework of two phase commit protocol. We assume that the sites communicate by a reliable broadcast which eventually deliver messages without any ordering guarantees.

In the abstraction, the global state of update transactions is represented by a variable *transstatus* in the abstract model of the transactions. The variable *transtatus* is defined as $transtatus \in trans \rightarrow TRANSSTATUS$, where $TRANSSTATUS = \{COMMIT, ABORT, PENDING\}$. The *transstatus* maps each transaction to its global state. An update transaction commits by updating abstract variable *database*. With respect to an update transaction, activation of the following events change the global transaction states.

- *StartTran(tt)* : The activation of this event *starts* a fresh transaction and the state of the transaction is set to *pending*.
- *CommitWriteTran(tt)* : This event models global commit of an update transaction. A *pending* update transaction commits atomically by updating the abstract database and it status is set to *commit*.
- *AbortWriteTran(tt)* : This event models global abort of an update transaction. A *pending* update transaction aborts by making no change in the abstract database and its status is set to *abort*.

In the refined model, a global update transaction can be submitted to any one site, called the coordinator site for that transaction. 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 starts 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 specifications of this event is given in the Fig. 2. In this refinement, the state of a transaction at a site is represented by a variable *sitetransstatus*. The variable *sitetransstatus* maps each transaction, at a site, to transaction states given by a set *SITETRANSTATUS*, where $SITETRANSTATUS = \{pending, commit, abort, precommit\}$. A transaction t is said to be active at a site s if it has acquired the locks of the object set at that site.

BeginSubTran *(tt∈ TRANSACTION ,ss∈ SITE)≅*

WHEN $\wedge$ $tt \in trans$
$\wedge$ $(ss \mapsto tt) \notin activetrans$
$\wedge$ $ss \notin dom(sitetransstatus)$
$\wedge$ $ran(transeffect(tt)) \neq \{\varnothing\}$
$\wedge$ $objectset(tt) \subseteq freeobject[\{ss\}]$
$\wedge$ $transstatus(tt)=PENDING$
$\wedge$ $\forall tz.(tz \in trans \wedge (ss \mapsto tz) \in activetrans$
$\Rightarrow objectset(tt) \cap objectset(tz) = \varnothing)$

THEN $activetrans := activetrans \cup \{ss \mapsto tt\}$
|| $sitetransstatus(tt)(ss) := pending$
|| $freeobject := freeobject - \{ss\} \times objectset(tt)$

END;

Fig. 2. Sub Transaction

Our model prevents starting sub-transaction at a site if any conflicting transaction is already active at that site. Following guard of *BeginSubTran(tt)* event 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 :

$$(ss \mapsto tz) \in activetrans \Rightarrow objectset(tt) \cap objectset(tz) = \varnothing$$

The guard $ss \notin dom(sitetransstatus(tt))$ 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. The guard $ran(transeffect(tt)) \neq \{\varnothing\}$ states that tt is an update transaction, i.e., $writeset(tt) \neq \varnothing$. 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.

- *BeginSubTran(tt)* : This event models *starting* a sub-transaction at a site. The status of the transaction tt at site ss is set to *pending*.
- *SiteAbortTx(ss,tt)* : This event models *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 *abort*.
- *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 *precommit*.
- *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 *abort*. The transaction is said to complete execution at the site.
- *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.

In our model, update messages from the coordinator site are broadcast using a reliable broadcast. A reliable broadcast imposes no restriction on the order in which messages are delivered to the participating sites. This may lead to the formation of the deadlocks due to race conditions 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 acquire locks on $objectset[T_i]$ at site S_i. Site S_i broadcast update message of T_i to participating sites. Similarly, another site S_j starts a transaction T_j , acquires locks on $objectset[T_j]$ at site S_j and broadcast update message of T_j to participating sites.
- The site S_i delivers update message of T_j from S_j and S_j delivers update message of 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. The abort of these conflicting update transactions may be avoided if a reliable broadcast also provides ordering guarantees on the message delivery such that all update messages are delivered to various participating site including the sender in a total order. In the remaining sections we formally model and analyze a system of total order broadcast and verify that the required ordering properties are satisfied.

4 Informal Specifications of a Total Order Broadcast

A reliable broadcast [20] eventually deliver the messages to all participating sites. A *total order* [16, 20] broadcast is a stronger notion of a reliable broadcast that delivers messages to all processes in a same delivery order. A *total order broadcast*[1] can be defined as a reliable broadcast which satisfies following requirement.

> *If processes p and q both deliver messages m1 and m2, then q delivers m1 before m2 if and only if p delivers m1 before m2.*

The *agreement* property of a reliable broadcast and *total order* requirements imply that all correct processes eventually deliver the same *sequence* of messages [20]. As shown in the Fig. 3 all processes has same delivery order of messages. However, as shown in Fig. 4 the delivery order violates total order requirement as delivery order at process *P1* and *P2* are different.

[1] The *Total Order Broadcast* is also known as Atomic Broadcast. Both of the terms are used interchangeably. However we prefer the former as the term *atomic* suggests the *agreement* [20] property rather than *total order*.

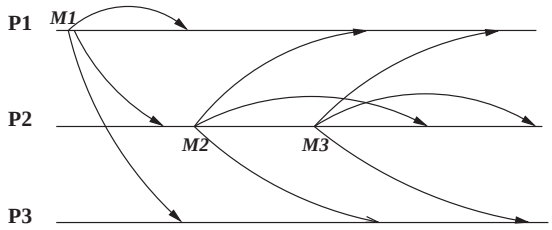

Fig. 3. Total Order

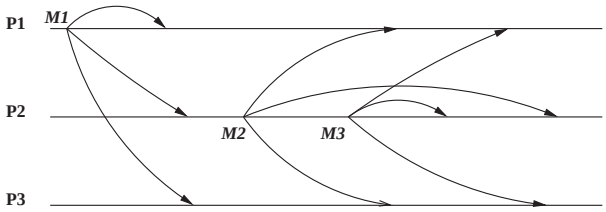

Fig. 4. Violation of Total Order

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. In our development we consider *Broadcast Broadcast*(BB) [16] variant of a sequencer based system. In sequencer based system, a specific process takes the role of a *sequencer* and becomes responsible for building a total order. The protocol consists of first broadcasting m to all destinations including the sequencer, followed by an 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 process *P2* broadcast a *computation message m.*

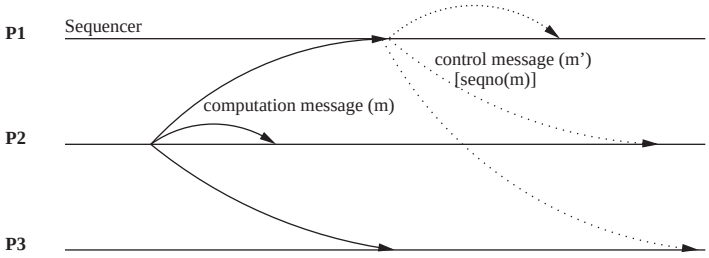

Fig. 5. Broadcast Broadcast variant

Upon delivery of m to a *sequencer* process, sequencer assigns a sequence number and broadcast its sequence number by a *control message*(m'). Upon receipt of the control messages, a destination process deliver its computation message according to the sequence numbers.

5 Abstract Model of Total Order Broadcast

The abstract model of total order broadcast system is given in Fig. 6 and Fig. 7. The *PROCESS* and *MESSAGE* sets define types for the model. The specification contains of four variables *sender*, *totalorder*, *tdeliver* and *delorder*.

MACHINE	*TotalOrder*
SETS	*PROCESS; MESSAGE*
VARIABLES	*sender, totalorder, delorder, tdeliver*
INVARIANT	$sender \in MESSAGE \nrightarrow PROCESS \land$
	$totalorder \in MESSAGE \leftrightarrow MESSAGE \land$
	$delorder \in PROCESS \nrightarrow (MESSAGE \leftrightarrow MESSAGE) \land$
	$tdeliver \in PROCESS \leftrightarrow MESSAGE$
INITIALISATION	$sender := \varnothing \quad \| \quad totalorder := \varnothing \quad \|$
	$delorder := PROCESS \times \{\varnothing\} \quad \| \quad tdeliver := \varnothing$

Fig. 6. Initial Part : Level-0

The *sender* is defined as a partial function from *MESSAGE* to *PROCESS*. The mapping $(m \mapsto p) \in sender$ 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 $(m1 \mapsto m2) \in totalorder$ indicate that message *m1* is *totally ordered before m2*. The variable *tdeliver* represent the messages delivered following a total order. A mapping of form $(p \mapsto m) \in tdeliver$ represents that a process p has delivered m following a *total order*. In order to represent the delivery order of messages at a process, variable *delorder* is used. A mapping $(m1 \mapsto m2) \in delorder(p)$ indicate that process p has delivered *m1* before *m2*.

The event *Broadcast* given in the Fig. 7 models the broadcast of a message. Similarly, the event *Order* models the construction of 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 ($pp \in PROCESS$, $mm \in MESSAGE$) $\widehat{=}$
  WHEN $mm \notin dom(sender)$
  THEN $sender := sender \cup \{mm \mapsto pp\}$
  END;
Order ($pp \in PROCESS$,$mm \in MESSAGE$) $\widehat{=}$
  WHEN $mm \in dom(sender) \ \wedge$
  $mm \notin ran(tdeliver) \ \wedge$
  $ran(tdeliver) \subseteq tdeliver[\{pp\}]$
  THEN $tdeliver := tdeliver \cup \{pp \mapsto mm\} \ \|$
  $totalorder := totalorder \cup (\ ran(tdeliver) \times \{mm\}) \ \|$
  $delorder(pp) := delorder(pp) \cup (\ tdeliver[\{pp\}] \times \{mm\})$
  END;
TODeliver ($pp \in PROCESS$, $mm \in MESSAGE$) $\widehat{=}$
  WHEN $mm \in dom(sender) \ \wedge$
  $mm \in ran\ (\ tdeliver\ ) \ \wedge$
  $pp \mapsto mm \notin tdeliver \ \wedge$
  $\forall m.(\ m \in MESSAGE \wedge (m \mapsto mm) \in totalorder$
  $\Rightarrow (pp \mapsto m) \in tdeliver)$
  THEN $tdeliver := tdeliver \cup \{pp \mapsto mm\} \ \|$
  $delorder(pp) := delorder(pp) \cup (\ tdeliver[\{pp\}] \times \{mm\})$
  END

Fig. 7. Events : Level-0

5.1 Constructing a Total Order

The event *Order* models the construction of an abstract total order on message mm its *first* ever delivery to a process pp. The following guards of this event ensures 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).

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

Later in the refinement we show that this is a function of a designated process called *sequencer*. As a consequence of the occurrence of *Order* event, the message mm is delivered to the process pp and variable *totalorder* is updated by mappings in $(ran(tdeliver) \times mm)$. 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 precedes 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*. As the guard $mm \in ran(tdeliver)$ implies that the *mm* has been delivered to at least one process and it also implies that the total order on the message *mm* has also been constructed. Later in the refinement we show that process *pp* represents a process other than the *sequencer* process. The guard of the event ensure that message *mm* has already been delivered elsewhere and that all messages which precedes *mm* in abstract total order has also been delivered to *pp*.

5.2 Invariant Properties of Total Order

After building an abstract model of a total order broadcast(Level-0), our goal was to formally verify that our model preserves the total ordering properties defined in the Section 4. The agreement and total order requirement imply that all correct process eventually deliver all messages in the same order [20]. Thus, we add following invariant as a primary invariant to our model.

$$m1 \mapsto m2 \in delorder(p) \Rightarrow m1 \mapsto m2 \in totalorder \tag{1}$$

This invariant at (1) state that if a process delivers any two messages then their delivery order at that process corresponds to their abstract total order. Subsequently, in order to prove that total order preserves the transitivity property, we add following as a primary invariant to our model.

$$\begin{aligned} & m1 \mapsto m2 \in totalorder \wedge m2 \mapsto m3 \in totalorder \\ & \Rightarrow \ m1 \mapsto m2 \in totalorder \end{aligned} \tag{2}$$

Lastly, to verify that the abstract total order is non-symmetric and non-reflexive, we add following invariant :

$$m1 \mapsto m2 \in totalorder \Rightarrow m2 \mapsto m1 \notin totalorder \tag{3}$$

$$m \in MESSAGE \Rightarrow m \mapsto m \notin totalorder \tag{4}$$

6 Proof Obligations and Invariant Discovery

In this section, we outline how the proof obligations generated due to the addition of the primary invariants given at (1), (2), (3) and (4) in Fig. 8 guide us discovering new invariants.

Verification of Total Ordering Property : In order to verify that our abstract model of total order broadcast satisfies the total order property, we add *Inv-1* given in Fig. 8 to our model. When we add this invariant to our model two proof obligations were generated associated with the event *Order* and *TODeliver*. Proof obligation associated with the event *Order* was discharged using interactive prover, however the proof obligation associated with *TODeliver* could not

Primary Invariants

/*Inv-1*/	$(m1 \mapsto m2) \in delorder(p) \Rightarrow (m1 \mapsto m2) \in totalorder$	*Total Order*
/*Inv-2*/	$(m1 \mapsto m2) \in totalorder \wedge (m2 \mapsto m3) \in totalorder \Rightarrow (m1 \mapsto m3) \in totalorder$	*Transitivity*
/*Inv-3*/	$(m1 \mapsto m2) \in totalorder \Rightarrow (m2 \mapsto m1) \notin totalorder$	*Non-symmetric*
/*Inv-4*/	$m \in MESSAGE \Rightarrow (m \mapsto m) \notin totalorder$	*Non-reflexive*

Fig. 8. Primary Invariants-I : Level-0

be discharged. Following is the simplified form of a proof obligation generated by the interactive prover.

$$TODeliver(PO1)$$
$$\begin{bmatrix} p \mapsto m1 \in tdeliver \wedge \\ p \mapsto m2 \notin tdeliver \wedge \\ m2 \in ran(tdeliver) \wedge \\ \Rightarrow \ m1 \mapsto m2 \in totalorder \end{bmatrix}$$

This state 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-5* in Fig. 9. Addition of *Inv-5* was sufficient to discharge *PO1*, however a new proof obligation associated with *TODeliver* was generated due to the addition of *Inv-5*. Following is the simplified form of the proof obligation.

$$TODeliver(PO2)$$
$$\begin{bmatrix} m1 \in ran(tdeliver) \wedge \\ m2 \in ran(tdeliver) \wedge \\ m2 \mapsto m1 \notin totalorder \wedge \\ \Rightarrow \ m1 \mapsto m2 \in totalorder \end{bmatrix}$$

This proof obligation require us to prove that if two messages *m1* and *m2* are delivered to any process(es) in the system then a total order exists among them, i.e., either *m1* precedes *m2* or *m2* precedes *m1* in abstract total order. In order to discharge the proof obligation we add another invariant *Inv-6* to our model. Addition of this invariant to the model further generate proof obligations.

After four round of invariant strengthening we arrive at a set of invariant given in Fig. 9 which were sufficient to discharge all proof obligations generated

	Invariants	Required By
/*Inv-5*/	$(p \mapsto m1) \in tdeliver \ \wedge (p \mapsto m2) \notin tdeliver$ $\wedge\ m2 \in \text{ran}(tdeliver)$ $\Rightarrow \ (m1 \mapsto m2) \in totalorder$	*TOdeliver*
/*Inv-6*/	$m1 \in ran(tdeliver) \ \wedge m2 \in ran(tdeliver)$ $\wedge (m2 \mapsto m1) \notin totalorder$ $\Rightarrow \ (m1 \mapsto m2) \in totalorder$	*Order, TOdeliver*
/*Inv-7*/	$(p \mapsto m1) \in tdeliver \ \wedge (p \mapsto m2) \in tdeliver$ $\wedge (m2 \mapsto m1) \notin totalorder$ $\Rightarrow \ (m1 \mapsto m2) \in totalorder$	*Order, TOdeliver*
/*Inv-8 */	$(p1 \mapsto m1) \in tdeliver \ \wedge (p1 \mapsto m2) \notin tdeliver$ $\wedge (p2 \mapsto m1) \in tdeliver \ \wedge (p2 \mapsto m2) \in tdeliver$ $\Rightarrow \ (m1 \mapsto m2) \in totalorder$	*Order, TOdeliver*

Fig. 9. Invariants-II : Level-0

due to addition of invariant *Inv-1* is a primary invariant. A brief description of the properties is given below.

- If a process p has delivered *m1* and but not *m2*, and if *m2* was delivered to at least one process elsewhere in the system then *m1* precedes *m2* in total order(*Inv-5*).
- If two messages *m1* and *m2* has been delivered anywhere in the system then a total order exist among them, such that, either *m1* precedes *m2* or *m2* precedes *m1* in total order. (*Inv-6*)
- If a process p has delivered two message *m1* and *m2* then either *m1* precedes *m2* or *m2* precedes *m1* in totalorder(*Inv-7*).
- Given two processes *p1* and *p2*, then for any two messages *m1* and *m2* if the process *p2* has delivered both messages and *p1* has delivered *m1* but not *m2* then *m1* precedes *m2* in total order(*Inv-8*).

Verification of Transitivity Property : Our next step was to verify that our model of total order broadcast also preserves transitive properties on abstract total order. In order to verify that *total order* is transitive, we add *Inv-2* given in Fig. 8 to our model. Addition of this invariant generate several proof obligations. Using the same strategy of invariants strengthening outlined in previous section, we arrive at a set of invariant that is sufficient to discharge all proof obligations generated due the addition of *Inv-2* as a primary invariants. A full set of invariant

	Invariants	Required By
/*Inv-9 */	$(m1 \mapsto m2) \in totalorder \ \wedge (p \mapsto m2) \in tdeliver \Rightarrow (p \mapsto m1) \in tdeliver$	*Broadcast,Order TOdeliver*
/*Inv-10 */	$m \in (\ dom\ (totalorder) \cup ran(totalorder)\ ) \Rightarrow m \in ran(tdeliver)$	*Order*
/*Inv-11 */	$m \notin dom(\text{sender}) \Rightarrow m \notin dom(totalorder)$ $m \notin dom(\text{sender}) \Rightarrow m \notin ran(totalorder)$ $ran(tdeliver) \subseteq \text{dom(sender)}$	*Broadcast, Order TOdeliver*

Fig. 10. Invariants-III : Level-0

are given in the Fig. 10. A brief description of these properties are outlined below.

- For any two messages *m1* and *m2* where *m1* is *totally ordered before m2* then a process p who delivered *m2* has also delivered *m1*(*Inv-9*).
- The total order is built for those messages which has been delivered to at least one process(*Inv-10*).
- A total order can not be build for the messages which were not sent and each message delivered at any process must be a sent message (*Inv-11*).

Verification of Non-Symmetric and Non-Reflexive Property : In order to prove the *non-symmetric* and *non-reflexive* property on total order we add primary invariants *Inv-3* and *Inv-4* given in Fig. 8 to our model. Using process outlined in the previous section, we are able to discharge the proof obligations generated due to addition of these primary invariants without having to add a new invariant.

7 Overview of the Refinement Chain

In the previous sections we outlined abstract model of a total order broadcast and the invariant properties of abstract total order. 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.

L0 This consist of 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.

L1 This is a refinement of abstract model which introduces the notion of the *sequencer*. In this refinement we outline how a total order on the messages are constructed by the *sequencer*.

L2 This is a very simple refinement giving more concrete specification of *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.

L3 In this refinement we introduce the notion of *computation* messages and *sequence numbers*. Global sequence number of the computation messages are generated by the sequencer. The delivery of the messages is done based on the sequence numbers.

L4 In this refinement we introduce notion of *control* messages. We also introduce the relationship of each *computation* message with the *control* messages.

L5 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 *control* message for it.

7.1 Introducing the notion of the Sequencer : Level-1

In the first refinement, given in Fig. 11, we introduce the notion of a sequencer. The sequencer is defined as a constant for this model as $sequencer \in PROCESS$. As shown in the refined specification of *Order* event given in Fig. 11, a message is first delivered to the sequencer process. It can be noticed that the the following guards in the abstract specification

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

are replaced by following.

$$pp = sequencer$$
$$(sequencer \mapsto mm) \notin tdeliver$$

Due to the guard $pp \neq sequencer$ shown in the specifications of *TODeliver*, a message *mm* is delivered to a process other than the sequencer. The replacement of the guards in the *Order* event generate new proof obligations. Using the same approach of invariant discovery as outlined in Section 5.2, we arrived at a set of invariants that was sufficient to discharge all proof obligations. These invariants are given in Fig. 12. A brief description of these invariants are given in the following steps.

- A message not delivered to the sequencer have not been delivered elsewhere. (*Inv-12*)
- If a total order on any message m has been constructed then it must have been delivered to the sequencer.(*Inv-13,14*)

Order ($pp \in PROCESS$,$mm \in MESSAGE$) $\cong$
WHEN $pp = sequencer \wedge$
$mm \in dom(sender) \ \wedge$
$(sequencer \mapsto mm) \notin tdeliver$
THEN $tdeliver := tdeliver \cup \{pp \mapsto mm\} \ \|$
$totalorder := totalorder \cup (\ ran(tdeliver) \times \{mm\})$
END;
TODeliver ($pp \in PROCESS$, $mm \in MESSAGE$) $\cong$
WHEN $pp \neq sequencer \wedge$
$mm \in dom(sender) \wedge$
$mm \in ran\ (\ tdeliver\ ) \wedge$
$pp \mapsto mm \notin tdeliver \ \wedge$
$\forall m.(\ m \in MESSAGE \wedge (m \mapsto mm) \in totalorder$
$\Rightarrow (pp \mapsto m) \in tdeliver)$
THEN $tdeliver := tdeliver \cup \{pp \mapsto mm\}$
END

Fig. 11. Total Order Broadcast : Level-1

	Invariants	**Required By**
/*Inv-12*/	$(sequencer \mapsto m) \notin tdeliver \Rightarrow m \notin ran(tdeliver)$	*Order,TOdeliver*
/*Inv-13*/	$m \in dom(totalorder) \Rightarrow (sequencer \mapsto m) \in tdeliver$	*Order*
/*Inv-14*/	$m \in ran(totalorder) \Rightarrow (sequencer \mapsto m) \in tdeliver$	*Order*

Fig. 12. Invariants-IV : Level-1

Order ($pp \in PROCESS$,$mm \in MESSAGE$) $\cong$
WHEN $pp = sequencer \ \wedge$
$mm \in dom(sender) \ \wedge$
$(sequencer \mapsto mm) \notin tdeliver$
THEN $tdeliver := tdeliver \cup \{pp \mapsto mm\} \ \|$
$totalorder := totalorder \cup (\ tdeliver[\{sequencer\}] \times \{mm\})$
END;

Fig. 13. Total Order Broadcast : Refined Order Event : Level-2

7.2 Second Refinement : Refinement of 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. 11, a total order is generated as $totalorder := totalorder \cup (ran(tdeliver) \times \{mm\})$. It state that all messages delivered at any process are ordered before the new message *mm*.

In the refined *Order* event the totalorder is constructed as $totalorder := totalorder \cup (tdeliver[\{sequencer\}] \times \{mm\})$. It state that all messages delivered to the sequencer are ordered before the new message *mm*.

Invariants		Required By
/*Inv-15*/	$ran(tdeliver) = tdeliver[\{sequencer\}]$	*Order*

Fig. 14. Invariants-V : Level-2

The specifications of this refinement are given in the Fig. 13. 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-15* given in the Fig. 14. This invariant was sufficient to discharge the proof obligations.

7.3 Third Refinement : Introducing Sequence Numbers

In the third refinement, given in Fig. 15, we introduce the notion of computation message and the sequence numbers. The new variables *computation*, *seqno* and *counter* are introduced in the refinement typed as follows :

$$computation \subseteq MESSAGE$$
$$seqno \in computation \nrightarrow Natural$$
$$counter \in Natural$$

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

$$(m \mapsto mm) \in totalorder \Rightarrow (pp \mapsto m) \in tdeliver$$

The above is replaced by following guard in this refinement.

$$seqno(m) < seqno(mm) \Rightarrow (pp \mapsto m) \in tdeliver$$

Order ($pp \in PROCESS, mm \in MESSAGE$) $\cong$
WHEN $pp = sequencer$
$mm \in dom(sender) \wedge$
$mm \in computation \wedge$
$(sequencer \mapsto mm) \notin tdeliver$
THEN $totalorder := totalorder \cup (\ tdeliver[\{sequencer\}] \times \{mm\})$ ||
$tdeliver := tdeliver \cup \{pp \mapsto mm\}$ ||
$seqno := seqno \cup \{mm \mapsto counter\}$ ||
$counter := counter + 1$
END;
TODeliver ($pp \in PROCESS, mm \in MESSAGE$) $\cong$
WHEN $pp \neq sequencer \wedge$
$mm \in dom(sender) \wedge$
$mm \in ran\ (\ tdeliver\ ) \wedge$
$pp \mapsto mm \notin tdeliver \wedge$
$\forall m.(\ m \in computation \wedge (\ seqno(m) < seqno(mm)\ )$
$\Rightarrow (pp \mapsto m) \in tdeliver)$
THEN $tdeliver := tdeliver \cup \{pp \mapsto mm\}$
END

Fig. 15. Total Order Broadcast : Level-3

	Invariants	**Required By**
/*Inv-16*/	$m1 \mapsto m2 \in totalorder$ $\Rightarrow seqno(m1) < seqno(m2)$	*Order,TOdeliver*
/*Inv-17*/	$m \in computation \wedge m \in dom(seqno)$ $\Rightarrow sequencer \mapsto m \in tdeliver$	*Order,TOdeliver*

Fig. 16. Invariants-VI : Level-3

The change of the guards in the *TODeliver* event generate new proof obligations. These proof obligations are discharged by adding new invariants given in the Fig. 16 to the model. Invariant *Inv-16* state that if *m1* precedes *m2* in abstract total order then the sequence number assigned to *m1* is less than the sequence number assigned to *m2*. The invariant *Inv-17* state that if a computation message has been assigned a sequence number then sequencer must have delivered it.

7.4 Fourth Refinement : Introducing Control Messages

In this refinement given in Fig. 17, 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 process 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 :

$$control \subseteq MESSAGE$$
$$messcontrol \in control \rightarrowtail computation$$

The variables *control* and *computation* are used to represent a computation or

Order ($pp \in PROCESS$,$mm \in MESSAGE$,$mc \in MESSAGE$) $\widehat{=}$
WHEN $pp = sequencer \ \wedge$
$mm \in dom(sender) \ \wedge$
$mm \in computation \wedge$
$(sequencer \mapsto mm) \notin tdeliver \ \wedge$
$mc \notin dom(messcontrol) \wedge$
$mm \notin ran(messcontrol)$
THEN $totalorder := totalorder \cup (tdeliver[\{sequencer\}] \times \{mm\}) \ \|$
$tdeliver := tdeliver \cup \{pp \mapsto mm\} \ \|$
$control := control \cup \{mc\} \ \|$
$messcontrol := messcontrol \cup \{mc \mapsto mm\} \ \|$
$seqno := seqno \ \cup \{mm \mapsto counter\} \ \|$
$counter := counter + 1$
END;
TODeliver ($pp \in PROCESS$, $mm \in MESSAGE$) $\widehat{=}$
WHEN $pp \neq sequencer \wedge$
$mm \in dom(sender) \ \wedge$
$mm \in ran (messcontrol) \ \wedge$
$pp \mapsto mm \notin tdeliver \ \wedge$
$\forall m.(m \in computation \wedge (\text{seqno}(m) < \text{seqno}(mm))$
$\Rightarrow (pp \mapsto m) \in tdeliver)$
THEN $tdeliver := tdeliver \cup \{pp \mapsto mm\}$
END

Fig. 17. Total Order Broadcast : Level-4

	Invariants	Required By
/*Inv-18*/	$ran(messcontrol) \subseteq ran(tdeliver)$	Order,TOdeliver
/*Inv-19*/	$ran(messcontrol) \subseteq computation$	Order,TOdeliver

Fig. 18. Invariants-VII : Level-4

ReceiveControl ($pp \in PROCESS$, $mc \in MESSAGE$) $\cong$
WHEN $mc \in control \wedge$
$(pp \mapsto mc) \notin receive$
THEN $receive := receive \cup \{pp \mapsto mc\}$
END
TODeliver ($pp \in PROCESS$, $mm \in MESSAGE$) $\cong$
WHEN $pp \neq sequencer \wedge$
$mm \in computation \wedge$
$(pp \mapsto mm) \notin tdeliver \wedge$
$(pp \mapsto messcontrol^{-1}(mm)) \in receive \wedge$
$\forall m.(\, m \in computation \wedge (seqno(m) < seqno(mm) \Rightarrow (pp \mapsto m) \in tdeliver)$
THEN $tdeliver := tdeliver \cup \{pp \mapsto mm\}$
END

Fig. 19. Total Order Broadcast: Receive Control : Level-5

	Invariants	Required By
/*Inv-20*/	$m \in computation \wedge messcomtrol^{-1}(m) \in receive \Rightarrow m \in ran(messcontrol)$	Order,TOdeliver

Fig. 20. Invariants-VIII : Level-5

a control message. The variable *messcontrol* is a partial injective function which defines relationship among a control message and its computation message. A mapping $(m1 \mapsto m2) \in messcontrol$ indicate that message *m1* is the *control message* related to the *computation message m2*. The set *ran(messcontrol)* contains the computation messages for which control messages has been sent by the sequencer. The guard $mm \in ran(tdeliver)$ of *TODeliver* event is replaced by the guard $mm \in ran(messcontrol)$ in this refinement. This indicate 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. The change in the guards of *Order* and *TODeliver* events generate proof obligations which are discharged by adding following invariant to the model.

7.5 Fifth Refinement : Introducing Receive Control Event

A new event *ReceiveControl* is introduced in this refinement. This event model receiving a control message at a process. A new variable *receive* is also introduced in this refinement typed as $receive \in PROCESS \leftrightarrow control$. A mapping $p \mapsto m \in receive$ indicate that process p has received a control message m. The specifications of the refined events are given in Fig. 19.

As shown in the *TODeliver* event at Level-5, the guard $mm \in ran(messcontrol)$ is replaced by the the guard $(pp \mapsto messcontrol^{-1}(mm)) \in receive$. This guard of the *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 generate proof obligations associated with the event *TODeliver*. In order to discharge these proof obligations we add the invarinats given in Fig. 20.

8 Related Work

Distributed algorithms can be deceptive, may have complex execution paths and may allow unanticipated behavior. Rigorous reasoning about such algorithms is required to ensure that an algorithm achieves what it is supposed to do [25]. The group communication primitives has been proposed to develop fault-tolerant distributed services. The total order broadcast is one primitive that deliver messages to the sites in a distributed system in same 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 [34]. The implementations of these group communication primitives has also been investigated for different distributed systems such Isis [10], Totem [29], Trans [27], Amoeba [36] and Transis [7]. The protocols in these systems use varying broadcast primitives and address group maintenance, fault tolerance and consistency services. The transaction mechanism in the management of replicated data is also considered in [6, 8, 31, 32, 34].

Group communication services have been studied as a basic building block for many fault tolerant distributed services, however the application of formal methods providing clear specifications and proofs of correctness is rare [16]. In [17], 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 [18, 33] the specification 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 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 execution. A series of invariants relating state variables and reachable states are proved by hand using the method of induction. In [37], a formal method is proposed to prove the total and causal order of multicast protocols. The formal results are provided in the paper that 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. In a similar work in [26], meta properties are used to express total order broadcast algorithm. The proof of correctness of the results are done by hand.

Instead, our approach of specifying the system and verification is based on the technique of abstraction and refinement. This formal approach carries a step-wise development from initial abstract specifications to a detailed design of a system in the refinement steps. Through the refinement proofs we verify that design of detailed system conforms to the abstract specifications. A refinement based approach to developing distributed systems in B is outlined in [12]. Use of refinement and decomposition rules in the development of telecommunications systems is outlined in [11]. The refinement approach of Event-B has also been used for the formal development of fault-tolerant communication systems [24] and fault-tolerant agent systems [23]. Other important work carried out using the refinement approach include verification of the IEEE 1394 tree protocol distributed algorithm [5], development of a secure communication system [13], development of a train system [2], verification of one copy equivalence criterion in a distributed database system [38]. The case study on development of Mondex purse system in Event-B [15] illustrates modelling strategies and the guidelines to achieve a high degree of automatic proofs.

9 Conclusions

In a replicated database, an update transaction modifies the requested data objects at various sites. A global update transaction may be submitted to any site and the effects of the update transaction are global, i.e., at commit all replicas at various sites must be updated. In case of abort, none of the sites update data objects. We have presented a rigorous design of distributed transactions for a replicated database in [38]. In this model, update messages from the coordinator site are assumed to broadcast using a reliable broadcast. A reliable broadcast imposes no restriction on the order in which messages are delivered to the participating sites. Unordered delivery of updates to the participating sites leads to the formation of deadlocks and the sites may abort conflicting transactions by timeouts. The failure of such transactions may be avoided if the updates are broadcast using a total order broadcast that delivers updates to the participating sites in a same order.

In this paper we have presented formal development of a system of total order broadcast. In the abstract model we outline how an abstract total order is con-

structed on the messages. Subsequently in a series of refinement steps we outline how an abstract total order can correctly be implemented by using the notion of control messages and sequence numbers. Instead of model checking, proving theorems by hand or proving correctness of the trace behavior, our approach consists of defining problem in the abstract model and introducing solutions or design details in the refinement steps. Through refinement checking we verify that the models in the refinement are valid refinement of abstract models. We used the *Click'n'Prove* B tool for proof management. This tool generates the proof obligations due to refinement and consistency checking, factorizes complex proof obligations in to relatively simpler proofs and helps discharge proof obligations by the use of automatic and interactive prover.

This case study illustrate how an incremental approach to system development can be used to obtain more concrete specifications. A powerful tool support helped us to discover several new invariants that helps to understand why a total order broadcast can correctly be implemented using sequence numbers. A clear relationship of computation and control message is outlined to indicate that our system generate exactly one control message for each computation message. In this case study approximately 75% of the proof obligations were discharged using automatic prover. The proof obligations generated by the B tool also help discovering new system invariants. The proofs and the invariants help to precisely understand why a design decision or a solution proposed in the refinement is correct. The over all proof statistics is given in Table 1.

Machine	Total POs	Completely Automatic	Required Interaction
Abstract Model	48	29	19
Refinement1	19	16	03
Refinement2	2	2	00
Refinement3	18	14	04
Refinement4	15	14	01
Refinement5	04	04	00
Overall	106	79	27

Table 1. Proof Statistics- Total Order Broadcast

References

1. J.-R. Abrial. *The B-Book: Assigning programs to meanings.* Cambridge University Press, 1996.
2. Jean-Raymond Abrial. Train systems. In Butler et al. [14], pages 1–36.
3. Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, and Laurent Voisin. An open extensible tool environment for Event-B. In Zhiming Liu and Jifeng He, editors, *ICFEM*, volume 4260 of *Lecture Notes in Computer Science*, pages 588–605. Springer, 2006.

4. Jean-Raymond Abrial and Dominique Cansell. Click'n prove: Interactive proofs within set theory. In David A. Basin and Burkhart Wolff, editors, *TPHOLs*, volume 2758 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2003.
5. Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. A mechanically proved and incremental development of ieee 1394 tree identify protocol. *Formal Asp. Comput.*, 14(3):215–227, 2003.
6. Divyakant Agrawal, Gustavo Alonso, Amr El Abbadi, and Ioana Stanoi. Exploiting atomic broadcast in replicated databases (extended abstract). In Christian Lengauer, Martin Griebl, and Sergei Gorlatch, editors, *Euro-Par*, volume 1300 of *Lecture Notes in Computer Science*, pages 496–503. Springer, 1997.
7. Yair Amir, Danny Dolev, Shlomo Kramer, and Dalia Malki. Membership algorithms for multicast communication groups. In Adrian Segall and Shmuel Zaks, editors, *WDAG*, volume 647 of *Lecture Notes in Computer Science*, pages 292–312. Springer, 1992.
8. Özalp Babaoglu, Alberto Bartoli, and Gianluca Dini. Replicated file management in large-scale distributed systems. In Gerard Tel and Paul M. B. Vitányi, editors, *WDAG*, volume 857 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 1994.
9. Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
10. Kenneth P. Birman, André Schiper, and Pat Stephenson. Lightweigt causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, 1991.
11. Michael Butler. Stepwise refinement of communicating systems. *Science of Computer Programming.*, 27(2):139–173, 1996.
12. Michael Butler. An approach to the design of distributed systems with B AMN. In Jonathan P. Bowen, Michael G. Hinchey, and David Till, editors, *ZUM*, volume 1212 of *Lecture Notes in Computer Science*, pages 223–241. Springer, 1997.
13. Michael Butler. On the use of data refinement in the development of secure communications systems. *Formal Aspects of Computing.*, 14(1):2–34, 2002.
14. Michael Butler, Cliff B. Jones, Alexander Romanovsky, and Elena Troubitsyna, editors. *Rigorous Development of Complex Fault-Tolerant Systems [FP6 IST-511599 RODIN project]*, volume 4157 of *Lecture Notes in Computer Science*. Springer, 2006.
15. Michael Butler and Divakar Yadav. An incremental development of the mondex system in Event-B. *Formal Aspects of Computing*, 20(1):61–77, 2008.
16. Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.
17. Alan Fekete, M. Frans Kaashoek, and Nancy Lynch. Implementing sequentially consistent shared objects using broadcast and point-to-point communication. *Journal of the ACM*, 45(1):35–69, 1998.
18. Alan Fekete, Nancy A. Lynch, and Alexander A. Shvartsman. Specifying and using a partitionable group communication service. *ACM Trans. Comput. Syst.*, 19(2):171–216, 2001.
19. Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
20. V. Hadzilacos and S.Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR 94 -1425, Cornell University,NY, 1994.
21. Bettina Kemme and Gustavo Alonso. A suite of database replication protocols based on group communication primitives. In *Proc. Intl. Conf. Distributed Computing System, Amsterdam, ICDCS*, pages 156–163, 1998.

22. Bettina Kemme, Fernando Pedone, Gustavo Alonso, André Schiper, and Matthias Wiesmann. Using optimistic atomic broadcast in transaction processing systems. *IEEE Trans. Knowl. Data Eng.*, 15(4):1018–1032, 2003.
23. Linas Laibinis, Elena Troubitsyna, Alexei Iliasov, and Alexander Romanovsky. Rigorous development of fault-tolerant agent systems. In Butler et al. [14], pages 241–260.
24. Linas Laibinis, Elena Troubitsyna, Sari Leppänen, Johan Lilius, and Qaisar A. Malik. Formal service-oriented development of fault tolerant communicating systems. In *Rigorous Development of Complex Fault-Tolerant Systems*, pages 261–287, volume 4157 of Lecture Notes in Computer Science, Springer,2006.
25. Leslie Lamport and Nancy A. Lynch. Distributed computing: Models and methods. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 1157–1199. 1990.
26. X Liu, R Renesse, M Bickford, C Krietz, and R Constable. Protocol switching : Exploiting meta-properties. In *Intl. Workshop on applied reliable group communication, WARGC 2001,IEEE Computer Science*, pages 37–42, 2001.
27. P. M. Melliar-Smith, Louise E. Moser, and Vivek Agrawala. Broadcast protocols for distributed systems. *IEEE Trans. Parallel Distrib. Syst.*, 1(1):17–25, 1990.
28. C Metayer, J R Abrial, and L Voison. Event-B language. RODIN deliverables 3.2, http://rodin.cs.ncl.ac.uk/deliverables/D7.pdf, 2005.
29. Louise E. Moser, P. M. Melliar-Smith, Deborah A. Agarwal, Ravi K. Budhia, and Colleen A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Commun. ACM*, 39(4):54–63, 1996.
30. M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems, Second Edition*. Prentice-Hall, 1999.
31. Marta Patiño-Martínez, Ricardo Jiménez-Peris, Bettina Kemme, and Gustavo Alonso. Middle-r: Consistent database replication at the middleware level. *ACM Trans. Comput. Syst.*, 23(4):375–423, 2005.
32. Fernando Pedone, Rachid Guerraoui, and André Schiper. The database state machine approach. *Distributed and Parallel Databases*, 14(1):71–98, 2003.
33. Roberto De Prisco, Alan Fekete, Nancy Lynch, and Alex Shvartsman. A dynamic view-oriented group communication service. In *PODC '98: Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, pages 227–236, New York, NY, USA, 1998. ACM Press.
34. Andre Schiper and Michel Raynal. From group communication to transactions in distributed systems. *Communication of the ACM*, 39(4):84–87, 1996.
35. Ioana Stanoi, Divyakant Agrawal, and Amr El Abbadi. Using broadcast primitives in replicated databases. In *Proc. of 18th IEEE Intl. Conf. on Distributed Computing System,ICDCS*, pages 148–155, 1998.
36. Andrew S. Tanenbaum, M. Frans Kaashoek, Robbert van Renesse, and Henri E. Bal. The amoeba distributed operating system - a status report. *Computer Communications*, 14(6):324–335, 1991.
37. C. Toinard, Gerard Florin, and C. Carrez. A formal method to prove ordering properties of multicast systems. *ACM Operating Systems Review*, 33(4):75–89, 1999.
38. Divakar Yadav and Michael Butler. Rigorous design of fault-tolerant transactions for replicated database systems using Event B. In *Rigorous Development of Complex Fault-Tolerant Systems*, pages 343–363, volume 4157 of Lecture Notes in Computer Science, Springer,2006.