Rigorous Design of Fault-Tolerant Transactions for Replicated Database Systems using Event B

Divakar Yadav[⋆] and Michael Butler[⋆⋆]

School of Electronics and Computer Science
University of Southampton
Southampton SO17 1BJ ,U.K
{dsy04r,mjb}@ecs.soton.ac.uk

Abstract. System availability is improved by the replication of data objects in a distributed database system. However, during updates, the complexity of keeping replicas identical arises due to failures of sites and race conditions among conflicting transactions. Fault tolerance and reliability are key issues to be addressed in the design and architecture of these systems. Event B is a formal technique which provides a framework for developing mathematical models of distributed systems by rigorous description of the problem, gradually introducing solutions in refinement steps, and verification of solutions by discharge of proof obligations. In this paper, we present a formal development of a distributed system using Event B that ensures atomic commitment of distributed transactions consisting of communicating transaction components at participating sites. 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 refinement. Through refinement we verify that the design of the replicated database confirms to the one copy database abstraction.

1 Introduction

A distributed system is a collection of autonomous computer systems that cooperate with each other for successful completion of a distributed computation. A distributed computation may require access to resources located at participating sites. A distributed transaction may span several sites reading or updating data objects. 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 [29]. Assuming that each site maintains a log and a recovery procedure, commit protocols [17, 29] ensure that all sites abort or commit a transaction unanimously despite multiple failures. Several versions of commit protocols were proposed to improve performance

[⋆] Divakar Yadav is a Commonwealth Scholar supported by the Commonwealth Scholarship Commission in the United Kingdom.

[⋆⋆] Michael Butler's contribution is part of the IST project IST 511599 RODIN (Rigorous Open Development Environment for Complex Systems).

dealing with various aspects such as site failures, blocking and even compensation. Distributed transaction execution within the framework of commit protocols ensures consistency and provides fault tolerance. There exist a number of broadcast-based communication paradigms, e.g., *centralized two phase commit*, *nested two phase commit*, *distributed two phase commit* [24] in which commit protocols are implemented.

Replication improves availability in a distributed database system. It is advantageous to replicate data objects when the transaction workload is predominantly read only. However, during updates, the complexity of keeping replicas identical arises due to site failures and conflicting transactions. The algorithms ensuring globally ordered delivery of messages may be coupled with the provisions to provide fault tolerance in the event of failures. Several approaches has been proposed for management of replicated data using group communication primitives [5, 18, 20, 26, 30]. The application of formal methods to a replication algorithm is considered in [16]. Group communication has also been investigated in Isis [8], Totem [23] and Trans [22]. The protocols in these system use varying broadcast primitives and address group maintenance, fault tolerance and consistency services. The transaction semantics in the management of replicated data is also considered in [5, 6, 26]. 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* [7, 24] 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.

The *One Copy Serializability* [7] is the highest correctness criterion for replica control protocols. It is achieved by coupling 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. The one copy equivalence and serial execution together provide one copy serializability which is supported in a *read anywhere write everywhere* approach [27]. For example, consider any serial execution of a transaction produced by system in the *read anywhere write everywhere* replica control. A transaction which writes to a data item does so by writing data everywhere. Thus from the view point of a transaction which reads the values produced by an earlier transaction, all copies were written *simultaneously.* So no matter which copy a transaction reads, it reads the same value written by an earlier transaction [7]. Though serializability is the highest correctness criteria, it is too restrictive in practice. Various degrees of isolation to address this problem has been studied in [20].

Our focus in this paper is on data replication. An update transaction which spans several sites issuing a series of read/write operations is executed in isolation at a given site. The basic idea used in this paper is to allow update transactions to be submitted at any site. This site, called the coordinating site, broadcasts update messages to replicas at participating sites. Upon receipt of update requests, each site starts a sub transaction if it does not conflict with any other *active* transactions at that site. The coordinating site decides to commit if a transaction commits at all participating sites. Atomic broadcast is a powerful service in the design of distributed applications. We assume that sites communicate by broadcasting messages following globally ordered delivery of messages [14, 32]. Advantages of group communication in varying degree of isolation can be found in [15].

The reliability of distributed systems is an important design criterion for developing new distributed services or updating existing ones. Reliability refers to both *resilience* of a system to various type of failures and its capability to *recover* from them [24]. These issues must be addressed in design, architecture and in the component infrastructure itself. It is not possible to simply add a fault tolerance module later on to make the system fault tolerant [19]. A system can be designed to be fault tolerant by exhibiting *well defined behavior* which facilitates the action 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 containing replicas and a replica should receive the updates in the same order they were sent.

Formal methods provide a systematic approach to the development of dependable complex systems. They use mathematical notations to describe and reason about systems. Event B is a formal technique developed by Abrial [3, 2] for distributed systems. In this paper we formally develop a model of transactions in B for a one copy database. In the refinement, the notion of replicated database is introduced. We address the one copy equivalence consistency criterion through this refinement. We also address the issues of fault tolerance and reliability of the system by allowing for transaction failure at sites in our refinement. By verifying the refinement, we verify that the design of the replicated database confirms to the one copy database abstraction.

The remainder of this paper is organized as follows: section 2 provides an introduction to the B Method, section 3 describes the system model informally, section 4 presents an abstract B model of transactions considering the database as single logical entity, section 5 presents a refinement of the abstract B model introducing details of replicated database, section 6 present some properties of system given as invariants and lastly section 7 concludes the paper.

2 B Method

The B Method [1, 12] is a model oriented state based method developed by Abrial for specifying, designing and coding software systems. The B Method provides a state based formal notation based on set theory for writing abstract models of systems. A system may be defined as an abstract machine. Abstract machines contains *sets, variables, invariants, initialization* and a set of *operations* defined on variables. The *sets* clause contains user defined sets that can be used in the rest of the machine. The variables describe the state of machine. The *invariants* are first order predicates and these invariants are to be preserved while updating the variables through the operations. The operations can have input and output parameters. Operation of machines are defined through generalized substitution which allow both *non deterministic* and *deterministic* assignments.

2.1 Event B

Event B [2, 3] is an event driven approach to system modelling based on B for developing distributed systems. This formal technique consists of the following steps :

- Rigorous description of abstract problem.
- Introduce solutions or details in refinement steps to obtain more concrete specifications.
- Verifying that proposed refinements are valid.

In Event B operations are referred to as events which occur spontaneously rather then being invoked. The events are guarded by predicates and these guards may be strengthened at each refinement step. The state variables are modified by a set of events. The invariants state properties that must be *satisfied* by variables and *maintained* by activation of events.

The development methodology supported in B Method is stepwise refinement. This is done by defining an abstract formal specification and successively refining it to an implementable specification through a number of correctness preserving steps. At each refinement step more concrete specifications of a system are obtained. The B Method requires the discharge of proof obligations for *consistency checking* and *refinement checking*. The B Tools Atelier B [31], Click'n'Prove [4], B-Toolkit [13] provide an environment for generation and discharge of proof obligations required for *consistency checking* and *refinement checking*. Applications of the B method to distributed system may be found in [3, 9–11, 25, 32]. In this work we have used Click'n'Prove.

2.2 B Notations

In this section we present some B notation frequently used in our model (Table-1). A more detailed explanation of these may be found in [1, 28]. Let A and B be two sets, then notation $\leftrightarrow$ defines the set of relations between A and B as

$$\mathrm{A} \leftrightarrow \mathrm{B} = \mathbb{P}(\mathrm{A} \times \mathrm{B})$$

where $\times$ is cartesian product of A and B. A mapping of element $\mathrm{a} \in \mathrm{A}$ and $\mathrm{b} \in \mathrm{B}$ in a relation $\mathrm{R} \in \mathrm{A} \leftrightarrow \mathrm{B}$ is written as $\mathrm{a} \mapsto \mathrm{b}$. The *domain* of a relation $\mathrm{R} \in \mathrm{A} \leftrightarrow \mathrm{B}$ is the set of elements of A that R relates to some elements in B defined as

$$\mathrm{dom(R)} = \{ \mathrm{a} \mid \mathrm{a} \in \mathrm{A} \land \exists\, \mathrm{b}.\ (\mathrm{b} \in \mathrm{B} \land \mathrm{a} \mapsto \mathrm{b} \in \mathrm{R}) \}$$

Similarly, the *range* of relation $\mathrm{R} \in \mathrm{A} \leftrightarrow \mathrm{B}$ is defined as set of elements in B related to some element in A :

$$\mathit{ran}(\mathrm{R}) = \{ \mathrm{b} \mid \mathrm{b} \in \mathrm{B} \land \exists\, \mathrm{a}.\ (\mathrm{a} \in \mathrm{A} \land \mathrm{a} \mapsto \mathrm{b} \in \mathrm{R}) \}$$

A relation $\mathrm{R} \in \mathrm{A} \leftrightarrow \mathrm{B}$ can be projected on a domain $\mathrm{U} \subseteq \mathrm{A}$ called *domain restriction*($\lhd$) defined as

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

A domain anti-restriction $\mathrm{U} \mathbin{⩤} \mathrm{R}$ removes all mappings whose first element is in U. The *domain anti-restriction* is defined as

$$\mathrm{U} \mathbin{⩤} \mathrm{R} = \{ \mathrm{a} \mapsto \mathrm{b} \mid \mathrm{a} \mapsto \mathrm{b} \in \mathrm{R} \land \mathrm{a} \notin \mathrm{U} \}$$

The *Relational image* R[U] where $\mathrm{U} \subseteq \mathrm{A}$ is defined as

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

dom(R)	domain of relation R	ran(R)	range of relation R
$\lhd$	domain restriction	$\mapsto$	mapping
$\Leftarrow\!\!\!\!+$	relational overide operator	R[A]	relational image of R over set A
$\mathbb{P}_1$	non empty power set	$\mathbb{P}$	power set
$\nrightarrow$	partial function	$\rightarrow$	total function

Table 1. Some frequently used B Notations

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

$$R_0 \Leftarrow\!\!\!\!+ R_1 = (\ \mathrm{dom}(R_1) \ntriangleleft R_0) \cup R_1$$

A *function* is a relation having some special properties. A *Partial Function* from set A to B (A $\nrightarrow$ B) is a relation which relates an element in A to *at most* one element in B. A partial function f $\in$ A $\nrightarrow$ B satisfies

$$\forall\ (a,b_1,b_2).(a \in A \wedge b_1 \in B \wedge b_2 \in B$$
$$\Rightarrow ((\ a \mapsto b_1 \in f \wedge a \mapsto b_2 \in f) \Rightarrow b_1{=}b_2)).$$

Similarly a *Total Function* f $\in$ A $\rightarrow$ B is a partial function where dom(f)=A.

3 System Model

In this section we present an informal model of a distributed database. Our system model consist of a sets of sites and data objects. The distributed database consists of sets of objects stored at different sites. Users interact with the database by *starting transactions*. The data objects are assumed to be replicated across all sites. The *Read Anywhere Write Everywhere* [7, 24] replica control mechanism is considered for updating replicas. We consider the case of full replication and assume all data objects are updateable. A transaction is considered as a sequence of read/write operations executed atomically, i.e., a transaction either *commit* or *abort* the effect of all database operations. The *one copy equivalence* requires that multiple copies of object must appear logically as a single object to a transaction, i.e., from the users perspective, using the replicated database must behave as a centralized database [7].

3.1 Transaction Types

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

- *Read-Only Transactions* : These transaction 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, all replicas of data objects maintained at all sites must be updated. In case of abort, none of the sites update the data object.

To illustrate the two cases, consider Read Only transaction T_1 and Update transaction T_2 defined over set of data object O_1 and O_2 respectively. The read-only transaction T_1 issues a sequence of *read* operations over data objects in O_1 and update T_2 issues a sequence of *read* or *write* operation over data objects in O_2. A transaction is termed an update transaction if it issues *at least* one *write* operation.

3.2 Commitment of Transactions

Transactions in *Read Anywhere, Write Everywhere* replica control execute as follows. A read transaction may be submitted at any site and its execution remain confined to that site. However, an update transaction is executed globally and the global commit decision of an update transaction is determined by the commit decisions of the components of the update transaction at participating sites. Consider an update transaction T_i submitted at a site S_i called the *coordinator* site. Since T_i issues *write* operations, the coordinator site of T_i broadcasts its operations to all participating sites. Participating site S_j, upon receipt of request from coordinator S_i, begins a subtransaction T_{ij}. Each T_{ij} is executed following a two phase locking scheme at participating site S_j. Coordinating site S_i waits for the intention to commit or abort from each participating site. An intention to either *commit* or *abort* a sub transaction is sent by S_j to coordinator S_i. The decision of a global commit or abort is taken at the coordinator site. Thus the decision of a global commit or abort of an update transaction is taken in the framework of a two phase commit protocol. This mechanism ensures *atomic commitment* of update transactions even in case of site failures. The commit or abort decision of an update transaction T_i is taken as follows,

- T_i *commits* if *all* T_{ij} *commit* at S_j.
- T_i *aborts* if *any* T_{ij} *aborts* at S_j.

3.3 Degree of Isolation

We consider the situations where read-only and update transactions may be submitted to any site. In order to ensure correct serial execution of transactions they must execute in isolation. Various degrees of isolation, e.g., *no isolation*, *read-write isolation* and *general isolation* are discussed in [15] in the context of replication. 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 general isolation. Two 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 O_i and O_j respectively and $O_i \cap O_j \neq \varnothing$. General isolation between T_i and T_j means no operation of T_i may be interleaved with operations of T_j.

4 Abstract Model of Transactions in B

The abstract data model of transactions is given in Fig.1 as a B machine. The operations of the machine are shown in Fig.2. The abstract model maintains a

```
MACHINE      replica11

DEFINITIONS  PartialDB  == (OBJECT ⇸ VALUE) ;
              UPDATE  ==  (PartialDB ⇸ PartialDB) ;
             ValidUpdate (update,readset) ==  ( dom(update)= readset → VALUE
                                              ∧ ran(update) ⊆ readset → VALUE )

SETS         TRANSACTION; SITE ;  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 → ℙ(OBJECT)
             ∧∀t.(t∈ trans ⇒ ValidUpdate (transeffect(t), transobject(t)) )

INITIALISATION   trans :=∅    || transstatus :=∅  || transeffect :=  {}
             || transobject :={}  || database :∈  OBJECT → VALUE
```

Fig. 1. Abstract Model of Transactions in B

notion of a *central* or *o*ne 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, in this paper, 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 $readset \subseteq OBJECT$. It will read from a partial projection of the database (pdb) on to $readset$, i.e.,

$$pdb = readset \lhd database$$

If it is an update transaction it will write to a subset of *readset* and the new values of the objects to be written may depend on the existing values of the objects in *readset.* Let the set of objects to be written be *writeset* where *writeset* $\subseteq$ *readset.* So we model an update to a database as function that takes a partial database (representing the current values of the objects in *readset*) and yields a partial database (representing the new values of the objects in *writeset*). A transaction is a read only-transaction if its $writeset = \varnothing$. 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,

$$UPDATE \triangleq PartialDB \nrightarrow PartialDB$$
$$\text{where } PartialDB \triangleq OBJECT \nrightarrow VALUE$$

An update function *update* maps a partial database (*pdb1*) where *pdb1* = *readset* $\lhd$ *database* to another partial database (*pdb2*) where *dom(pdb2)* = *writeset*. The update function *update* updates the database as follows,

$$database := database \mathbin{\lhd\!\!\!+} update\ (pdb1)$$

We say that an update *update* is valid w.r.t a set of objects *readset* whenever,

$$\begin{aligned} &dom(update) = readset \rightarrow \text{VALUE} \\ \wedge\ &ran(update) \subseteq readset \rightarrow \text{VALUE} \end{aligned}$$

Our model of database updates is sufficiently general to model atomic series of read-only and update transactions. A brief description of machine is given below.

- TRANSACTION, SITE, OBJECT and VALUE are defined as a deferred sets. The TRANSSTATUS is an enumerated set containing value *COMMIT,ABORT and PENDING*. These values are used to represent the global status of transactions.
- The database is represented by a variable *database* as a total function from OBJECT to VALUE. A mapping (o $\mapsto$ v) $\in$ *database* indicates that object *o* has value *v* in the database.
- The variable *trans* represents set of *started* transactions. The variable *transstatus* maps each started transaction to TRANSSTATUS.
- The variable *transobject* is a total function which maps a transaction to a set of objects. The set *transobject(t)* represents the set of data objects read or written by a transaction *t*.
- The variable *transeffect* is a total function which maps each transaction to an object update function *UPDATE*. An object update function *f* $\in$ *PartialDB* $\rightarrow$ *PartialDB* maps data objects and their corresponding value to updateable objects and update values.
- A transaction *t* is a read-only transaction if *ran(transeffect(t))* = $\{\varnothing\}$.
- The invariant *t* $\in$ *trans* $\Rightarrow$ *ValidUpdate(transeffect(t),transobject(t))* indicate that all objects to be updated must be a part of transaction objects.

4.1 Starting a Transaction

The event *StartTran(tt)*, given in Fig.2, models starting a new transaction *tt*. The guards given in the *WHEN* statement prevents restarting *tt*. The *ANY* statement sets the variables *transobject(tt)* and *transeffect(tt)* so that *transobject(tt)* is a non empty set of objects and *transeffect(tt)* is some valid update on the objects. A transaction *tt* is considered as *read-only* if the *ran(transeffect(tt))* is set to an empty set and it is considered an *update transaction* if *ran(transeffect(tt))* contains *at least* one mapping of the form (o$\mapsto$v). The status of transaction *tt* is set to *PENDING*.

```
StartTran( tt∈ TRANSACTION ) ≅
    WHEN  tt ∉ trans
    THEN   trans := trans ∪{tt}
           ||  transstatus(tt) := PENDING
           ||  ANY updates,objects
             WHERE  updates ∈ UPDATE  ∧ objects ∈ ℙ1 (OBJECT)
                    ∧ ValidUpdate (updates,objects)
             THEN transobject(tt) := objects || transeffect(tt) := updates
    END        END;

CommitWriteTran( tt∈ TRANSACTION) ≅
    WHEN  tt ∈ trans ∧ transstatus(tt) =PENDING ∧ ran(transeffect(tt)) ≠ {∅}
    THEN   transstatus(tt) := COMMIT
           ||  LET pdb BE pdb =  transobject(tt) ◁ database IN
             database  := database <+ transeffect(tt)(pdb) END
    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;
```

Fig. 2. Operations of abstract transaction model

4.2 Commitment and Abortion of Update Transactions

The event *CommitWriteTran(tt)* models *commitment* of an update transaction. As a consequence of the occurrence of this event, the abstract *database* is updated with the effects of the transaction and its status is set to *commit*.

The event *AbortWriteTran(tt)* models *abort* of an update transaction. As a consequence of occurrence of this event, the transaction status is set to *abort* and its effects are not written to the database. The B specification of these operations are given in Fig.2.

4.3 Commitment of Read Only Transactions

The event *ReadTran(tt)*, given in Fig.2, models *commitment* of a read-only transaction *tt*. The guards of this events ensure that 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.

```
REFINEMENT   replica22
REFINES       replica11

SETS            SITETRANSSTATUS={commit,abort,precommit,pending}

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

INVARIANT      activetrans ∈ SITE ↔ trans
             ∧ coordinator ∈  TRANSACTION → SITE
              ∧ sitetransstatus ∈ trans ⇸ (SITE ⇸SITETRANSSTATUS)
              ∧ replica ∈ SITE → ( OBJECT → VALUE)
             ∧ freeobject ∈ SITE ↔ OBJECT

INITIALISATION    trans :=∅  || transstatus :=∅  || activetrans :=∅
              ||  coordinator :∈  TRANSACTION → SITE
              || sitetransstatus :=∅     ||  transeffect := {}   || transobject := {}
              || freeobject := SITE * OBJECT
              ||   ANY  data  WHERE  data ∈ OBJECT → VALUE
                  THEN replica :=  SITE *{data} END
```

Fig. 3. Initial part of Refinement

5 Refinement of Transactional Model

The initial part of the refinement of the abstract model is given in Fig.3. The B specification of events of the refinement are introduced later in this section. The abstract B model of a transaction maintains a notion of abstract *central database*. In the refinement the notion of *replicated database* is introduced. It may be noted that in the abstract model given in Fig.2, an update transaction perform updates on an abstract central database whereas in the refined model, an update transaction updates replicas at each sites. Similarly, a read-only transaction reads the data from the replica at the site of submission of the transaction. A brief description of the refinement is given below.

- New variables *coordinator*, *replica*, *activetrans*,*freeobject* and *sitetranstatus* are introduced in refinement. The variable *coordinator* is defined as a total function from *TRANSACTION* to *SITE*. A mapping of form $(t \mapsto s) \in$ *coordinator* imply that site s is a 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 $(s \mapsto (o \mapsto v)) \in$ *replica* indicate that site s currently has value v for object o.
- Variable *activetrans* keeps a record of transactions running at various site. A mapping $(s \mapsto t) \in$ *activetrans* indicate 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.

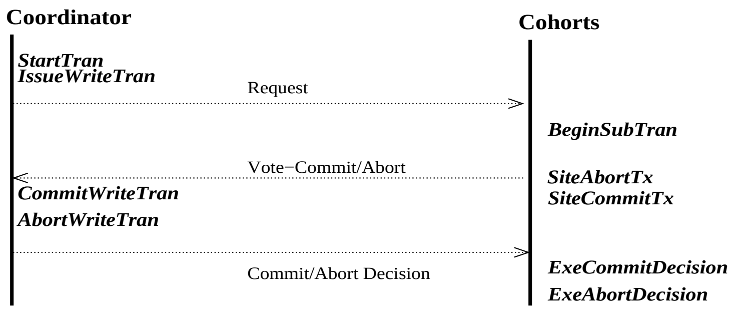


Fig. 4. Events of Update Transaction

- The variable *sitetransstatus* maintains the status of all started transaction at various sites. A mapping of form $(t \mapsto (s \mapsto \text{commit})) \in$ *sitetransstatus* indicate that t has committed at site s.
- The new events such as *IssueWriteTran*, *BeginSubTran*, *SiteAbortTx*, *SiteCommitTx*, *ExeAbortDecision* and *ExeCommitDecision* are introduced in operations.

An informal logical ordering of the occurrence of various events of the refinement for an update transaction is given in Fig.4. These events are triggered as given below.

i The events *StartTran(tt)* and *IssueWriteTran(tt)* occur at the coordinating site of *tt*. Upon *issue* of *tt* at the coordinator, the coordinator broadcast request messages to all sites to start a subtransaction which updates the replica at that site.

ii The event *BeginSubTran(tt,ss)* starts a subtransaction 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 commitment or abortion of an update transaction *tt* at *ss*. Participating sites communicate their decision to the coordinator of tt by sending either a *Vote-Commit* or *Vote-Abort* message.

iii Upon receipt of a *Vote-Commit/Abort* message, the coordinator site triggers either the event *AbortWriteTran(tt)* or *CommitWriteTran(tt)*. The coordinating site communicates its decision by broadcasting a *commit/abort decision* message. Upon receipt of a *commit/abort decision* message from the coordinator, the participating site *ss* decides to abort or commit *tt* by triggering either *ExeAbortDecision(ss,tt)* or *ExeCommitDecision(ss,tt)* event.

With reference to update transactions, some of the events are coordinator site events while others are participating sites event as shown in Fig.4. A brief description of events of refinement is given below.

```
StartTran(tt) ≙
WHEN    tt ∉ trans
THEN    trans := trans ∪{tt}   ||  transstatus(tt) := PENDING
        || sitetransstatus(tt)(coordinator(tt)) := pending
        ||  ANY updates,objects
            WHERE  updates ∈ UPDATE  ∧ objects ∈ ℙ1 (OBJECT)
                        ∧ ValidUpdate (updates,objects)
            THEN transobject(tt) := objects || transeffect(tt) := updates
END         END;

IssueWriteTran(tt) ≙
WHEN   tt ∈ trans  ∧ (coordinator(tt) ↦ tt) ∉ activetrans  ∧  transstatus(tt)=PENDING
       ∧ ran(transeffect(tt))≠{∅}  ∧ transobject(tt) ⊆ freeobject[{coordinator(tt)}]
       ∧ ∀tz.(tz ∈ trans ∧ (coordinator(tt) ↦tz)∈ activetrans
                 ⇒ transobject(tt) ∩ transobject(tz)=∅)
THEN   activetrans := activetrans ∪ {coordinator(tt)↦tt}
       || sitetransstatus(tt)(coordinator(tt)):= precommit
       || freeobject := freeobject - {coordinator(tt)}*transobject(tt)
END;
```

Fig. 5. Refinement : Coordinator Site Events-I

5.1 Starting and Issuing a Transaction

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

$tz \in \text{trans} \wedge (\text{coordinator(tt)} \mapsto tz) \in \text{activetrans}$
$\Rightarrow \text{transobject}(tt) \wedge \text{transobject } (tz) = \varnothing$

The B specification for events *StartTran(tt)* and *IssueWriteTran(tt)* of the refinement are given in Fig.5.

5.2 Commitment and Abortion of Update Transactions

Refined B specifications for the commit and abort events of update transaction *tt* are given in Fig.6. An update transaction *tt* globally commits only if all participating sites are ready to commit it, i.e., it has *pre-committed* and *active* at all sites. Thus, following conditions must hold for each site *s* before committing update transaction *tt*.

- sitetransstatus(tt)(s)= precommit
- (s $\mapsto$ tt) $\in$ activetrans

As a consequence of the occurrence of the *commit* event at the coordinator, the replica maintained at the coordinator site is updated with the transaction

```
CommitWriteTran(tt) $\cong$
WHEN  $tt \in trans \wedge ran(transeffect(tt)) \neq \{\emptyset\} \wedge (coordinator(tt) \mapsto tt) \in activetrans$
          $\wedge\ transstatus(tt)=PENDING \wedge \forall s.(s \in SITE \Rightarrow sitetransstatus(tt)(s)= precommit)$
          $\wedge\ \forall s,o \cdot (s \in SITE \wedge o \in OBJECT \wedge o \in transobject(tt) \Rightarrow (s \mapsto o) \notin freeobject)$
          $\wedge\ \forall s.(s \in SITE \Rightarrow (s \mapsto tt) \in activetrans)$
THEN  $transstatus(tt) := COMMIT \ \|\ activetrans := activetrans - \{coordinator(tt) \mapsto tt\}$
          $\|\ sitetransstatus(tt)(coordinator(tt)) := commit$
          $\|\ freeobject := freeobject \cup \{coordinator(tt)\} * transobject(tt)$
          $\|$ LET pdb BE $pdb = transobject(tt) \lhd replica(coordinator(tt))$ IN
              $replica(coordinator(tt)) := replica(coordinator(tt)) \mathbin{\lhd\!\!\!-} transeffect(tt)(pdb)$ END
END;

AbortWriteTran(tt) $\cong$
WHEN  $tt \in trans \wedge ran(transeffect(tt)) \neq \{\emptyset\} \wedge (coordinator(tt) \mapsto tt) \in activetrans$
          $\wedge\ transstatus(tt)=PENDING \wedge \exists s.(s \in SITE \wedge sitetransstatus(tt)(s)= abort)$
THEN  $transstatus(tt) := ABORT \ \|\ activetrans := activetrans - \{coordinator(tt) \mapsto tt\}$
          $\|\ sitetransstatus(tt)(coordinator(tt)) := abort$
          $\|\ freeobject := freeobject \cup \{coordinator(tt)\} * transobject(tt)$
END;

$val \longleftarrow ReadTran(tt,ss) \cong$
WHEN  $tt \in trans \wedge transstatus(tt)=PENDING \wedge transobject(tt) \in freeobject[\{ss\}]$
          $\wedge\ ss = coordinator(tt) \wedge ran(transeffect(tt))=\{\emptyset\}$
THEN  $val := transobject(tt) \lhd replica(ss) \ \|\ sitetransstatus(tt)(ss) := commit$
          $\|\ transstatus(tt) := COMMIT$
END
```

Fig. 6. Refinement : Coordinator Site Events - II

effects, data objects held for transaction *tt* are declared *free* and the status of the transaction at the coordinator site is set to *commit.* Similarly, the guard of *AbortWriteTran(tt)* means that an update will abort if it has aborted at any participating site,i.e.,

$\exists\ s\ .(\ s \in \text{SITE} \wedge \text{sitetransstatus(tt)}(s)\text{= abort})$

Further Refinement of Commit Event The event *CommitWriteTran(tt)* can be further refined under following observations.

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

These observations can be included as invariants in a further refinement allowing the guards of the *CommitWriteTran(tt)* event to be simplified to

$\text{tt} \in \text{trans}$
$\wedge\ \text{ran(transeffect(tt))} \neq \{\emptyset\}$
$\wedge\ \text{transstatus(tt)=PENDING}$
$\wedge\ \forall\ s\ .\ (\ s \in \text{SITE} \wedge \text{sitetransstatus(tt)}(s)\text{= precommit})$

5.3 Read Only Transaction

The specifications of executing a read-only transaction is given in Fig.6. A *pending* read-only transaction *tt* returns the value of objects defined by *transobject(tt)* from the replica at its coordinator. The necessary condition for occurrence of this event is given below,

- $\text{transstatus(tt)=PENDING} \wedge \text{ran(transeffect(tt))}= \{\varnothing\}$
- $\text{transobject(tt)} \subseteq \text{freeobject}[\{\text{ss}\}]$

As a consequence of occurrence of this event, transaction *tt* read the objects from replica at site *ss* as $val := transobject(tt) \lhd replica(ss)$. It may be noted that in the abstract model given in Fig.2, a read-only transaction read the objects from abstract database as $val := transobject(tt) \lhd database$. In refinement checking, we need to prove that $(\text{ss} \mapsto \text{oo}) \in \text{freeobject} \Rightarrow \text{database(oo)}= \text{replica(ss)(oo)}$ to show that refinement is valid. This is explained further in section 6.

5.4 Starting a Sub-Transaction

The *BeginSubTran(tt,ss)* is an event of starting a subtransaction of *tt* at participating site *ss*. The specification of this event is given in Fig.7. The guard of

```
BeginSubTran(tt,ss) ≅
WHEN    tt ∈ trans  ∧ sitetransstatus(tt)(coordinator(tt)) = precommit
          ∧ (ss↦tt)∉ activetrans  ∧ ss ≠ coordinator(tt)  ∧ ran(transeffect(tt))≠{∅}
          ∧ transobject(tt) ⊆ freeobject[{ss}]  ∧ transstatus(tt)=PENDING
          ∧ ∀tz.(tz ∈ trans ∧ (ss ↦tz)∈ activetrans  ⇒ transobject(tt)∩ transobject(tz)=∅)
THEN    activetrans := activetrans ∪ {ss↦tt}
          ||  sitetransstatus(tt)(ss) := pending
          || freeobject := freeobject - {ss}*transobject(tt)
END;

SiteCommitTx(tt,ss) ≅
WHEN    (ss↦tt)∈ activetrans  ∧ sitetransstatus(tt)(ss)= pending
           ∧ ss ≠coordinator(tt)   ∧ ran(transeffect(tt))≠{∅}  ∧ transstatus(tt)=PENDING
THEN      sitetransstatus(tt)(ss) := precommit
END;

SiteAbortTx(tt,ss) ≅
WHEN    (ss↦tt)∈ activetrans ∧ sitetransstatus(tt)(ss)= pending
          ∧ ss ≠coordinator(tt)  ∧ ran(transeffect(tt))≠{∅}  ∧ transstatus(tt)=PENDING
THEN    sitetransstatus(tt)(ss) := abort
          || freeobject := freeobject ∪ {ss}*transobject(tt)
          || activetrans := activetrans -{ss ↦tt}
END;
```

Fig. 7. Refinement : Participating Site Events -I

```
ExeAbortDecision(ss,tt) ≘
WHEN    tt∈ trans  ∧  (ss↦tt)∈ activetrans   ∧ transstatus(tt) =ABORT
          ∧ ss ≠ coordinator(tt) ∧ ran(transeffect(tt))≠{∅}
THEN   sitetransstatus(tt)(ss):= abort   || activetrans := activetrans -{ss ↦tt}
          || freeobject := freeobject ∪ {ss}*transobject(tt)
 END;

ExeCommitDecision(ss,tt) ≘
WHEN     tt∈ trans  ∧  (ss↦tt)∈ activetrans   ∧ transstatus(tt) =COMMIT
           ∧ ss ≠ coordinator(tt) ∧ ran(transeffect(tt))≠{∅}
THEN    activetrans := activetrans -{ss ↦tt}     || sitetransstatus(tt)(ss) := commit
          || freeobject := freeobject ∪ {ss}*transobject(tt)
          || LET pdb BE pdb =  transobject(tt) ◁ replica(ss)  IN
               replica(ss)  :=  replica(ss) ⩤ transeffect(tt)(pdb) END
END;
```

Fig. 8. Refinement : Participating Site Events -II

BeginSubTran(tt) ensures that a sub transaction of *tt* is started at site *ss* when all active transactions *tz* running at *ss* are not in *conflict* with *tt* and transaction *tt* has precommitted at coordinator site of *tt*.

- (ss $\mapsto$ tz) $\in$ activetrans $\Rightarrow$ transobject(tt) $\wedge$ transobject(tz) $= \varnothing$
- sitetransstatus(tt)(coordinator(tt))= precommit

As a consequence of occurrence of this event, transaction *tt* becomes *active* at site *ss* and *sitetransstatus* of *tt* at *ss* is set to pending.

5.5 Pre-Commitment and Abortion of Subtransaction

A participating site *ss* can independently decide to either pre-commit or abort a subtransaction. The events *SiteCommitTx(tt,ss)* and *SiteAbortTx(tt,ss)*, given in Fig.7, model pre-committing or aborting a subtransaction 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 set all *objects* of transaction *tt* free and a subtransaction is removed from list of active transactions at that site. On occurrence of both of these events, a participating site communicates its decision to the coordinating site by sending a *Vote-Abort* or *Vote-Commit* message.

5.6 Implementing Coordinator Decision of Global Commit

A global commit or abort decision of a transaction is communicated by the coordinator site to participating sites. A transaction globally commits if it commits at all participating site. The event of *ExeCommitDecision(tt,ss) and ExeAbortDecision(tt,ss)* given in Fig.8 model commit and abort of *tt* at *ss* once the decision of global abort or commit has been taken by the coordinating site. In the case of global commit, each site updates its replica separately.

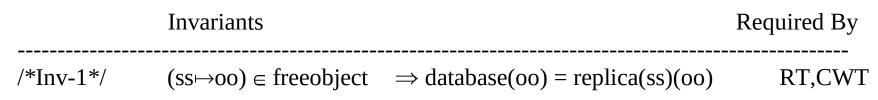
RT/ST	Read/StartTran	IWT	IssueWriteTran	CWT	CommitWriteTran
AWT	AbortWriteTran	BST	BeginSubTran	SAT	SiteAbortTx
SCT	SiteCommitTX	ECT	ExeCommitDecision	EAT	ExeAbortDecision

Table 2. Events Code

In our model messaging among the site is not dealt explicitly. Transaction may deadlock due to race conditions in replicated database. It is our assumption that ordered delivery of messages may be used to detect deadlock arising due to two simultaneous update requests from two different sites. B Specification of causal ordering of messages for fault tolerant transactions and their implementation with logical clock has been proposed in [32].

6 Gluing Invariants

The *One copy equivalence* consistency criterion requires us to prove that the refinement (replicated database) is a valid refinement of the abstract transaction model (abstract central database), i.e., a read-only transaction submitted at any site reads the correct value of the abstract database. We have replaced the abstract variable *database* in the abstract model by the variable *replica* in the refinement. An abstract machine is refined by applying the standard technique of data refinement. If a statement S that acts on variable a, is refined by another statement T that acts on variable b under invariants I then we write $S \sqsubseteq_I T$. The invariant I is called the Gluing Invariant and it defines the relationships between a and b. Replacing the abstract variable *database* in machine *replica11* by concrete variable *replica* in refinement *replica22* results in proof obligations generated by B tool. Initially, the only proof obligations that can not be proved involves the relationship between *database* and *replica*. These proof obligations were associated with the events *ReadTrans* and *CommitWriteTran*. In order to prove these we added the Gluing Invariants-I shown in Fig.9.



	Invariants	Required By
/*Inv-1*/	(ss↦oo) ∈ freeobject ⇒ database(oo) = replica(ss)(oo)	RT,CWT

Fig. 9. 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. When invariant Inv-1 is added to the refined machine, the B tool generates further proof obligations associated with several other events. Discharging these additional proof obligations required further invariants given at Invariants-II shown in Fig.10. A brief description of these invariants is given in following steps.

	Invariants	Required By
/*Inv-2*/	$(\text{coordinator}(t) \mapsto t) \in \text{activetrans} \wedge o \in \text{transobject}(t)$ $\Rightarrow \text{database}(o) = \text{replica}(\text{coordinator}(t))(o)$	AWT,CWT,EAD,ECD
/*Inv-3*/	$(s \mapsto t1) \in \text{activetrans} \wedge (s \mapsto t2) \in \text{activetrans}$ $\wedge\ \text{transobject}(t1) \cap \text{transobject}(t2) \neq \varnothing \Rightarrow t1=t2$	ST,IWT,BST
/*Inv-4*/	$\text{transstatus}(t)= \text{COMMIT} \wedge (s \mapsto t) \in \text{activetrans}$ $\wedge\ o \in \text{dom}(\text{transeffect}(t)(\text{transobject}(t) \lhd \text{replica}(s)))$ $\Rightarrow \text{database}(o)= \text{transeffect}(t)(\text{transobject}(t) \lhd \text{replica}(s))(o))$	CWT,AWT,RT,SCT

Fig. 10. Invariants -II

- If a transaction t is *active* at its coordinator then all transaction objects o $\in$ transobject(t) in the abstract database have the same value in the replica at coordinator(Inv-2).
- If two conflicting transaction t_1 and t_2 are active at a site s, they must represent the same transaction, i.e., t_1=t_2 (Inv-3). This also implies that two different conflicting transaction can not be *active* at the same time at the same site s.
- For a committed transaction t which is *active* at one of the site s, the new values of objects defined by *transeffect(t)* reflects the value of those objects in the abstract database(Inv-4).

Following a similar approach, in order to preserve the Invariants in Fig.10, we have to prove another set of invariants given by Invariants-III in Fig.11. The brief description of invariants in Fig.11 are given below.

- For a committed transaction t which is active at any participating site s, the value of the objects which are given in set *transobject(t)* but not to be updated by t, the site s reflects the value of those objects as in abstract database(Inv-5).

	Invariants	Required By
/*Inv-5*/	$\text{transstatus}(t)= \text{COMMIT}$ $\wedge\ o \in \text{transobject}(t) \wedge (s \mapsto t) \in \text{activetrans}$ $\wedge\ o \notin \text{dom}(\text{transeffect}(t)(\text{transobject}(t) \lhd \text{replica}(s)))$ $\Rightarrow \text{database}(o)= \text{replica}(s)(o)$	CWT,AWT,BST,ECD SAT,SCT
/*Inv-6*/	$\text{transstatus}(t)=\text{ABORT}$ $\Rightarrow \text{sitetransstatus}(t)(\text{coordinator}(t))= \text{abort}$	AWT,EAD,ECD,ST
/*Inv-7*/	$\text{transstatus}(t)= \text{COMMIT}$ $\Rightarrow \text{sitetransstatus}(t)(\text{coordinator}(t))= \text{commit}$	CWT,AWT,EAD,ECD,ST

Fig. 11. Invariants -III

- If a transaction t commits or aborts globally, it must have either committed or aborted locally at its coordinator as the case may be (Inv-6,7).

Finally the B tools generate more proof obligations to preserve Invariant-III which in turns requires Invariants-IV shown in Fig.12. The brief description of Invariants-IV is given below.

	Invariants	Required By
/*Inv-8*/	$\text{transstatus}(t) \neq \text{COMMIT} \wedge (s \mapsto t) \in \text{activetrans}$ $\wedge\ o \in \text{transobject}(t) \Rightarrow \text{database}(o) = \text{replica}(s)(o)$	CWT,AWT,EAD, ECD,RT
/*Inv-9*/	$\text{transstatus}(t) \neq \text{PENDING} \wedge \text{ran}(\text{transeffect}(t)) \neq \{\varnothing\}$ $\Rightarrow (\text{coordinator}(t) \mapsto t) \notin \text{activetrans}$	ST,IWT, SAT,SCT

Fig. 12. Invariants -IV

- A transaction t which has not globally *committed* but is still *active* at some site s, then for all objects o $\in$ transobject(t), value of object o at replica of s represent its value in abstract database(Inv-8).
- The Inv-8 refers to situations where a transaction is not committed. Therefore, it also involves situations where the transaction global status is either *PENDING* or *ABORT*.
- An update transaction whose status is not *PENDING* must not be *active* at its coordinator site(Inv-9). 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 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 four 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 refinement is a valid refinement of the abstract specification.

7 Conclusions

In this paper we have presented a formal approach to the 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 refinement of the abstract model, we introduced the notion of replicated database. The replica control mechanism presented in the paper 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 one replica. The various events given in the B 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.

Distributed algorithms [21] are difficult to verify and their verification has long been an issue of study. The work reported in [16] applies formal modelling to a replica control strategy and considers proof of correctness. They use I/O automata to model an algorithm and then prove properties about all trace behaviors of the automation. Instead of proving trace properties, we prove that our model of the algorithm is a correct refinement of a abstract model of single copy database. Also [16] does not consider transaction failures at sites.

The system development approach considered is based on Event B, which facilitates incremental development of dependable systems. The work was carried out on the B4Free 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. These proofs helped us to understand the complexity of problem and the correctness of the solutions. They also helped us to discover new system invariants providing a clear insight to the system. Our experience with this case study strengthens our believe that abstraction and refinement are valuable technique for modelling complex distributed system.

References

1. J R Abrial. *The B Book : Assigning Programs to Meaning*, Cambridge University Press,1996.
2. J R Abrial. *Extending B without changing it.* (For Distributed System).Proc. of 1st Conf. on B Method, pp 169-191, 1996
3. J R Abrial, D Cansell, D Mery. *A Mechanically Proved and Incremetal development of IEEE1394 Tree Identify Protocol.* Formal Aspect of Computing, Vol 14, PP215-227, 2003
4. J R Abrial, D Cansell : *Click'n'Prove - Interactive Proofs within Set Theory*,2003
5. D Agrawal, G Alonso, I Stanoi. *Exploiting Atomic Broadcast in Replicated Database.* Proc. of Europar97, 1997
6. O Babaoglu, A Bartoli, G Dini. *Replicated file management in large scale distributed system.* Proc. of 8th Intl. workshop on Distributed Algorithms. WDAG94,pp1-16,LNCS,Springer,1994
7. P A Bernstein, V Hadzilacos, N Goodman. *Concurrency Control and Recovery in Database System.* Addision Wesley,1987
8. K P Birman, A Schiper, P Stephenson. *Lighweight causal and atomic group multicast.* ACM Transaction on Computer System,Vol9,No 3,pp 272-314, 1991.
9. M Butler. *On the use of Data Refinement in the Development of Secure Communications Systems.* Formal Aspects of Computing, 14 : 2-34,2002
10. M Butler. *An Approach to Design of Distributed Systems with B AMN.* Proc. 10th Int. Conf. of Z Users: The Z Formal Specification Notation (ZUM), LNCS 1212, pp. 223-241,1997
11. M Butler, M Walden. *Distributed System Development in B.* Proc. of Ist Conf. in B Method, Nantes, pp155-168,1996
12. D Cansell, D Mery. *Foundations of B Method.* Computing and Informatics, Vol 22,1-31,2003

13. B Core UK Ltd. *B-Toolkit Manuals*,1999
14. X Défago, A Schiper, Péter Urbán. *Total order broadcast and multicast algorithms: Taxonomy and survey.* ACM Computing Surveys (CSUR), Volume 36, Issue 4, pp 372-421, Dec 2004.
15. R Ekwall, A Schiper. *Replication : Understanding the advantage of atomic broadcast over quorum systems.* Journal of Universal Computer Science, Vol 11, No 5, pp 703-711, 2005
16. A Fekete, M Frans Kaashoek, N Lynch. *Implementing Sequentially Consistent Shared Objects using Broadcast and Point-To-Point Communication. 15th IEEE Conf.Distributed Computing System*,ICDCS95, 1995.
17. J Gray, A Reuter. *Transaction Processing : Concepts and Technique*, Morgan Kaufmann, 1993.
18. J Holliday. *Replicated Database Recovery using Multicast Communication. IEEE Intl. Symposium on Network Computing and Application*,NCA2001, pp 104-107,2001
19. M Jandl, A Szep, R Smeikal, K M Goeschka. *Increasing avialability by sacrificing data integrity.* Proc. 38 Hawaii Intl. Conf. on System Sciences, 2005
20. B Kemme, G Alonso. *A New Approach to developing and implementing eager database replication protocols. ACM Transaction on Database System*, Vol 25, Issue 3, pp 333-379, 2000.
21. Nancy A Lynch. *Distributed Algorithms*, Morgan Kaughman,1996
22. P Melliar, Y Amir, L Moser, V Agrawala. *Broadcast protocols for Distributed Systems*, IEEE Transactions on Parallel and Distributed System,1(1), pp17-25,1990
23. L Moser, P Mellier, D Agrawal, R Budhia, C Papadopoulos, *TOTEM : A fault tolerant multicast group communication*, Communication of ACM,39,4, PP 54-63, 1996
24. M T Ozsu, P Valduriez. *Principles of Distribted Database Systems.* Prentice Hall, 1999
25. A Rezazadeh, M Butler. *Some Guidelines for formal developement of web based application in B Method.* Proc. of 4th Intl. Conf. of B and Z users, Guildford, LNCS, Springer, pp 472-491, April 2005.
26. M. Patino-Martinez, R. Jimenez-Peris, B. Kemme, G. Alonso. *Consistent Database Replication at the Middleware Level.* ACM Transactions on Computer Systems (TOCS).no. 4, vol. 23, pp. 375-423, Nov. 2005
27. B Silaghi, P Keleher, B Bhattacharjee. *Multi-Dimensional Quorum Sets for Red-Few Write-Many Replica Control Protocols.* In Proc. of the 4th CCGRID/GP2PC Chicago, IL, April 2004.
28. S Schneider. *The B-Method.* Palgrave Publications, 2001.
29. A Silberschatz, H Korth, S Sudarshan . *Database System Concepts*, McGrawHill, 2001
30. I Stanoi, D Agrawal , A.El Abbadi. *Using Broadbast Primitives in Replicated Data.* Proceddings of 18 IEEE Intl. Conf. on Distributed Computing System,ICDCS98,pp 148-155,1998
31. Steria- Atelier-B User and Reference Manuals, 1997
32. D Yadav, M Butler. *Application of Event B to Global Causal Ordering for Fault Tolerant Transactions.* Proc. of REFT 2005, Newcastle upon Tyne, pp 93-103, 2005.