Proceedings of the Federated Conference on
Computer Science and Information Systems pp. 1397–1401
ISBN 978-83-60810-51-4

Modeling of Multiversion Concurrency Control System Using Event-B

Raghuraj Suryavanshi
Institute of Engineering and Technology
GBTU, Lucknow 226021, INDIA
Email: suryavanshi.cse@ietlucknow.edu

Divakar Yadav
Faculty of Mathematics and Computer Science
South Asian University, New Delhi 110067, India
Email: dsyadav@cs.sau.ac.in

***Abstract*—Concurrency control in a database system involves the activity of controlling the relative order of conflicting operations, thereby ensuring database consistency. Multiversion concurrency control is timestamp based protocol that can be used to schedule the operations to maintain the consistency of the databases. In this protocol each write on a data item produces a new copy (or version) of that data item while retaining the old version. A systematic approach to specification is essential for the production of any substantial system description. Formal methods are mathematical technique that provide systematic approach for building and verification of model. We have used Event-B as a formal technique for construction of our model. Event-B provides complete framework by rigorous description of problem at abstract level and discharge of proof obligations arising due to consistency checking. In this paper, we outline formal construction of model of multiversion concurrency control scheme for database transactions using Event-B.**

***Index Terms*—Database system, Formal Method, Transaction, Event-B, Verification, Multiversion.**

I. Introduction

CONCURRENCY control is the activity of coordinating concurrent access to a database while preserving the consistency of the data. A good concurrency control mechanism should permit parallel execution of transactions to achieve high degree of concurrency. The concurrency control techniques are divided into two broad categories: Optimistic concurrency control and Pessimistic concurrency control [1]. Optimistic algorithms assumes that conflict is rare therefore, it delay the synchronization of transaction until transactions are near to their completion, whereas Pessimistic algorithms synchronize the concurrent execution of transactions early in their execution life cycles. Pessimistic algorithm are further categorized into lock-based and timestamp-based algorithm [1], [2].

Multiversion concurrency control is pessimistic time-stamp based algorithm [2], [3], [4]. In this algorithm the updates of data items do not modify the database but each write operation creates a new version of data item while retaining the old version. Each version has three fields read-timestamp, write-timestamp and its value. The data items are accessed through transactions. A transaction may be read only transaction or update transaction. Each transaction is assigned a unique time stamp by the system when it is submitted to the system.

In this paper, a model of multiversion concurrency control system is outlined. We have considered a model of centralized system where a transaction may read or create the version of data item. The model contains a *Submit-Transaction* event that models the event of submission of a transaction. In this event a unique time stamp is assigned to transaction. The events *Read-Operation, Write-Operation and Abort-Trancastion* models the event of reading data item value, creation of new version, abortion of transaction respectively. The transaction that read the value of any data item selects the version which has highest write-timestamp value and less than transaction-timestamp. After reading the version value, the read-timestamp of that version is set to as largest of current read-timestamp and transaction-timestamp. For the update transactions, the transaction selects the version which has highest write-timestamp value and less than transaction-timestamp, and it must also be ensured that selected version has not already been read by some younger transaction i.e., read-timestamp of version should be less than transaction-timestamp. If the operation is permitted a new version of that data item is created whose read-timestamp and write-timestamp is same as transaction timestamp.

The remainder of paper is organized as follows: Section 2 briefly outline modeling approach and B notations, Section 3 describes system model and informal description about events, Section 4 presents abstract model of multiversion concurrency control scheme for database system. Finally, Section 5 concludes the paper.

II. Modeling Approach

A functional specification of system describes its behavior. More specifically, it describes the interactions that the system offer to its user. A specification contains significant information about the system. The difficulties in specification is managing the large volume of detailed system that is required to formulate an accurate specification. A systematic approach to specification is essential for the production of any substantial system description. The B Method [5], [6], [7] offers one such approach. It represents the complete framework by mathematical development of a System. Event-B [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], a variant of B, is a formal technique that consists of describing rigorously the problem in an abstract model, introducing solutions or design details in the refinement steps to obtain more concrete specifications, and verifying that proposed solutions are correct. An Event-B model is composed of two constructs, machine and context

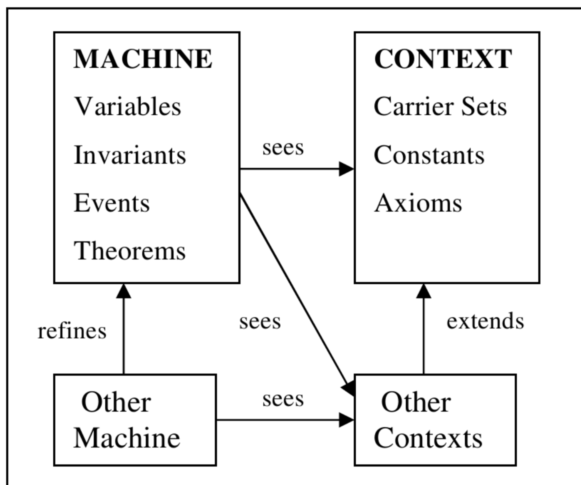

Fig. 1. Relationship between Machines and Contexts

(see Fig. 1). Machines represents the dynamic behavior of the model, contain system variables, invariants, Theorems, and events. The machine also maintains some local state information through its variables. Variables are represented using mathematical constructs such as sets, binary relations, functions, numbers, etc. The variables are constrained by the invariants. The invariant clause of machine provides the information about the variables. The values of variable changes as the machine executes, however the invariant describes the properties of variables that must be satisfied as execution progresses. The theorem of machine must follow from the context and the invariants of that machine. An event is made up of three elements; its name, guards and the actions. The guards are the predicates that must be satisfied for the event to trigger. An action is an assignment statement to a state variable and is achieved by a generalized substitution. The initial position of the model is shown by initialization event. We can add new variables, invariants and events in the refinement step. The purpose of refinement step is to provide implementations of more concrete specifications. The Context clause of machine defines the static part of model. It contains carrier sets, constants, axioms. The context may be seen by machine directly or indirectly. There may exist several relationships between machines and contexts. A machine can be a refinement of one and only one machine. This refined machine contains more concrete specifications of model. A context can extend one or more contexts. A machine can see several contexts.

The B Method requires the discharge of proof obligations for consistency checking and refinement checking. There are several B tools such as Rodin [16], [17], Click'n'Prove [18], Atelier B [19], B-Toolkit [20] that provide an environment for writing specifications and discharging proof obligations arising due to consistency and refinement checking. In this work, we have used Rodin platform. It is an open extensible tool for specification and verification of Event-B. It contains modeling element like event, variables, invariants and components like context and machines. It is embedded by various plugins such as proof-obligation generator, model checkers, provers, UML transformers, etc.

Event-B notations are set theoretic notations. The syntax and detail description of notations are outlined in [10]. Some of the notation are explained here.
Let A and B be two sets, then the relational constructor ($\leftrightarrow$) defines the set of relations between A and B as :

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

where $\times$ is cartesian product of A and B.
The *Relational image* R[U] where U⊆A is defined as:

$$R[U] = \{b \mid \exists a \cdot a \mapsto b \in R \wedge a \in U\}$$

A *function* is a relation with certain restrictions. The function may be a partial function ($\nrightarrow$) or a total function($\rightarrow$). 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 *total function* from set A to B ($A \rightarrow B$) is a partial function where *dom*(f)=A i.e. each element of set A is related to exactly one element of set B. Given $f \in A \nrightarrow B$ and $a \in$ *dom*(f), *f(a)* represents the unique value that a is mapped to by *f*.

III. System Model

We have considered multiversion concurrency control system to coordinate the concurrent execution of transaction. Each time when a transaction is submitted to the system a new time stamp is assigned to it. The transaction may read or write the value of data item. The write operation doesn't modify the value of data item. It creates a new version of data item. For each version of the data item, system maintains three values; the value of the version, read-timestamp of version which is the largest timestamp value of all the transactions that have successfully read that version and the write-timestamp which is the time stamp value of the transaction that created the version. Transaction timestamp is used to keep track of timestamp values of each version. The informal description of events are as follows:

1) *Submit Transaction:* When a fresh transaction is submitted to the system it creates an entry of transaction and the objects needed by it. This event also assigns a new timestamp value to the submitted transaction.
2) *Read Operation:* If the transaction wants to read the value of data item then it selects that version of data item which has maximum write-timestamp value that is less than transaction time stamp. After performing read operation the the read-timestamp of that version is set to as largest of current read-timestamp and transaction-timestamp.
3) *Write Operation:* The write operations are performed by update transaction. Upon activation of this event a version of data item is selected that has maximum write-timestamp value and less than the transaction time stamp, and at the same time it also ensures that selected version has not already been read by some other transaction whose time stamp is greater than transaction-time

stamp. This condition states that read-timestamp of selected version must be less than transaction-timestamp. The writing operation creates a new version having its value, read-timestamp and write-timestamp. The read and write-timestamp of new version is initialized with transaction timestamp.

4) *Abort Operation:* This event models the abortion of a update transaction. An update transaction is aborted when the selected version has already been read by some other transaction whose time stamp is greater than transaction-time stamp. In this case read-timestamp of selected version will be greater than transaction time stamp.

IV. Abstract Model of Multiversion Concurrency Control Scheme for Database Systems

We begin with the transactional model where a unique time stamp is assigned to the transaction when it is submitted to the system. In a context seen by machine, *transaction*, *dataitem*, *datavalue* are defined as deferred set. The transaction may read or write the value of *dataitem*. The *transactionstatus* is enumerated set containing the element *commit*, *abort*, *pending*. After the submission of transaction its status is set to be *pending*. For each data item *di* in the system, the database holds a number of versions of it. Each versions have read and write-timestamp value. Therefore, each *di* may have several read and write time-stamp values. Each time a new version is created during writing operation. Therefore, data item *di* having write-timestamp values *w1, w2, w3.....wk* shows the write-timestamp values of different versions of the data item *di* (The value of *k* depends on how many times writing is done on data item *di*). The variables, invariants and initialisation of machine are given in Fig. 2. The variable *vwtimestamp* represents version's write-timestamp. It is declared as:

$$vwtimestamp \in dataitem \leftrightarrow Natural$$

The operator $\leftrightarrow$ defines the set of relations between *dataitem* and Natural (write-timestamp). The 'Natural' represents a set of natural numbers in B. This ensures that each *dataitem di* may have several distinct write-timestamp values. A mapping of the form *(di↦wi)∈vwtimestamp* represents that a version of *di* have the write-timestmp *wi* (natural number). The read-timestamp of version is modeled as:

$$vrtimestamp \in \{\text{vwtimestamp}\} \rightarrow Natural$$

Variable *vrtimestamp* represents version's read-timestamp value. It maps each write-timestamp value of version to a set of natural number through a function. Therefore, this ensures that each data item for which version is created, there does exist a corresponding read-timestamp (natural number). For any version of data item *di* and write-timestamp *wi*, *vrtimestamp*($\{di \mapsto wi\}$) gives the read time stamp value of that version. The variable *versionvalue* represents value of the version. It is modeled as $versionvalue \in vv$ and in the context *vv* is declared as:

Variables :
trans, versionvalue, transactiontimestamp, counter, transdataitem, vwtimestamp vrtimestamp, transstatus

Invariants :
inv1: *trans ⊆ transaction*
inv2: *versionvalue ∈ vv*
inv3: *transactiontimestamp ∈ trans→Natural*
inv4: *counter ∈ Natural*
inv5: *vwtimestamp ∈ (dataitem↔Natural)*
inv6: *transdataitem ∈ trans→Pow(dataitem)*
inv7: *vrtimestamp ∈ {vwtimestamp}→Natural*
inv8: *transstatus ∈ trans→transactionstatus*

Initialisation
act1: *trans ≔ ∅*
act2: *counter ≔ 1*
act3: *versionvalue ≔ vv0*
act4: *transactiontimestamp ≔ ∅*
act5: *transdataitem ≔ { }*
act6: *vwtimestamp ≔ dataitem×{0}*
act7: *vrtimestamp ≔ {dataitem×{0}}×{0}*
act8: *transstatus ≔ ∅*

Fig. 2. Variables, Invariants and Initialisation of Machine

$$vv = (dataitem \leftrightarrow Natural) \rightarrow (datavalue)$$

For any version of data item *di* and write-timestamp *wi*, *versionvalue*($\{di \mapsto wi\}$) gives the value of version. We have initialized variable *versionvalue* with *vv0*, where in the context *vv0* is declared as: $vv0 \in vv$. The description of other variables are as follows:

(i) The variable *trans* represents a set of started transactions.
(ii) The variable *transstatus* maps each started transaction to *transactionstatus*. Thus every transaction will have one of the following states; pending, abort or commit
(iii) The variable *transactiontimestamp* is defined as a total function which maps a transaction to natural number. This ensures that each started transaction have timestamp value which is any natural number.
(iv) The variable *transdataitem* represent a set of dataitem required by the transaction. It maps each fresh transaction to a set of dataitems. For any transaction *tr*, *transdataitem(tr)* gives the dataitems required by the transaction *tr*.

Submit-Transaction
Any *tr, ditem* **Where**
grd1: $tr \in transaction$
grd2: $tr \notin trans$
grd3: $ditem \in Pow(dataitem)$
Then
act1: $trans \coloneqq trans \cup \{tr\}$
act2: $transactiontimestamp(tr) \coloneqq counter$
act3: $counter \coloneqq counter+1$
act4: $transdataitem(tr) \coloneqq ditem$
act5: $transstatus(tr) \coloneqq pending$
End

Fig. 3. Submit-Transaction Event

Read-Operation
Any *tr, di, wtav, maxwts, readvalue* **Where**
grd1: $tr \in trans$
grd2: $di \in dataitem$
grd3: $di \in transdataitem(tr)$
grd4: $wtav = vwtimestamp[\{di\}]$
grd5: $\exists x \cdot (x \in wtav \wedge x < transactiontimestamp(tr))$
grd6: $maxwts = max(\{x \mid x \in wtav \wedge x < transactiontimestamp(tr)\})$
grd7: $\{di \mapsto maxwts\} \in dom(vrtimestamp)$
grd8: $transstatus(tr) = pending$
grd9: $\{di \mapsto maxwts\} \in dom(versionvalue)$
grd10: $readvalue = versionvalue(\{di \mapsto maxwts\})$
Then
act1: $vrtimestamp(\{di \mapsto maxwts\}) \coloneqq max(\{vrtimestamp(\{di \mapsto maxwts\}), transactiontimestamp(tr)\})$
act2: $transstatus(tr) \coloneqq commit$

Fig. 4. Read-Operation Event

A. Transaction Submission

The event (*Submit-Transaction*) models the submission of transaction (see Fig. 3). The guard *grd2* $tr \notin trans$ ensures that *tr* is a fresh transaction. After the submission of transaction, it is added into the *trans* set *(act1)*. The action *act2* assigns a new timestamp value to the transaction. Each time a timestamp value is assigned to a *transaction*, the *counter* variable is incremented by one *(act3)*. This ensures that each fresh transaction is assigned a unique timestamp. The action *act4* represents the dataitems required by transaction *tr*. Status of transaction *tr* is set to *pending* through the action *act5*.

B. Reading the Version Value

The event *Read-Operation* models the reading of version value (see Fig. 4). If a transaction *tr* reads the value of data item *di* then it selects that version of *di* which has the maximum write-timestamp value that is less than transaction timestamp. The guard *grd4* is written as: $wtav=vwtimestamp[\{di\}]$, *wtav* is relational image of data item *di* under the relation *vwtimestamp*. It contains all write-timestamp values of *di*. The guard *grd5* is modeled as:

$$\exists x.(x \in wtav \wedge x < transactiontimestamp(tr))$$

It ensures that there are some write-timestamp values of data item *di* which are less than transaction-timestamp *tr*. Finally, the guard *grd6* selects the maximum write-timestamp value of version which is less than transaction timestamp *tr*. It is written as:

$$maxwts = max(x :\mid x \in wtav \wedge x < transactiontimestamp(tr))$$

Therefore, the version whose write-timestamp is *maxwts* is selected for reading operation. The variable *readvalue* performs the reading of version value. It returns the version value of data item *di* whose write-timestamp is given by *maxwts(grd10)*. After reading the value of version, read-timestamp of that version is set to as largest of current read-timestamp and transaction-timestamp *(act1)*. The status of transaction *tr* is set as *commit* through action *act2*.

Write-Operation
Any *tr, di, wtav, newvv, maxwts* **Where**
grd1: $tr \in trans$
grd2: $newvv \in datavalue$
grd3: $di \in dataitem$
grd4: $di \in transdataitem(tr)$
grd5: $wtav = vwtimestamp[\{di\}]$
grd6: $transstatus(tr) = pending$
grd7: $\exists x \cdot (x \in wtav \wedge x < transactiontimestamp(tr))$
grd8: $maxwts = max(\{x \mid x \in wtav \wedge x < transactiontimestamp(tr)\})$
grd9 $\{di \mapsto maxwts\} \in dom(vrtimestamp)$
grd10: $vrtimestamp(\{di \mapsto maxwts\}) < transactiontimestamp(tr)$
Then
act1: $versionvalue \coloneqq versionvalue \cup \{(\{di \mapsto transactiontimestamp(tr)\}) \mapsto newvv\}$
act2: $vwtimestamp \coloneqq vwtimestamp \cup \{di \mapsto transactiontimestamp(tr)\}$
act3: $vrtimestamp \coloneqq vrtimestamp \cup \{\{di \mapsto transactiontimestamp(tr)\} \mapsto transactiontimestamp(tr)\}$
act4: $transstatus(tr) \coloneqq commit$
End

Fig. 5. Write-Operation event

C. Creation of New Version (Writing Operation)

The event *Write-Operation* models the creation of new version (see Fig. 5.). If any transaction *tr* wishes to perform

a write operation on data item *di* then it must be ensured that the data item *di* has not already been read by some other transaction whose time stamp value is greater than transaction timestamp *tr*, i.e., read-timestamp of selected version must be less than transaction time stamp *tr*. Thus, in the writing operation following must hold:

(i) The version of data item *di* which has largest write-timestamp that is less than transaction timestamp *tr* is selected for operation (same as version selection in reading operation). It is ensured by guard *grd8* given below:
maxwts = max(x :| x∈ wtav ∧ x<transactiontimestamp(tr))

(ii) Read-timestamp of that selected version should be less than transaction timestamp (ensures that version has not already been read by some other transaction whose time stamp value is greater than *tr*). The following guard *grd10* ensures this condition:
vrtimestamp({di↦maxwts})<transactiontimestamp(tr)

The above states that read-timestamp of selected version (The version which satisfy first condition, i.e., whose write-timestamp *maxwts*) is less than transaction timestamp *tr*. If both the conditions are satisfied then new version is created having new version value *newvv (act1)*. The write and read time stamp of new version is set as a transaction timestamp *(act2 & act3)* respectively. The status of transaction *tr* is set as *commit* through *act4*.

D. Abort Transaction

An update transaction *tr* will be aborted if the data item *di* on which transaction *tr* wishes to perform a write operation has already been read by some other transaction whose timestamp value is greater than timestamp of *tr*. Precisely, it is violation of guard *grd10* of writing operation given below:

$vrtimestamp(\{\text{di}\mapsto\text{maxwts}\}) > transactiontimestamp(tr)$

This event set the status of transaction to be *abort*.

V. Conclusions

Multiversion concurrency control is timestamp based protocol that reduces restart overhead of transactions. In this protocol an update transaction doesn't modify the database but it creates a new version of a data item while retaining the old version. Therefore, each data item have multiple versions containing its value, read-timestamp and the write timestamp. Formal analysis of such systems are required to precisely understand the behavior of systems and to verify that required properties are satisfiable.

In this paper, multiversion concurrency control scheme for database transactions is specified using Event-B. This work is carried out on Rodin tool [16], [17]. The tool supports generation and discharge of proof obligations arising due to consistency checking. In this approach, the proof obligations are generated by B tools and they provide an environment for discharging proof obligation. Modeling guidelines outlined in [14] were used and these guidelines helped us in modeling and discharging proof obligations generated due to consistency checking. Total thirty seven proof obligations were generated by the system out of which thirty three were discharged automatically while remaining four required interaction with the prover. This case study strengthen our believe that Event-B can be used to construct the model of database systems providing a deeper insight into why a system should work. In our future work we plan to strengthen the invariant conditions and add more concrete design details, by using logical clock, in the refinement steps for distributed environment.

References

[1] M. Ozsu and P. Valduriez: Principles of Distributed Database Systems. Pearson Education (Singapore) Pte.Ltd. India (2004).

[2] P. Bernstein and N. Goodman: Timestamp Based Algorithms for Concurrency Control in Distributed Database Systems. In: Proc. of 6th Int. Conf. on Very Large Databases (1980).

[3] P. Bernstein and N. Goodman: Multiversion Concurrency Control-Theory and Algorithms. ACM Trans. Database Systems, vol. 8, no. 4, pp. 465–483, (1983).

[4] P. Bernstein, V. Hadzilacos and N. Goodman: Concurrency Control and Recovery in Database Systems. Addison-Wesley (1987).

[5] M. Butler: An Approach to Design of Distributed Systems with B AMN. In: Proc. 10th Int. Conf. of Z Us-ers: The Z Formal Specification Notation (ZUM), LNCS 1212, pp. 223-241, (1997).

[6] M. Butler and M. Walden: Distributed System Development in B. In: Proc. of Ist Conf. in B Method, Nantes, pp. 155–168, (1996).

[7] A. Rezazadeh and M. Butler: Some Guidelines for formal development of web based application in B Method. In: Proc. of 4th Intl. Conf. of B and Z users, Guildford, LNCS, Springer, pp 472-491, (2005).

[8] R. Banach: Retrenchment for Event-B: UseCase-wise development and Rodin integration. Formal Aspects of Computing, 23, pp. 113–131, (2011).

[9] S. Hallerstede: On the purpose of Event-B proof obligations. Formal Aspects of Computing, 23: pp. 133–150, (2011).

[10] D. Yadav and M. Butler: Rigorous Design of Fault-Tolerant Transactions for Replicated Database Systems Using Event B. In: Butler M., Jones C.B., Romanovsky A, Troubitsyna E. (eds.) Rigorous Development of Complex Fault-Tolerant Systems. Lecture Notes in Computer Science, vol. 4157, Springer, Heidelberg, pp.343-363,(2006).

[11] S. Hallerstede and M. Leuschel: Experiments in program verification using Event-B. Formal Aspects of Computing, 24: pp. 97–125, (2012)

[12] R. Suryavanshi and D. Yadav:Formal Development of Byzantine Immune Total Order Broadcast System using Event-B. In: ICDEM 2010, F. Andres and R. Kannan (eds.) LNCS, Vol. 6411, Springer, pp.317-324, (2010).

[13] D. Yadav and M. Butler: Application of Event B to Global Causal Ordering for Fault Tolerant Transactions. In: Proc. of REFT 2005, Newcastle upon Tyne, pp. 93-103, (2005).

[14] M. Butler and D. Yadav: An incremental development of the mondex system in Event-B. Formal Aspects of Computing, 20(1):61-77, (2008).

[15] D. Yadav and M. Butler: Formal Development of a Total Order Broadcast for Distributed Transactions Using Event-B. Lecture Notes in Computer Science 5454, springer-Verlag Berlin Heidelberg, pp.152-176, (2009).

[16] 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).

[17] J. R. Abrial: A system development process with Event-B and the Rodin platform. In: Lecture Notes In Computer Science 4789, Springer, pp. 1–3, (2007).

[18] J. R. Abrial and D. Cansell: Click-n-Prove—Interactive Proofs within Set Theory, (2003).

[19] Steria, Atelier-B User and Reference Manuals, (1997).

[20] B Core UK Ltd. B-Toolkit Manuals, (1999).