MODELING OF DISTRIBUTED MUTUAL EXCLUSION SYSTEM USING EVENT-B

Raghuraj Suryavanshi[1] and Divakar Yadav[2]

[1]Institute of Engineering and Technology, GBTU, Lucknow, INDIA
[2] Department of Computer Science, South Asian University, New Delhi 110067, India
suryavanshi.cse@ietlucknow.edu;dsyadav@cs.sau.ac.in

ABSTRACT

The problem of mutual exclusion arises in distributed systems whenever shared resources are concurrently accessed by several sites. For correctness, it is required that shared resource must be accessed by a single site at a time. To decide, which site execute the critical section next, each site communicate with a set of other sites. A systematic approach is essential to formulate an accurate speciation. Formal methods are mathematical techniques 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 is event driven approach which is used to develop formal models of distributed systems .It supports generation and discharge of proof obligations arising due to consistency checking. In this paper, we outline a formal construction of model of Lamport's mutual exclusion algorithm for distributed system using Event-B. We have considered vector clock instead of using Lam-port's scalar clock for the purpose of message's time stamping.

KEYWORDS

Formal Methods, Distributed System, Vector Clock, Event-B, Formal Specifications, Mutual Exclusion.

1. INTRODUCTION

In distributed system, the problem of mutual exclusion arises when several sites access shared resources concurrently. To ensure the correctness, it is necessary that the shared resource must be accessed by a single site at a time. The mutual exclusion problem in a single computer system, where shared memory exist, can be solved by using shared variables i.e., semaphores. In distributed systems, shared memory does not exist and the resources may be distributed. Therefore, approaches based on shared variable may not be applicable. To solve the problem of mutual exclusion in distributed system, the approaches based on message passing are used. The mutual exclusion algorithm can be categorized as token based [1], [2] and non token based algorithm [1], [3], [4]. In the first category a unique token is shared among all the sites. A site is allowed to enter its critical section if it contains the token. In non token based algorithm, a site communicates with a set of other sites to decide who should execute the critical section next. Non token based mutual exclusion algorithms use timestamps to order requests for the critical section.

Jan Zizka (Eds) : CCSIT, SIPP, AISC, PDCTA - 2013
pp. 477–491, 2013. © CS & IT-CSCP 2013 DOI : 10.5121/csit.2013.3653

In this paper, formal construction of non token based mutual exclusion algorithm for distributed system is outlined. We have considered Lamport's algorithm [1], [3] for formal development of our model. In this algorithm, each site maintains a request queue, which contains its own times tamped request for mutual exclusion and also request messages received from other sites [3]. If any site *Sx* wants to enter the critical section, it broadcasts a time stamped request message *REQUEST-X* to all the sites and makes an entry for request message *REQUEST-X* in its request queue. When a site *Sy* receives the request message *REQUEST-X* sent by site *Sx*, It makes an entry of Sx's request(*REQUEST-X*)in its request queue and returns a time stamped reply message *REPLY-Y* to site Sx. After receiving the time stamped reply messages from all the sites, there questing site *Sx* enters the critical section if following conditions hold :

1. Time stamp of all received messages are greater than time stamp of request message *REQUEST-X*.
2. Time stamp of *REQUEST-X* is minimum among all requests present in request queue of site *Sx*.

After executing the critical section site *Sx* removes the entry of request message (*REQUEST-X*) from its request queue and broadcasts a time stamped release message *RELEASE-X* to all the sites. When a site *Sy* receives the release message *RELEASE-X* from site *Sx*, It removes *Sx's* request *REQUEST-X* from its request queue. When a site removes a request from its request queue then it may possible that next minimum times tamped request is own request, enabling it to enter the critical section. This algorithm executes critical section requests in the increasing order of timestamps.

A functional specification of system describes its behavior. A specification contains significant information about the system. The B Method provides a systematic approach to formulate an accurate specification. we develop our model in the spirit embedded in Event-B. The model contains a *BROADCAST-REQ* event that models the event for requesting critical section. In this event a requesting site broadcasts a time stamped request message to all sites. Delivery of time stamped request message is shown by *DELIVER-REQ* event. The event *REPLY* models the event for sending time stamped reply message from a site (receiver of request message) to requesting site. The event *REPLY-RECEIVE* models the receiving of time stamped reply message at the requesting site. At the same time this event also count how many sites have sent the reply messages. The execution and releasing of critical section is shown by the event *EXECUTE-CS* and *RELEASE-CS* respectively. After the execution of critical section, the requesting site broadcasts a timestamped release message to all sites. The broadcasting of time stamped release message is shown by the event *BROADCAST-RELEASE*. The event *DELIVER-RELEASE* models the delivery of times tamped release message at all sites.

The remainder of this paper is organized as follows: Section 2 briefly outline Event B and Rodin platform, Section 3 describes system model and informal description about events, Section 4 presents Event−B Model of mutual exclusion for distributed system. Section 5 concludes the paper

2. EVENT-B AND RODIN PLATFORM

The B Method [5], [6], [7] is a model oriented state based method. It represent the complete mathematical development of a Discrete Transition System. Event-B represents a further evolution of the B method, which has been simplified and is now centered around the general notion of events. Event-B [8], [9],[10], [11], [12],[13], [14], [15], [16], [17] is event driven approach used to develop formal models of distributed systems. It is made of several components of two kinds: machines and contexts. Machines represent the dynamic part of model. This part is used to provide behavioral properties of model. It contains the variables, invariants, theorems, and events of a project. A machine is made of a state, which is defined by means of variables. Variables correspond to mathematical objects: sets, binary relations, functions, numbers, etc. These variables are constrained by invariants and these invariants are to be preserved while change the value of variables. The theorem of machine must follow from the context and the invariants of that machine. Moreover, a machine can be refined by other machines, but each machine can refine only one machine. Contexts contain the static part of model. It contains sets, constants, axioms, theorems. Sets may be enumerated or carrier. Axioms are used to describe the properties of those sets and constants. The context may be seen by machine directly or indirectly.

Besides its state, a machine contains a number of events which specify how the state may evolve. An event is made up of three elements its name, guards and actions. The guards are the necessary conditions for the event to occur. An event known as initialization event has no guard and it gives initial position of the model. An event can be specified in one of following three forms:

$$\text{Event} \triangleq \text{any k where P(k,v) then S(k,v) end}$$
$$\text{Event} \triangleq \text{when P(v) then S(v) end}$$
$$\text{Event} \triangleq \text{begin S(v) end}$$

Where k denotes parameters that are local to event, v denotes variable of machine containing the event, P(...) is a predicate denoting the guards of event and S(...) denotes the actions that updates some variables. Event-B notations are set theoretic notations. The syntax and description of notations are outlined in [10].

The Event-B Method requires the discharge of proof obligations for consistency checking. What is to be proved is stated in terms of proof obligations of a model. Proof obligations serve to verify properties of a model. They also serve to demonstrate that a model is sound with respect to some behavioral semantics. In this work, we have used Rodin platform. It is an open extensible tool for specification and verification of Event-B. The tool provides a seamless integration between modeling and proving. It also provide an environment for generation and discharge of proof obligations. It is embedded by various plugins such as proof-obligation generator, model checkers, provers, UML transformers, etc.

3. SYSTEM MODEL

We have considered a distributed system having a set of sites where every site maintains a request queue. The request queue contains timestamped request messages. In our model, time stamping of messages are done through vector clock [18]. In a system of vector clock, every site maintains a vector of size N to represent what that site believes to be the logical time at all other sites (N is the total number of sites in the system). Assume each site Si maintains a vector clock VT_{Si}, where $V_{T_{Si}}(i)$represents a local logical time at S_i while $VT_{Si}(j)$ represents the site Si's latest knowledge of the time at site Sj . Precisely $VT_{Si}(j)$ ($i \neq j$)represents the local time at site $_{Sj}$ when the most recent message was sent from Sj to Si directly or indirectly. Each time when a message is sent by any particular site a vector time stamp is assigned to message. While sending a message M from site Si to Sj , sender process Si updates its own time (*ith* entry of vector) by updating $VT_{Si}(i)$as $VTSi(i) := VT_{Si}(i) + 1$. The message time stamp VTM of message M is generated as $VT_M(k) := VT_{Si}(k), \forall k \in (1..N)$,,where N is the number of sites in system. A site Si increments its own local time $VT_{Si}(i)$only at the time of sending a message.

When a recipient site Sj receives a time stamped message it updates its knowledge by updating own vector clock. Site Sj updates its vector clock VT_{Sj} afterthe delivery of message M as $VT_{Sj}(k) := Max\ (VTSj(k), VTM(k))$. Therefore, inthe vector clock of site Sj , $VT_{Sj}(i)$ indicates the number of messages delivered to site Sj sent by site Si. The delivery order of messages between every pair of sites must follow FIFO order. The FIFO ordering property says that: *If a particular site broadcasts a message M1 before it broadcast a message M2, then each recipient process delivers M1 before M2.* The informal description of events are as follows:

1. *Request for Critical Section:* Any site which wants to enters the critical section, broadcasts a time stamped request message to all the sites. When a site broadcasts a message it increments its own vector time stamp by one and modified vector time stamp is assigned to message. It also creates an entry of time stamped request message in its request queue.

2. *Delivery of Request Message:* When a site receives the time stamped request message, it makes an entry of received request in its request queue. The delivery order of request message must follow the FIFO order. This ensures that all the messages which are previously sent by requesting site before the request message have already been delivered. During the delivery of request message receiving site also updates its knowledge by updating own vector time stamp with the time stamp of request message.

3. *Reply to Requesting Site:* After the delivery of timestamped request message at any site, it sends a corresponding time stamped reply message to requesting site. For assigning time stamp to message, a receiving site increments its own vector time stamp by one and modified vector time stamp is assigned to reply message.

4. *Receive Reply Message:* The requesting site receives the times tamped reply message sent by all sites. It makes an entry for each received reply message.The requesting site also count the total number of replied site. Each time when it receives a reply message it increments the value of total number of replied site by one. The requesting site also updates its vector time stamp with the time stamp of replied messages.

5. *Execution of Critical Section:* After receiving the reply messages from all sites, a requesting site enters critical section if the time stamp of all received messages are greater than time stamp of its request message and also the time stamp of own request message is minimum among all request messages present in request queue.

6. *Release Critical Section:* After performing execution of critical section the requesting site release it and removes the entry of request message from its request queue.

7. *Broadcast Release Message:* The requesting site broadcasts a time stamped release message to all the sites so that they can also remove the entry of request message (which is previously sent by it) from their request queue.

8. *Receive Release Message:* After the delivery of timestamped release message at any site, it removes the entry of corresponding request message from its request queue and updates its vector time stamp with the time stamp of release message.

4. EVENT-B MODEL OF MUTUAL EXCLUSION FOR DISTRIBUTED SYSTEM

Our Event-B model contains a context and a machine having eight events. In a context seen by machine *SITE* and *MESSAGE* represent carrier set. The *status* is defined as enumerated set containing the element *pending, reqcs, execs, releasecs*. The *type* is also defined as enumerated set and contains the element *request, reply, release*. Variable *sender* is defined as a partial function from *MESSAGE* to *SITE*. A mapping of the form $(m \mapsto s) \in sender$ indicates that message *m* was sent by a site *s*. The variable *msgsend* is subset of *MESSAGE* and it contains only those messages which are sent by any site. The variable *reqsites* is subset of *SITE* and it contains only those sites which have sent request messages. The variable *vtss* represents vector time stamp of site. It is declared as:

$$vtss \in SITE \rightarrow (SITE \rightarrow Natural)$$

It is a total function which maps every site to a vector function. The vector function maps each site to a natural number. The *'Natural'* represents a set of natural numbers in B. Therefore, vector time stamp of any site *Si*, *vtss(Si)* is a vector. The length of vector depends on number of sites present in the set *SITE*. Assume there are *K* sites in the system then *vtss(Si)* is a vector of

$((S1 \mapsto N1),(S2 \mapsto N2), (S3 \mapsto N3).........(Si \mapsto Ni)........(Sk \mapsto Nk))$.

Every time when a message is sent by site *Si*, it increments its own clock value vtss(*Si*)(*Si*) by one. Therefore, vector time stamp of site *Si* after sending single message is

$((S1 \mapsto N1),(S2 \mapsto N2), (S3 \mapsto N3)........(Si \mapsto Ni + 1)........(Sk \mapsto Nk))$.

The variable *vtsm* represents vector time stamp of message. It is defined as:

Variables:

sender, msgsend, vtss, vtsm, sitestatus, messagetype, reqsites, requestqueue replymsgsent, replymsgrec, deliver, delorder, counter, totalrepliedsite

Invariants :

inv1: $sender \in MESSAGE \nrightarrow SITE$

inv2: $msgsend \subseteq MESSAGE$

inv3: $vtss \in SITE \rightarrow (SITE \rightarrow Natural)$

inv4: $vtsm \in MESSAGE \rightarrow (SITE \rightarrow Natural)$

inv5: $sitestatus \in SITE \rightarrow status$

inv6: $messagetype \in msgsend \rightarrow type$

inv7: $reqsites \subseteq SITE$

inv8: $requestqueue \in SITE \leftrightarrow (MESSAGE \nrightarrow SITE)$

inv9: $replymsgsent \in (MESSAGE \leftrightarrow MESSAGE) \nrightarrow SITE$

inv10: $replymsgrec \in SITE \leftrightarrow (MESSAGE \leftrightarrow MESSAGE)$

inv11: $deliver \in SITE \leftrightarrow MESSAGE$

inv12: $delorder \in SITE \nrightarrow (MESSAGE \leftrightarrow MESSAGE)$

inv13: $counter \in Natural$

inv14: $totalrepliedsite \in SITE \rightarrow Natural$

Fig. 1. Variables and Invariants of Machine

$$vtsm \in MESSAGE \rightarrow (SITE \rightarrow Natural)$$

It is a total function which maps every message to a vector function. Vector time stamp of any message *mm (vtsm(mm))* is also a vector. Every time when a message *mm* is sent by site *Si*, it increments its own clock value by one and modified vector timestamp of site is assigned to message *mm*. This creates thevector timestamp of message *mm*. The *vtss(Si)(Si)* represents the number of messages sent by site *Si*. The description of other variables are as follows (see Fig. 1):

(i) The variable *sitestatus* is defined as a total function which maps each site to *status*. Thus every site in the set *SITE* will have one of the following states;*pending, reqcs, execs, releasecs*.

(ii) The variable *messagetype* is defined as:

$messagetype \in msgsend \rightarrow type$

It is a total function which maps every sent message to type. This ensures that every sent message will have one of the following states; *request, reply,release*.

(iii) The variable *requestqueue* is declared as:

$requestqueue \in SITE \leftrightarrow (MESSAGE \nrightarrow SITE)$

The operator ↔ defines the set of relations between *SITE* and request messages sent by corresponding sites. A mapping of the form $(ss \mapsto (m \mapsto s)) \in$

BROADCAST-REQ $\triangleq$
Any *ss, mm, nvts* **Where**
grd1: $ss \in SITE$
grd2: $ss \notin reqsites$
grd3: $ss \in dom(sitestatus)$
grd4: $sitestatus(ss) = pending$
grd5: $mm \in MESSAGE$
grd6: $mm \notin msgsend$
grd7: $mm \notin dom(sender)$
grd8: $nvtss \in (SITE \rightarrow Number)$
grd9: $nvtss = vtss(ss) \Leftarrow \{ss \mapsto vtss(ss)(ss)+1\}$
grd10: $\{mm \mapsto ss\} \notin requestqueue[\{ss\}]$
Then
act1: $vtsm(mm) := nvtss$
act2: $sender := sender \cup \{mm \mapsto ss\}$
act3: $reqsites := reqsites \cup \{ss\}$
act4: $msgsend := msgsend \cup \{mm\}$
act5: $messagetype(mm) := request$
act6: $requestqueue := requestqueue \cup \{ss \mapsto \{mm \mapsto ss\}\}$
act7: $sitestatus(ss) := reqcs$
END

Fig. 2. Broadcasting of request message

requestqueue indicates that request queue of site *ss* has a request message *m* sent by site *s*. Relational image of site *Si* under the relation *requestqueue* is represented by *requestqueue[{Si}]* and it contains all request messages sent by corresponding sites i.e., if site *Si* receives three request messages *M*1, *M*2,*M*3 sent by sites *S*1, *S*2, *S*3 respectively then *requestqueue[{Si}]* contains $((M_1 \mapsto S_1),(M_2 \mapsto S_2),\ (M_3 \mapsto S_3))$. The vector time stamp of messages can be found from the variable *vtsm*.

(iv) When a site receives a request message it sends a corresponding reply message to requesting site. A reply of request message sent by a site is represented by variable *replymsgsent*. It is defined as:

$$replymsgsent \in (MESSAGE \leftrightarrow MESSAGE) \nrightarrow SITE$$

A mapping $\{(\{mm \mapsto m\}) \mapsto ss\} \in replymsgsent$ indicates that a reply message *m* of a request message *mm* has been sent by a site *ss*.

(v) The variable *replymsgrec* represents receiving of reply message of a request message at requesting site.

(vi) The variable *deliver* represents delivery of message at a site. A mapping of form $(s \mapsto m) \in deliver$ *deliver* represents that a site *s* has delivered message *m*.

DELIVER-REQ $\triangleq$
Any ss, mm, s **Where**
grd1: $ss \in SITE$
grd2: $ss \in reqsites$
grd3: $s \in SITE$
grd4: $s \in dom(delorder)$
grd5: $mm \in MESSAGE$
grd6: $mm \in msgsend$
grd7: $messagetype(mm) = request$
grd8: $(mm \mapsto ss) \in sender$
grd9: $\{mm \mapsto ss\} \notin requestqueue[\{s\}]$
grd10: $(s \mapsto mm) \notin deliver$
grd11: $\forall m,k \cdot (m \in MESSAGE \land k \in SITE \land (m \mapsto ss) \in sender \land vtsm(m)(k) < vtsm(mm)(k) \Rightarrow (s \mapsto m) \in deliver)$
Then
act1: $deliver \coloneqq deliver \cup \{s \mapsto mm\}$
act2: $delorder(s) \coloneqq delorder(s) \cup (deliver[\{s\}] \times \{mm\})$
act3: $requestqueue \coloneqq requestqueue \cup \{s \mapsto \{mm \mapsto ss\}\}$
act4: $vtss(s) \coloneqq vtss(s) \Leftarrow (\{k \mid k \in SITE \land vtss(s)(k) < vtsm(mm)(k)\} \lhd vtsm(mm))$
End

Fig. 3. Delivery of request message

(vii) The variable *delorder* represents delivery order of messages at a site. A mapping $(m1 \mapsto m2) \in delorder(s)$ indicate that site *s* has delivered *m1* before *m2*.

(viii) The variable *totalrepliedsite* maps each site to *'Natural'* number. Variable *counter* is a integer type which is used to count number of sites from which requesting site has received the reply messages. A mapping $(s \mapsto n) \in$ *totalrepliedsite* represents that 'n' number of sites has sent reply message to site *s*. Each time when a requesting site receives a reply message from other sites the value of *counter* is incremented by one. Initially, the status of all site is set to as *pending* and the value of variable *counter* is zero. The vector time stamp of all sites and messages are initialized with zero. The remaining variables contain null values.

Broadcasting and Delivery of Request Message : The event *BROADCAST-REQ* models the broadcasting of request message (see Fig. 2). A site *ss* which wants to enters critical section, broadcasts a timestamped request message *mm* to all site. The guard *grd6 & grd7* ensures that message *mm* has not been sent previously. At the time of broadcasting a message *mm*, site *ss* increments its own clock value *vtss(ss)(ss)* by one *(grd9)*. The modified vector timestamp of site is assigned to message *mm(act1)*. The guard *grd10* is written as:

REPLY $\triangleq$
Any *ss, mm, nvtss, m* **Where**
grd1: $ss \in SITE$
grd2: $mm \in MESSAGE$
grd3: $mm \in msgsend$
grd4: $mm \in dom(sender)$
grd5: $messagetype(mm) = request$
grd6: $m \in MESSAGE$
grd7: $m \notin msgsend$
grd8: $m \notin dom(sender)$
grd9: $\{mm \mapsto m\} \notin dom(replymsgsent)$
grd10: $ss \notin \{sender(mm)\}$
grd11: $nvtss \in (SITE \rightarrow Natural)$
grd12: $nvtss = vtss(ss) \lessdot \{ss \mapsto vtss(ss)(ss)+1\}$
Then
act1: $vtsm(m) := nvtss$
act2: $msgsend := msgsend \cup \{m\}$
act3: $messagetype(m) := reply$
act4: $sender := sender \cup \{m \mapsto ss\}$
act5: $replymsgsent := replymsgsent \cup \{(\{mm \mapsto m\}) \mapsto ss\}$
End

Fig. 4. Sending of Reply Message

$$\{mm \mapsto ss\} \notin requestqueue[\{ss\}]$$

It ensures that request queue of site *ss* does not contain a request message *mm* which is sent by it. The action *act2* ensures broadcasting of message *mm* by site *ss* and actions *act3, act4* add the site *ss* and message *mm* in the set *reqsites* and *msgsend* repectively. The type of message *mm* is set to as a *request* through the action *act5*. The action *act6* adds the message *mm* sent by site *ss* in the request queue of *ss*. The action *act7* changes the status of site *ss* from *pending* to *reqcs*.

The event *DELIVER-REQ* models the delivery of request message (see Fig.3).The request message *mm (grd7)* which is sent by site *ss (grd8)* has not been delivered at site *s* is ensured by guard *grd10*. The site *ss* is requesting site is ensured by guard *grd2*. The guard *grd9* ensures that request message *mm* sent by site *ss* is not present in the request queue of site s. The guard *grd11* ensures FIFO order delivery of message. It confirms that all the messages which are sent by site *ss* before message *mm* has been already delivered to site *s*. As a consequence of occurrence of this event delivery of message *mm* is done at site*s (act1)* and request is added in the request queue of site *s (act3)*. The delivery order at site *s* is also updated such that all messages delivered at site *s* must

REPLY-RECEIVE $\triangleq$

Any ss, mm, m, s **Where**

grd1: $ss \in SITE$

grd2: $ss \in reqsites$

grd3: $s \in SITE$

grd4: $mm \in msgsend)$

grd5: $messagetype(mm) = request$

grd6: $(mm \mapsto ss) \in sender$

grd7: $m \in msgsend$

grd8: $messagetype(m) = reply$

grd9: $ss \in dom(delorder)$

grd10: $(\{mm \mapsto m\}) \mapsto s \in replymsgsent$

grd11: $\{mm \mapsto ss\} \in requestqueue[\{ss\}]$

grd12: $\{m \mapsto s\} \notin requestqueue[\{ss\}]$

grd13: $(ss \mapsto (\{mm \mapsto m\})) \notin replymsgrec$

grd14: $\forall mg, k \cdot (mg \in MESSAGE \wedge k \in SITE \wedge (mg \mapsto s) \in sender \wedge vtsm(mg)(k) < vtsm(m)(k) \Rightarrow (ss \mapsto mg) \in deliver)$

Then

act1: $deliver := deliver \cup \{ss \mapsto m\}$

act2: $delorder(ss) := delorder(ss) \cup (deliver[\{ss\}] \times \{m\})$

act3: $replymsgrec := replymsgrec \cup \{ss \mapsto (\{mm \mapsto m\})\}$

act4: $totalrepliedsite(ss) := counter$

act5: $counter := counter + 1$

act6: $vtss(ss) := vtss(ss) \mathbin{<\!\!+} (\{k / k \in SITE \wedge vtss(ss)(k) < vtsm(m)(k)\} \lhd vtsm(m))$

End

Fig. 5. Delivery of Reply Message

precede *mm (act2)*. For maintaining the latest knowledge about the system, site *s* updates its vector time stamp. It is expressed as *act4* :

$$vtss(s) := vtss(s) \mathbin{<\!\!+} (\{k :| k \in SITE \wedge vtss(s)(k) < vtsm(mm)(k)\} \lhd vtsm(mm))$$

The operator $\mathbin{<\!\!+}$ (overload operator) updates the values in the vector clock of site *s* by corresponding values in the vector timestamp of message *mm (vtsm(mm))* wherever values in the recipient site clock *(vtss(s)(k))* are less than corresponding values in the message time stamp *(vtsm(mm)(k))*.

Sending and Delivery of Reply Message: The event *REPLY* is given in Fig.4. This event models the sending of timestamped reply message of corresponding request message. The message *mm* is request message is ensured by guard *grd5*. A reply message *m* of request message *mm* has not been sent is ensured by guards *grd7& grd9*. The guard *grd8* ensures that message *m* is a fresh message and has not been previously sent by any site. As a consequences of occurrence of this event, incremented vector time stamp value of site *ss* is assigned to message *m (act1)* and it is added in the set *msgsend (act2)*. The type of message *m* is set to as *reply* through action *act3*. The action *act4* makes site *ss* as a sender of *m*. The action *act5* is written as:

$$replymsgsent := replymsgsent \cup \{(\{mm \mapsto m\}) \mapsto ss\}$$

It updates variable *replymsgsent* and creates the entry of reply message *m* of request message *mm* sent by site *ss*. The event *REPLY-RECEIVE* models the delivery of reply message at requesting site (see Fig. 5). Site *ss* is a requesting site is ensured by guard *grd2*.

A request message *mm* has already been sent by site *ss* is ensured by guards *grd4, grd5* & *grd6*. A reply mesaage *m* of *mm* has been sent by site *s* is ensured by guard *grd10*. The guard *grd13* ensures that reply *m* of corresponding request message *mm* has not been received by site *ss*. The guard *grd14* ensures FIFO order delivery of message. The action *act1* makes the delivery of message *m* at sites *ss* and action *act2* updates the delivery order of messages such that all messages delivered at site *ss* must precede *m*. The action *act3* is written as:

$$replymsgrec := replymsgrec \cup \{ss \mapsto (\{mm \mapsto m\})\}$$

This makes receiving of reply message *m* of request message *mm* at site *ss*. This event also count how many sites have sent the reply message to requesting site. Each time when a reply message is received by requesting site the value of *total replied* site is incremented by one *(act4 & act5)*. For maintaining the latest knowledge about the system, site *ss* updates its vector time stamp through the action *act6* .

Execution and Releasing of Critical Section: The *EXECUTE-CS* event, given in Fig. 6, models the execution of critical section. A requesting site *ss* executes the critical section if following condition holds:

(i) Site *ss* has received the reply messages from all sites and time stamp of all received messages are greater than time stamp of request message which is sent by site *ss*.

(ii) Time stamp of all request messages which are present in the request queue of site *ss* are greater than the time stamp of request message sent by site *ss*. The guard *grd2* ensures that site *ss* is requesting site and guard *grd4* ensures that status of site *ss* is *reqcs*. The *message type* of *mm* is *reques*t is ensured by guard *grd7*. The guard *grd8* ensures that request queue of site *ss* contains a request message *mm* which is sent by it. The guard *grd9* ensures that site *ss* has received the reply messages from all the sites. The guard *grd10* ensures that time stamp of request message *mm* is less than time stamp of all received replied messages. The guard *grd11* ensures that time stamp of request message *mm* is minimum among all the requests messages present in request queue of site *ss*. The status of site *ss* is set to as *execs* through the action *act1*.

EXECUTE-CS $\triangleq$
Any *ss, mm* **Where**
grd1: $ss \in SITE$
grd2: $ss \in reqsites$
grd3: $ss \in dom(sitestatus)$
grd4: $sitestatus(ss) = reqcs$
grd5: $mm \in MESSAGE$
grd6: $mm \in msgsend$
grd7: $messagetype(mm) = request$
grd8: $\{mm \mapsto ss\} \in requestqueue[\{ss\}]$
grd9: $totalrepliedsite(ss) = card(SITE) - 1$
grd10: $\forall s, m \cdot (s \in SITE \wedge m \in MESSAGE \wedge m \in dom(messagetype) \wedge messagetype(m) = reply \wedge \{mm \mapsto m\} \in replymsgrec[\{ss\}] \Rightarrow vtsm(mm)(s) < vtsm(m)(s))$
grd11: $\forall s, m \cdot (s \in SITE \wedge m \in MESSAGE \wedge \{m \mapsto s\} \in requestqueue[\{ss\}] \Rightarrow (\forall k \cdot k \in SITE \wedge vtsm(mm)(k) < vtsm(m)(k)))$
Then
act1: $sitestatus(ss) \coloneqq execs$
End
RELEASE-CS $\triangleq$
Any *ss, mm* **Where**
grd1: $ss \in SITE$
grd2: $ss \in reqsites$
grd3: $ss \in dom(sitestatus)$
grd4: $sitestatus(ss) = execs$
grd5: $mm \in MESSAGE$
grd6: $mm \in msgsend$
grd7: $messagetype(mm) = request$
grd8: $\{mm \mapsto ss\} \in requestqueue[\{ss\}]$
Then
act1: $requestqueue \coloneqq requestqueue \setminus \{ss \mapsto \{mm \mapsto ss\}\}$
act2: $sitestatus(ss) \coloneqq releasecs$
End

Fig. 6. Execution and Releasing of Critical Section

The *RELEASE-CS* event models the releasing of critical section (see Fig.6).After performing execution of critical section the requesting site release it and removes the entry of request message from its request queue. The site *ss* is requesting site is ensured by guard *grd2*. The guard *grd4* ensures that status of site *ss* is *execs*. This event set the status of site *ss* as *releasecs (act2)* and removes entry of its request from its request queue *(act1)*.

BROADCAST-RELEASE $\triangleq$
Any $ss, mm, nvtss$ **Where**
grd1: $ss \in SITE$
grd2: $ss \in reqsites$
grd3: $ss \in dom(sitestatus)$
grd4: $sitestatus(ss) = releasecs$
grd5: $mm \in MESSAGE$
grd6: $mm \notin msgsend$
grd7: $mm \notin dom(sender)$
grd8: $nvtss \in (SITE \rightarrow Natural)$
grd9: $nvtss = vtss(ss) \mathbin{<\!\!\!+} \{ss \mapsto vtss(ss)(ss)+1\}$
Then
act1: $vtsm(mm) := nvtss$
act2: $msgsend := msgsend \cup \{mm\}$
act3: $sender := sender \cup \{mm \mapsto ss\}$
act4: $messagetype(mm) := release$
act5: $reqsites := reqsites \setminus \{ss\}$
act6: $sitestatus(ss) := pending$
End

Fig. 7. Broadcasting of Release message

Broadcast and Delivery of Release message: The event *BROADCAST-RELEASE* is given in Fig. 7. After releasing of critical section site *ss* broadcasts a time stamped release message to all sites so that they can also remove the entry of request message previously sent by it. The guard *grd4* ensures that site *ss* has performed the execution of critical section. Before broadcasting a reply message *mm* site *ss* increments its vector time stamp *(grd9)* and this modified vector time stamp is assigned to the message *mm (act1)*. The action *act4* set the type of message *mm* as *release* message. The action *act5* removes site *ss* from request set *reqcs*. The status of site *ss* is set to as *pending* through the action *(act6)*.

The event *DELIVER-RELEASE* models the delivery of release message (see Fig. 8). The guard *grd3* ensures that message *mm* is release message. A site *ss* has sent the release message *mm* is ensured by guard *grd4*. In the request queue of site *s* (recipient site) there is an entry of request message *m* sent by site *ss* is ensured by guard *grd9*. The guard *grd10* ensures FIFO order delivery of messages. The delivery of message *mm* at site *s* is done through action *act1*. The action *act2* updates the delivery order such that all the messages which are previously delivered to site *s* must precede message *mm*. The action *act3* updates the vector time stamp of site *s*. The action *act4* removes the entry of request message *m* sent by site *ss* from the request queue of site *s*. Removing a request from request queue makes possible that next minimum time stamped request is own request, enabling it to enter the critical section.

DELIVER-RELEASE $\triangleq$

Any *ss, mm, s, m* **Where**

grd1: $ss \in SITE$

grd2: $mm \in msgsend$

grd3: $messagetype(mm) = release$

grd4: $(mm \mapsto ss) \in sender$

grd5: $s \in dom(delorder)$

grd6: $m \in msgsend$

grd7: $(m \mapsto ss) \in sender$

grd8: $messagetype(m) = request$

grd9: $\{m \mapsto ss\} \in requestqueue[\{s\}]$

grd10: $\forall mg, k \cdot (mg \in MESSAGE \land k \in SITE \land (mg \mapsto ss) \in sender \land vtsm(mg)(k) < vtsm(mm)(k) \Rightarrow (s \mapsto mg) \in deliver)$

Then

act1: $deliver := deliver \cup \{s \mapsto mm\}$

act2: $delorder(s) := delorder(s) \cup (deliver[\{s\}] \times \{mm\})$

act3: $vtss(s) := vtss(s) \mathbin{\lhd\!\!-} (\{k \mid k \in SITE \land vtss(s)(k) < vtsm(mm)(k)\} \lhd vtsm(mm))$

act4: $requestqueue := requestqueue \setminus \{s \mapsto \{m \mapsto ss\}\}$

End

Fig. 8. Delivery of Release message

5 . CONCLUSIONS

In distributed system, due to absence of global clock and shared memory traditional technique like semaphore may not be appropriate for solving the problem of mutual exclusion. To decide which site execute the critical section, a site communicates with other sites by sending a message. We have considered Lamport's mutual exclusion algorithm [1], [3] for formal construction of our model. In this algorithm, each site maintains a request queue, which contains its own timestamped request for mutual exclusion and also request messages received from other sites [3]. We have considered vector clock [18] instead of using Lamport's scalar clock for assigning the time stamp to messages. In a system of vector clock, every site maintains a vector to represent what that site believes to be the logical time at all other sites.

In this paper, modeling of distributed mutual exclusion system is specied using Event-B. This work is carried out on Rodin tool [16], [17]. The Rodin tool is intended to support construction and verification of Event-B models. The tool takes the formal text of model and produces proof obligations. It provides an environment to discharge of proof obligations arising due to consistency checking. Modeling guidelines outlined in [14] were used and these guidelines helped us in modeling and discharging proof obligations generated due to consistency checking. Total sixty four proof obligations were generated by the system and all of them were discharged automatically. The proofs and invariants together helped us to reason about the system design.We also found that vector clock can also be used to implement Lamport's mutual exclusion instead of using scalar clock.

REFERENCES

[1] Singhal, M., Shivratri, N.G.: Advanced Concepts in Operating Systems. Tata Mc- GrawHill Book Company, India (2005)

[2] Raymond, K.: A Tree-Based algorithm for Distributed Mutual Exclusion. In: ACM transactions on computer systems, vol.7, pp. 61-77, (1989)

[3] Lamport, L.: Time, Clocks and Ordering of Events in Distributed Systems. In:communications of the ACM, July (1978)

[4] Ricart, G. and Agrawala, A. K.: An Optimal algorithm for Mutual Exclusion in Computer Networks. In: communications of the ACM, (1981)

[5] Butler, M.: An Approach to Design of Distributed Systems with B AMN. In: Proc. 10th Int. Conf. of Z Users: The Z Formal Speci_cation Notation (ZUM), LNCS1212, pp. 223-241, (1997).

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

[7] Rezazadeh, A. and Butler, M.: 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] Banach, R.: Retrenchment for Event-B: UseCase-wise development and Rodin integration. Formal Aspects of Computing, 23, pp. 113131, (2011).

[9] Hallerstede, S.: On the purpose of Event-B proof obligations. Formal Aspects of Computing, 23: pp. 133150, (2011).

[10] Yadav, D. and Butler, M.: Rigorous Design of Fault-Tolerant Transactions for Replicated Database Systems Using Event B. In: Butler M., Jones, C.B.(eds.) LNCS, vol. 4157, Springer, Heidelberg, pp.343-363,(2006).

[11] Hallerstede, S. and Leuschel, M.: Experiments in program veri_cation using Event-B. Formal Aspects of Computing, 24: pp. 97125, (2012)

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

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

[15] Yadav, D. and Butler, M.: 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] Metayer, C., Abrial, J.R. and Voison, L.: Event-B language. RODIN deliverables 3.2, http://rodin.cs.ncl.ac.uk/deliverables/D7.pdf, (2005).

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

[18] Baldoni, R. and Raynal, M.: Fundamentals of Distributed Computing: A Practical Tour of Vector Clock Systems. In: IEEE Distributed Systems Online, Vol. 3, no. 2,(2002)