Verification of Liveness Properties in Distributed Systems

Divakar Yadav and Michael Butler [⋆]

Institute of Engineering and Technology, U P Technical University, Lucknow INDIA
School of Electronics and Computer Science, University of Southampton, UK
divakar.yadav@ietlucknow.edu,mjb@ecs.soton.ac.uk

Abstract. This paper presents liveness properties that need to be preserved by Event-B models of distributed systems. Event-B is a formal technique for development of models of distributed systems related via refinement. In this paper we outline how enabledness preservation and non-divergence are related to the liveness properties of the B models of the distributed systems. We address the liveness issues related to our model of distributed transactions and outline the construction of proof obligations that need to be discharged to ensure liveness.

Key words: Formal Methods, Distributed Systems, Event-B, Liveness Properties

1 Introduction

Distributed systems are hard to understand and verify due to complex execution paths and unanticipated behavior. A rigorous reasoning of these systems is required to precisely understand behavior of such systems. Safety and liveness are two important issues in the development of the distributed systems [9]. The distinction between safety and liveness properties is motivated by different tools and techniques for proving them and various interpretations of these properties are discussed in [8]. Informally, as described in [9], the safety property expresses that something *bad* will not happen during a system execution. A liveness property expresses that something *good* will eventually happen.

Event-B [10], a variant of classical B [1], is a formal technique for developing models of distributed systems. Event-B supports refinement and provides a complete framework for development of models of distributed systems. Existing B tools provides an environment for the generation of proof obligations for consistency and refinement checking and to discharge them. This technique consists of first describing the abstract problem, introducing solutions or details in refinement steps to obtain more concrete specifications and verifying that proposed solutions are valid. A specific development in this approach is made of a

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

series of more and more refined models. Each model is made of static properties (invariants) and dynamic properties (events). A list of state variables is modified by activation of finite list of events. The events are guarded by predicates and these guards may be strengthened at each refinement step. An approach to incremental design of distributed systems and guidelines for construction of Event-B models are outlined in [3–5].

Existing B tools provide a strong proof support to aid reasoning about the safety properties by generating the proof obligations and providing an environment to discharge the proof obligations. In addition to the safety properties, it is also useful to verify that the models of distributed systems are live. In this paper we outline issues related to liveness in Event-B models and present the guidelines to address these issues in the Event-B development of the distributed systems. A case study of distributed transactions [11] for replicated database is used to outline construction of proof obligations.

2 Safety and Liveness in the Event-B Models

With regard to the safety, the most important property which we want to prove about the models of distributed systems is *invariant preservation*. The invariant is a condition which must hold permanently on the state variables. By *invariant preservation* we mean proving that the actions of the events do not violate the invariants. Existing B tools generate proof obligations for following.

1. *Consistency Checking* : Consistency of a machine is established by proving that the actions associated with each event modifies the variables in such a way that the invariants are preserved under the hypothesis that the invariants hold initially and the guards are true. Discharging proof obligations generated due to consistency checking proves that the machine is consistent with respect to the invariants.
2. *Refinement Checking* : The refinement of a machine consists of refining its state and events. The gluing invariants relate the state of the refinement, represented by the concrete variables, to the abstract state represented by the abstract variables. An event in the abstraction may be refined by one or more events, and discharging proof obligations generated due to refinement checking ensure that gluing invariants are preserved by actions of the events in the refinement.

In order to ensure that the models of the distributed are *live* and eventually makes *progress* we need to prove that Event-B models are *non-divergent* and *enabledness* preserving [10].

3. *Non-Divergence* : In an incremental development approach using Event-B, new events and the variables can be introduced in the refinement steps. Each new event of the refinement refines a *skip* event in the abstraction and defines actions on the new variables. Proving the non-divergence requires us to prove that the new events do not take control forever.

4. *Enabledness Preservation* : By enabledness preservation, we mean that whenever some events (or group of events) is enabled at the abstract level then the corresponding events (or group of events) are eventually enabled at the concrete level.

Existing B tools [2, 10] generate proof obligations due to consistency and refinement checking and they provide an environment to discharge them using automatic prover or by interaction. To ensure liveness in Event-B models we need to prove that models of distributed system are *Non-Diergent* and *Enabledness* preserving.

3 Event-B Models of Distributed Transactions

Our system model consist of sets of sites and data objects.We considered two types of transactions namely, Read-only and Update Transaction. In our model we considered *read anywhere, write everywhere* approach. A read-only transaction read the data locally while the update transactions are processed within the framework of a two phase commit protocol [7] to ensure global atomicity. Two update transactions are said to be in *conflict* if atleast one data object appears in *write set* of both transactions. To meet the strong consistency requirements, *conflicting* transactions need to be executed in isolation. We ensure this property by not *starting* a transaction at a site if any conflicting update transaction is *active* at that site. The commit or abort decision of a global transaction T_i is taken at the coordinator site within the framework of a two phase commit protocol. A global transaction T_i *commits* if *all* T_{ij} *commit* at S_j. The global transaction T_i *aborts* if *some* T_{ij} *aborts* at S_j.

A formal refinement based approach using Event-B to model and analyze distributed transaction is given in [11]. In our abstract model, an update transaction modifies the abstract one copy database through a single atomic event. In the refinement, an update transaction consists of a collection of interleaved events updating each replica separately. The transaction mechanism on the replicated database is designed to provide the illusion of atomic update of a one copy database. Through the refinement proofs, we verify that the design of the replicated database conforms to the one copy database abstraction despite transaction failures at a site. We assume that the sites communicate by a reliable broadcast [6] which eventually deliver messages without any ordering guarantees.

3.1 Abstract Transaction States

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

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

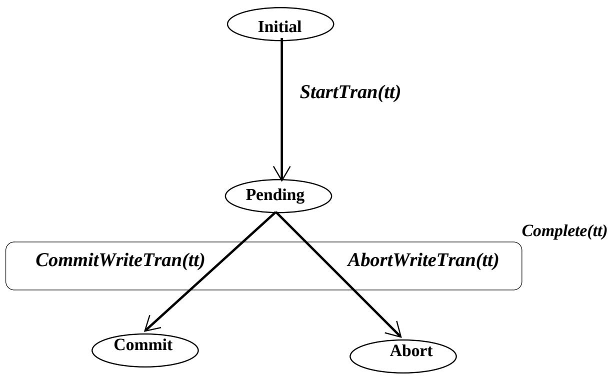

Fig. 1. Transaction States in the Abstract Model

The transitions in the transaction states due to the activation of events in the abstract model of the transactions are outlined in the the Fig. 1. The *CommitWriteTran(tt)* and *AbortWriteTran(tt)* together are represented as *Complete(tt)*, as both of these event models the completion of a update transaction.

3.2 Refined Transaction States

In the refined model, a global update transaction can be submitted to any one site, called the coordinator site for that transaction. Upon submission of an update transaction, the coordinating site of the transaction broadcasts all operations of the transaction to the participating sites by an *update* message. Upon receiving the update message at a participating site, the transaction manager at that site starts a sub-transaction. The activity of a global update transaction at a given site is referred as a sub-transaction.

The *BeginSubTran(tt,ss)* event models starting a sub-transaction of *tt* at participating site *ss*. In this refinement, the state of a transaction at a site is represented by a variable *sitetransstatus*. The variable *sitetransstatus* maps each transaction, at a site, to transaction states given by a set *SITETRANSTATUS*, where *SITETRANSTATUS*={*pending*, *commit*, *abort*, *precommit*}. A transaction *t* is said to be active at a site *s* if it has acquired the locks on the object set at that site. Our model prevents starting sub-transaction at a site if any conflicting transaction is already active at that site. Following guard of *Begin-SubTran(tt)* event ensures that a sub-transaction of *tt* starts at a site *ss* when no active transaction *tz* running at *ss* is in *conflict* with *tt* :

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

As a consequence of the occurrence of this event, transaction *tt* becomes *active* at site *ss* and the *sitetransstatus* of *tt* at *ss* is set to pending. Instead of giving the specifications of all events of the refinement in the similar detail, brief descriptions of the new events in this refinement are outlined below.

- *BeginSubTran(tt)* : This event models *starting* a sub-transaction at a site. The status of the transaction *tt* at site *ss* is set to *pending.*
- *SiteAbortTx(ss,tt)* : This event models *local abort* of a transaction at a site. The transaction is said to complete execution at the site. The status of the transaction *tt* at site *ss* is set to *abort.*
- *SiteCommitTx(ss,tt)* : This event models *precommit* of a transaction at a site. The status of the transaction *tt* at site *ss* is set to *precommit.*
- *ExeAbortDecision(ss,tt)* : This event models *abort* of a *precommitted* transaction at a site. This event is activated once the transaction has globally aborted. The status of the transaction *tt* at site *ss* is set to *abort.* The transaction is said to complete execution at the site.
- *ExeCommitDecision(ss,tt)* : This event models *commit* of a *precommitted* transaction at a site. This event is activated once the transaction has globally committed. The status of the transaction *tt* at site *ss* is set to *precommit.* The replica at the site is updated with the transaction effects and the transaction is said to complete execution at this site.

4 Non-Divergence

New events and the variables can be introduced in the refinement. Each new event of the refinement refines a *skip* event and defines a computation on new variables. In order to show that the model of system is non-divergent, we need to show that the new events introduced in the refinement do not together diverge, i.e., run forever. For example, if the new events in the refinement, such as, *BeginSubTran*, *SiteAbortTx* or *SiteCommitTx* take the control forever then the events of global commit/abort are never activated and a global commit decision may never be achieved. In order to prove that the new events do not diverge, we use a construct called *variant.* A variant V is a variable such that $V \in \mathbb{N}$, where $\mathbb{N}$ is a set of natural numbers. For each new event in the refinement we should be able to demonstrate that the execution of each new event decrease the variant and variant never goes below zero. This allow to us prove that a new event cannot take control forever, since a variant can not be decreased indefinitely.

The state diagram for a concrete transaction state transitions at a site is shown in the Fig. 2. A transaction state at each participating site is first set to

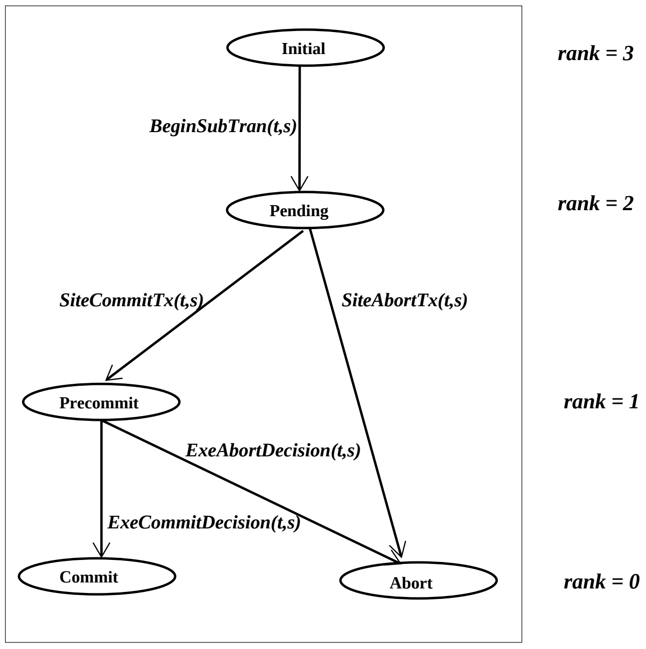

Fig. 2. Concrete Transaction States in the Refinement

pending by the activation of *BeginSubTran*. The activation of the event *SiteCommitTx* changes the status from *pending* to *precommit* while the activation of *SiteAbortTx* sets the status from *pending* to *abort* at that site. A transaction in the *precommit* state at a site changes the state to either *commit* or *abort* by the activation of event *ExeCommitDecision* or *ExeAbortDecision* respectively. As shown in the Fig. 2, each state is represented by a rank. The *initial* state represents a state of a transaction *tt* at a participating site *ss* when it is not active, i.e., the *sitetranssstatus* of *tt* at *ss* is not defined ($ss \notin dom(sitetranstatus(tt))$). After submission of a transaction, a transaction first become *active* at the coordinator site. Subsequently, due to the activation of the event *BeginSubTran(tt,ss)*, subtransactions are started separately at different sites, i.e., at each activation of this event, *tt* becomes active at participating sites *ss*. As shown in the figure, new events in the refinement change the state of a transaction at a site such that each time the rank is decreased.

A variant in the refinement is defined as a variable *variant* :

$$variant \in trans \rightarrow Natural$$

and initialized as $variant := \varnothing$.

When a fresh transaction *tt* is submitted by the activation of the event *StartTran(tt)*, the initial value of variant is set as *varinat(tt)* $:= 3 * N$, where N is total number of the sites in the system. All new events in the refinement decrease the variant by one. For example, events *SiteCommitTx(tt,ss)* and *SiteAbortTx(tt,ss)* decrease the variant by one and change the status of a transaction from a *pending* state to *precommit* or *abort* state. Since activation of the new events in the refinement decrease the variant, the rank of state is changed from three to zero such that *variant(tt)* will always be greater than or equal to zero.

In order to prove that the activation of the new events given in the Fig. 2 do not diverge, we need to prove that the changes in the state of a transaction at a site corresponds to the decrement in the rank from three to zero. Also, the variable *variant* is decreased each time a new event in the refinement is activated. Thus, we construct invariant property involving variable *variant* that need to be satisfied by the action of the events in the refinement. This property is given the Fig 3.

$$
\begin{array}{l}
\forall t \cdot (t \in trans \Rightarrow \\
\quad variant(t) \geq (\quad 3 * card(SITE - activetrans^{-1}[\{t\}] \\
\qquad\qquad\qquad +2 * card(sitetransstatus(t)^{-1}[\{pending\}] \\
\qquad\qquad\qquad +1 * card(sitetransstatus(t)^{-1}[\{precommit\}] \\
\qquad\qquad\qquad +0 * card(sitetransstatus(t)^{-1}[\{commit, abort\}] \\
\qquad\qquad\qquad) \\
\quad)
\end{array}
$$

Fig. 3. Invariant used in variant Proofs

In this expression, $activetrans^{-1}[\{\mathrm{t}\}]$ returns a set of sites where transaction *t* is in active state. Similarly, $sitetransstatus(\mathrm{t})^{-1}[\{pending\}]$ returns a set of site where a transaction *t* is in *pending* state. In order to prove that the new events in the refinement do not diverge, we have to show that the above invariant property on a variable *variant* holds on the activation of the events in the refinement. In order to prove this invariant property we need to add invariants (1) and (2) to the model. The invariant (1) states that if a transaction *t* is not *active* at a site *s* then the variable *variant* is greater than or equal to zero. The invariant (2) state that the variable *variant* is greater than or equal to zero if the status of a transaction *t* at site *s* either *precommit*, *pending*, *abort* or *commit*.

$$\forall(s,t) \cdot (t \in trans \wedge s \in SITE \wedge (s \mapsto t) \notin activetrans \Rightarrow variant(t) \geq 0) \tag{1}$$

$$\begin{array}{l}\forall(s,t) \cdot (t \in trans \wedge s \in SITE \wedge \\ sitetransstatus(t)(s) \in \{pending, precommit, abort, commit\} \Rightarrow variant(t) \geq 0)\end{array} \tag{2}$$

Once the above invariants are added to the model, the B Tool generate the proof obligations associated with the events in the refinement. These proof obligations may be discharged using automatic/interactive prover of the tool. By discharging the proof obligations we ensure that the model of distributed system is non-divergent. The same strategy can be followed for each level of the refinement chain of Event-B development to show non-divergence in development of distributed systems.

5 Enabledness Preservation

With respect to the liveness, the freedom from deadlock is an important property in a distributed database system. In our model of transactions, it require us to prove that each transaction eventually *completes* the execution, i.e., either it commits or aborts. It require us to prove that if a transaction *completes* the execution in the abstract model of a system, then it must also *complete* the execution in the concrete model. We ensure this property by enabledness preservation. The enabledness preservation in Event-B requires us to prove that the guards of the one or more events in the refinement are enabled under the hypothesis that the guard of one or more events in the abstraction are also enabled [10]. Precisely, let there exist events E_1^a, E_2^a ..., E_n^a in the abstraction and a corresponding event E_i^r in the refinement refines the abstraction event E_i^a. The events H_1^r, ..., H_k^r are the new events in the refinement. A weakest notion of enabledness preservation can be defined as follows :

$$\begin{aligned}&Grd(E_1^a) \vee Grd(E_2^a) ... \vee Grd(E_n^a)\\ &\Rightarrow\\ &Grd(E_1^r) \vee Grd(E_2^r) ... \vee Grd(E_n^r) \vee Grd(H_1^r) \vee Grd(H_2^r) ... \vee Grd(H_k^r)\end{aligned} \tag{3}$$

Weakest notion of enabledness preservation given at (3) state that if one of the event in the abstraction is enabled then one or more events in refinement are also enabled. The strongest notion of the enabledness is defined at (4). It state that if the event E_i^a in the abstraction is enabled then either the refining event E_i^r is enabled or one of the new events are enabled.

$$Grd(E_i^a) \Rightarrow Grd(E_i^r) \vee Grd(H_1^r) \vee Grd(H_2^r) ... \vee Grd(H_k^r) \tag{4}$$

A concrete model outlined in Fig. 2 may be deadlocked due to the race conditions, i.e., if updates are delivered to the sites without any order. To ensure that all updates are delivered to all sites in the same order, we need to *order* the update transactions such that all sites deliver updates in the same order. It may achieved if a site broadcasts an update using a total order broadcast [12]. In next section, we outline how the enabledness preservation properties relates to our model of transactions in the presence of abstract ordering on the update transactions.

Abstract Ordering on Transactions : To ensure that our concrete model of transactions does not block and makes progress, we introduce a new event *Order* in the refinement. The very purpose of introducing new event *Order(tt)* is to ensure that the transactions are executed at all sites in a predefined abstract order on the transactions. The abstract ordering on the transaction can be realized by introducing explicit *total ordering* [6] on the messages in the further refinements. To model an abstract order on the transactions we introduce new variables *tranorder* and *ordered* typed as follows:

$$\begin{aligned}&tranorder \subseteq trans \leftrightarrow trans\\ &ordered \subseteq trans\end{aligned}$$

A mapping of the form *t1* $\mapsto$ *t2* $\in$ *tranorder* indicate that a transaction *t1* is ordered before *t2*, i.e., at all sites *t1* will be processed before *t2*. The set variable *ordered* contains the transactions that has been ordered.

Proof Obligations for Enabledness Preservation : In this section, we outline the proof obligations to verify that the refinement is enabledness preserving. Our objective is a prove that if a transaction *completes* in the abstraction then it also *completes* in the refinement. The weakest notion of enabledness preservation[1] given at (3) requires us to prove following :

$$\begin{aligned}
&Grd(StartTran(t) \vee CommitWriteTran(t) \vee \; Grd(AbortWriteTran(t))\\
&\Rightarrow Grd(StartTran^{*}(t))\\
&\vee\, Grd(Order(t))\\
&\vee\, Grd(BeginSubTran(t,s))\\
&\vee\, Grd(SiteCommitTx(t,s))\\
&\vee\, Grd(SiteAbortTx(t,s))\\
&\vee\, Grd(CommitWriteTran^{*}(t))\\
&\vee\, Grd(AbortWriteTran^{*}(t))
\end{aligned} \tag{5}$$

The property given at (5) is not sufficient as it state that if one of the events in *StartTran*, *AbortWriteTran* or *CommitWriteTran* are enabled in the abstraction then one of the refined event or the new events are enabled in the refinement. It does not guarantee that if a transaction t completes in abstraction then it also completes in the refinement. What we need to prove is that if either *AbortWrite-Tran* or *CommitWriteTran* in the abstraction is enabled then one of the refined event or new event in the refinement are enabled. According to the strongest notion of enabledness preservation given at (4), it requires us to prove (6), (9) and (10).

$$\begin{aligned}
&Grd(StartTran(t))\\
&\Rightarrow Grd(StartTran^{*}(t))\\
&\vee\, Grd(Order(t))\\
&\vee\, Grd(BeginSubTran(t,s))\\
&\vee\, Grd(SiteCommitTx(t,s))\\
&\vee\, Grd(SiteAbortTx(t,s))\\
&\vee\, Grd(CommitWriteTran^{*}(t))\\
&\vee\, Grd(AbortWriteTran^{*}(t))
\end{aligned} \tag{6}$$

The property at (6) state that if the guard of *StartTran* event is enabled then the guard of refined *StartTran* or the guards of new events are enabled.

[1] An event E in the abstract model is defined as E^{*} in the refinement.

This property is provable due to following observations.

$$Grd(StartTran(t) \Rightarrow Grd(StartTran^{*}(t)) \tag{7}$$

In order to prove this property, the following proof obligation needs to be discharged. This proof obligation is trivial and can be discharged by the automatic prover of the tool.

$$\forall t(t \in TRANSACTION \wedge t \notin trans \Rightarrow t \notin trans) \tag{8}$$

$$\begin{aligned}
&Grd(CommitWriteTran(t))\\
&\Rightarrow Grd(StartTran^{*}(t))\\
&\vee\ Grd(Order(t)\\
&\vee\ Grd(BeginSubTran(t, s))\\
&\vee\ Grd(SiteCommitTx(t, s))\\
&\vee\ Grd(SiteAbortTx(t, s))\\
&\vee\ Grd(CommitWriteTran^{*}(t))\\
&\vee\ Grd(AbortWriteTran^{*}(t))
\end{aligned} \tag{9}$$

The property at (9) state that if the guard of *CommitWriteTran* event is enabled than the guard of refined *CommitWriteTran* or the guards of new events are enabled. This property is too storing to prove due to following reasons. A transaction may not *commit* in the refinement until some other transaction either do not *commit* or *abort*. It may happen due to the interleaved action of the events in refinement such that some other transaction has started at some site and the commit of the transaction t depends on commit or abort of other transaction. Also, due same reasons the property at (10) is also not provable.

$$\begin{aligned}
&Grd(AbortWriteTran(t))\\
&\Rightarrow\ Grd(StartTran^{*}(t))\\
&\vee\ Grd(Order(t)\\
&\vee\ Grd(BeginSubTran(t, s))\\
&\vee\ Grd(SiteCommitTx(t, s))\\
&\vee\ Grd(SiteAbortTx(t, s))\\
&\vee\ Grd(CommitWriteTran^{*}(t))\\
&\vee\ Grd(AbortWriteTran^{*}(t))
\end{aligned} \tag{10}$$

We observe that the proof obligations constructed due to the weakest notion of the enabledness preservation are not sufficient to prove that if a transaction completes in abstraction then it also completes in the refinement. Also, we observe that strongest notion of enabledness preservation is too strong to prove this property, therefore unprovable. What we really need a notion of enabledness preservation that is stronger than the weakest notion(see property 3) and weaker than the strongest notion(see property 4). This can be defined as below.

1. If the event *StartTran* is enabled in the abstraction then it is also enabled in the refinement.
2. If the completion event, i.e.,either *CommitWriteTran* or *AbortWriteTran* events are enabled in the abstract model then these completion events are also enabled in the refinement.

We have already outlined that the first property is preserved by the our model of transaction given as property (7). For second property, we further construct the property given at (11).

$$\begin{aligned}
&Grd(CommitWriteTran(t)) \vee \; Grd(AbortWriteTran(t)) \\
&\Rightarrow \; Grd(Order(t)) \\
&\vee \; Grd(BeginSubTran(t,s)) \\
&\vee \; Grd(SiteCommitTx(t,s)) \\
&\vee \; Grd(SiteAbortTx(t,s)) \\
&\vee \; Grd(CommitWriteTran^{*}(t) \\
&\vee \; Grd(AbortWriteTran^{*}(t)
\end{aligned} \tag{11}$$

We observe that property (11) is also not provable because a transaction t cannot complete its execution until some other conflicting transaction, already active, do not complete. Using the same strategy outlined above we construct a new property to illustrate that if the events corresponding to a completion of a transaction t_x in the abstraction are enabled then the new events *Order*, *BeginSubtran*, *SiteCommitTx*, *SiteAbortTx* are enabled for other transactions t_y or the refined *Complete* events are also enabled for t_x. Once these properties are added to the Event-B model of distributed transactions, the B Tool generate proof obligations due to these additions. In order to discharge new proof obligations we are required to add fresh invariants until the B prover is able to discharge all proof obligation.

The proof obligations outlined above are specific to the model of the transactions, however, using the same strategy the proof obligations for the other models of the distributed systems may be generated. Discharging these proof obligations ensures that the model of distributed system is enabledness preserving. It can be noticed that same strategy should be used to formulate the proof obligations for each level of refinement.

6 Conclusions

In this paper, we addressed the issue of liveness in the B models of distributed system. The safety and liveness are two important issues in the design and development of distributed systems. The safety properties express that something *bad* will not happen during system execution. A liveness property expresses that something *good* will eventually happen during the execution. With regards to the safety properties, the existing B tools generate proof obligations for consistency and refinement checking. These proof obligations may be discharged by using automatic/interative provers of B Tools. To ensure livness in the models of distributed systems, it is useful to state that the model of the system under development is *non-divergent* and *enabledness* preserving. In order to show that the new event in the refinement do not take control forever, i.e., models are non-divergent, we outlined a method for construction of an invariant property on variant. To ensure that a concrete models also make progress and do not block more often than its abstraction, it is necessary to prove that if an abstract model makes a *progress* due to the activation of events then the concrete model also make a progress due to the activation of the events in the refinement. We ensure this property by enabledness preservation. A method for construction of proof obligations for ensuring enabledness preservation is outlined in this paper. The proof obligations corresponding to both *non-divergent* and *enabledness* preserving can be discharged using automaic/interactive prover of B Tools to ensure that models of distributed system are live.

References

1. J. R. Abrial: The B Book. Assigning programs to meanings. Cambridge University Press (1996)
2. Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, and Laurent Voisin. An open extensible tool environment for Event-B. In Z. Liu and J. He, editors, ICFEM 2006, Lecture Notes in Computer Science, volume 4260, pages 588-605. Springer (2006)
3. Michael Butler : Incremental Design of Distributed Systems with Event-B, Marktoberdorf Summer School 2008 Lecture Notes, `http://eprints.ecs.soton.ac.uk/16910` (2008)
4. Michael Butler : An approach to the design of distributed systems with B AMN, In Jonathan Bowen,Michael Hinchey, and David Till, Eds, ZUM, Lecture Notes in Computer Science, ,volume 1212, Springer (2008)
5. Michael Butler and Divakar Yadav: An incremental development of mondex system in Event-B. Formal Aspects of Computting. 20(1) : 61-77. Springer (2008)
6. X.Defago, A.Schiper, P. Urban: Total order broadcast and multicast algorithms: Taxonomy and Survey. ACM Computing Survey, 36(4), 372-421 (2004)
7. Jim Gray and Andreas Reuter : Transaction Processing: Concepts and Techniques, Morgan Kaufmann, ISBN 1-55860-190-2 (1993)
8. E. Kindler: Safety and Liveness Properties : A Survey. Bulletin of the European Association for Theoitical Computer Science. 53 : 268-272 (1994)
9. L. Lamport: Proving the Correctness of Multiprocess Programs. IEEE Transactions on Software Eng. Vol.3.No.2, 125-143 (1977)
10. C. Metayer, J. R. Abrial, L. Voison : Event-B Language. RODIN deliverables 3.2. `http://rodin.cs.ncl.ac.uk/deliverables/D7.pdf.` (2005)
11. Divakar Yadav and Michael Butler : Rigorous Design of Fault-Tolerant Transactions for Replicated Database Systems Using Event B. volume 4157, LNCS, 343-363, Springer. (2006)
12. Divakar Yadav and Michael Butler : Formal Development of a Total Order Broadcast for Distributed Transactions Using Event B, volume 5454, LNCS,152-176, Springer (2009)