Modeling and Verification of Deadlock Potentials of a Concurrency Control Mechanism in Distributed Databases Using Hierarchical Colored Petri Net

Saeid Pashazadeh, *Senior Member, IACSIT*

***Abstract*—Formal methods are widely used for studying and verifying characteristics of concurrency control mechanisms (CCMs) and protocols in distributed databases. Colored Petri net (CPN) has high modeling capabilities and is one of the best methods for formal analysis and verification of CCMs. In this paper, a novel model of CCM based on two phase locking (2PL) using CPN has presented. State space analysis of model permits us to prove that all schedules of concurrent execution of transactions using 2PL in a case study is deadlock free nor not. Then a simple case study along with its state space analysis has presented. Results show that CPN is proper method for modeling CCM using 2PL and proving deadlock freeness property of all schedules of transactions.**

***Index Terms*—Colored Petri net, concurrency control mechanism, verification, two phase locking, state space analysis**

I. Introduction

Concurrent execution of transactions in distributed databases faces with a lot of problems. Serializable schedules of transactions are correct and equivalence with serial execution of them and preserves databases consistency. CCM is used for isolation and noninterference execution among conflicting transactions to preserve database consistency through consistency preserving execution of transactions. Locking is one of the mechanisms of concurrency control. But simple locking and unlocking of shared data do not guarantee the serializability of transactions [1]. 2PL protocol is one of the well known locking protocols that enforce conflict serializabilty. 2PL has two phases, locking and unlocking phases. In 2PL scheduling all locking operations of transactions must precede their first unlock operation. But 2PL may cause the deadlock. Conservative 2PL is one of the extensions of 2PL which prevents deadlock but greatly decrease the concurrency level [2]. Formal modeling of CCMs is useful in studying different characteristics of them.

II. Related Works

Petri nets are formal methods that benefits from easy graphical user interface. Analyzing the performance of transaction database systems with continuous deadlock detection is studied using stochastic Petri net model [3]. Using Extended Place/Transition nets for formal modeling and performance analysis purposes is done for performance evaluation study on distributed database CCMs [4]. Modeling two phase commit protocol for transaction management in distributed environment is studied using time Petri net [5]. Formal specification for concurrency control of database transactions using conventional Petri net based on 2PL protocol which can ensure the serialization of concurrency scheduling is also studied [6]. Quantitative performance study of 2PL in parallel database systems is performed using a novel simulation-based methodology [7]. This methodology employs a Petri net model for captures the characteristics of parallelism and synchronization at the workload level in the higher level and a queuing network model for captures queuing aspects of the system at the physical resource level at the lower level. Standard 2PL and the primary-copy methods are modeled using high level Petri net (colored Petri net) too [8]. In this paper, a new model of concurrency control based on the 2PL is introduced using colored Petri net.

Manuscript received March 9, 2012; revised April 18, 2012.

Saeid Pashazadeh is with the Department at Faculty of Electrical and Computer Engineering in University of Tabriz in Iran (e-mail: s_pashazadeh@yahoo.com).

III. Colour Sets, Initial Markings and Models of The System

Fig. 1 shows the top level model of the system and Fig. 2 shows the model of each transaction. Colour sets that are used in the models are as follows:

```
colset   RESOURCE = string;
colset   SEQUENCE = int;
colset   ACCESS = string;
colset   LOCK = string;
val      TransactionsNO = 3;
colset   TRANSACTION = index
                        Trans with 1.. TransactionsNO;
colset   BOOLEAN = bool
colset   SEQxACCESSxRES =  product SEQUENCE *
                              ACCESS *RESOURCE;
colset  TRANSLIST = list  TRANSACTION;
colset  TRANSLISTxLOCKxRES  = product TRANSLIST*
                              LOCK*RESOURCE;
```

The colour set RESOURCE defined to represent the name of resources, and its type is the set of all text strings. Colour set SEQUENCE is defined to be equal to the set of all integers and is used for representing sequence number of transaction's instructions (starting from 1). The colour set ACCESS is defined of type text string and is used to model the type of access that each process wants to have on a resource. “LX” is abbreviation for exclusive lock, “LS” for shared lock, “R” for read, “W” for write, and “UL” for unlock access in the model.

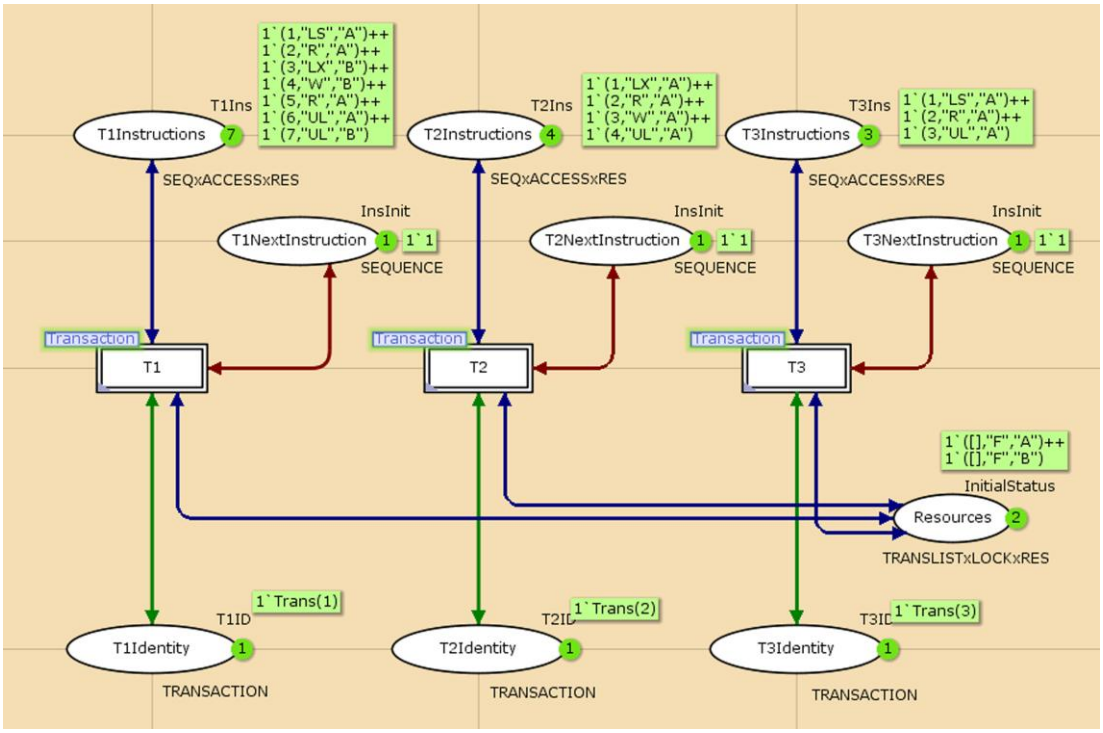

Fig. 1. Top-level module of the hierarchical resource management model.

Colour set LOCK is of type text string and is used for representing the type of lock that currently exists on a resource. “LX” and “LS” is as mentioned before and “F” represents that resource is free and no lock exists on it.

Constant NoTransactions determines the number of model's transactions. This constant is used in the definition of the colour set TRANSACTION such that the colours in this colour set match the number of transactions. The index colour set TRANSACTION is used for modeling the identity of the transactions. This colour set contains three colours: Trans(1), Trans(2), and Trans(3) which identifies the three transactions of the model.

The colour set SEQxACCESSxRES is used to model the instructions of each transaction. Instructions have sequence number, type of operation, and a resource name. The colour set TRANSLIST is defined to model the list of transactions. This list will be used when few processes has shared lock on a resource. Colour set TRANSLISTxLOCKxRES is defined to model the existing locks on a resource. Colour set BOOLEAN is defined of type bool. Defined constants in the model are as follows:

```
val     InsInit = 1`1;
val     ETL = [] : TRANSLIST;
val     InitialStatus =1`(ETL,”F”,”A”)++1`([],”F”,”B”);
val     T1Ins = 1`(1, “LS”,”A”)++1`(2, “R”,”A”)++
            1`(3,”LX”,”B”)++1`(4,”W”,”B”)++
            1`(5,”R”,”A”)++1`(6,”UL”,”A”)++
            1`(7,”UL”,”B”)
val     T1ID = 1`Trans(1)
```

Constant *InsInit* represents the initial marking of place *NextInstrcution* that is shown in the Fig. 2. It represents the next instruction of each transaction that will be executed. Constant *ETL* is defined of colour set TRANSLIST, represents an empty list of transactions, and is used in the definition of constant *InitalStatus*. Constant *InitialStatus* represents that two existing resources of the system are free. Constant *T1Ins* represents the instructions of transaction number one. Constant *T1ID* represents identifier of transaction one (Trans(1)). Variables that are used in the model are as follows:

```
var     S : SEQUENCE;
var     A: ACCESS;
var     OA,NA: LOCK;
var     C: BOOLEAN;
var     R : RESOURCE;
var     CT: TRANSACTION;
var     TL,NTL: TRANSLIST;
```

Varible *OA* represents the old lock type on a resource and *NA* represents the new lock type of it. Variable *CT* represents the current transaction that model runs its instruction. Variable *TL* represents the list of transactions that have specific lock on a resource and *NTL* represent the new list of transactions on a resource after executing current instruction.

IV. Description of Model's Functions

Most of the functionality of the CPN's models is based on the user written functions in functional ML language [9]. Model's functions and their operation are as described follows. In this part, all of the model's functions and their operation are described.

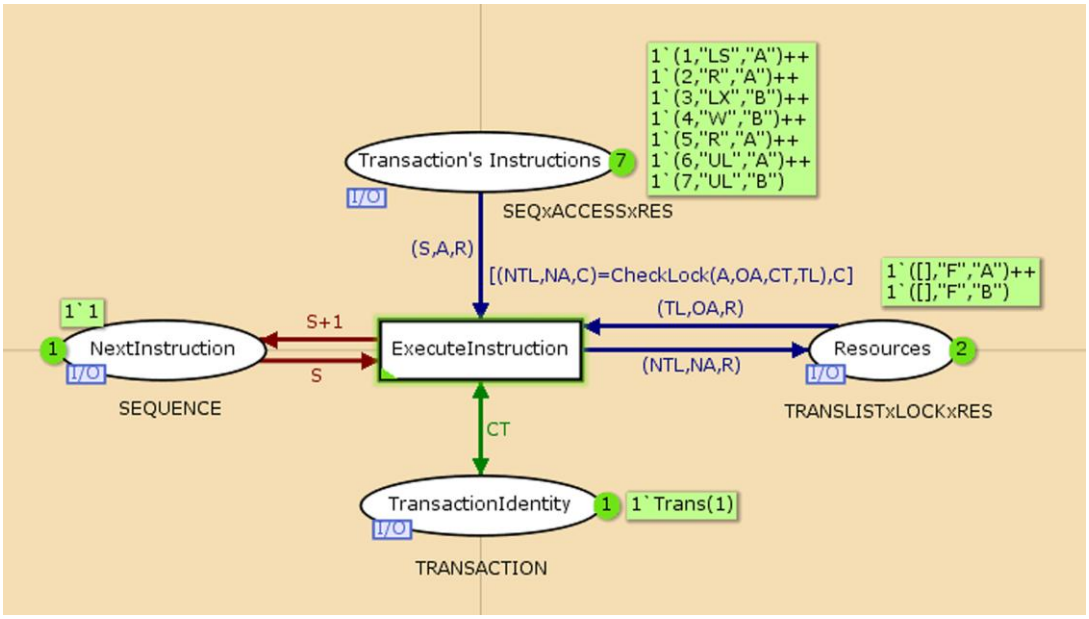

Fig. 2. CPN model of transaction module instance corresponding to transaction T_1.

Function *getTransIndex* takes a transaction identity of colset TRANSACTIAON as first parameter and a list of transcations with colset TRANSLIST as second parameter and returns the index of transaction's position in the list (counting from 0). If list do not contains this transaction, then function returns -1 as result.

```
fun getTransIndex( t , h::L ) : int =
 let   val i =0
 in    if ( h=t) then   i
       else
            if  (getTransIndex(t, L) <> ~1) then
                 getTransIndex(t, L) +1
            else    ~1
 end
| getTransIndex( _,[] ) = ~1;
```

Function *eliminateTrans* takes a transaction identity t of colset TRANSACTION as first parameter and a list of transactions' identities h::L of colset TRANSLIST as second parameter. If the transaction in first parameter exists in the list of transactions in the second parameter, then function returns a list that occurs from eliminating transaction t from the list. In otherwise returns the unchanged list of transactions.

```
fun eliminateTrans (T, L)=
   let   val  index = getTransIndex(T, L)
   in    if (index <> ~1)  then
                 List.take(L,index)^^List.drop(L,index+1)
         else    L
    end
  | eliminateTrans (T,[]) = [];
```

Function *isExists* takes a transaction identity of colset TRANSACTION as first parameter and a list of transactions' identities of colset TRANSLIST as second parameter. This function returns true if the first parameter is exists in the list of second parameter. In otherwise this function returns false.

```
fun isExists(T , TL ) =
   let  val  n = getTransIndex(T,TL)
   in   if n <> ~1 then
          true
    else
          false
   end
   | isExists ( _ , [] ) = false;
```

Function *checkLock* is the most important function of the model that is shown in Fig. 2. First formal parameter of it, RAcc is of colset ACCESS and represents the required access on a resource by current instruction. Second parameter ELock is of colset LOCK and represents the current existing lock on the resource that current instruction wants to have RAcc access on it. Third parameter CurT is of colset TRANSACTION and represents the identity of transaction that current instruction of under study belongs to it. Fourth parameter TList is of colset TRANSLIST and represents the list of transactions that have lock of type ELock on the resource that will be used by current instruction.

This function returns a record (with three fields) as the result. Third field of the result is of colset BOOLEAN. Function returns true for the third parameter when the current instruction can execute based on the locks compatibility rules. Function return false in two different cases. First case is when the current instruction is not permitted to execute based on the existing locks on the resource and lock compatibility rules. After execution of some other instructions may be this instruction can be executed. In this case, second filed of output result which is of colset LOCK, will represent the final lock type on the resource after executing the current instruction. Second case occurs when the instruction has conflict based on the 2PL algorithm. E.x. if a transaction wants to have write operation on a resource before applying an exclusive lock on it. In latter case, the second field of output record contains the error message for better identifying the errors of transactions' instructions for easier trace of the model in execution time. First field of the result is of colset TRANSLIST and represents the list of transactions that have lock on the resource after successful execution of current instruction.

```
fun checkLock( RAcc, ELock , CurT, TList) =
  if  RAcc = “LX” then
         if ELock = “LX” then
               if  isExists( CurT , TList ) then
```

```
                (TList,"Duplicate LX Error",false)
            else
                (TList,ELock,false)
        else if ELock = "LS" then
            if isExists( CurT , TList ) then
                if List.length(TList) > 1 then
                    (TList,ELock,false)
                else
                    (TList,RAcc,true)
            else
                (TList,ELock,false)
        else if ELock = "F" then
            (CurT::TList,RAcc,true)
        else
            (TList,"Unknown Old Access Error",false)
else  if RAcc = "LS" then
        if ELock= "LX" then
            if isExists( CurT , TList ) then
                (TList, "LX to LS Error",false)
            else
                (TList,ELock,false)
            else if ELock = "LS" then
                if isExists( CurT , TList ) then
                    (TList,"Duplicate LS Error",false)
                else
                    (CurT::TList,ELock,true)
            else if ELock = "F" then
                    (CurT::TList,RAcc,true)
                else
                (TList,"Unknown Old Access Error",false)
else if RAcc ="UL" then
        if isExists( CurT , TList ) then
          let val NewTList =eliminateTrans(CurT,TList)
            in
              if List.null NewTList then
                    (NewTList,"F",true)
              else
                    (NewTList, ELock,true)
            end
        else
                (TList,"Unlocks No Lock Error",false)
else if RAcc ="R" then
        if isExists( CurT , TList ) then
                (TList,ELock,true)
        else
                (TList,"No Lock Error",false)
else if RAcc="W" then
        if isExists( CurT , TList ) then
            if ELock = "LX" then
                (TList,ELock,true)
            else
                (TList, "No X Lock Error",false)
        else
            (TList,"No Lock Error",false)
else   (TList,"unknown Access",false);
```

Operation of model by means of function *checkLock* is summarized in Table I, Table II and Table III. Table I represents the operation of the model when transaction CurT requires access RAcc on a resource which transactions of list TList have lock ELock on it and transaction CurT exists in the list of TList. Table II summerises the cases which transaction CurT do not exists in the list TList.

Table III sumerrises the cases which no lock exists on the resource or any unknown lock is presented by mistake on the resource. If RAcc is beyond the range of predefined accesses the function returns record (TList,"unknown Access",false) as result. Fig. 3shows the structure chart of models' functions. Our model contains three transaction and two types of resources as a simple case study. Instructions of each transaction and resources of the system are as shown in Fig. 1.

TABLE I: OPERATION OF SYSTEM WHEN CURRENT REQUESTING TRANSACTION EXISTS IN THE LIST OF TRANSACTIONS THAT HAVE LOCK ON THE RESOURCE.

		Existing Lock	
		Exclusive Lock	***Shared Lock***
Required Access	***Exclusive Lock***	*(TList, "Duplicate LX Error", false)*	If length(TList) > 1 then **(TList, ELock, false)** else (TList, RAcc, true)
	Shared Lock	*(TList, "LX to LS Error", false)*	*(TList, "Duplicate LS Error", false)*
	Unlock	Let val NewTList = eliminateTrans(CurT,TList) in if List.null NewTList then (NewTList, "F", true) else (NewTList, ELock, true) end	
	Read	(TList, ELock, true)	
	Write	(TList, ELock, true)	*(TList, "No X Lock Error", false)*

TABLE II: OPERATION OF SYSTEM WHEN CURRENT REQUESTING TRANSACTION DOES NOT EXISTS IN THE LIST OF TRANSACTIONS THAT HAVE ANY LOCK ON THE RESOURCE

		Existing Lock	
		Exclusive Lock	***Shared Lock***
Required Access	***Exclusive Lock***	**(TList, ELock, false)**	**(TList, ELock, false)**
	Shared Lock	**(TList, ELock, false)**	(CurT::TList, ELock, true)
	Unlock	*(TList,"Unlocks No Lock Error",false)*	
	Read	*(TList, "No Lock Error", false)*	
	Write	*(TList, "No Lock Error", false)*	

TABLE III. OPERATION OF SYSTEM WHEN NO LOCK EXISTS ON THE RESOURCE OR IN APPEARANCE OF UNKNOWN LOCK.

		Existing Lock	
		No Lock	*else*
Required Access	*Exclusive Lock*	(CurT::TList, RAcc, true)	*(TList,"Unknown Old Access Error",false)*
	Shared Lock	(CurT::TList, RAcc, true)	*(TList,"Unknown Old Access Error",false)*
	Unlock	*(TList,"Unlocks No Lock Error",false)*	-
	Read	(TList, ELock, true)	-
	Write	*(TList, "No X Lock Error", false)*	-

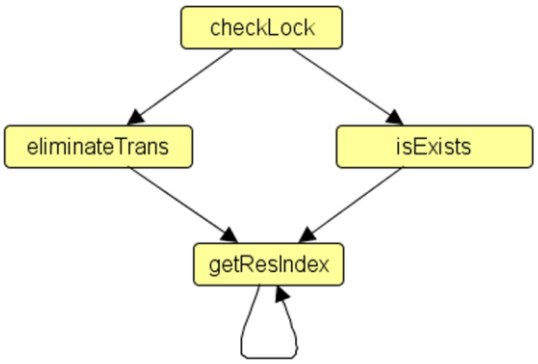

Fig. 3. Structure chart of model's functions.

V. STATE SPACE ANALYSIS

My proposed model presents a simple case study that contains three transaction and two types of resources. Fig. 4 shows the state space of the system's model. Summary report of state space analysis is as follows.

```
Statistics
------------------------------------
State Space
Nodes:  102
Arcs:   174
Secs:   0
Status: Full
Scc Graph
Nodes:  102
Arcs:   174
Secs:   0
Home Properties
------------------------------------
Home Markings
[102]
Liveness Properties
------------------------------------
Dead Markings
[102]
Dead Transition Instances
None
Live Transition Instances
None
Fairness Properties
------------------------------------
.No infinite occurrence sequences
```

State space of model has 102 nodes and 174 arcs. Deadlock of Petri net appears as a node with no outgoing arc. State space analysis shows that the Perti net have a single deadlock state. This state is the deadlock state of the Petri net and is not deadlock state of the system. This state is shown with node 102 in the Fig. 4. This deadlock is desirable and represents a system state that instructions of all transactions executed completely and no lock exists on the resources. This deadlock state of Petri net represents the desirable final state of model's run. However, appearing of a deadlock state which some of instructions of any transactions do not executed represent a deadlock state of the system. This condition shows that some of the transactions felled in the deadlock state and never can finish. Automatic analysis of the system state did by using appropriate computational tree logic (CTL) formulas [10].

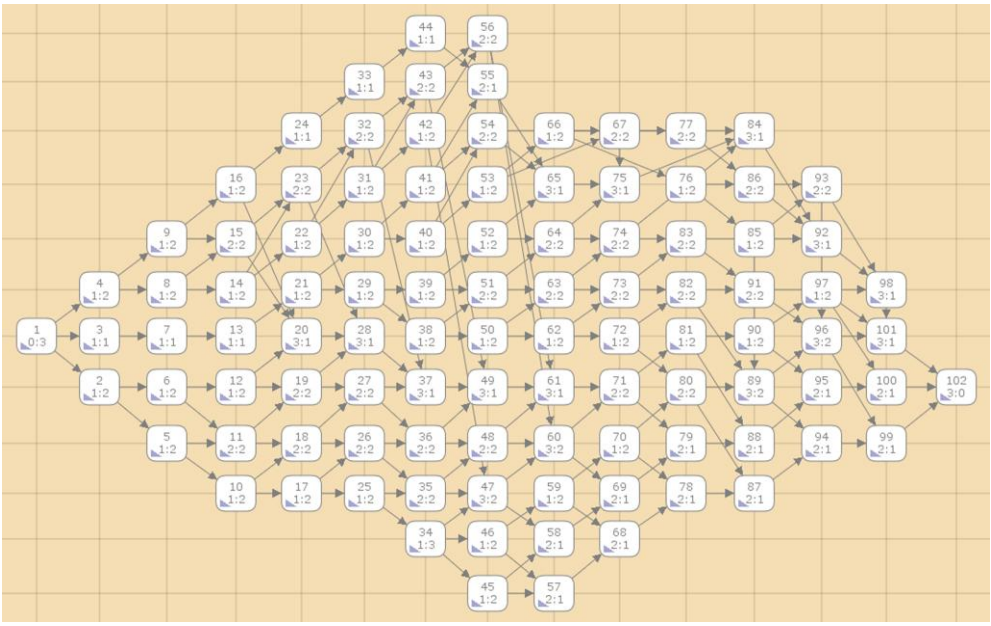

Fig. 4. State space of the system with given sample data.

VI. Conclusion

CPN is flexible and powerful method for modeling and formal analysis of the distributed nondeterministic systems. In this paper a new novel, scalable and extendable model of 2PL CCM using hierarchical CPN presented. State space analysis of the model is done using CTL formulas. State space analysis permits us to prove that all schedules of transactions are deadlock free or not. State space calculation and analysis is done fast and completed in few seconds. Model can easily change for modeling and study of other CCMs like strict 2PL. Model of CPN that used for formal verification, can easily extend for performance analysis too.

References

[1] C. J. Date, *An Introduction to Database Systems*, 8th ed. USA: Pearson Education, 2004, pp. 465-494.

[2] G. Weikum and G. Vossen, Transactional Information Systems Theory, Algorithms, and the Practice of Concurrency Control and Recovery, The Morgan Kaufmann Series in Data Management Systems, J. Gray, Ed. Academic Press, London, United Kingdom, E. W. Dijkstra, "Co-operating Sequential Process," in *Programming Languages*, F. Genuys, Ed. London: Academic Press, pp. 125-166, 2002.

[3] I.-R. Chen and R. Betapudi, "A Petri net model for the performance analysis of transaction database systems with continuous deadlock detection," in *Proc. ACM symposium on Applied computing*, Phoenix, Arizona, USA, 1994, pp. 539-544.

[4] M. T. Ozsu, "Modeling and analysis of distributed database concurrency control algorithms using an extended Petri net formalism," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 10, pp. 1225-1240, Oct. 1985.

[5] B. B. Sarkar and N. Chaki, "Modeling & analysis of transaction management for distributed database environment using Petri nets," in *Proc. World Congress on Nature & Biologically Inspired Computing (NaBIC 2009)*, Dec. 9-11, 2009, pp. 918-923.

[6] H. Jie and L. Fengying and W. Huijiao, "Petri net based model for concurrent control of database system," in *Proc. International Conference on Intelligent Computing and Integrated Systems (ICISS)*, Oct. 22-24, 2010, pp. 813-815.

[7] B.-C. Jenq, B. C. Twichell, and T. W. Keller, "Locking performance in a shared nothing parallel database machine," *IEEE Transactions on Knowledge and Data Engineering*, vol.1, no.4, pp. 530-543, Dec. 1989.

[8] K. Voss, "Prototyping and verifying distributed database systems using executable high-level Petri net models," in *Proc. IEEE International Conference on Systems, Man, and Cybernetics, "Computational Cybernetics and Simulation"*, vol.4, Oct. 12-15, 1997, pp. 3395-3400.

[9] L. C. Paulson, *ML for the Working Programmer*, 2nd ed. NY, USA: Cambridge university press, 1996.

[10] C. Baier and J. P. Katoen, *Principles of Model Checking (Representation and Mind Series)*, Cambridge, Massachusetts, USA: The MIT Press, 2008, pp. 229-433.

Saeid Pashazadeh is Assistant Professor of Software Engineering and chair of Information Technology Department at Faculty of Electrical and Computer Engineering in University of Tabriz in Iran. He received his B.Sc. in Computer Engineering from Sharif Technical University of Iran in 1995. He obtained M.Sc. and Ph.D. in Computer Engineering from Iran University of Science and Technology in Iran in 1998, 2010 respectively. He was Lecturer in Faculty of Electrical Engineering in Sahand University of Technology in Iran from 1999 until 2004. His main interest is in the development, modeling and formal verification of distributed systems, and computer security. He is member of IEEE and senior member of IACSIT.