Multi-step transactions specification and verification in a mobile database community

Rafat Alshorman
Department of Computer Science
Loughborough University
Loughborough, LE11 3TU, UK
Email: R.Alshorman@lboro.ac.uk

Walter Hussak
Department of Computer Science
Loughborough University
Loughborough, LE11 3TU, UK
Email: W.Hussak@lboro.ac.uk

***Abstract*—Executions of concurrent multi-step transactions interleave steps in ways that improve the throughput of the particular transactions processing system. In this paper, we use temporal logic to specify and verify formally the correctness of local and mobile transactions executing concurrently on a mobile database. The correctness condition is that of serializability which we specify in CTL (Computational Tree Logic). The reason for using a temporal logic such as CTL, is that the method can be extended to verifying infinite schedules modelling mobile environments such as MDBCs (mobile database communities). The verification is carried out using the symbolic model checking NuSMV. We verify that a local scheduler based on timestamps serializes local and mobile multi-step transactions.**

I. Introduction

In a MDBC (mobile database community), autonomous, distributed, heterogenous and mobile databases are interconnected through a wireless communication infrastructure such as a MANET (mobile ad hoc network). There are two kinds of transactions to be executed concurrently on any mobile database participating in the MDBC, mobile transactions and local transactions. Formally, a MDBC consists of:

1) a set $\mathcal{MDBC} = \{mdbs_1, mdbs_2, mdbs_3, \ldots, mdbs_k\}$ of mobile databases systems, $k > 1$;
2) a set $\underline{\mathcal{L}} = \{\mathcal{L}_1, \ldots, \mathcal{L}_k\}$ of sets of local transactions, where each $\mathcal{L}_h$ denotes the set of local transactions executing at the mobile database system $mdbs_h$, and;
3) a set $\mathcal{M} = \{M_1, \ldots, M_p\}$ of mobile transactions.

In 1) we assume that the mobile databases in the MDBC comprise *pairwise disjoint sets of objects (or data items)*. Also, that any mobile transaction update of an object in one mobile database does not depend on the state of other mobile databases. As in [1], a mobile transaction M_i, here, is *module-structured*, i.e. its operations are grouped into subsequences, called *modules*, where each module contains the operations for one mobile database participating in the MDBC. For example, let $mdbs_i$ and $mdbs_j$ participate in the MDBC, such that $mdbs_i$ contains the set of data items $\{a, b, c, d\}$, $mdbs_j$ contains the set data items $\{e, f, g, h\}$. An instance of M_i as a module-structured transaction with modules is given below:

$$M_i = \underbrace{r_i(a)w_i(a)r_i(b)w_i(b)}_{\text{module1}} \overbrace{r_i(f)w_i(f)r_i(g)w_i(g)}^{\text{module2}}$$

The correctness criterion for the concurrent execution of these transactions is that of serializability. Serializability in this context means that the concurrent transactions (local and mobile) should execute in a schedule or *history* whose effect is equivalent to a serial execution of the transactions. The basic problem is to determine whether histories, generated by some scheduler, are serializable. A *serializable* history is a history S, if it is *equivalent* to some serial history of the same transactions. This definition of serializable history requires us to define the equivalence of two histories. Basically, we say that the two histories are *equivalent* if they produce the same final database state, and every read operation reads the same write operation in the both histories (see, for example, [5]). The objective of this paper is to specify the correctness of transactions (local and mobile) executing concurrently on a mobile database in terms of serializability using specifications written in CTL (Computational Tree Logic). The reason for using a temporal logic such as CTL, is that the method can be extended in order to verify infinite schedules as occur in mobile environments where transactions are incoming and outgoing in a continuous stream. The importance of temporal logic in computer science is clear, especially in the specification and verification of critical reactive systems. Model checkers such as NuSMV of many variants of temporal logics have been developed to the extent that they can deal with a huge number of states and verify real-world systems.

This paper is organized as follows. In Section II, we give concurrency techniques used to ensure the serializability of histories in terms of protocols that guarantee serializability. Our multi-step transactions definition and model is given in Section III. The encoding into CTL is given in Section IV. Section V contains the verification and implementation in NuSMV of the multi-step transactions model. We give some conclusions in Section VI.

II. Concurrency techniques for ensuring serializability

Basically, there are two main kinds of protocols used to ensure the serializability of concurrent transactions, *locking* protocols and protocols using *timestamps* [6]. Locking protocols lock objects (data items) to prevent multiple transactions from accessing the objects concurrently, as we discuss in the

next subsection. Timestamp protocols use a timestamp, as a unique identifier associated with every transaction and generated by the system, to ensure the serializability of concurrent transactions by ordering transactions timestamps.

A. Locking Protocols

One of the most popular locking protocols is Two-Phase Locking (2PL). It is used to manage concurrent transactions accessing database systems ensuring that the history produced from these transactions is serializable. The idea behind this protocol is to associate a lock and waiting queue with every object (the object size depends on the object or data item granularity that has been used) in the database. When a transaction T_i tries to access the data item x, the scheduler first checks the associated lock. If there is no transaction holding the lock, then the scheduler grants the lock to T_i. Otherwise the transaction T_i is added to the waiting queue associated with x. This means that, the scheduler will not grant T_i the lock until T_j gives it up [2]. Deadlock may occur which such a locking technique. As such, deadlock prevention schemes based on timestamps [6], have been developed to deal with transactions involved in a deadlock situation.

B. Timestamp Protocols

Timestamp-based concurrency control is a method of non-lock concurrency control; hence, deadlock cannot occur. In this paper we choose a scheduler based on a timestamp protocol which we implement in NuSMV in Section V and check whether histories generated satisfy the correctness condition of serializability. We use the basic timestamp ordering (TO) algorithm as a scheme to ensure that the histories produced are guaranteed to be 'conflict' serializable. A history S is *conflict serializable* if it is 'conflict equivalent' to some serial history of the same transactions. Two histories are *conflict equivalent* if the order of any conflicting operations is the same in both. Two operations are *conflicting* if they are from different transactions, they are on the same object or data item and at least one of them is a write operation [2]. The idea of the basic timestamp ordering (TO) algorithm is to order the transactions according to their timestamps. We will denote the timestamp of transaction T_i as $TS(T_i)$. A transaction T_i obtains a timestamp $TS(T_i)$ when it starts, i.e. when the first step of transaction T_i has arrived. Consequently, $TS(T_i) < TS(T_j)$ when T_i starts before T_j and we say that T_i is *older* than T_j. The history produced by the TO algorithm is equivalent to the serial order of transactions corresponding the order of their timestamps. The TO algorithm associates two timestamps with every object x, a read timestamp and a write timestamp denoted by $RTS(x)$ and $WTS(x)$, respectively. $RTS(x)$ is the largest timestamp among all transactions that have successfully read object x. This means that $RTS(x) = TS(T_i)$, if T_i is the youngest transaction to have read object x successfully. $WTS(x)$ is the largest timestamp among all transactions that have successfully written to object x. This means that $WTS(x) = TS(T_i)$, if T_i is the youngest transaction to have written to object x successfully. The basic TO algorithm compares the timestamps of transactions with $RTS(x)$ and $WTS(x)$ to make sure that the timestamp order of the transactions is not violated as follows:

1) T_i issues a write operation on object x:
   - If $RTS(x) > TS(T_i)$ or if $WTS(x) > TS(T_i)$, then abort and rollback T_i and reject the write operation on object x.
   - If the above condition is not satisfied, execute the write operation and $WTS(x) = TS(T_i)$.
2) T_i issues a read operation on object x:
   - If $WTS(x) > TS(T_i)$, then abort and rollback T_i and reject the read operation on object x.
   - If $WTS(x) \leq TS(T_i)$, execute the read operation and $RTS(x) = Max\{TS(T_i), RTS(x)\}$.

There is another reason to choose a scheduler based on the basic TO algorithm to be implemented in our NuSMV model - it can be used to give any transaction higher priority than the others. This is useful in our model as we will discuss in Section III.

III. Multi-step transactions model

Dividing the transactions into sets of steps improves system throughput and allows transactions interleaving to gain more parallelism. An example of a multi-step transaction is a user entering data using a sequence of forms. At the end of the sequence, the application processes the input data. Desktop applications using wizards to simplify operations are a good example of multi-step transactions. In an e-commerce site, the checkout flow can be seen as a multi-step transaction where a user, starting from the shopping cart page, goes through the shipping, billing, confirmation and finally a thank you page. Formally, a database system consists of a set D of data items and a set $T = \{T_1, \ldots, T_n\}$ of transactions. A *read step* $r_i(x)$, and a *write step* $w_i(x)$ is defined for $x \in Z \subseteq D$. We propose multi-step transactions with m data items of the form:

$$T_i = r_i(x_1)w_i(x_1)r_i(x_2)w_i(x_2)\ldots r_i(x_m)w_i(x_m)$$

for $1 \leq i \leq n$. These execute in the mobile database system asynchronously at any point of time. A local transaction scheduler in any mobile database system must decide on-line if it can grant each arriving read and write request immediately without violating the serializability correctness criteria. We assume that the mobile transactions have *higher priority* than local transactions. The reason for our priority assumption is that at any time a mobile transaction may transiently disconnect from the network (due to communication disruption or to save power), so it is reasonable to give it priority over local transactions in the history.

IV. Encoding into CTL

A. Multi-step Serializability

In line with the definition of multi-step transactions in Section III, we assume that we have $n-1$ transactions with m data items contending to execute at the local host for the mobile database $mdbs_i$, all with the same priority, and one

mobile transaction denoted by M_n, similar to the local ones (the same operations sequence) but with higher priority, such that:

$$\begin{aligned} T_1 &= r_1(x_1)w_1(x_1)\dots r_1(x_m)w_1(x_m) \\ T_2 &= r_2(x_1)w_2(x_1)\dots r_2(x_m)w_2(x_m) \\ &\vdots \\ T_{n-1} &= r_{n-1}(x_1)w_{n-1}(x_1)\dots r_{n-1}(x_m)w_{n-1}(x_m) \\ M_n &= r_n(x_1)w_n(x_1)\dots r_n(x_m)w_n(x_m) \end{aligned}$$

We assume that the history should satisfy the following condition to be conflict serializable:

1) If a transaction T_i begins executing a read operation on data item x_j, no read or write operation on data item x_j by any other transactions occurs (executes) until the write operation of T_i completes its execution on data item x_j, i.e.:

$$r_i(x_j) \underbrace{\dots\dots\dots\dots\dots\dots\dots\dots}_{\text{no } r_k(x_j) \text{ or } w_k(x_j)} w_i(x_j)$$

2) If T_i precedes T_k in accessing a data item x_j in a history, then T_i should precede T_k in accessing all other data items for both read and write operations. For example, assume that we have 3 transactions with $m = 2$ (two-step transactions), i.e.:

$$\begin{aligned} T_1 &= r_1(x_1)w_1(x_1)r_1(x_2)w_1(x_2) \\ T_2 &= r_2(x_1)w_2(x_1)r_2(x_2)w_2(x_2) \\ T_3 &= r_3(x_1)w_3(x_1)r_3(x_2)w_3(x_2) \end{aligned}$$

and suppose that T_1 precedes T_2 and T_2 precedes T_3 in arriving at the scheduler. The following history S would be conflict serializable:

$$\begin{aligned} S = {} & r_1(x_1)w_1(x_1)r_1(x_2)r_2(x_1)w_2(x_1)w_1(x_2) \\ & r_3(x_1)w_3(x_1)r_2(x_2)w_2(x_2)r_3(x_2)w_3(x_2) \end{aligned}$$

As T_1 precedes T_2 and T_3 in accessing a data item x_1 in a history S, every data item operation (read/write) belonging to T_1 should precede (in the history) the operations belong to T_2 and T_3 on the same data item. As T_2 precedes T_3 a similar condition should hold for them.

B. Computational Tree Logic (CTL)

Computational Tree Logic (CTL) is a temporal logic where the model of time is a like-tree structure in which the future is not determined. This means, there are different paths in the future. It is useful to specify and verify the correctness of computer systems, whether they are hardware, software, or a combination [7] and achieves polynomial-time model checking [4]. It is worthwhile using CTL to specify multi-step transactions to gain polynomial-time model checking and as a first step to specifying infinite histories of multi-step transactions.

C. Encoding Serializability Into CTL

The encoding of the serializability conditions of section IV into CTL will be as follows:

$$\bigwedge_{1\le i\le n}\ \bigwedge_{1\le j\le m}\ \bigwedge_{1\le k\le n, k\neq i} AG[r_i(x_j) \Longrightarrow A((\neg r_k(x_j) \wedge \neg w_k(x_j))U w_i(x_j))] \quad (1)$$

$$\bigwedge_{1\le i\le n}\ \bigwedge_{1\le j\le m}\ \bigwedge_{1\le k\le n, k\neq i}\ \bigwedge_{1\le l\le m, l>j} AG[(w_i(x_j)\wedge EX r_k(x_j)) \Longrightarrow A(\neg r_k(x_l)U w_i(x_l))] \quad (2)$$

The CTL formula (1) represents the serializability condition 1. It specifies that at any point of time *for all paths always* (AG), if $r_i(x_j)$ is true then, *for all paths* (A) no read or write operation on the data item x_j will occur by any other transactions *until* (U) $w_i(x_j)$ occurs. The CTL formula (2) represents the serializability condition 2. At any point of time, for all paths always, if a write operation of T_i on data item x_j, $w_i(x_j)$, occurs and *there exists a path such that at the next point in time* (EX) a read operation belonging to a different transaction T_k, on the same data item x_j, $r_k(x_j)$ occurs then, for all paths (A) no read operation of T_k on any subsequent data item x_l ($l > j$) should occur until the write of T_i to x_l $w_i(x_l)$ occurs. Therefore, if T_i precedes T_k on the first data item x_1 in a history, then T_i should precede T_k for accesses to all other data items x_l, where $l > 1$.

V. Verification of the Multi-Step Transactions Model

In NuSMV, the system to be verified is modeled as a finite state transition system, and the specifications are expressed in either CTL or LTL (Linear Temporal Logic). Then, by exploring the state space of the state transition system, it is possible to check automatically if the implementation satisfies the specification. The termination of model checking is guaranteed by the finiteness of the model. One of the most important features of model checking is that, when a specification is found not to hold, a counterexample is produced [8]. We specify and verify our multi-step transactions model as a finite state machine in the input language of NuSMV [8] in Subsection D. We make additional assumptions on the read and write steps of multi-step transactions as follows:

1) If a write operation on x_j has completed execution, it becomes false at the next point of time. This means that any write operation is true during execution and then becomes false at the next point of time and remains false to the end of schedule. Formally, in CTL

$$\bigwedge_{1\le i\le n}\ \bigwedge_{1\le j\le m} AG[w_i(x_j) \Longrightarrow AX\neg w_i(x_j)] \quad (3)$$

2) If the write operation to x_j has occurred, a corresponding read operation to x_j should have occurred beforehand. This means there are no occurrences of

writes before reads to the same data item by the same transaction. Formally, in CTL

$$\bigwedge_{1\leq i\leq n}\ \bigwedge_{1\leq j\leq m} AG[w_i(x_j) \Longrightarrow r_i(x_j)] \qquad (4)$$

We use keywords in the NuSMV model having the following meaning:

- MODULE : Either the main module or a subroutine.
- VAR : Defines variables.
- SPEC : Specifies the serializability conditions in CTL.
- ASSIGN : Defines transition relations for variables.
- init : Defines initial conditions of the variables.
- next : Defines a relationship between values of variables in a particular state and its successor state.
- Process: Defines a collection of parallel processes, whose actions are interleaved asynchronously.
- - - : Indicates a comment.

In our NuSMV model, given in Subsection D, there are two multi-step transactions - a local transaction and a mobile transaction (lines 90-91). They are contending to execute on the local host for the mobile database $mdbs_i$.

A. The Model Variables

The operations of the transactions are declared as ops1 and ops2 in the main module (lines 77-78) so that every transaction contains four steps (operations). We denote $r_1(x_1)$ and $w_1(x_1)$ by r1x and w1x, respectively. We denote x_2 by y, therefore $r_1(x_2)$ is r1y in the NuSMV model. The variables opt1 and opt2 (line 1) in module Tran are the formal parameters that receive ops1 and ops2, respectively.

We assume that the system timestamp is a counter TS taking values from 0 to 16 (line 80) and that every transaction may rollback at any operation a multiple of times; so the maximum number of rollbacks for every transaction will be 4 (number of operations) multiplied by 2 (number of transactions). Each transaction has a timestamp as given in lines 82-83. The timestamp TS_l2 (line 83) is for the mobile transaction TM (line 91). It takes the values 15..16 so as to give the mobile transaction priority over the local transaction T1 (line 90).

Every data item has two timestamps, a read and a write timestamp, denoted by RTS_x (or RTS_y) and WTS_x (or WTS_y), respectively (lines 85-88). The local variable abort (line 3) takes the value true if the transaction has to rollback; otherwise it is false. A variable corresponding to a scheduler action sch is declared in line 89. The required properties of the scheduler including serializability are given in the SPEC section in lines 107-113.

B. Implementation Of The Timestamp Protocol

As we mentioned in Section II, the basic TO algorithm is a scheme to ensure that the histories produced are conflict serializable. The basic TO algorithm compares the timestamps of transactions with $RTS(x)$ and $WTS(x)$ to make sure that the timestamp order of the transactions is not violated. When the transaction T_i issues a read operation on data item x (or y) first of all, it should check if the write timestamp of the data item x is greater than the timestamp of the transaction T_i ($WTS(x) > TS(T_i)$). If so, the transaction T_i should abort. We encode this condition in the variable abort (lines 58-61). The variable abort takes the value true at the next point of time if the previous condition is satisfied and then this will affect the value of opt1 (or opt2) which will return to the value none as in lines 8 and 16. This means that the transaction T_i aborts and returns to the initial state. Moreover, it will not be added to the schedule (or history) (line 69) and the timestamp of the transaction T_i will be affected at the next point of time, when the value of opt1 (or opt2) change from none to r1x (or rmx). This implies that it should set a new timestamp to the transaction T_i as in lines 25 and 30.

If the condition ($WTS(x) > TS(T_i)$) is not satisfied, then the operation should be executed as in lines 9-12 and 17-20. Consequently, the transaction timestamp will be unchanged (lines 26 and 31) and the schedule sch will add the operation (lines 71-72). The read timestamp of the data item $RTS(x)$ (or $RTS(y)$) will take the maximum value of $TS(T_i)$ and $RTS(x)$ as is encoded in lines 34-37 and 41-44. When the transaction T_i issues a write operation on data item x (or y) the basic TO algorithm checks whether $RTS(x) > TS(T_i)$ or $WTS(x) > TS(T_i)$ are satisfied or not. If one of them is satisfied, then the transaction T_i aborts (lines 62-65) and opt1 will return to the none state (line 8 of subsection D) and the timestamp will be set to a new value in the next point of time (lines 25 and 30). The schedule sch will remain in the same state without adding the operation (line 69). On the other hand, if the condition ($RTS(x) > TS(T_i)$ or $WTS(x) > TS(T_i)$) is not satisfied, then the operation will execute (lines 9-13 and 17-20). The timestamp will remain as in its previous state (lines 26 and 31) and the write timestamp of data item x (or y) will take the timestamp of the transaction (lines 48-49 and 53-54). The schedule sch will add the operation as at lines 71 and 72.

C. Example Of The Model Run

In this subsection we describe, briefly, runs of the model in the NuSMV model checker. The CTL specifications are evaluated by NuSMV in order to determine their truth or falsity in the finite state machine model. When a specification is discovered to be false, NuSMV constructs and prints a counterexample, i.e. a trace of the FSM that falsifies the property. In NuSMV a CTL specification is given as a CTL formula introduced by the keyword SPEC (lines 107-113). For our example, we specify the formulae (1) and (2) in NuSMV in lines 107-109. The run of NuSMV produces true which means that the serializability conditions are true on the histories produced by the basic TO algorithm here. Formulae (3) and (4) are specified in lines 110-112.

Assume that the condition 2 of conflict serializability is

exchanged for the condition:

$$\bigwedge_{1\le i\le n}\ \bigwedge_{1\le j\le m}\ \bigwedge_{1\le k\le n, k\neq i}\ \bigwedge_{1\le l\le m, l>j} AG[(w_i(x_j)\wedge EXr_k(x_j)) \Longrightarrow E(r_k(x_l)Uw_i(x_l))]. \quad (5)$$

This allows for T_i to precede T_k in accessing a data item x_j and yet for T_i to not necessarily precede T_k in accessing all other data items in both read and write operations. We specify formula (5) in NuSMV in line 113. We find the specification is false and a counterexample is produced to show that the new serializability condition is not true on the histories produced by the basic TO algorithm.

D. NuSMV Code

```
1 MODULE Tran(Tran_no,opt1,TS,RTS_x,WTS_x,RTS_y,
                WTS_y,sch,TS_l1,TS_l2,opt2)

2 VAR
3 abort:boolean;

4 ASSIGN
5 init(abort):=0;
6 --When abort =1, the opt should return to none value

7 next(opt1):=case
8 (next(abort)=1):none;
9   (opt1=none)&(next(abort)!=1)& (Tran_no=0d3_1) : r1x;
10 (opt1=r1x)&(next(abort)!=1)&   (Tran_no=0d3_1) : w1x;
11 (opt1=w1x)&(next(abort)!=1)&  (Tran_no=0d3_1) : r1y;
12 (opt1=r1y)&(next(abort)!=1)&   (Tran_no=0d3_1) : w1y;
13 1                                              : opt1;
14 esac;

15 next(opt2):=case
16 (next(abort)=1) :none;
17 (opt2=none)&(next(abort)!=1)& (Tran_no=0d3_2) : rmx;
18 (opt2=rmx)&(next(abort)!=1) & (Tran_no=0d3_2) : wmx;
19 (opt2=wmx)&(next(abort)!=1)& (Tran_no=0d3_2) : rmy;
20 (opt2=rmy)&(next(abort)!=1) & (Tran_no=0d3_2) : wmy;
21 1                                              : opt2;
22 esac;

23 next(TS_l1):=
24 case
25 (next(opt1)=r1x)& (Tran_no = 0d3_1) & (TS<16):TS+1;
26 1                                              :TS_l1;
27 esac;

28 next(TS_l2):=
29 case
30 (next(opt2)=rmx)& (Tran_no = 0d3_2) :16;
31 1                                    :TS_l2;
32 esac;

33 next(RTS_x):= case
34    ((next(opt1)=r1x))&    (WTS_x   <=    next(TS_l1))&
      (RTS_x >next(TS_l1)) : RTS_x;
35    ((next(opt1)=r1x))&    (WTS_x   <=    next(TS_l1))&
      (RTS_x <=next(TS_l1)): next(TS_l1);
36    ((next(opt2)=rmx))&    (WTS_x   <=    next(TS_l2))&
      (RTS_x >next(TS_l2)) : RTS_x;
37    ((next(opt2)=rmx))&    (WTS_x   <=    next(TS_l2))&
      (RTS_x <=next(TS_l2)): next(TS_l2);
38 1                                  : RTS_x;
39 esac;

40 next(RTS_y):= case
41    ((next(opt1)=r1y))&    (WTS_y   <=    next(TS_l1))&
      (RTS_y >next(TS_l1)) : RTS_y;
42    ((next(opt1)=r1y))&    (WTS_y   <=    next(TS_l1))&
      (RTS_y <=next(TS_l1)): next(TS_l1);
43    ((next(opt2)=rmy))&    (WTS_y   <=    next(TS_l2))&
      (RTS_y >next(TS_l2)) : RTS_y;
44    ((next(opt2)=rmy))&    (WTS_y   <=    next(TS_l2))&
      (RTS_y <=next(TS_l2)): next(TS_l2);
45 1                                  : RTS_y;
46 esac;

47 next(WTS_x):= case
48    (next(opt1)=w1x) & ( WTS_x <= next(TS_l1) ) &
      ( RTS_x <= next(TS_l1) ) : next(TS_l1);
49    (next(opt2)=wmx) & ( WTS_x <= next(TS_l2) ) &
      ( RTS_x <= next(TS_l2) ) : next(TS_l2);
50 1                                  : WTS_x;
51 esac;

52 next(WTS_y):= case
53    (next(opt1)=w1y) & ( WTS_y <= next(TS_l1) ) &
      ( RTS_y <= next(TS_l1) ) : next(TS_l1);
54    (next(opt2)=wmy) & ( WTS_y <= next(TS_l2) ) &
      ( RTS_y <= next(TS_l2) ) : next(TS_l2);
55 1                                  :WTS_y;
56 esac;

57 next(abort):= case
58 (opt1=r1x)& (WTS_x > TS_l1)                          : 1;
59 (opt2=rmx)& (WTS_x > TS_l2)                          : 1;
60 (opt1=r1y)& (WTS_y > TS_l1)                          : 1;
61 (opt2=rmy)& (WTS_y > TS_l2)                          : 1;
62 (opt1=w1x)& ((RTS_x > TS_l1)|(WTS_x > TS_l1))        : 1;
63 (opt2=wmx)& ((RTS_x > TS_l2)|(WTS_x > TS_l2))        : 1;
64 (opt1=w1y)& ((RTS_y > TS_l1)|(WTS_y > TS_l1))        : 1;
65 (opt2=wmy)& ((RTS_y > TS_l2)|(WTS_y > TS_l2))        : 1;
66 1                                                    : 0;
67 esac;

68 next(sch):=case
69 (next(abort) = 1)                    : sch;
70 ((opt1=w1y)&(opt2=wmy))              : sch;
71 (Tran_no=0d3_1)&(next(abort) = 0) : next(opt1);
72 (Tran_no=0d3_2)&(next(abort) = 0) : next(opt2);
73 1                                    : sch;
```

```
74 esac;

75 MODULE main

76 VAR
77 ops1 : {none,r1x,w1x,r1y,w1y};
78 ops2 : {none,rmx,wmx,rmy,wmy};
79 −−Global time stamp
80 TS : 0..16;
81 −−transactions timestamps
82 TS_1 : 0..16;
83 TS_2 : 15..16;
84 −−Read/Write time stamp to data item X and Y
85 RTS_x  : 0..16;
86 WTS_x : 0..16;
87 RTS_y  : 0..16;
88 WTS_y : 0..16;
89 sch : {none,r1x,w1x,r1y,w1y,rmx,wmx,rmy,wmy};

90 T1  : process Tran(0d3_1,ops1,TS,RTS_x,WTS_x,RTS_y,
                    WTS_y,sch,TS_1,TS_2,ops2);
91 TM : process Tran(0d3_2,ops1,TS,RTS_x,WTS_x,RTS_y,
                    WTS_y,sch,TS_1,TS_2,ops2);

92 ASSIGN
93 init(ops1)     :={none};
94 init(ops2)     :={none};
95 init(TS)       :=1;
96 init(TS_1)     :=1;
97 init(TS_2)     :=16;
98 init(RTS_x)   :=0;
99 init(WTS_x)  :=0;
100 init(RTS_y) :=0;
101 init(WTS_y):=0;

102 next(TS):= case
103    TS<16 :TS+1;
104 1           :TS;
105 esac;

106 init(sch):=none;

107 SPEC AG (sch=r1x − > A [(sch!=rmx & sch!=wmx)
                  U sch=w1x])
108 SPEC AG (sch=rmx − > A [(sch!=r1x & sch!=w1x)
                  U sch=wmx])
109 SPEC AG ((sch=wmx & EX sch=r1x) − > A[sch!=r1y
                  U sch=wmy] )
110 SPEC AG (sch=wmx − > AX sch!=wmx)
111 SPEC AG (sch=w1x − > AX sch!=w1x)
112 SPEC AG (sch=w1x − > sch=r1x)
113 SPEC AG ((sch=wmx & EX sch=r1x) − > E[sch=r1y
                  U sch=wmy] )

114 FAIRNESS
115 running
116 −−By adding the declaration:FAIRNESS running
117 −−to the module, we can effectively force every instance
      of the module to execute infinitely often.
```

VI. Conclusion

In this paper, we have shown that serializability conditions can be easily encoded into CTL demonstrating how schedulers can be verified using such model checkers. In fact the verification runs in polynomial time, i.e. if we are given a scheduler specified by a transition system of size n and a serializability condition expressed as a CTL formula of size m, then the CTL model checking algorithms runs in $O(nm)$ time [7]. This is a first step towards specifying infinite histories of multi-step transactions using temporal logic and extending previous work on two-step transactions [5] and [3]. Infinite histories are growing in importance with the emergence of new technologies such as mobile transactions. It is useful to conduct proofs of such infinite histories using fully automated techniques to avoid numerous disadvantages of manual proofs. Further work will look to extend the NuSMV model checker by adding scripts to generate CTL specifications automatically for such kinds of problems.

Acknowledgment

We would like to thank ZPU (Zarqa Private University) for its grant in making this work possible.

References

[1] A. Brayner, J.A. Filho, Increasing Mobile Transaction Concurrency in Dynamically Configurable Environments, Proceedings of the 25th IEEE International Conference on Distributed Computing Systems Workshops, ICDCS Workshops 2005, 637-641.

[2] C.H. Papadimitriou, The Theory of Database Concurrency Control. Computer Science Press, Pockville, Maryland, 1986.

[3] W. Hussak, Specifying Strict Serializability of Iterated Transactions in Propositional Temporal Logic, International Journal of Computer Science, 2(2), 2007, 150-156, ISSN 1306-4428.

[4] Ph. Schnoebelen, Specifying Systems, The Complexity of Temporal Logic Model Checking, Advances in Modal Logic, Volume 4, 1-44, 2002, by World Scientific Publishing Co. Ltd.

[5] W. Hussak, Serializable Histories in Quantified Propositional Temporal Logic, International Journal of Computer Mathematics, 81(10), 2004, 1203-1211, ISSN 1029-0265.

[6] R. Elmasri, S. Navathe, Fundamental of Database Systems Addison-Wesley, Fourth Edition , ISBN 0-8053-1755-4.

[7] E. Clarke Jr., O. Grumberg, and D. Peled *Model checking*. The MIT press,Cambridge, Massachusetts 1999.

[8] A. Cimatti, E. Clarke, F. Giunchiglia, M. Roveri, NuSMV: A New Symbolic Model Verifier. In Proceedings of 11th Conference on Computer-aided Verification, Lecture Notes in Computer Science, Springer, Trento, Italy, vol. 1633, 1999, 495-499.

[9] Ph. Schnoebelen, The Complexity of Temporal Logic Model Checking, Advances in Modal Logic 2002, 393-436.