Using Temporal Logics to Specify and Verify Multi-step Transactions in Mobile Environments

by

Rafat Alshorman

A Doctoral Thesis

Submitted in partial fulfilment of the requirements for the award of

Doctor of Philosophy

of

Loughborough University

3 July 2009

Copyright 2009 Rafat Alshorman
Abstract

The advent of mobile and web technologies has given rise to unlimited numbers of concurrent users executing their transactions in databases in continuous streams. In order to prove correctness, a method of modelling the behaviour of such transactions is required. Most approaches to proving the correctness of the concurrent execution of transactions, have relied on mathematical proofs. These have numerous disadvantages such as: the person who performs the proof needs to be an expert in mathematical proof techniques, the possibility of human error occurring in manual proofs, and if a simplified model is used not all system behaviours are covered and not all properties can be proved. With regard to the last point, most models assume a fixed finite number of transactions.

In the first part of this thesis, we present a model of an unlimited number of multi-step transactions occurring in web and mobile environments over time, where a finite number of the possible different transactions repeat or 'iterate' infinitely often. We define temporal logics for specifying and verifying required properties of the model. The attraction of using temporal logics is the availability of powerful model checkers that can perform verification automatically. These verify by an exhaustive search of the state space of the model containing all possible system behaviours. We define two common conditions on transactions which we prove result in a simple specification of the main serializability property in temporal logic. The first condition is when transactions access the same set of data items in different orders, and the second condition is when transactions access different sets of contiguous data items.

In the second part of the thesis, we apply the specification method to model executions of multi-step transactions occurring in mobile environments. A protocol that can schedule an unlimited number of multi-step transactions accessing a finite set of data items is specified, along with its required properties, using two types of temporal logic - namely CTL (Computational Tree Logic) and LTL (Linear-time Temporal Logic). We conduct automatic verification of serializability of the protocol, in order to check its correctness. We also compare the complexity and suitability of CTL and LTL for such protocols. An extension of the protocol is specified to deal with environments having a mixture of local and mobile tran-
transactions. Later, we consider a justifiable modification of the proposed protocol. The model checker shows how this modification can make the modified protocol an incorrect protocol. Finally, one of the advantages of using temporal logic to model infinitely many transactions, is that the property of starvation can be specified. We give an example for which the run of the model checker demonstrates the presence of starvation.
Acknowledgements

First of all, I would like to thank my supervisor, Dr Walter Hussak, for being so kind and supportive during my stay at Loughborough University and also for showing me the right way of research, and for helping me many times.

I would also like to thank my DoR (Director of Research), Dr. Helmut Bez, for keeping an eye on my progress and providing many useful suggestions.

I am very grateful to my wife Duaa, daughter Raneem, parents, brothers and sisters, for their support, encouragement and love.

I have met many people during the course of my PhD. I thank everyone with whom I have had interactions, both in research and otherwise.

Finally, I am thankful for the financial support I have received from ZPU (Zarqa Private University).
Contents

Abstract iv
Acknowledgements vi

1 Introduction 1
1.1 Basic concept of database transactions 3
1.1.1 The need for concurrency control 4
1.1.2 Formal transaction definition of a transaction 6
1.1.3 Characteristics of schedules based on serializability 6
1.1.4 Conflict graph 7
1.1.5 Multi-step transactions and infinite histories 8
1.2 Verification of concurrency control protocols in mobile systems 9
1.3 Temporal Logic 10
1.3.1 Temporal properties 10
1.3.2 Temporal operators 11
1.3.3 Expressing system properties in temporal logic 12
1.3.4 Infinite histories and temporal logic 14
1.4 Model Checking 15
1.4.1 The process of model checking 17
1.5 Thesis organization 17

2 Literature Review and Research Methodology 20
2.1 Introduction 20
2.2 Related work 20
2.3 Motivation 21
2.4 Methodology 21

3 Multi-step Transactions Accessing The Same Set Of Data Items 23
3.1 Introduction 23
3.2 A model of concurrent multi-step transactions 24
3.2.1 Steps and histories 24
CONTENTS

3.2.2 Serializability ........................................... 25
3.3 A condition for serializability of multi-step transactions .... 26
3.4 Specification of serializability in CTL .................... 33
  3.4.1 Syntax ................................................. 33
  3.4.2 Semantics of CTL ................................... 34
  3.4.3 Specification of the multi-step transactions model .... 35
  3.4.4 Specification of serializability ...................... 38
  3.4.5 The complexity gain of checking two transactions .... 39
3.5 Applications ................................................. 40

4 Multi-step Transactions Accessing Ordered Sets Of Data Items 42
  4.1 Introduction ............................................. 42
  4.2 Concurrent multi-step transactions model ................ 42
    4.2.1 Histories .......................................... 44
  4.3 Cycle reduction in conflict graphs ....................... 45
  4.4 The serializability condition ........................... 53
  4.5 Applications ............................................. 56

5 A Timestamp-based Protocol For Multi-step Transactions Accessing Ordered Sets Of Data Items 57
  5.1 Introduction .............................................. 57
  5.2 A timestamp-based protocol .............................. 57
    5.2.1 Accessing rules ................................... 58
  5.3 Linear temporal logic specifications .................... 61
    5.3.1 Syntax of LTL ...................................... 61
    5.3.2 Semantics of LTL .................................. 61
  5.4 Properties of read and write propositions ............... 63
  5.5 Encoding the accessing protocol and serializability condition into LTL ...................................... 65
  5.6 NuSMV model .............................................. 71
    5.6.1 Modelling .......................................... 72
    5.6.1.1 The Model Variables ............................ 73
    5.6.2 Specifications ..................................... 74
    5.6.3 Verification ....................................... 80
  5.7 Encoding into CTL ....................................... 82
    5.7.1 The Model Variables ................................ 83
    5.7.2 Specifications ..................................... 84
    5.7.3 Verification ....................................... 87
  5.8 Comparison of LTL and CTL ............................... 87
CONTENTS

6 Specification and Verification of Mobile Multi-step Transactions with Priority 97
   6.1 Introduction .................................................. 97
   6.2 Mobile Transactions .......................................... 97
   6.3 Application .................................................... 98
   6.4 The modified protocol ...................................... 101
   6.5 Transactions model .......................................... 103
       6.5.1 Properties of read and write propositions .......... 103
       6.5.2 The correspondence between paths and histories .... 104
   6.6 Encoding into LTL .......................................... 106
   6.7 The corresponding NuSMV model .............................. 108
       6.7.1 The model variables .................................. 108
       6.7.2 Specifications ......................................... 109
   6.8 Verification of the transactions model .................... 113
   6.9 Verification of starvation freedom ....................... 115

7 Conclusion .................................................................... 122
   7.1 Future work ..................................................... 125

References ........................................................................ 127

A NuSMV model for multi-step transactions with LTL specification 134

B NuSMV model for multi-step transactions with CTL specification 145

C NuSMV model for iterated transactions with LTL specification 154

D NuSMV model for iterated transactions with CTL specifications 167

E NuSMV model for modified protocol ................................. 173
List of Figures

1.1 Concurrent execution of two transactions. .................. 5
1.2 conflict graph ............................................. 8
1.3 Model checking. ............................................. 16

3.1 Conflict Graph for \( h_T \) .................................. 30
3.2 Conflict Graph for \( h_T' \) .................................. 33

4.1 The conflict graph \( G(h) \) for Lemma 4.3. .................. 46
4.2 The conflict graph \( G(h) \) which contains \( T_i, T_j, T_k \) and \( T_l \). .................. 46
4.3 We can either reduce the cycle or make a cycle of length two. ......... 47
4.4 We have a cycle of length two if either \( T_j \prec^h T_l \) or \( T_l \prec^h T_j \) ......... 47
4.5 No cycle in \( G(h) \) ............................................ 49
4.6 Represents Lemma 4.6 ........................................ 51
4.7 Cycle in \( G(h) \) of length \( n \) .................................. 52
4.8 \( G(h) \), as in Figure 4.7, after applying Lemma 4.5 .................. 53
4.9 \( G(h_T) \) is a subgraph of \( G(h) \). .......................... 55
4.10 Ordered set .................................................. 56

5.1 Cycle of length two ........................................... 68
5.2 Transaction executes its operations sequentially. .................. 75
5.3 No two transactions are reading the same data item simultaneously. ....... 75
5.4 Output generated by NuSMV for \( \sigma_1 \) ......................... 81
5.5 (a)CTL notion of time over the states; (b) LTL notion of time over the states .................. 83
5.6 Time and space for the model of LTL specifications .................. 89
5.7 Time and space for the model of CTL specifications .................. 90
5.8 Transaction can iterate infinitely many times ......................... 91

6.1 Mobile environment global architecture ....................... 98
6.2 Mobile and local transactions in a registration system .................. 99
6.3 ‘Infinitely many times’ means a loop .......................... 108
6.4 Set of states iterating infinitely often ......................... 108
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.5</td>
<td>$\sigma_{sc_1}$ and $\sigma_{sc_2}$ hold, if both transactions are of the same type.</td>
<td>115</td>
</tr>
<tr>
<td>6.6</td>
<td>$\sigma_{sc_1}$ does not hold.</td>
<td>119</td>
</tr>
<tr>
<td>6.7</td>
<td>$\sigma_{sc_2}$ does not hold.</td>
<td>120</td>
</tr>
<tr>
<td>6.8</td>
<td>$\sigma_s$ does not hold.</td>
<td>121</td>
</tr>
</tbody>
</table>
List of Tables

1.1 Temporal logic operators ........................................ 12
Chapter 1

Introduction

Recent advances in the development of portable devices and wireless communication networks have lead to the emergence of mobile computing. In mobile computing environments, users have the opportunity to access information and services regardless of their physical location or movement behaviour. This means that a large community of concurrent users can submit their transactions to the database to be executed [51]. Examples of these applications are mobile auctions, stock trading and electronic commerce applications. In stock trading, submitting buy or sell transactions on the Internet has existed for some time. In electronic commerce applications, customers carrying portable devices (cell phones, laptops, PDAs) may purchase flight tickets from any airlines with their credit cards. These applications involve dealing with a huge numbers of transactions accessing databases whose consistency must be preserved in spite of updates. The component in the database system responsible for scheduling the operations of concurrent transactions to achieve consistency is the concurrency control system (or the scheduler). The scheduler orders operations belonging to different transactions by means of a concurrency control protocol [22].

The number of transactions, in most concurrency control protocols of traditional database systems, that are allowed to be executed concurrently is finite or bounded. However, in recent database systems, especially in mobile environments, the number of possible concurrent transactions is unbounded. Therefore, the schedules produced are infinite. In order to prove correctness of such systems, a verification technique that works with this view of infinite schedules is needed. Most verification techniques of such concurrency control protocols are based on finite state representations of system behaviours. These techniques cannot be directly applied to those systems of concurrent transactions where behaviours may refer to past steps of the ongoing computation or where the number of concurrent transactions is unbounded. In such cases, even simple transactions can generate infinite state systems [50].
One of the most famous examples that refers to the importance of specifying and verifying the correctness of the protocols that are used in environments where the number of users increase beyond known bounds is the Skype\(^1\) services outage [66, 67]. This problem was caused by a massive restart of users' computers across the globe within a very short time as they rebooted after receiving a routine set of patches through Windows update [67]. A huge number of users came to the system in a continuous stream to access and request the services in terms of login transactions. The flood of attempted Skype logins together with a lack of Skype network resources, at that time, led to an outage in services. However, this event revealed a previously unseen software bug within the network resource allocation algorithm which prevented the self-healing function from working quickly. Regrettably, as a result of this disruption, Skype was unavailable to the majority of its users for approximately two days and prevented millions of registered users from accessing and making internet telephone calls using Skype software [66]. Skype lost a lot of its business and reputation during 48 hours of services outage. This demonstrates the limitation of some of the current protocols to deal with huge numbers of users transactions, and shows the need for specifying and verifying the correctness of the protocols that are used in such environments.

In general, mobile users submit transactions to servers that contain databases and participate in the mobile environment for execution. Since the architecture of a mobile computing system is distributed in nature [52], transactions are decomposed into a set of subtransactions, each of which executes in a different database participating in the mobile environment. Each database in a mobile environment contains a finite set of data items and therefore, the number of different transactions is also finite. However, the database system continuously reacts with other components in the environment in terms of transactions, and so the schedule will be infinite. For example, consider a number of field engineers working in a communication company in the same geographical area who need to share some data in a cooperative way and sometimes refer to the main database server of the company. Anyone of them may submit a transaction that may be required to access data items stored in different databases of their colleagues and the main database of the company, all to be executed in a continuous stream.

This chapter is organized as follows. In Section 1.1, we shall discuss the concept of concurrent database transactions in general and the need for concurrency control. The formal definition of transactions and the characteristics of a correct schedule will be discussed in Subsections 1.1.2 and 1.1.3 respectively.

\(^1\)Skype is a VoIP (Voice over Internet Protocols) telephony company that enables its users to make free voice calls with other Skype users and also low-cost calls to landlines and mobiles around the world. Recently, the number of Skype users exceeded 170 millions, about 10 millions of which are online at the same time [67]. For more information see [65].
Chapter 1. Introduction

The basic technique used for testing the correctness of schedules is introduced in Subsection 1.1.4. Multi-step transactions and infinite histories are defined in Subsection 1.1.5. In Section 1.2, some existing verification techniques for concurrency control protocols in mobile systems are discussed. The attraction of using temporal logic to reason about concurrent and reactive systems and their properties in general, and in specifying and verifying infinite histories in particular, is discussed in Section 1.3. In Section 1.4, the technique of model checking with temporal logic, as well as the process of model checking in general, is introduced.

1.1 Basic concept of database transactions

Transaction processing systems are systems with large databases and hundreds or maybe thousands of concurrent users that are executing database transactions such as reservations, banking, stock markets, supermarkets and other similar systems [22]. A database transaction can be defined as an operation or series of operations, carried out by users or applications, which access or changes the contents of a database. The transaction operations which can be performed on the databases are categorized as follows:

- Access operation (or read operation): returns the value stored in a data item \(x\). We shall denote it by \(\text{Read}(x)\).

- Update operation (or write operation): updates the value of a data item \(x\) to a new value and we denote it by \(\text{Write}(x)\).

The definition of a database is a collection of related data which represent a part of the real-world [22, 24]. But, in the real-world, we have restrictions that make it in a legal state. For example, in human resources databases no employee age can be negative, and in a travel agency database no bus or aircraft can have more passengers than the capacity of the bus or aircraft. These restrictions called integrity constraints of a database provide a framework for ensuring that any change made to the database by users or applications does not result in a loss of data consistency. A consistent state of a database is a collection of all stored values of the data items at a particular time such that these data values achieve the integrity constraints of the database. A transaction should transform the database from one consistent state to another consistent state. To make sure that the database will be maintained in a consistent state, any transaction executing in the database should have the following properties (called ACID properties):

- Atomicity: All operations of a transaction must either succeed or fail.
• Consistency: A transaction must either (a) leave the system in a consistent state or (b) abort. If a transaction cannot achieve a consistent state, it must return to its initial state.

• Isolation: The behaviour of a transaction is not affected by other transactions being executed simultaneously. A transaction must serialize all access to shared resources and guarantee that concurrent programs do not corrupt each other's operations.

• Durability: The effects of a committed transaction are permanent. Even if the system fails, the changes resulting from a transaction are permanent and durable.

1.1.1 The need for concurrency control

In general, concurrency control is the activity of coordinating the actions of processes that execute in parallel way, access shared data, and therefore possibly interfere with each other. Concurrency control, in the context of transaction processing systems, aims to ensure that the database will preserve the ACID properties (or maybe some of them) on the transactions that concurrently execute in the system [27]. Many concurrency control and recovery protocols have been developed to achieve this purpose [28]. Now, the question is, why might the execution of concurrent transactions breach the ACID properties? The answer to this question is explained in the following example:

Suppose that we have a data item \( x \) in a database with initial value 5 \( (x = 5) \), two transactions such that

1. Transaction1: \( \text{Read}(x); \quad x = x - 3; \quad \text{Write}(x); \)

2. Transaction2: \( \text{Read}(x); \quad x = x + 7; \quad \text{Write}(x); \)

and the concurrent execution of the two transactions is as in Figure 1.1. In Figure 1.1, the final value of the data item \( x \) equals 12. But, assume that \( x \) represents the number of available seats in a bus. This means that the first transaction should cancel the reservations for three passengers \( (x = x - 3) \) and the second transaction should reserve seven seats. Therefore, the final value of \( x \) should be 9 \( (x = 9) \). But, the interleaving operations, which are shown in Figure 1.1, produce \( x = 12 \). This means that the final value of the concurrent transactions is incorrect. The reason is that the second transaction (Transaction2) reads the value of \( x \) before the first transaction (Transaction1) changes it in the database. Hence, the updated value resulting from the first transaction is lost. This problem is called the lost update problem.
Chapter 1. Introduction

Figure 1.1: Concurrent execution of two transactions.

<table>
<thead>
<tr>
<th>Transaction1</th>
<th>Transaction2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td></td>
</tr>
<tr>
<td>Read(x)</td>
<td>Read(x)</td>
</tr>
<tr>
<td>$x = x - 3$</td>
<td>$x = x + 7$</td>
</tr>
<tr>
<td>Write(x)</td>
<td>Write(x)</td>
</tr>
</tbody>
</table>

More problems could occur due to the concurrent execution of the transactions such as the unrepeatable read and the temporary update. The unrepeatable read may occur when the first transaction reads several values and the second transaction updates some of them during execution of the first. This means that a transaction can read different values for the same data item at different times. The temporary update (or dirty read) problem may occur when one transaction can access (or read) intermediate results of another transaction before it has completed [22, 23]. In all of these problems, which may occur from the concurrent execution of transactions, we notice that some of the ACID properties are breached. For example, in the lost update problem, we violate the consistency and isolation properties, because the final result of the execution of the transactions leaves the database in an inconsistent state. Also, we notice from the previous example in Figure 1.1, that the transaction Transaction1 is affected by transaction Transaction2. This means that we violate the isolation property. The preservation of the atomicity and durability properties is the responsibility of a subsystem in the DBMS called the transaction recovery subsystem [22]. The consistency property is the responsibility of the programmer who writes the transaction, or the DBMS that checks the integrity constraints. In this thesis, we focus on specifying and verifying the isolation property which is the responsibility of the concurrency control system. The isolation property asserts that the behaviour of a transaction should not be affected by other transactions being executed simultaneously (or concurrently) in the system. A serial execution of the transactions (without any interleaving among the transactions) will maintain the isolation property. The execution order of a sequence of database operations generated by a set of transactions is called a schedule or history. In the next Subsection 1.1.3, we shall introduce some characteristics of schedules that are used to preserve the isolation property. First of all, we give a formal definition of a transaction.
1.1.2 Formal transaction definition of a transaction

A transaction is a collection of one or more operations on one or more databases, which reflects a single real-world transition. Formally (as in [27]) a transaction $T_i$ is a partially ordered set, with ordering relation $\ll_i$, where

1. $T_i \subseteq \{r_i(x), w_i(x) \mid x$ is a data item} $\cup \{a_i, c_i\}$, where $a_i$ means abort, $c_i$ means commit, $r_i(x)$ means read operation and $w_i(x)$ means write operation on data item $x$

2. $a_i \in T_i$ iff $c_i \notin T_i$

3. if $b$ is $c_i$ or $a_i$ (whichever is in $T_i$), for any other operation $s \in T_i, s \ll_i b$

4. if $r_i(x), w_i(x) \in T_i$; then either $r_i(x) \ll_i w_i(x)$ or $w_i(x) \ll_i r_i(x)$.

Condition (1) defines the operations in the transaction. Condition (2) says if a commit operation occurs (meaning that the transaction completes execution), then an abort does not occurs and vise versa. Condition (3) says that the commit or abort (whichever is present) must follow all other operations. In condition (4) the relation $\ll_i$ specifies the order of execution of read and write operations.

1.1.3 Characteristics of schedules based on serializability

A serial schedule is a schedule $S$ if, for every transaction $T_i$ participating in the schedule, all the operations of $T_i$ are executed consecutively in the schedule. Otherwise, the schedule is called a nonserial schedule. A serializable schedule is a schedule $S$, if it is equivalent to some serial schedule of the same transactions. This means that being serializable implies that the schedule is a correct schedule (leaves the database in a consistent state) [24]. The equivalence of schedules can be divided into two definitions:

1. Result equivalent: Two schedules are called result equivalent if they produce the same final state of the database.

2. Conflict equivalent: Two schedules are said to be conflict equivalent if the order of any two conflicting operations is the same in both schedules.

Two operations are conflicting if:

- they are by different transactions,
- they are on the same object or data item,
- and at least one of them is a write.
To make use of the conflict equivalent definition, a conflict serializable can be defined as a schedule $S$ that is conflict equivalent to some serial schedule $S'$. In the literature, various notions of serializability are introduced such as view serializability and strict serializability [24]. These notions are either NP-complete problem for deciding whether a given schedule is serializable, such is the case for the problem of view serializability, or do not affect the correctness of the final state of the database, as in strict serializability [24, 29]. Conflict serializability represents a good correctness basis for concurrency control protocols with acceptable performance. In other words, conflict serializability is a strong notion of correctness, but it has an efficient algorithm (or technique) for testing whether the schedule is correct in this sense [15, 24]. In the next subsection, we shall introduce a technique called precedence graph (or conflict graph) used to validate or test the conflict serializability of a schedule.

1.1.4 Conflict graph

Conflict graphs are widely used for testing conflict serializability in a polynomial time [24, 22, 27]. Given a schedule $S$, the conflict graph is a directed graph $G = (V, A)$ where $V$ is set of nodes and $A$ is set of directed edges created as follows:

1. Create a node for each transaction.

2. Create a directed edge $T_i \rightarrow T_j$, if $T_j$ reads the value of an item written by $T_i$.

3. Create a directed edge $T_i \rightarrow T_j$ if $T_j$ writes a value into an item after it has been read by $T_i$.

4. Create a directed edge $T_i \rightarrow T_j$ if $T_j$ writes a value into an item after it has been written by $T_i$.

According to this, if an edge $T_i \rightarrow T_j$ exists in the conflict graph for $S$ then, in any serial schedule $S'$ conflict equivalent to $S$, $T_i$ must appear before $T_j$. If the conflict graph contains a cycle, the schedule is not conflict serializable. For example, suppose we have two transactions such that

$$T_1 = r_1(x)w_1(x)r_1(y)w_1(y)$$
$$T_2 = r_2(x)w_2(x)r_2(y)w_2(y)r_2(z)w_2(z)$$
and they are interleaved in three histories as follows:

\begin{align*}
    h_1 &= r_1(x)r_2(x)w_1(x)w_2(x)r_1(y)w_1(y)r_2(y)w_2(y)r_2(z)w_2(z) \\
    h_2 &= r_1(x)w_1(x)r_2(x)w_2(x)r_2(y)w_2(y)r_1(y)w_1(y)r_2(z)w_2(z) \\
    h_3 &= r_1(x)w_1(x)r_2(x)w_2(x)r_1(y)w_1(y)r_2(y)w_2(y)r_2(z)w_2(z).
\end{align*}

The corresponding conflict graphs for the histories \( h_1 \), \( h_2 \) and \( h_3 \) are in the Figures 1.2(a), 1.2(b) and 1.2(c) respectively. Therefore, the histories \( h_1 \) and \( h_2 \) are not conflict serializable histories because the corresponding conflict graphs (Figure 1.2(a) and Figure 1.2(b) respectively) are cyclic. On the other hand, the history \( h_3 \) is conflict serializable. The reason is that the corresponding conflict graph (Figure 1.2(c)) is acyclic.

\begin{figure}[h]
\centering
\begin{subfigure}{0.3\textwidth}
\centering
\begin{tikzpicture}
    \node (t1) at (0,0) [circle,draw] {\( T_1 \)};
    \node (t2) at (1,0) [circle,draw] {\( T_2 \)};
    \draw[->] (t1) -- node[above] {\( x,y \)} (t2);
    \draw[->] (t2) -- node[below] {\( x \)} (t1);
\end{tikzpicture}
\caption{conflict graph for \( h_1 \)}
\end{subfigure}
\hspace{1cm}
\begin{subfigure}{0.3\textwidth}
\centering
\begin{tikzpicture}
    \node (t1) at (0,0) [circle,draw] {\( T_1 \)};
    \node (t2) at (1,0) [circle,draw] {\( T_2 \)};
    \draw[->] (t1) -- node[above] {\( x \)} (t2);
    \draw[->] (t2) -- node[below] {\( y \)} (t1);
\end{tikzpicture}
\caption{conflict graph for \( h_2 \)}
\end{subfigure}
\hspace{1cm}
\begin{subfigure}{0.3\textwidth}
\centering
\begin{tikzpicture}
    \node (t1) at (0,0) [circle,draw] {\( T_1 \)};
    \node (t2) at (1,0) [circle,draw] {\( T_2 \)};
    \draw[->] (t1) -- node[above] {\( x,y \)} (t2);
\end{tikzpicture}
\caption{conflict graph for \( h_3 \)}
\end{subfigure}
\caption{conflict graph}
\end{figure}

\subsection{Multi-step transactions and infinite histories}

Dividing the transactions into sets of steps to produce ‘multi-step transactions’ improves system throughput and allows transactions interleaving to gain more parallelism [21]. Examples of multi-step transactions are when users enter data using a sequence of forms. At the end of the sequence, the application does something with the input data. Desktop applications use wizards to simplify operations. In an e-commerce site, the checkout flow can be seen as a multi-step transaction. Booking e-ticket from travel agencies can also be seen as a multi-step transaction. Students registration systems which are multi-step transactions, will be seen in Chapter 6. Formally, a database system consists of a set \( D \) of data items and a set \( T = \{T_1, T_2, \ldots, T_n\} \) of transactions. A multi-step transaction \( T_i \) is a sequence (totally ordered set) of read and write steps , as in [25], where every read step \( r_i(x) \) precedes a write step \( w_i(x) \) such that

\[ T_i = r_i(x_1)w_i(x_1) \ldots r_i(x_m)w_i(x_m) \]
Chapter 1. Introduction

An infinite history $h$, in this thesis, is a history where a finite set of transactions iterate (or repeat) infinitely many times. The reason for this is as follows. In most real-world applications, the set of data items is a finite set. Hence, the number of different transactions we can create to access these data items is also finite. Thus, the infinitely many transactions that occur over time, are actually a finite number of different transactions iterating infinitely often and thereby producing infinite histories. Originally, this definition of transactions that iterate infinitely many times was introduced in the context of operating systems [30]. Infinite histories in the context of concurrent database transactions have been considered in [15, 14, 34, 35]. Most of these concentrate on specifying infinite histories using different specification languages based on temporal logics.

1.2 Verification of concurrency control protocols in mobile systems

Most of the existing approaches that are used to specify and verify the correctness of concurrency control protocols in mobile environments are based on mathematical proofs. For example, the work in [53] defines a protocol that uses a combination of the traditional protocols for mobile environments such as 2-Phase-Commit, 3-Phase-Commit, and consensus and proves the correctness of the proposed protocol using mathematical proof rules. In [51], a new protocol for processing both read-only and update mobile transactions is defined. The correctness proof of the proposed protocol is made using the mathematics of graph theory. Also, the work in [54], uses mathematical rules and graph theory to proof the correctness of the protocols. The problem of these approaches is that the use of mathematical proofs has numerous disadvantages, such as they require considerable expertise by the person who is to carry out the proof manually [14] and human error is common in such manual proofs. There are some of the existing approaches using automatic proving techniques as in [55] which are based on logic programming. The work in [56] introduces a model checking verification approach to reason about the behaviour of mobile systems using specifications written in the $\pi$-calculus and modal logic. The verification is based on a system (called HAL) which is able to interface with several model checkers to determine whether or not certain properties hold for a given specification. Fully automated techniques avoid numerous disadvantages of manual proofs.
1.3 Temporal Logic

Temporal logics are logics where truth values of formulae may change over time. The formulae of temporal logic can express facts about past, present, and future states. The term temporal logic has been broadly used to cover all approaches to the representation of temporal information within a logical framework. The significance of temporal logic in computer science is indisputable, especially in verification of reactive and concurrent systems [37]. Such systems are increasingly used in real-world applications where failure is considered as fatal, such as electronic commerce, high-speed communication networks, traffic control systems, and automated manufacturing. Model checkers of many types of temporal logics have been developed to the extent that they can quickly verify real-world systems with a huge number of states. Temporal logic has been proposed as applying both to the specification and verification of program behaviour, and to the specification of system behaviour. Temporal logics are useful for specifying concurrent systems in describing ordering of events in time without explicitly introducing time. As reactive and concurrent systems become more complicated, components may continuously interact with each other and their environment without terminating. It is essential to specify and verify some of the important properties of such systems [32, 31]. In the next subsection, we shall discuss some of these properties. In Subsection 1.3.2, we shall informally explain the temporal operators that are used in most types of temporal logics. Subsection 1.3.3 discusses how we can express the typical properties of reactive and concurrent systems using temporal logic. The specification of infinite histories using temporal logics is discussed in Subsection 1.3.4.

1.3.1 Temporal properties

The typical properties of reactive and concurrent systems that need to be specified and verified are safety properties and liveness properties, as they have been categorized in [32, 38, 36] as follows

- Safety:
  A safety property asserts that nothing bad ever happens during the execution of the system. Examples of safety property are mutual exclusion, freedom from deadlocks and partial correctness. We can observe the violation of the safety property in a finite behaviour of the system because this violation will happen at a particular point in time.

- Liveness:
  A liveness property asserts that something good will happen eventually. We
need the liveness properties because the safety properties prohibit undesirable things from happening but cannot make sure that something good must happen sometime in the future. Liveness properties are relevant to the infinite behaviours of the systems because the eventuality asserts something should happen later in undetermined time. Guaranteed accessibility, total correctness and responsiveness are examples of liveness property.

Also, another property can be introduced in the context of concurrent systems called the fairness property [57, 41] as follows:

- Fairness:
  
  A fairness property asserts that if something is requested, then it will be granted. In other words, a fairness property includes an assumption on the progress of individual processes in a system. Three definitions of fairness can be introduced: unconditional fairness, strong fairness and weak fairness. **Strong fairness** asserts that every process enabled infinitely often is executed infinitely often. **Weak fairness** asserts that every process enabled almost everywhere is executed infinitely often. **Unconditional fairness** asserts that every process is executed infinitely often. For example, assume that we want a scheduler that never ignores any process forever. This means that if such a scheduler is present, the system will guarantee the progress. The important thing to note is that unconditional, weak and strong fairness describe liveness properties [57].

### 1.3.2 Temporal operators

Temporal logic contains two kinds of operators (1) ordinary logical operators ($\land, \lor, \neg, \Rightarrow, \Leftarrow, \ldots$). (2) temporal operators which are used in most types of temporal logic such as LTL (Linear-Time Temporal Logic) and CTL (Computational Tree Logic). Temporal logic considers the necessity concept as well as possibility i.e. if $\beta$ is a formula, then $\Diamond \beta$ is a temporal logic formula that asserts that $\beta$ is possibly true, and $\Box \beta$ is a temporal logic formula that asserts that $\beta$ is necessarily true. The following table summarizes some of the temporal operators semantics:
Table 1.1: Temporal logic operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Informally</th>
</tr>
</thead>
<tbody>
<tr>
<td>∆α or Fα</td>
<td>α will be true at some time in the future.</td>
</tr>
<tr>
<td>□α or Gα</td>
<td>α will always be true in the future.</td>
</tr>
<tr>
<td>α U β or α Until β</td>
<td>α will always be true until β becomes true.</td>
</tr>
<tr>
<td>○α or Xα</td>
<td>α will be true &quot;next&quot;</td>
</tr>
<tr>
<td>Eα</td>
<td>Exists: there exists at least one path starting from the current state where α holds.</td>
</tr>
<tr>
<td>Aα</td>
<td>All: α has to hold on all paths starting from the current state.</td>
</tr>
</tbody>
</table>

1.3.3 Expressing system properties in temporal logic

In this subsection, we shall express some of the system properties, that are mentioned in Subsection 1.3.1, using temporal logic as in [41]:

Safety properties:

1. Mutual Exclusion:
   No two processes use the same resource at the same time. For example, assume that we have two processes α and β that run asynchronously, i.e., either Process α or β makes a step but not both, and the order of execution is undetermined. We write the mutual exclusion formally as in the following

   $$□¬((α = R) ∧ (β = R))$$

   where α = R means that the process α uses the resource R.

2. Freedom from Deadlocks:
   At any time, at least one process is enabled to progress. This can be expressed as follows

   $$□(enabled_1 ∨ ··· ∨ enabled_k)$$

   where enabled_i is true if process i has an action that can be executed (for 1 ≤ i ≤ k).

3. Partial Correctness:
   If α is satisfied when the program starts, then β will be satisfied if the program reaches a distinguished state γ.

   $$α → □(γ → β)$$
Liveness properties:

1. Guaranteed Accessibility:
   Once a process is in a current state, it will eventually go to the next state. For example, the computations that execute both Process $\alpha$ and Process $\beta$ infinitely often will hold:
   \[
   \square((\alpha = i) \implies \Diamond(\alpha = (i + 1))) \land \square((\beta = i) \implies \Diamond(\beta = (i + 1)))
   \]
   where the processes $\alpha$ and $\beta$ can be in state 1 (i.e. $\alpha = i$) and then in the state $i + 1$.

2. Responsiveness:
   If a request is issued, it will eventually be granted. This property can be formally represented in temporal logic as follows
   \[
   \square(\alpha \implies \Diamond \beta)
   \]
   where $\alpha$ is request and $\beta$ means granted.

3. Total Correctness:
   If $\alpha$ is satisfied when the program starts, then the program terminates in a distinguished state $\gamma$ where $\beta$ is satisfied.
   \[
   \alpha \implies \Diamond (\gamma \land \beta)
   \]

Fairness properties:

1. Weak Fairness:
   Every process enabled almost everywhere is executed infinitely often such that:
   \[
   \bigwedge_{1 \leq i \leq k} (\Diamond \square enabled_i \implies \square \Diamond executed_i)
   \]
   This means a constantly enabled event must occur infinitely often.

2. Strong Fairness:
   Every process enabled infinitely often is executed infinitely often:
   \[
   \bigwedge_{1 \leq i \leq k} (\square \Diamond enabled_i \implies \square \Diamond executed_i)
   \]
   This means that the event that becomes enabled infinitely often (and may become disabled) must occur infinitely often.
3. Unconditional Fairness:
   Every process is executed infinitely often such that:
   \[ \bigwedge_{1 \leq i \leq k} (\diamond \square \text{enabled}_i) \]
   This means that the processes can be executed at any time.

### 1.3.4 Infinite histories and temporal logic

As we have discussed above, temporal logic has a power to deal with reasoning about concurrent programs, therefore it seems promising for specifying and verifying infinite histories that are produced from the infinitely iterating transactions in a database system. The correctness criterion for the concurrent execution of the transactions is that of serializability [24]. Therefore, the basic problem is to determine whether such infinite histories, generated by some scheduler, are serializable. Serializability of database transactions was first defined in linear temporal logic in [33]. Partial Order Temporal Logic has also used to specify and verify serializability in [42] and QPTL (Quantified Propositional Temporal Logic) in [35]. Now, it is useful to conduct proofs of such infinite histories using fully automated techniques to avoid the numerous disadvantages of manual proofs. This requires using temporal logics for which there are available model checkers. The attraction of the temporal logics CTL and LTL is the availability of powerful industrial-strength model checkers such as NuSMV [10] and SPIN [43]. Model checkers carry out exhaustive checks on the correctness of an execution against the specification of a correctness criterion, and are fully automatic and therefore require no special expertise to carry out the verification. In Section 1.4, we shall discuss the model checking technique in general.

The idea of verifying a scheduler producing infinite histories, is to specify the scheduler in a temporal logic (such as CTL or LTL) by a formula \( \varphi_p \) whose models correspond to the generated histories, and the serializability condition in a formula \( \varphi_{\text{sercon}} \). The scheduler will be correct (always generates serializable histories), if the following formula holds

\[ \varphi_p \implies \varphi_{\text{sercon}}. \]

In [44, 45, 40], serializability is considered to be a safety property. Other properties of infinite histories, that can be specified and verified using temporal logic, include ‘starvation’. Simply, starvation can be defined as the case when the system as a whole makes progress, but some individual transaction does not for an indefinite period of time [22]. Starvation is considered to be a liveness property in [46, 48]. Starvation and deadlock are the most common problems that
may occur to the schedulers of database transactions [22]. Deadlock-freedom, as in Subsection 1.3.1, is a safety property and can be specified in temporal logic. Moreover, if we require that transactions iterate infinitely many times, we can specify this as a fairness property.

The importance of temporal logic to computer science is seen in the fact that the person who introduced temporal logic to the computing community (Amir Pnueli) won the ACM A. M. Turing Award in 1996 [58], for outstanding contributions to program and systems verification.

1.4 Model Checking

Model checking is an automated technique or process that, given a finite-state model of a system and a formal properties (specifications) written in a logic, systematically checks whether these properties hold for (a given state in) that model [9]. In model checking, the system to be verified is represented by a model $M$. The model $M$ consists of all possible system executions (behaviours) in a mathematical structure like a finite state transition system. The system property to be satisfied is represented by formula $\phi$ in a logic that is interpreted over such structures (e.g. temporal logic). The system correctness is shown by checking that the formula is satisfied by the model, i.e. $M \models \phi$ [49]. This check is performed by exhaustively exploring the state space of the model to ensure that all possible system behaviours satisfy the property or not [47].

Model checking has two main advantages. Firstly, it is fully automatic, and so requires no special expertise in logic and theorem proving. Secondly, if the system contains an error (the system does not satisfy the property), model checking will produce a counterexample that can be used to pinpoint the source of that error; see Figure 1.3. The counterexample is a trace of system executions (behaviours) that violate the property that is being checked. The main disadvantage of model checking is having to deal with the state space explosion problem. This problem occurs in systems where many components can interact with each other, or systems with data structures containing many different values. In such cases the number of states can be huge. The first approach that was developed to solve the state explosion problem was by McMillan. McMillan [9] realized that by using a symbolic representation for the transition graphs, much larger systems could be verified. The new symbolic representation was based on ordered binary decision
diagrams (OBDD^2). McMillan developed a model checking system called SMV, which extracts a transition system represented as a OBDD from a program and uses a OBDD-based search algorithm to determine whether the system satisfies its specification. Since then, it became possible to verify systems that have more than 10^20 states [59].

Model checking requires a precise and unambiguous statement of the properties to be examined; this is typically done in temporal logic [9]. Model checking has a number of advantages over traditional approaches that are based on simulation and testing. Simulation and testing are involved in making experiments to the system before applying it in the field. Even though the techniques that are based on simulation or testing are inexpensive and widely used, they usually suffer from extremely low coverage and cannot provide a full guarantee of correctness [60]. While simulation is performed on an abstraction or a model of the system, testing is performed on the actual product [61]. Model checking is not only considered to be automatic and usually quite fast [9] but also can guarantee the correctness of the system against the desired properties that are specified [62]. The ACM A. M. Turing Award in 2007 was granted to Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis for their efforts in developing model checking into a highly effective verification technology, widely adopted in the hardware and software industries [58].

^2These are data structures used to represent boolean functions. OBDDs provide a canonical form, like NNF (Negation Normal Form), that is more compact than CNF (Conjunctive Normal Form) or DNF (Disjunctive Normal Form) and also they have very efficient algorithms to manipulate them.
1.4.1 The process of model checking

The main blocks in the process of model checking are:

- **Modelling:** This step includes converting a design into a form which will be accepted by the model checking tool using the description language of the model checker. The modelling of design may require the use of abstraction to ignore or eliminate the unimportant or irrelevant details.

- **Specification:** This step involves writing the properties that the design must satisfy. Usually, this is achieved by using temporal logic, which models the behaviour of the system over time.

- **Verification:** This step explores the state space of the state transition system to verify whether the specifications or system properties meet the design or not, and give a counterexample if the specification is found not to hold. Counterexamples are one of the most useful features of model checking because they allow users to quickly understand why the specification is not satisfied. Ideally the verification is completely automatic, but often in practice there is human assistance to analyze the verification results, in the case of negative results, and this process may requires modification to the system to correct it. An error trace can also result from incorrect modelling of the system or from an incorrect specification. The error trace can also be useful in identifying and fixing these two problems.

In chapter 5 and 6, we shall show how we apply the main blocks of the process of model checking to a timestamp-based protocol that is defined in Section 5.2.

1.5 Thesis organization

The rest of this thesis is organized in the following way:

- **Chapter 2:**
  In Chapter 2, we present the related work and show how the research point emerged. Moreover, we present the research methodology used for the research in this thesis and highlight the motivation for our research.

- **Chapter 3:**
  In Chapter 3, we give a mathematical model of concurrent multi-step transactions. We prove that acyclicity of conflict graphs corresponds to serializability in the case of an infinite number of transactions. Also, we give and prove a simpler correspondence in the case where transactions access the
same set of data items. We define and prove conditions for serializability in a way that can be encoded into the logic $\text{CTL}$ which is introduced in this chapter. Applications for the kind of transactions considered in this chapter, are given in the context of web and mobile environments.

- **Chapter 4:**
  In chapter 4, we give and prove a serializability condition for multi-step transactions accessing contiguous data items from an ordered set of data items. As in Chapter 3, this serializability condition can be encoded efficiently into $\text{CTL}$ and $\text{LTL}$. Applications for the kind of transactions considered in this chapter, are given in the context of web and mobile environments.

Chapters 5 and 6, which are the main chapters of this thesis, describe and apply the model checking technique to infinite histories of multi-step transactions in mobile environments.

- **Chapter 5:**
  In chapter 5, we define a protocol based on timestamps that aims to ensure the serializability of multi-step transactions accessing an ordered set of data items. Also, we specify the serializability condition from chapter 4 and the timestamps-based protocol, using $\text{LTL}$ and $\text{CTL}$. The logic $\text{LTL}$ is introduced in this chapter. The specification and verification of the serializability condition and the protocol are carried out using the symbolic model checker $\text{NuSMV}$. We verify that the protocol serializes the multi-step transactions. We compare between $\text{CTL}$ and $\text{LTL}$ in terms of suitability for specifying and verifying timestamps-based protocols. This chapter is a stepping stone to modelling typical protocols found in mobile environments.

- **Chapter 6:**
  In chapter 6, we modify the protocol which is defined in chapter 5, to one relevant to mobile environments. We justify the modification of the protocol. We specify the serializability condition, that is defined in chapter 4, and the modified protocol using $\text{LTL}$. We verify that the protocol serializes the multi-step transactions and does not lead to starvation. Finally, we show how a further slight modification to the protocol can produce histories that are incorrect in terms of both serializability and starvation.

- **Chapter 7:**
  Chapter 7 summarizes the results and gives some conclusions.

**Origin of the chapters material.**
Some of the material contained in this thesis has been published in the form of
articles, and has been restructured and extended here. Initial work for Chapter 6, was published in [21]. Chapter 4 has been published in [68], and Chapter 3 has been published in [69]. The more extensive work of Chapters 5 and 6 is being prepared for a possible submission to an IEEE transactions.
Chapter 2

Literature Review and Research Methodology

2.1 Introduction

Standard database techniques model fixed finite schedules of transactions [64], [24]. Recently, with the emergence of new techniques such as web transactions and mobile databases, where an unlimited number of transactions may be incoming and outgoing to the databases in continuous streams, the importance of representing infinite histories has been recognised [35], [14], [34].

One way of representing infinite histories is as models of temporal logic formulae. A benefit of using temporal logic is the availability of powerful model checkers such as NuSMV [10] and SPIN [43]. Model checkers can carry out exhaustive checks of a correctness criterion such as serializability, and are fully automatic and therefore require no special expertise to carry out the verification.

2.2 Related work

The model of infinite schedules, where transactions repeat infinitely often, was originally considered and investigated in [30]. Also, the proposed application of such histories was in the scheduling problem of service processes in operating systems. The temporal logics that have been used to specify concurrent database transactions include the partial-order temporal logic ISTL in [42], quantified propositional temporal logic QPTL in [35], LTL in [14], a first-order temporal logic in the first part of [33] and a monodic fragment of first-order temporal logic in [34]. With the exception of LTL these are, at best, of exponential space complexity, and, at worst, undecidable. Most of these temporal logics (except LTL) do not have model checkers to carry out full automatic verification. Moreover, the drawback
of these techniques is that proving the basic serializability correctness condition is impractical, as encoding (the absence of) conflict cyclicity within large sets of transactions results in prohibitively large temporal logic formulae.

In [35] and [34] transactions iterate infinitely many times and each iteration of a transaction is called an occurrence, and every occurrence of a particular transaction comprises the same two (read and write) steps. In this thesis, we shall improve this to a case of multi-step transactions. Also, we shall show that, under certain common assumptions on the graph structure of data items accessed by multi-step transactions, conflict cyclicity need only be checked within all possible pairs of transactions. This results in formulae of considerably reduced size in any temporal-logic-based approach to proving serializability, and scales to arbitrary numbers of transactions. This makes testing for serializability efficient and easy to encode into the widely used temporal logics CTL and LTL.

2.3 Motivation

This thesis focuses on specifying and verifying infinite histories of multi-step transactions accessing a finite set of data items with different properties, using temporal logics and model checkers. Much work has been done in modelling mobile environments in order to determine performance. The aim here is to model mobile environments in order to determine correctness. In mobile computing environments infinite histories are produced, by the newer technologies of web and mobile transactions, in which transactions are continuously accessing the data items of the databases. The main desired properties that need to be specified and verified are serializability and starvation. We model protocols that produce infinite histories and use the NuSMV model checker based on the temporal logics CTL and LTL specifications to prove or disprove that the models satisfy these properties. We define a protocol, based on timestamps, to apply the technique to, and verify its correctness for serializability. We compare the suitability of LTL and CTL for specifying and verifying such protocols. We make a little change on the protocol and show the effect of this change with respect to the correctness of the protocol. Finally, we consider the specification and verification of (the absence of) starvation for this kind of problem.

2.4 Methodology

In this section, we discuss some important methodological assumptions which will be used throughout the thesis. The objective of this thesis is to specify the correctness of transactions executing concurrently on a database in terms of serializability
using specifications written either in CTL or LTL. The reason for using temporal logics such as CTL and LTL, 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 logic have been developed to the extent that they can deal with a huge number of states and verify real-world systems [21].

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 [9] and achieves polynomial-time model checking [8].

LTL is a temporal logic where the model of time is a path in which the future is determined. LTL is used for specifying general reactive and concurrent systems [4, 3]. It is worthwhile using CTL or LTL to specify multi-step transactions to gain full automatic verification by using model checkers.

We shall model the protocols which are used to ensure the serializability of concurrent transactions as finite state transition systems for which the specifications are expressed in either CTL or LTL. Then, by exploring the state space of the state transition system, it is possible to check automatically if the protocol satisfies its desired specifications or not. 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 [10]. We specify and verify schedules of multi-step transactions model as a finite state machine in the input language of NuSMV. These schedules are produced based on the behaviour of protocol over time. Also, the multi-step transactions model is characterized by a set of properties which ensures that the NuSMV model meets the defined model. The desired properties, that the system should satisfy, are expressed in either CTL or LTL. Finally, the model checker will produce true if specification of the desired property satisfies all possible system behaviours. Otherwise, a counterexample will be produced to show the source of the error.
Chapter 3

Multi-step Transactions Accessing The Same Set Of Data Items

3.1 Introduction

As concurrent users access and update databases in terms of transactions, a reliable condition of correctness is needed for the execution of these transactions. The established correctness condition is that of 'serializability', where an interleaved schedule of concurrent transactions is equivalent to a serial 'schedule' (or 'history') of the transactions. Most work on serializability has modelled histories to be finite with a known fixed bound [64], [24]. Recently, with the emergence of new techniques such as web transactions and mobile databases, where an unlimited number of transactions may be incoming and outgoing to the databases in continuous streams, the importance of representing infinite histories has been recognised [35], [14], [34].

One way of representing infinite histories is as models of temporal logic formulae. A benefit of using temporal logic is the availability of powerful model checkers such as NuSMV [10] and SPIN [43]. Model checkers can carry out exhaustive checks of a correctness criterion such as serializability, and are fully automatic and therefore require no special expertise to carry out the verification. The drawback with model checking is that even the most powerful model checkers cannot overcome the theoretical worst-case complexity of model checking inherent from the temporal logic being used. The most benign temporal logic in this respect is CTL which can check whether executions represented by a finite-state machine satisfy a specification with time complexity $O((|S| + |R|).|f|)$, where $|S|$ is the number of states in the finite state machine, $|R|$ the number of transitions, and $|f|$ is the length of the specification formula. This is marginally better than for LTL which has a corresponding time complexity of $O((|S| + |R|).2^{|f|})$ [9]. However, the tem-
poral logics that have been used to specify transactional concurrency include the partial-order temporal logic ISTL in [42], quantified propositional temporal logic QPTL in [35], LTL in [14], a first-order temporal logic in the first part of [33] and a monodic fragment of first-order temporal logic in [34]. With the exception of LTL these are, at best, of exponential space complexity, and, at worst, undecidable.

In this chapter, we give a computationally efficient specification of serializability in CTL. The serializability condition expressed in CTL is based on acyclicity of conflict graphs. To be able to use such a condition, we prove that acyclicity of conflict graphs corresponds to serializability for infinite schedules. We then assume the further property for our transactions, that they access the same set of data items in different orders. We show that serializability then corresponds to the efficient condition where only cycles of length two need be checked, and this condition is used for the CTL specification. This work advances that of [35] and [14], which both deal with two-step transactions, to the more normal case of multi-step transactions. We also produce the specification in the slightly more efficient CTL rather than LTL. This chapter is organized as follows. In Section 3.2, we give a mathematical model of concurrent multi-step transactions. In Section 3.3, the results on acyclicity of conflict graphs and serializability for infinite schedules are given. From these, serializability is characterized mathematically in a way to be encoded into CTL. The CTL specification is given in Section 3.4. Applications that satisfy this model are given in Section 3.5.

3.2 A model of concurrent multi-step transactions

3.2.1 Steps and histories

The model of concurrent two-step transactions in [35] comprises \( n \) transactions \( \{T_1, \ldots, T_n\} \) occurring infinitely many times, with each transaction containing a read step and a write step each accessing a finite number of data items. In this chapter, we define transactions as containing multiple alternate read and write steps, each accessing a single data item. We shall denote a read step and the corresponding write step on the data item \( x_j \) by transaction \( T_i \), as \( r_i(x_j) \) and \( w_i(x_j) \), respectively, and the set of data items accessed by all transactions as \( D \). We say that two steps are conflicting if they belong to different transactions, they access the same data item and at least one of them is a write step. Later in this chapter, we shall assume that, given transactions \( T_i \) and \( T_j \), the data items accessed by both are the same, but that the order of access of data items by
transaction $T_i$ is not necessarily the same as that by $T_i'$. Precisely, we will assume a finite set of data items $D = \{x_1, \ldots, x_m\}$, and an infinite set of (multi-step) transactions $T = \{T_i : i \in \mathbb{N}_1\}$, where $\mathbb{N}_1$ is the set of positive integers, such that all $T_i \in T$ are of the form,

$$T_i = r_{i_1}(x_{i_1})w_{i_1}(x_{i_1}) \ldots r_{i_m}(x_{i_m})w_{i_m}(x_{i_m})$$

where $\{x_{i_1}, \ldots, x_{i_m}\} = D$.

A schedule or history $h$ of $T$ is an interleaved sequence of all the read and write steps, of all the transactions in $T$, such that, for each $i \geq 1$, the subsequence of $h$ compromising the steps of $T_i$ is exactly the sequence of steps of $T_i$ occurring in the order that they do in $T_i$. For a history $h$, $<_h$ will denote the (irreflexive) total order between all the read and write steps of $h$. If $T' \subseteq T$, then the projection of $h$ to $T'$, denoted $h_{T'}$, is the history of $T'$, obtained from $h$, by deleting all steps of transactions not in $T'$.

### 3.2.2 Serializability

The required correctness condition of 'serializability' is that concurrent multi-step transactions should execute in a history whose effect is 'equivalent' to a serial execution of all the $T_i \in T$. Our definitions of equivalence and serializability are based on those in [24].

**Definition 3.1.** Histories $h_1$ and $h_2$ of $T = \{T_i : i \in \mathbb{N}_1\}$ are equivalent, written as $h_1 \sim h_2$, iff for all $i, i' \geq 1, i \neq i'$, and for all $x \in D$,

1. if $r_i(x) <_{h_1} w_{i'}(x)$, then $r_i(x) <_{h_2} w_{i'}(x)$,
2. if $w_i(x) <_{h_1} w_{i'}(x)$, then $w_i(x) <_{h_2} w_{i'}(x)$ and
3. if $w_i(x) <_{h_1} r_{i'}(x)$, then $w_i(x) <_{h_2} r_{i'}(x)$

**Definition 3.2.** A history $h$ of $T = \{T_i : i \in \mathbb{N}_1\}$ is serializable iff there is a serial history $h_S$ of $T$ of the form, for each $i \in \mathbb{N}_1$,

$$h_S = \ldots \frac{\ldots r_i(x) \ldots w_i(y) \ldots}{\text{only (all) steps of } T_i} \ldots$$

such that $h \sim h_S$. 


3.3 A condition for serializability of multi-step transactions

In [35], serializability of infinite histories is characterized in terms of 'detachable' steps for certain finite subsequences of steps. We shall determine serializability in terms of acyclicity of 'conflict graphs' - a technique widely used for finite histories [24]. We define conflict graphs in Definition 3.3 and in Theorem 3.4 give conditions for which acyclicity of conflict graphs correspond to serializability in the case of an infinite number of transactions. In Lemma 3.5, we give a simpler correspondence in the case where transactions access the same set of data items. This result is used to prove the main result, Theorem 3.7, which gives the conditions for serializability that will form the basis of the specification in CTL in Section 3.4.

Definition 3.3. A directed graph is a pair \( G = (V, A) \), where \( V \) is a set of elements called nodes, denoted \( \text{nodes}(G) \), and \( A \subseteq V \times V \) is a set of elements called arcs, denoted \( \text{arcs}(G) \). A walk in a directed graph \( G = (V, A) \) is a sequence of nodes \((v_1, v_2, \ldots, v_n)\) such that \((v_i, v_{i+1}) \in A\) for \( i = 1, \ldots, n - 1 \). A walk with no nodes repeated is called a path; it is a cycle when only the first and last node coincide. For each history \( h \), there is a directed graph \( G(h) \) called the precedence graph or conflict graph of \( h \). This graph has the transactions of \( h \) as its nodes, and contains an arc \((T_i, T_j)\), where \( T_i \) and \( T_j \) are distinct transactions of \( h \), whenever there is a step of \( T_i \) which conflicts with a subsequent (in \( h \)) step of \( T_j \).

Theorem 3.4. A history \( h \) of an infinite number of multi-step transactions \( T = \{T_i : i \in N\} \), accessing data items in some finite set \( D \) (though not necessarily accessing the same data items), is serializable iff the conflict graph \( G(h) \) is acyclic.

Proof. If

Let \( h \) be a history of \( T \) such that \( G(h) \) is acyclic. Assume that, for some \( T_i \in T \), we have the following infinite regression of arcs:

\[
\ldots, (T_{k_n+1}, T_{k_n}), \ldots, (T_{k_2}, T_{k_1}), (T_{k_1}, T_i)
\]

where \( \{T_{k_1}, \ldots, T_{k_n}, \ldots\} \subseteq T \). Then, as only finitely many data items are accessed by the transactions, and as each step may be preceded by only finitely many steps in \( h \), there exist \( l > j \geq 0 \) such that we have the following order of steps in \( h \) (assuming, without loss of generality, that the arcs in (3.1) are as the result of write-read conflicts):

\[
w_{k_i+1}(x) <_h r_k(x) \ldots <_h w_{k_j+1}(x) <_h r_{k_j}(x)
\]
Chapter 3. Multi-step Transactions Accessing The Same Set Of Data Items 27

and $w_{kl}(x)$ does not precede $r_{kl}(x)$ in $h$, i.e.

$$r_{kl}(x) <_h w_{kl}(x)$$

From (3.2) and (3.3), we produce the cycle

$$(T_{k'}, T_{k-1}), \ldots, (T_{k'+1}, T_{k''}), (T_{k''}, T_{k'})$$

This contradiction shows that (3.1) cannot occur. It follows that we can define, inductively, the sequence $i^1, i^2, \ldots$ thus:

\[
\begin{align*}
  i^1 &= \min\{k \in \mathbb{N}_1 : \text{for all } i \neq k, (T_i, T_k) \notin \text{arcs}(G(h))\} \\
  \ldots \\
  i^n &= \min\{k \in \mathbb{N}_1 : \text{for all } i \notin \{k, i^1, \ldots, i^{n-1}\}, (T_i, T_k) \notin \text{arcs}(G(h))\} \\
  \ldots
\end{align*}
\]

Firstly, we show that $\{i^1, \ldots, i^n, \ldots\} = \mathbb{N}_1$. Suppose, on the contrary, that there is some $i' \in \mathbb{N}_1$ such that $i' \neq i^n$ for any (all) $n \in \mathbb{N}_1$. We can choose $i'$ to be such that

$$(T_i, T_{i'}) \implies T_i = i^n \text{ for some } n \in \mathbb{N}_1$$

(3.4)

The reason that we can choose $i'$ satisfying (3.4) is that otherwise we could generate, inductively, $k^1, \ldots, k^n, \ldots \notin \{i^1, \ldots, i^n, \ldots\}$ giving an infinite regression of arcs:

$$\ldots, (T_{k'+1}, T_{k''}), \ldots, (T_{k^2}, T_{k^1}), (T_{k'}, T_{i'})$$

as in (3.1) which we have shown cannot occur. Intuitively, $T_{i'}$ is the 'earliest' transaction for which $T_{i'} \neq i^n$ for any $n \in \mathbb{N}_1$. Now choose $n' \in \mathbb{N}_1$ to be such that:

(a) $i^{n+1} \geq i'$

(b) all the steps of any $T_{i^n}$, where $n \geq n'$, come after the steps of $T_{i'}$ in $h$.

Assume that we have that

$$(T_i, T_{i'}) \in \text{arcs}(G(h)) \text{ for some } i \in \mathbb{N}_1, i \notin \{i^1, \ldots, i^n\}$$

(3.5)

This implies, by (3.4), that

$$T_i = T_{i^n} \text{ for some } n \in \mathbb{N}_1$$
which implies, by the assumption at (3.5) that \( i \neq i^1, \ldots, i^{n'} \),

\[ T_i = T_{i'} \text{ for some } n > n' \]

This, in turn, implies, by (b), that all the steps of \( T_i = T_{i'} \) come after the steps of \( T_i' \) in \( h \) and so \((T_i, T_i') \notin \text{arcs}(G(h))\). This contradicts (3.5) and so the assumption at (3.5) cannot hold. But, then,

\[
i^{n'+1} = \min \{k \in \mathbb{N}_1 : \text{ for all } i \neq k, i^1, \ldots, i^{n'}, (T_i, T_k) \notin \text{arcs}(G(h))\} \quad (3.6)
\]

As (3.5) cannot hold, by (3.6) and (a) we have that

\[
i' > i^{n'+1} \geq i'
\]

This last contradiction means that the assumption that \( i' \neq i^n \text{ for all } n \in \mathbb{N}_1 \) is false. It follows that \( \{i^1, \ldots, i^n, \ldots\} = \mathbb{N}_1 \).

We construct the serial history \( h_S \) of \( T \) given by \( h_S = \)

\[
\underbrace{\ldots r_1(x) \ldots w_{i^1}(y) \ldots}_{\text{all steps of } T_{i^1}} \quad \underbrace{\ldots r_{n}(s) \ldots w_{i^n}(t) \ldots}_{\text{all steps of } T_{i^n}}
\]

and show that \( h \sim h_S \) by showing that Definition 3.1(1), (2) and (3) hold. For Definition 3.1(1), suppose that \( r_i(x) <_h w_{i'}(x) \) for transactions \( T_i, T_i' \in T \). Then, \((T_i, T_i') \in \text{arcs}(G(h))\). In the sequence \( i^1, i^2, \ldots \) above, we cannot have \( i = i^n \) and \( i' = i'^n \) for some \( n' < n \) as, from the definition of \( n' \), that would imply that \((T_i, T_i') \notin \text{arcs}(G(h))\). Thus, \( h_S \) is of the form

\[
\underbrace{\ldots r_1(c) \ldots w_{k}(d) \ldots}_{\text{all steps of } T_i} \quad \underbrace{\ldots r_{n'}(c) \ldots w_{i'}(f) \ldots}_{\text{all steps of } T_{i'}}
\]

and Definition 3.1(1) holds as required. The proof of Definition 3.1(2) and (3) are similar.

Only if

Let \( h \) be a serializable history. This means that there is a serial history \( h_S \) such that \( h \sim h_S \). This implies, by Definitions 3.1 and 3.3, that \( G(h) = G(h_S) \). As \( G(h_S) \) is necessarily acyclic, since it must be a subgraph of the total order under which the transactions occur in \( h_S \), we conclude that \( G(h) \) is acyclic.

In the case where all transactions access the same set of data items, serializability is guaranteed if \( G(h) \) has no cycle of length 2.

**Lemma 3.5.** Let \( h \) be a history of multi-step transactions \( T = \{T_i : i \in \mathbb{N}_1\} \) acce-
sing the same set of data items \( D \) (in possibly different orders). Then, if \( G(h) \) has a cycle, there are transactions \( T_i, T_j \) such that \( G(h) \) has the cycle \((T_i, T_j), (T_j, T_i)\).

**Proof.** Assume that \( G(h) \) has a cycle

\[
(T^n, T'^0), (T^n, T'^1), \ldots, (T^n, T'^{n-1}), (T'^0, T'^n)
\]

where \( n > 2 \), but no such cycle for \( n = 2 \). We will derive a contradiction. Choose any \( x \in D \). Then, for \( 1 \leq j \leq n - 1 \),

\[
w_j(x) \prec_h r_{j+1}(x)
\]

otherwise \((T^n, T'^j), (T^n, T'^{j+1})\) is an arc in \( G(h) \) and, from (3.7), \((T^n, T'^{j+1})\) is also an arc in \( G(h) \) giving a cycle between \( T^n \) and \( T'^{j+1} \) contrary to our assumption that there are no cycles of length 2. From (3.8) we have that

\[
r_i(x) \prec_h w_i(x) \prec_h \cdots \prec_h w_n(x) \prec_h r_i(x)
\]

The contradiction, from (3.9), that \( r_i(x) \prec_h r_i(x) \), means that our assumption that there is no cycle between two transactions is incorrect. \( \square \)

To demonstrate Lemma 3.5, we give the following example. Let 3 transactions access the set of data item \( D \) in the following way:

\[
T_1 = r_1(x_1)w_1(x_1)r_1(x_2)w_1(x_2)r_1(x_3)w_1(x_3)
\]

\[
T_2 = r_2(x_2)w_2(x_2)r_3(x_1)w_2(x_1)r_3(x_3)w_2(x_3)
\]

\[
T_3 = r_3(x_1)w_3(x_1)r_3(x_2)w_3(x_2)r_3(x_3)w_3(x_3)
\]

where \( D = \{x_1, x_2, x_3\} \) and \( T' = \{T_i : 1 \leq i \leq 3\} \subseteq T \). Also, suppose that we have a cycle in the conflict graph \( G(h_{T'}) \) containing all transactions in \( T' \) and corresponding to the following history \( h_{T'} \) (see also Figure 3.1):

\[
h_{T'} = r_1(x_1)w_1(x_1)r_1(x_2)w_2(x_2)r_2(x_2)w_2(x_1)r_3(x_3)w_3(x_3)r_3(x_3)w_2(x_3)
\]

We notice, from Figure 3.1, that there is a cycle of length 3 and also a cycle of length 2. The reason is that transaction \( T_1 \) precedes \( T_2 \) in all data items \( x_1, x_2 \) and \( x_3 \), \( T_2 \) precedes \( T_3 \) in \( x_1, x_2 \), and \( T_3 \) precedes \( T_1 \) in \( x_2 \). This implies that \( T_3 \) precedes \( T_2 \) in \( x_3 \). This means that we have a cycle between \( T_2 \) and \( T_3 \). Also, as \( T_1 \) precedes \( T_2 \) in \( x_1 \) and \( x_2 \), and \( T_2 \) precedes \( T_3 \) in \( x_1 \) and \( x_2 \), this implies that \( T_1 \) precedes \( T_3 \) in \( x_1 \) and \( x_2 \). Therefore, we have a cycle between \( T_1 \) and \( T_3 \).
Chapter 3. Multi-step Transactions Accessing The Same Set Of Data Items

Definition 3.6. We say that $T_i$ comes before $T_{i'}$ in $h$ iff $w_i(x) <_h r_{i'}(y)$, where $x$ and $y$ are the first data items accessed by $T_i$ and $T_{i'}$ respectively.

Theorem 3.7. A history $h$ of multi-step transactions $T = \{ T_i : i \in \mathbb{N}_1 \}$ is serializable iff for any two distinct transactions $T_i$ and $T_{i'}$, one of them, $T_i$ say, is such that

(i) $T_i$ comes before $T_{i'}$ in $h$, and

(ii) for all $x \in D$, $w_i(x) <_h r_{i'}(x)$

Proof. If

Let $h$ be not serializable. We show that there are $T_i$ and $T_{i'}$ such that the conditions (i) and (ii) do not both hold. To have $h$ not serializable means, by Theorem 3.4 and Lemma 3.5, that there is a cycle in the precedence graph $G(h)$, $(T_i, T_{i'}), (T_{i'}, T_i)$. Assume that (i) holds for $T_i$ and $T_{i'}$, i.e. $T_i$ comes before $T_{i'}$. Here, letting $x$ and $y$ denote the first data items accessed by $T_i$ and $T_{i'}$ respectively, there are a limited number of cases causing the cycle:

\[
\ldots w_i(x) \ldots r_{i'}(y) \ldots w_{i'}(y) \ldots r_i(z) \ldots w_i(z) \ldots w_i(z) \ldots (3.10)\\
\ldots w_i(x) \ldots r_{i'}(y) \ldots w_{i'}(y) \ldots r_i(z) \ldots w_i(z) \ldots w_{i'}(z) \ldots (3.11)\\
\ldots w_i(x) \ldots r_{i'}(y) \ldots w_{i'}(y) \ldots r_{i'}(z) \ldots r_i(z) \ldots w_i(z) \ldots w_{i'}(z) \ldots (3.12)\\
\ldots w_i(x) \ldots r_{i'}(y) \ldots w_{i'}(y) \ldots r_{i'}(z) \ldots r_i(z) \ldots w_i(z) \ldots w_{i'}(z) \ldots (3.13)
\]

In the all cases (3.10)-(3.13) condition (ii) is breached because $r_{i'}(z) <_h w_i(z)$ (underlined).

Only if

Assume that we have that $h$ does not satisfy conditions (i) and (ii) for all $T_i, T_{i'}$. We show that $h$ is not serializable. Firstly, suppose that condition (i) holds, but that condition (ii) does not hold for some $T_i$ and $T_{i'}$. Then, if $x$ and $y$ are the first
Chapter 3. Multi-step Transactions Accessing The Same Set Of Data Items 31

data items accessed by \( T_i \) and \( T_{i'} \) respectively,

\[
w_i(x) <_h r_{i'}(y)
\] (3.14)
giving the arc \((T_i, T_{i'})\) in \( G(h) \). As condition (ii) does not hold, there is \( x \in D \) such that

\[
r_{i'}(z) <_h w_i(z)
\]
This gives the arc \((T_{i'}, T_i)\) and hence a cycle in \( G(h) \). By Theorem 3.4, \( h \) is not serializable. Secondly, suppose condition (i) does not hold for some \( T_i, T_{i'} \). Then, by Definition 3.6, if \( x \) and \( y \) are the first data items accessed as above,

\[
r_i(x) <_h w_{i'}(y)
\] (3.15)
and

\[
r_{i'}(y) <_h w_i(x)
\] (3.16)

From (3.15), if \( x = y \), \((T_i, T_{i'})\) is an arc in \( G(h) \), and from (3.16) \((T_{i'}, T_i)\) is an arc in \( G(h) \). This gives a cycle and shows, by Theorem 3.4, that \( h \) is not serializable. But, if \( x \neq y \), \( T_i \) accesses \( y \) later, and \( T_{i'} \) accesses \( x \) later. Thus, by (3.16),

\[
r_{i'}(y) <_h w_i(x) <_h r_i(y) <_h w_{i'}(y)
\] (3.17)
and, by (3.15),

\[
r_i(x) <_h w_{i'}(y) <_h r_{i'}(x) <_h w_i(x)
\] (3.18)

From (3.17), \((T_{i'}, T_i)\) is an arc in \( G(h) \) and, from (3.18), \((T_i, T_{i'})\) is an arc in \( G(h) \) giving a cycle.

In order to explain how the serializability conditions, that are defined in Theorem 3.7, can be used to verify whether a history \( h \) is serializable or not, we give the following example. Suppose that we have 3 transactions accessing the set of data items \( D \) in the following way:

\[
T_1 = r_1(x_1)w_1(x_1)r_1(x_2)w_1(x_2)r_1(x_3)w_1(x_3)
\]
\[
T_2 = r_2(x_1)w_2(x_1)r_2(x_2)w_2(x_2)r_2(x_3)w_2(x_3)
\]
\[
T_3 = r_3(x_2)w_3(x_2)r_3(x_3)w_3(x_3)r_3(x_1)w_3(x_1)
\]

where \( D = \{x_1, x_2, x_3\} \). Let \( h_{T'} \) be the history of \( T' = \{T_i : 1 \leq i \leq 3\} \), where
Chapter 3. Multi-step Transactions Accessing The Same Set Of Data Items

\[ T' \subseteq T, \]

\[ h_{T'} = r_1(x_1)w_1(x_1)r_1(x_2)r_2(x_1)w_2(x_1)w_1(x_2)r_2(x_2)w_2(x_2)r_3(x_2)w_3(x_2)r_1(x_3)w_1(x_3) \]
\[ w_1(x_3)r_2(x_3)w_2(x_3)r_3(x_3)w_3(x_3)r_3(x_1)w_3(x_1) \]

Firstly, we chop the history \( h_{T'} \) into sets of histories each containing two different transactions. Then, we verify whether the serializability conditions are satisfied for each set in order to make sure that the history \( h_{T'} \) is serializable. Otherwise, if \( h_{T'} \) is not serializable, the main history \( h \) will not be serializable. For the following

\[ h_{\{T_1,T_2\}} = r_1(x_1)w_1(x_1)r_1(x_2)r_2(x_1)w_2(x_1)w_1(x_2)r_2(x_2)w_2(x_2)r_1(x_3)w_1(x_3) \]
\[ r_2(x_3)w_2(x_3) \]

we notice that \( T_1 \) comes before \( T_2 \) (\( w_1(x_1) <_h r_2(x_1) \)). This means that condition (i) of Theorem 3.7 is satisfied. Also, we notice that \( T_1 \) precedes \( T_2 \) in accessing all data items of \( D \), i.e., for all \( x \in D \), \( w_1(x) <_h r_2(x) \). This means that condition (ii) of Theorem 3.7 is satisfied. For

\[ h_{\{T_1,T_3\}} = r_1(x_1)w_1(x_1)r_1(x_2)r_2(x_1)w_1(x_2)r_3(x_2)w_3(x_2)r_1(x_3)w_1(x_3)r_3(x_3)w_3(x_3) \]
\[ r_3(x_1)w_3(x_1) \]

we notice that \( T_1 \) comes before \( T_3 \) (\( w_1(x_1) <_h r_3(x_2) \)). Also, \( T_1 \) precedes \( T_3 \) in accessing all data items of \( D \), i.e., for all \( x \in D \), \( w_1(x) <_h r_3(x) \). This means that conditions (i) and (ii) of Theorem 3.7 are satisfied. Finally, for

\[ h_{\{T_2,T_3\}} = r_2(x_1)w_2(x_1)r_2(x_2)w_2(x_2)r_3(x_2)w_3(x_2)r_2(x_3)w_2(x_3)r_3(x_3)w_3(x_3) \]
\[ r_3(x_1)w_3(x_1) \]

both conditions of serializability of Theorem 3.7 are also satisfied. Therefore, we can say that the history \( h_{T'} \) is serializable and the corresponding conflict graph \( G(h_{T'}) \), in Figure 3.2, shows that \( h_{T'} \) is serializable because it does not have a cycle. Therefore, from Theorem 3.7 and the example above, we see that it is sufficient to check the serializability conditions for every two transactions, participating in \( h \), to make sure that \( h \) is serializable. In the next section (Section 3.4), we shall encode these serializability conditions into CTL.
3.4 Specification of serializability in CTL

We present a CTL specification of infinite histories composed of $n$ transactions each accessing all of $m$ data items, and repeating infinitely often. The aggregate of all the repetitions of the $n$ transactions will constitute the infinite number of transactions $\{T_i : i \in \mathbb{N}_1\}$ of the previous section. Such concurrent repeating or 'iterating' transactions were originally investigated in [30] and temporal logic models have been given in [35] and [14]. In [35] and [14] each iteration of a transaction is called an occurrence, and every occurrence of a particular transaction comprises the same two (read and write) steps. We improve this to a case of multi-step transactions where, different occurrences of particular transactions access the same data items, but in possibly different orders. So, the order of access of data items may be different between different transactions and between different occurrences of the 'same' transaction. Actually, in our model here, different occurrences of the 'same' transaction bear no relation to each other. As such, we model, not so much the same $n$ transactions iterating, but a more general case of an infinite number of (possibly totally unrelated) transactions where there is a limit of $n$ on how many are active at any given time.

The syntax for CTL is given in Subsection 3.4.1 and the semantics in Subsection 3.4.2. The specification of the multi-step transactions model is in Subsection 3.4.3 and serializability is specified in 3.4.4. The complexity gain of checking two transactions is discussed in Subsection 3.4.5.

3.4.1 Syntax

The alphabet of CTL consists of a set of propositions symbols $p_0, p_1, \ldots$, distinguished read/write step propositional symbols $r_i(x_j), w_i(x_j)$ ($1 \leq i \leq n, 1 \leq j \leq m$), booleans $\neg, \lor, \land, T, \bot$, quantifiers $E, A$, and temporal operators $X, F, G$ and $U$. Formulæ in CTL are those generated by:

$$\phi ::= p_i \mid r_i(x_j) \mid w_i(x_j) \mid \neg \phi \mid \phi_1 \lor \phi_2 \mid \phi_1 \land \phi_2 \mid AX \phi$$
Chapter 3. Multi-step Transactions Accessing The Same Set Of Data Items

\[ \mathbf{EX} \phi \mid \mathbf{AF} \phi \mid \mathbf{EF} \phi \mid \mathbf{AG} \phi \mid \mathbf{EG} \phi \mid \mathbf{A}[\phi_1 \mathbf{U} \phi_2] \mid \mathbf{E}[\phi_1 \mathbf{U} \phi_2] \]

Note that, despite their appearance, \( r_i(x_j) \) and \( w_i(x_j) \) are propositions, and not predicates, in the logic. The symbols \( \bot \) and \( \top \) will also be used to denote the truth values false and true respectively and the abbreviations \( \Rightarrow \) and \( \Leftrightarrow \) will have their usual logical meaning.

3.4.2 Semantics of CTL

An interpretation for CTL, \( I(s_a) \), at a given state \( s_a \in S \), where \( S \) is a set of states, assigns truth values \( p_i^{I(s_a)}, r_i(x_j)^{I(s_a)} \) and \( w_i(x_j)^{I(s_a)} \) (\( \in \{ \bot, \top \} \)) to propositional symbols \( p_i, r_i(x_j) \) and \( w_i(x_j) \), respectively. A interpretation \( I \) over \( S \), is a set of interpretations \( I = \{ I(s_a) \mid s_a \in S \} \). A Kripke structure \( M \) is a triple \( \langle S, R, I \rangle \), where \( S \) is a set of states, \( R \subseteq S \times S \) a transition relation such that, for all \( s \in S \), there exists \( s' \in S \) with \( (s, s') \in R \), and \( I \) is an interpretation over \( S \). A path in \( M \) is an infinite sequence of states, \( \pi = s_a, s_{a+1}, \ldots \), such that, for every \( b \geq a, (s_b, s_{b+1}) \in R \). The set of paths that start in state \( s_a \) is denoted \( \text{Paths}(s_a) \).

As each state in a Kripke structure is required to have at least one successor, it follows that \( \text{Paths}(s_a) \neq \{ \} \) for any state \( s_a \). The semantics of a CTL formula \( \phi \) is given by the truth relation \( M, s_a \models \phi \) which means that \( \phi \) holds at state \( s_a \) in the Kripke structure \( M \). The relation \( \models \) is defined inductively as follows:

\[
M, s_a \models p_i \text{ iff } p_i^{I(s_a)} = \top
\]

\[
M, s_a \models r_i(x_j) \text{ iff } r_i(x_j)^{I(s_a)} = \top
\]

\[
M, s_a \models w_i(x_j) \text{ iff } w_i(x_j)^{I(s_a)} = \top
\]

\[
M, s_a \models \neg \phi \text{ iff } M, s_a \not\models \phi
\]

\[
M, s_a \models \phi_1 \lor \phi_2 \text{ iff } M, s_a \models \phi_1 \text{ or } M, s_a \models \phi_2
\]

\[
M, s_a \models \phi_1 \land \phi_2 \text{ iff } M, s_a \models \phi_1 \text{ and } M, s_a \models \phi_2
\]

\[
M, s_a \models \mathbf{AX} \phi \text{ iff, for all } \pi \in \text{Paths}(s_a), M, s_{a+1} \models \phi
\]

\[
M, s_a \models \mathbf{EX} \phi \text{ iff there exists } \pi \in \text{Paths}(s_a) \text{ such that } M, s_{a+1} \models \phi
\]

\[
M, s_a \models \mathbf{AF} \phi \text{ iff, for all } \pi \in \text{Paths}(s_a), \text{ there exists } b \geq a \text{ such that } M, s_b \models \phi
\]

\[
M, s_a \models \mathbf{EF} \phi \text{ iff there exists } \pi \in \text{Paths}(s_a) \text{ and } b \geq a \text{ such that } M, s_b \models \phi
\]

\[
M, s_a \models \mathbf{AG} \phi \text{ iff, for all } \pi \in \text{Paths}(s_a), \text{ and, for all } b \geq a, M, s_b \models \phi
\]

\[
M, s_a \models \mathbf{EG} \phi \text{ iff there exists } \pi \in \text{Paths}(s_a) \text{ such that, for all } b \geq a, M, s_b \models \phi
\]
Chapter 3. Multi-step Transactions Accessing The Same Set Of Data Items

3.4.3 Specification of the multi-step transactions model

The read and write step propositions have the following intuitive meanings:

\( r_i(x_j) \sim \) active transaction \( T_i \) has read data item \( x_j \)

\( w_i(x_j) \sim \) active transaction \( T_i \) has written to data item \( x_j \)

The multi-step transactions model is characterized by the following properties:

\((C1)\) Read/write alternation
A transaction \( T_i \) cannot have read two distinct data items without having written to one of them, i.e. \( r_i(x_j) \) and \( r_i(x_{j'}) \) cannot both be true if \( w_i(x_j) \) and \( w_i(x_{j'}) \) are both false.

\((C2)\) Write implies read
A transaction \( T_i \) can only have written to \( x_j \) if it has read \( x_j \), i.e. if \( w_i(x_j) \) is true, then \( r_i(x_j) \) must be true.

\((C3)\) Read/write steps remain true to transaction end
If a read/write step has taken place, the corresponding propositions remain true until the transaction ends, i.e. \( r_i(x_j)/w_i(x_j) \) once true, remain true until all other steps \( r_i(x'_{j}) \) and \( w_i(x'_{j}) \) \( (x'_{j} \in D) \) are true.

\((C4)\) End of transaction occurrence
After a transaction occurrence ends, at most one read step \( r_i(x_j) \) and no write steps \( w_i(x_j) \) can be true in any next state.

\((C5)\) At most one step occurs at each successive state
No two distinct steps can both be false in a state, and then both true in a next state.

Given a state \( s_a \), and a path \( \pi \in \text{Paths}(s_a) \), there corresponds a sequence of read and write step propositions that become true in \( s_a, s_{a+1}, \ldots \). In this way, \( \pi \) yields a history of infinitely many occurrences of the transactions \( T_1, \ldots, T_n \). We illustrate this correspondence between paths and histories as follows.
In the depiction above, we have $D = \{x, y, z\}$ and transactions

$$T_1 = r_1(x)w_1(x)r_1(y)w_1(y)r_1(z)w_1(z)$$
Chapter 3. Multi-step Transactions Accessing The Same Set Of Data Items

and

\[ T_2 = r_2(x)w_2(x)r_2(z)w_2(z)r_2(y)w_2(y) \]

Interpretations for read and write step propositions are given for successive states, and the top of each column displays the unique proposition that becomes true in the particular state. The corresponding history \( h \) is:

\[ h = r_1(x)w_1(x)r_1(y)r_2(x)w_2(x)w_1(y)r_1(z)r_2(z)w_2(z)r_2(y)w_2(y)w_1(x) \ldots \]

We encode the conditions (C1)-(C5) as \( \sigma_1, \sigma_2, \sigma_3, \sigma_4 \) and \( \sigma_5 \) respectively, below. We use an extra proposition \( endT_i \) to mark the states at which an occurrence of \( T_i \) ends, i.e. the states at which \( r_i(x_j) \) and \( w_i(x_j) \) are true for all \( x_j \). This is defined in \( \sigma_0 \) as follows:

\[
\sigma_0 = \bigwedge_{1 \leq i \leq n} \text{AG}(endT_i \iff \bigwedge_{1 \leq j \leq m} (r_i(x_j) \land w_i(x_j)))
\]

Conditions (C1)-(C5) are given below:

(C1) Read/write alternation
A transaction \( T_i \) cannot have read two distinct data items without having written to one of them, i.e. \( r_i(x_j) \) and \( r_i(x_{j'}) \) cannot both be true if \( w_i(x_j) \) and \( w_i(x_{j'}) \) are both false.

\[
\sigma_1 = \bigwedge_{1 \leq i \leq n} \bigwedge_{1 \leq j \neq j' \leq m} \neg \text{EF}(r_i(x_j) \land r_i(x_{j'}) \land \neg w_i(x_j) \land \neg w_i(x_{j'}))
\]

(C2) Write implies read
A transaction \( T_i \) can only have written to \( x_j \) if it has read \( x_j \), i.e. if \( w_i(x_j) \) is true, then \( r_i(x_j) \) must be true.

\[
\sigma_2 = \bigwedge_{1 \leq i \leq n} \bigwedge_{1 \leq j \leq m} \text{AG}(w_i(x_j) \Rightarrow r_i(x_j))
\]

(C3) Read/write steps remain true to transaction end
If a read/write step has taken place, the corresponding propositions remain true until the transaction ends, i.e. \( r_i(x_j) / w_i(x_j) \) once true, remain true until all other steps \( r_i(x_{j'}) \) and \( w_i(x_{j'}) \) \((x_{j'} \in D)\) are true.

\[
\sigma_3 = \bigwedge_{1 \leq i \leq n} \bigwedge_{1 \leq j \leq m} \text{AG}((r_i(x_j) \land \neg endT_i \Rightarrow AXr_i(x_j)) \land (w_i(x_j) \land \neg endT_i \Rightarrow AXw_i(x_j)))
\]
(C4) End of transaction occurrence
After a transaction occurrence ends, at most one read step \( r_i(x_j) \) and no write steps \( w_i(x_j) \) can be true in any next state.

\[
\sigma_4 = \bigwedge_{1 \leq i \leq n} \text{AG}(\text{end} T_i \Rightarrow \text{AX} \bigvee \bigwedge_{1 \leq j \leq m, 1 \leq j' \leq m} (\neg r_i(x_{j'}) \land \neg w_i(x_{j'}) \land \neg w_i(x_j)))
\]

(C5) At most one step occurs at each successive state
No two distinct steps can both be false in a state, and then both true in a next state.

\[
\sigma_5 = \bigwedge_{1 \leq i, i' \leq n} \bigwedge_{1 \leq j, j' \leq m, \text{i \neq i' or \text{j \neq j'}}} \text{AG}[\neg((\neg r_i(x_j) \land \neg r_i(x_{j'})) \land \text{EX}(r_i(x_j) \land r_i(x_{j'})))]
\]

We denote by \( \sigma_{\text{trans}} \) the specification of the transactions model, i.e.

\[
\sigma_{\text{trans}} = \sigma_0 \land \sigma_1 \land \sigma_2 \land \sigma_3 \land \sigma_4 \land \sigma_5
\]

3.4.4 Specification of serializability
We encode conditions (i) and (ii) of Theorem 3.7. We make use of additional propositions \( \text{before}_{i,i'} \) (\( 1 \leq i \neq i' \leq n \)), each of which is true in a state if the current occurrence of \( T_i \) comes before the current occurrence of \( T_{i'} \). We have that \( \text{before}_{i,i'} \) becomes true either if \( T_i \) has performed a write step and \( T_{i'} \) has not performed any read steps, or in a state which comes after a state in which the occurrence of \( T_{i'} \) ended and \( T_i \) had previously performed a write step. This is specified as \( \sigma_6 \):

\[
\sigma_6 = \bigwedge_{1 \leq i \neq i' \leq n} \text{AG}[\neg \text{before}_{i,i'} \Rightarrow \text{AX}(\neg \text{before}_{i,i'} \cup
\left( \left( \bigvee_{1 \leq j \leq m} w_i(x_j) \land \neg(r_{i'}(x_{j'}) \land w_{i'}(x_{j'})) \land \text{before}_{i,i'} \right) \lor
\left( \bigvee_{1 \leq j \leq m} w_i(x_j) \land \text{end} T_{i'} \land \neg \text{before}_{i,i'} \land \text{AX} \text{before}_{i,i'} \right) \right)]
\]
Also, we need to ensure that before\(_{i,i'}\), once true, remains true until the end of the occurrence of \(T_i\), and then becomes false. This is given by \(\sigma_7\):

\[
\sigma_7 = \bigwedge_{1 \leq i, i' \leq n} \text{AG}(\text{before}_{i,i'} \land \neg \text{end}_i \Rightarrow \text{AX} \text{before}_{i,i'} \land (\text{end}_i \Rightarrow \text{AX} \neg \text{before}_{i,i'}))
\]

Theorem 3.7 condition (i) can then be encoded as \(\sigma_8\) which states that, if \(T_i\) and \(T_{i'}\) are active, one of them must come before the other:

\[
\sigma_8 = \bigwedge_{1 \leq i, i' \leq n} \text{AG}(\bigvee_{1 \leq j, j' \leq m} r_i(x_j) \land r_{i'}(x_{j'})) \Rightarrow (\text{before}_{i,i'} \lor \text{before}_{i',i})
\]

Theorem 3.7 condition (ii) is encoded as \(\sigma_9\):

\[
\sigma_9 = \bigwedge_{1 \leq i, i' \leq n} \bigwedge_{1 \leq j, j' \leq m} \text{AG}(\text{before}_{i,i'} \Rightarrow \neg (r_{i'}(x_j) \land \neg w_i(x_j)))
\]

We denote by \(\sigma_{sz}\) the specification of the serializability condition, i.e.

\[
\sigma_{sz} = \sigma_6 \land \sigma_7 \land \sigma_8 \land \sigma_9.
\]

Now, we have given a method using CTL for specifying and verifying the correctness of concurrent executions of multi-step transactions produced by schedulers. For example, a scheduler might be specified as a finite-state machine in a model checker such as NuSMV, corresponding to a structure \(\text{Sched}\) for CTL. The specification would then be checked to see that the transactions model had been specified in the correct way. This would mean running the NuSMV model checker to show that

\[
\text{Sched, } s_a \models \sigma_{trans}
\]

Serializability could then be verified by using the NuSMV model checker to show that

\[
\text{Sched, } s_a \models \sigma_{sz}
\]

### 3.4.5 The complexity gain of checking two transactions

In Subsection 3.3, we have shown that the acyclicity of conflict graphs corresponds to serializability (Theorem 3.4) and that the existence of a cycle of any length in the conflict graphs for our transactions, implies the existence of a cycle of length two (as in Lemma 3.5). In other words, if the corresponding conflict graph of a history has a cycle (the history is not serializable) then there is a cycle between two transactions participating in the history. This means that checking for the existence of cycle of length two in a conflict graph suffices to verify whether
the corresponding history is serializable or not. These results are very useful for
improving the performance of model checking. Assume that we have \( n \) transactions
in a scheduler at a point in time, and we want to verify whether the history that
contains the transactions is serializable or not. Now, to check for the existence of
a cycle of length \( i \), the time complexity will be as follows:

\[
TC_i(n) = \frac{n(n-1) \ldots (n-i+1)}{i-1} = \prod_{j=0}^{i-1} (n-j)
\]

Therefore, the time complexity of checking for a cycle of length two \((i = 2)\), will
be

\[
TC_2(n) = n(n-1).
\] (3.19)

On the other hand, to check for the existence of a cycle of any length, we check
for the existence of cycles of length 2, 3, \ldots, \( n \) as follows

\[
TC(n) = \sum_{i=2}^{n} TC_i(n) = \sum_{i=2}^{n} \prod_{j=0}^{i-1} (n-j)
\] (3.20)

where \( TC(n) \) denotes the time complexity of checking for a cycle of any length.
From (3.19), it is easy to see that \( TC_2(n) \) has complexity of \( O(n^2) \). But, \( TC(n) \),
from (3.20), has complexity of \( O(n!) \). Clearly, the performance of checking two
transactions is much better than \( n \) transactions. Moreover, encoding the serial-
izability conditions using two transactions make the length of the specification
formula \(|f|\) much less than encoding serializability conditions using \( n \) transac-
tions. Given that the CTL model checking algorithm runs in \( O((|S| + |R|).|f|) \)
time, this means that the model checker needs much less time to verify the spe-
cification formula for two transactions than to verify the specification formula for
\( n \) transactions.

3.5 Applications

We have defined a serializability condition that scales well with increasing numbers
of transactions and data items. However, this has come at a price as we have
added the assumption that transactions access the same set of data items, albeit
in different orders. In fact, there are many applications where this assumption is
realistic. For example, people booking meals at restaurants over mobile phones.
Some may book the main course first, then maybe dessert, then starters, and finally tea or coffee. Others may choose to book in a different order. The availability of one course may influence the choice of another course and serializability of the booking transactions for the whole meals would be the appropriate correctness condition. In the next chapter we consider other, quite different, assumptions on the transactions model that result in a similar serializability condition that only needs to check for cyclicity between pairs of transactions and has a similar efficient encoding into CTL.
Chapter 4

Multi-step Transactions Accessing Ordered Sets Of Data Items

4.1 Introduction

In Chapter 3, we introduced infinite histories composed of \( n \) multi-step transactions each accessing all of \( m \) data items, and repeating infinitely often. In this chapter, we make different assumptions. We allow transactions to access possibly different subsets of data items of an ordered set of all data items \( D \). However, each transaction should access contiguous data items in the same order as they occur in \( D \). So, the accessed data items may be different between different transactions and between different occurrences of the same transaction. We show that, under these different but common assumptions on the graph structure of data items accessed by the transactions, conflict cyclicity still only needs to be checked within all possible pairs of transactions. Again, this results in formulae of considerably reduced size in any temporal-logic-based approach to proving serializability, and also scales to arbitrary numbers of transactions.

This chapter is structured as follows. First of all, in Section 4.2, we give a model for infinite numbers of concurrent multi-step transactions accessing ordered data. In Section 4.3, we prove that, in this model, a cycle exists in the (infinite) conflict graph if and only if a cycle exists between two transactions. A formal condition for serializability, that can be encoded into temporal logic, is given in Section 4.4. Applications that satisfy this model are discussed in Section 4.5.

4.2 Concurrent multi-step transactions model

The problem with specifying serializability is with specifying the existence of cycles in the infinite conflict graph of a history. As we saw in Chapter 3, there are \( n! \)
ways that a cycle can occur between \( n \) transactions and an encoding into temporal logic results in a formula whose length is of order a factorial of the number of propositions that represent different active transactions at any point in time. In the case of a scheduler dealing with at most \( n \) active transactions at any point in time, exhaustive proofs of serializability are beyond even the most powerful model checkers available for any realistic value of \( n \). However, in the case where transactions access the same set of data items as in Chapter 3, checks only need to be performed for cycles between pairs of transactions. In the general case, where transactions access different sets of data items, it is not sufficient to check for cycles between pairs of transactions. Consider the case of 3 transactions \( T_1, T_2 \) and \( T_3 \) which access the sets of data items \( \{x, y\}, \{y, z\} \) and \( \{x, z\} \) respectively, where \( x, y \) and \( z \) are all different, and the history \( h_1 \) whose steps occur in the following order:

\[
h_1 = r_3(x)w_3(x)r_1(x)w_1(x)r_1(y)w_1(y)r_2(y)w_2(y)r_2(z)w_2(z)r_3(z)w_3(z)
\]

History \( h_1 \) has the following cycle of conflicts:

\[
T_3 \rightarrow T_1 \rightarrow T_2 \rightarrow T_3
\]

(and is therefore not serializable). However, as can be checked, there is no cycle between two transactions. In fact, as the history \( h_2 \) below shows, the absence of a cycle between any \( n - 1 \) transactions does not guarantee the absence of a cycle between \( n \) transactions:

\[
h_2 = r_n(x_1)w_n(x_1)r_1(x_1)w_1(x_1)r_1(x_2)w_1(x_2)\ldots r_i(x_i)w_i(x_i)r_i(x_{i+1})w_i(x_{i+1})\ldots
\]

\[
\ldots r_{n-1}(x_{n-1})w_{n-1}(x_{n-1})r_{n-1}(x_n)w_{n-1}(x_n)r_n(x_n)w_n(x_n)
\]

Here,

\[
T_1, \ldots, T_i, \ldots, T_n
\]

access sets of data items

\[
\{x_1, x_2\}, \ldots, \{x_i, x_{i+1}\}, \ldots, \{x_n, x_1\}
\]

where \( x_1, \ldots, x_n \) are distinct. History \( h_2 \) has the cycle

\[
T_n \rightarrow T_1 \rightarrow \ldots T_{n-1} \rightarrow T_n
\]
but no cycle of fewer than \( n \) transactions - if \( T_i \ (1 \leq i \leq n) \) is removed from the history, the (acyclic) order of conflicts is:

\[
T_{i+1} \rightarrow \ldots \rightarrow T_n \rightarrow T_1 \rightarrow \ldots \rightarrow T_{i-1}
\]

We notice that in the examples \( h_1 \) and \( h_2 \) above, it is impossible to define an order on the data items, so that each transaction only accesses sets of contiguous data items. In this chapter, we show that if such an order exists, then to check for conflict cyclicity within sets of transactions, it is sufficient to check for conflict cyclicity between pairs of transactions. This results in a test for serializability of concurrent multi-step transactions, similar to that in Chapter 3, but for an entirely different class of transactions where the sets of data items accessed by transactions need not be the same.

### 4.2.1 Histories

We shall denote the set of multi-step transactions by \( T = \{T_i : i \in \mathbb{N}_1\} \), where \( \mathbb{N}_1 \) is the set of positive integers. A history (or schedule) \( h \) for the set \( T \) is an interleaved sequence of all the read and write steps of all the transactions in \( T \) such that the subsequence of \( h \) comprising the steps of \( T_i \) is exactly the sequence of steps of \( T_i \) occurring in the order that they do in \( T_i \). As in Chapter 3, for a history \( h \), we denote \( s_i <_h s'_i \) if step \( s_i \) of \( T_i \) occurs before step \( s'_i \) of \( T'_i \) in \( h \). In the next definition, we define formally an order on the set of the data items from which data is accessed by transactions.

**Definition 4.1.** Let \( D = \{x_1, x_2, \ldots, x_m\} \) be an irreflexively totally ordered, by \( <_D \) say, set of data items such that

\[
x_1 <_D \ldots <_D x_m
\]

and \( T = \{T_i : i \in \mathbb{N}_1\} \) be the set of transactions participating in history \( h \). Denote by \( D_i \) the totally ordered set of data items accessed in turn by transaction \( T_i \) assumed to be of the form

\[
D_i = \{x_a, x_{a+1}, \ldots, x_{b-1}, x_b\},
\]

where \( 1 \leq a \leq b \leq m \) and \( D_i \subseteq D \). For the remainder of this chapter, if a set of data items \( D' \subseteq D \) is denoted by \( \{x_a, \ldots, x_b\} \), this will mean that \( x_a <_D \ldots <_D x_b \). We shall denote the case of transaction \( T_i \) preceding transaction \( T'_i \) in accessing data items \( x_p, \ldots, x_s \) in both read and write operations over history \( h \), as \( T_i <^{x_p, \ldots, x_s}_h T'_i \) where \( x_p, x_{p+1}, \ldots, x_s \in D_i \cap D'_i \).
Chapter 4. Multi-step Transactions Accessing Ordered Sets Of Data Items

For example, assume that $D_i$ and $D_j$ are:

$$D_i = \{x_1, x_2, x_3\}$$

$$D_j = \{x_2, x_3, x_4, x_5\}$$

where $x_1 <_D x_2 <_D x_3 <_D x_4 <_D x_5$. Then, in the history

$$h = \ldots r_i(x_2)w_i(x_2)\ldots r_i(x_3)w_i(x_3)\ldots r_j(x_2)w_j(x_2)\ldots r_j(x_3)w_j(x_3)\ldots$$

we have that $T_i <_{h}^{x_2,x_3} T_j$.

### 4.3 Cycle reduction in conflict graphs

In this section, we prove a succession of properties of infinite histories culminating in Theorem 4.7. Theorem 4.7 is the main result and is used to give a condition for serializability in Section 4.4 that can be encoded efficiently into CTL or LTL.

The first two lemmas, Lemma 4.2 and Lemma 4.3, give basic properties of the order $<_{h}^{x_1,\ldots,x_n}$ from Definition 4.1.

**Lemma 4.2.** Let $h$ be a history with no cycle of length 2 in its conflict graph, and suppose that $x \in D$ and transactions $T_i$ and $T_j$ are such that $x \in D_i \cap D_j$. Then,

$$\text{either } T_i <_{h}^{x} T_j \text{ or } T_j <_{h}^{x} T_i \quad (4.1)$$

**Proof.** By the definition of $<_{h}^{x}$ in Definition 4.1, the only way (4.1) can fail is if two reads $r_i(x)$ and $r_j(x)$ occur before the two writes $w_i(x)$ and $w_j(x)$. But, then the conflict graph would have the cycle $(T_i, T_j, (1j, Ti))$. As $Ti <_{h}^{x} T_j$, there is another arc $(Ti, T_j)$ in $G(h)$ thereby completing a cycle of length 2 and contradicting the assumption that $G(h)$ has no cycle of length 2. $\square$

**Lemma 4.3.** Let $h$ be a history with no cycle of length two in its conflict graph $G(h)$, $T_i <_{h}^{x_1,x_2} T_j$, and suppose that $x_a$ lies between $x_{a_1}$ and $x_{a_2}$, i.e. $x_{a_1} <_D x_a <_D x_{a_2}$. Then, $T_i <_{h}^{x_a} T_j$, (see Figure 4.1).

**Proof.** As $x_{a_1}, x_{a_2} \in D_i \cap D_j$, and as, by Definition 4.1, $D_i$ and $D_j$ contain contiguous elements, we have that $x_a \in D_i \cap D_j$. If $T_i <_{h}^{x_a} T_j$ does not hold then, by Lemma 4.2, we must have that $T_j <_{h}^{x_a} T_i$. This implies that there is an arc in $G(h)$ such that $(T_j, T_i)$. As $T_i <_{h}^{x_1} T_j$, there is another arc $(T_i, T_j)$ in $G(h)$ thereby completing a cycle of length 2 and contradicting the assumption that $G(h)$ has no cycle of length 2. $\square$

For the next property Lemma 4.4 below, consider the case of three transactions.
Chapter 4. Multi-step Transactions Accessing Ordered Sets Of Data Items

Figure 4.1: The conflict graph $G(h)$ for Lemma 4.3.

$T_i$, $T_j$, and $T_k$ accessing data items such that the data items accessed by $T_j$ straddle those accessed by $T_i$ and $T_k$ as follows:

$$D = \{ ..., x_{k1}, x_{k2}, ..., x_{k\ell}, \ldots, x_{a1}, x_{a2}, \ldots, x_{a\ell}, \ldots \}$$

We claim that $T_i$, $T_j$, and $T_k$ cannot be part of a cycle in $G(h)$ of length $n$ where $n > 2$ if there is no cycle of length 2. For example, assume that there are 4 ordered sets of data items $D_i \supseteq \{x_a, x_c\}$, $D_j \supseteq \{x_b, x_a\}$, $D_k \supseteq \{x_z, x_b\}$ and $D_l \supseteq \{x_z, x_c\}$ accessed by corresponding transactions $T_i, T_j, T_k$ and $T_l$ respectively, which form a cycle as in Figure 4.2 below. Since $D_l$ contains $x_z$ and $x_c$ it follows, by Definition 4.1, that $x_b$ and $x_a$ are also in $D_l$, i.e.,

$$D_l \supseteq \{x_z, x_b, x_a, x_c\}.$$

Thus, $x_a \in D_i \cap D_l$ and so, by Lemma 4.2, either $T_i <^z_h T_l$ or $T_i <^e_h T_l$. If $T_i <^z_h T_l$ then, by transitivity of $<^z_h$, we can reduce the cycle, as in Figure 4.3(a). But, if $T_i <^e_h T_l$ then, we have a cycle of length two; see Figure 4.3(b). Now, for the case of Figure 4.3(a), from above we have that $x_b \in D_j \cap D_l$. Therefore, by Lemma 4.2, either $T_j <^z_h T_l$ or $T_l <^z_h T_j$. If $T_l <^z_h T_j$ then, by transitivity, we
have a cycle of length two; see Figure 4.4(b). But, if \( T_j <^h_x T_i \), then we also have a cycle of length two, see Figure 4.4(a). Below, Lemma 4.4 shows that in fact our claim is true for \( n \geq 3 \) number of transactions. Lemma 4.4 will be used to prove Lemma 4.5.

![Figure 4.3: We can either reduce the cycle or make a cycle of length two.](image)

![Figure 4.4: We have a cycle of length two if either \( T_j <^h_x T_i \) or \( T_i <^h_x T_j \).](image)

**Lemma 4.4.** Suppose that \( D_i, D_j \) and \( D_k \) are the sets of data items accessed by \( T_i, T_j \) and \( T_k \), respectively, and are of the form

\[
\begin{align*}
D_i &= \{ x_{a'_1}, x_{a'_2}, \ldots, x_{a'_l}, \ldots, x_c, \ldots \} \\
D_j &= \{ \ldots, x_{b'_1}, x_{b'_2}, \ldots, x_{b'_u}, \ldots, x_{a'_1}, x_{a'_2}, \ldots, x_{a'_l}, \ldots \} \\
D_k &= \{ \ldots, x_{b'_1}, x_{b'_2}, \ldots, x_{b'_u} \}
\end{align*}
\]

where, for all \( x_{a'_i} \in \{ x_{a'_1} : 1 \leq i \leq l \} \) and \( x_{b'_g} \in \{ x_{b'_1} : 1 \leq g \leq u \} \), \( x_{b'_g} <_D x_{a'_i}, x_c \notin D_j, T_i <^h_x T_j \) and \( T_j <^h_x T_k \). Then, there is no cycle in \( G(h) \) of minimum length \( n \), where \( n > 2 \) is of the form:

\[
(T_i, T_j), (T_j, T_k), \ldots, (T_i, T_{i+1}), \ldots, (T_s, T_i)
\]

and \( T_s <^h_x T_i \).

**Proof.** Assume, on the contrary, that we have a cycle of minimum length \( n > 2 \),

\[
(T_i, T_j), (T_j, T_k), (T_1, T_2), \ldots, (T_s, T_i) \quad (s \geq 0)
\]
Chapter 4. Multi-step Transactions Accessing Ordered Sets Of Data Items 48

(Where $s = 0$ is the case $T_s = T_k$). Let $x_0 = x_{y'}$, $x_1$, ..., $x_s = x_c$, $x_{s+1} = x_{a'}$ be such that, putting $T_i = T_{s+1}, T_j = T_{s+2},$ and $T_k = T_0$:

$T_{s+1}(= T_i) <_{h^s}^x T_{s+2}(= T_j), T_{s+2}(= T_j) <_{h^s}^y T_0(= T_k), ..., T_l <_{h^s} T_{l+1}, ...

\ldots, T_s <_{h^s} T_{s+1}(= T_i)$

We have that $x_c >_D x_{a'} >_D x_{y'},$ but, clearly, we cannot have

$x_{a'} >_D x_{y'} >_D x_1 >_D \ldots >_D x_l >_D \ldots >_D x_c >_D x_{a'}$

Therefore, there is some $l$, with $0 \leq l \leq s - 1$, such that

$x_l <_D x_{l+1} >_D x_{l+2}$ \hspace{1cm} (4.3)

There are two cases to consider corresponding to (4.3):

Case (i) $x_{l+2} <_D x_l <_D x_{l+1}$. In this case, as

$x_{l+2} <_D x_l <_D x_{l+1},$

and

$T_l <_{h^l} T_{l+1} <_{h^{l+1}} T_{l+2} <_{h^{l+2}} T_{l+3}$

we have that $x_{l+1}, x_{l+2}$ and therefore $x_l$ belong to $T_{l+2}$. As no cycle of length 2 exists between $T_{l+1}$ and $T_{l+2}$, we must have that $T_{l+1} <_{h^l} T_{l+2}$. Therefore, as $T_l <_{h^l} T_{l+1}$, by transitivity we have that $T_l <_{h^i} T_{l+2}$. This produces a cycle reduction of (4.2), which is a contradiction.

Case (ii) $x_l <_D x_{l+2} <_D x_{l+1}$. In this case, as

$x_l <_D x_{l+2} <_D x_{l+1},$

and

$T_l <_{h^l} T_{l+1} <_{h^{l+1}} T_{l+2} <_{h^{l+2}} T_{l+3}$

we have that $x_l, x_{l+1}$ and therefore $x_{l+2}$ belong to $T_{l+1}$. As no cycle of length 2 exists between $T_{l+1}$ and $T_{l+2}$, we must have that $T_{l+1} <_{h^l} T_{l+2}$. By transitivity, from $T_{l+1} <_{h^{l+2}} T_{l+2}$ and $T_{l+2} <_{h^{l+2}} T_{l+3}$, we get $T_{l+1} <_{h^{l+2}} T_{l+3}$ giving a cycle reduction. This contradiction completes the proof of the lemma. \hfill $\Box$

Now, consider the case where $z <_D y <_D x$ and $T_1, T_2$ and $T_3$ access the
following sets of data items

\[ D_1 = \{x\}, D_2 = \{y, x\} \text{ and } D_3 = \{z, y\}. \]

Also, consider a history \( h \), which contains \( T_1, T_2 \) and \( T_3 \), of the form:

\[
\begin{align*}
    h = \ldots r_2(y) \ldots w_2(y) \ldots r_3(y) \ldots w_3(y) \ldots r_1(x) \ldots w_1(x) \\
    \ldots r_2(x) \ldots w_2(x) \ldots 
\end{align*}
\]

The corresponding conflict graph \( G(h) \), for the history \( h \), is shown in Figure 4.5. This shows a situation that Lemma 4.5 can remove, i.e. Lemma 4.5 asserts that if we have cycle in \( G(h) \) of length \( n \), where \( n > 2 \), then, any three consecutive transactions \( T_i, T_j \) and \( T_k \), participating in \( G(h) \), should contain \( x_{a'} \) and \( x_{b'} \) such that \( x_{a'} <_D x_{b'} \), \( T_i <^h_{a'} T_j \) and \( T_j <^h_{b'} T_k \). We need such \( x_{a'} \) and \( x_{b'} \) of Lemma 4.5 along with Lemma 4.6 to prove the main result Theorem 4.7 that reduces to cycles of length 2.

![Figure 4.5: No cycle in \( G(h) \)]

**Lemma 4.5.** Let \( h \) be a history with a cycle in \( G(h) \) of minimum length \( n \), where \( n > 2 \), containing arcs \((T_i, T_j)\) and \((T_j, T_k)\), and \( x_a, x_b \) such that \( T_i <^h_{a'} T_j \) and \( T_j <^h_{b'} T_k \). Then, there exist \( x_{a'}, x_{b'} \) such that \( x_{a'} <_D x_{b'} \) and \( T_i <^h_{a'} T_j \) and \( T_j <^h_{b'} T_k \).

**Proof.** Assume, on the contrary, that we have a cycle in \( G(h) \) of length \( n \), where \( n > 2 \), containing arcs \((T_i, T_j), (T_j, T_k)\) such that \( T_i <^h_{a'} T_j \) and \( T_j <^h_{b'} T_k \) for some \( x_a, x_b \), but that there are no \( x_{a'}, x_{b'} \) such that \( x_{a'} <_D x_{b'} \) and \( T_i <^h_{a'} T_j \) and \( T_j <^h_{b'} T_k \). This means that, for all \( x_{a'} \in D_i \cap D_j \) such that \( T_i <^h_{a'} T_j \) and, for all \( x_{b'} \in D_j \cap D_k \) such that \( T_j <^h_{b'} T_k \), we have that \( x_{b'} <^h_{b'} T_k \). Therefore, \( D_i \) contains data items \( x_{a'} \in D_i \cap D_j = \{x_{a'_1}, x_{a'_2}, \ldots, x_{a'_i}\} \), \( D_k \) contains data items \( x_{b'} \in D_j \cap D_k = \{x_{b'_1}, x_{b'_2}, \ldots, x_{b'_j}\} \) and \( D_j \) contains all data items \( x_{a'} \in D_i \cap D_j \) and \( x_{b'} \in D_j \cap D_k \) so that \( D_j \supset\{x_{b'_1}, x_{b'_2}, \ldots, x_{b'_j}, x_{a'_1}, x_{a'_2}, \ldots, x_{a'_i}\} \), i.e. \( D_j \) is of the form:

\[
D_j = \{\ldots, x_{b'_1}, \ldots, x_{b'_j}, \ldots, x_{a'_1}, \ldots, x_{a'_i}, \ldots\} \quad (4.4)
\]
We show, from (4.4), that, in fact, $D_i$ and $D_k$ should be as follows

$$D_i = \{x_{a'_1}, x_{a'_2}, \ldots, x_{a'_l}, \ldots\} \text{ and }$$

$$D_k = \{x_{b'_1}, x_{b'_2}, \ldots, x_{b'_u}\} \text{ (4.5)}$$

Firstly, assume that (4.5) does not hold, i.e. we can find $x_c \in D_i$ such that $x_c <_D x_{a'_1}$ and

$$D_i = \{x_{a'_1}, x_{a'_2}, \ldots, x_{a'_l}, \ldots\}. \text{ (4.6)}$$

From (4.4), we can choose $x_c$ to be such that $x_c \in D_j$ and therefore $x_c \in D_i \cap D_j$. If $x_c$ is such that $T_i <^{x_c}_h T_j$ then, $x_c \in \{x_{a'_i} : 1 \leq i \leq l\}$ and this contradicts our assumption that $x_c <_D x_{a'_1}$. On the other hand, if $T_j <^{x_c}_h T_i$, we have $(T_j, T_i)$ in $G(h)$ which with $(T_i, T_j)$, from $T_i <^{x_c}_h T_j$, completes a cycle of length 2 which contradicts the hypothesis of this lemma. Thus, we have now shown that (4.5) must hold.

Secondly, assume that (4.6) does not hold, i.e. we can find $x_c$ such that $x_{b'_u} <_D x_c$ and $x_c \in D_k \cap D_j$. If $x_c$ is such that $T_j <^{x'_u}_h T_k$ then, $x_c \in \{x_{b'_i} : 1 \leq i \leq u\}$ and this contradicts our assumption that $x_{b'_u} <_D x_c$. On the other hand, if $T_k <^{x'_u}_h T_j$, we have an arc $(T_k, T_j)$ in $G(h)$ which, along with the arc $(T_j, T_k)$, forms a cycle of length 2 contradicting the hypothesis of this lemma. Thus (4.6) must hold.

We shall now show that our main assumption that no $x_{a'} <_D x'_{b'}$ exists such that $T_i <^{x'}_h T_j$ and $T_j <^{x'}_h T_k$, leads to a contradiction. Since, we have a cycle, there is $x_c$ and a transaction $T_{i-1}$ such that $T_{i-1} <^{x_c}_h T_i$. We cannot choose such $x_c$ in $\{x_{a'_i} : 1 \leq i \leq l\}$ because then we could reduce the cycle as we would have $x_c \in T_j$ and therefore $T_{i-1} <^{x_c}_h T_j$ and an arc $(T_{i-1}, T_j)$. Also, we cannot choose $x_c$ such that $x_{a'_i} <_D x_c$ and $x_c \in D_j$ as we could reduce the cycle because then $T_{i-1} <^{x_c}_h T_j$. However, if $x_{a'_i} <_D x_c$ and $x_c \notin D_j$, we have that

$$D_i = \{x_{a'_1}, x_{a'_2}, \ldots, x_{a'_l}, \ldots, x_c, \ldots\}$$

and $x_c \notin D_j$ giving the conditions of Lemma 4.4. Application of Lemma 4.4 shows that $(T_i, T_j)$ and $(T_j, T_k)$ could not form part of a cycle. This contradiction completes the proof.

**Lemma 4.6.** If $h$ is a history with a cycle in $G(h)$ of minimum length $n$, where $n > 2$, and there are $T_i, T_j$ and $T_k$ such that $T_i <^{x'}_h T_j$, $T_j <^{x'_u}_h T_k$, and $x_{a'} <_D x'_{b'}$, then $x_{a'} <_D x'_{b'}$, as in Figure 4.6.

**Proof.** Assume, on the contrary, that $x_{b'_u} <_D x_{a'}$. As $x_{a'} <_D x'_{b'}$, then $x_{a'}, x'_{b'}$ and
$x'_{y}$ will be ordered such that

$$x'_{y} <_{D} x'_{a'} <_{D} x'_{y}$$

(4.7)

As $x'_{y} \in D_{j} \cap D_{k}$ and $x'_{y} \in D_{j} \cap D_{k}$ then, by Lemma 4.3 and (4.7), $x'_{a'} \in D_{j} \cap D_{k}$.

Therefore, we should have either $T_{j} <_{h} x'_{a'} T_{k}$ or $T_{k} <_{h} x'_{a'} T_{j}$. Now, if $T_{j} <_{h} x'_{a'} T_{k}$, then we can reduce the cycle via $T_{i} <_{h} x'_{a'} T_{k}$ giving a contradiction. But, if $T_{k} <_{h} x'_{a'} T_{j}$, then this gives an arc $(T_{k}, T_{j})$ which with the arc $(T_{j}, T_{k})$ from $T_{j} <_{h} x'_{a'} T_{k}$, completes a cycle of length 2 which is also a contradiction.

Figure 4.6: Represents Lemma 4.6

From the previous lemma, we conclude that if we have the data item $x_{a} \in D$ such that $T_{i} <_{h} x_{a} T_{j}$, $\{x_{h_{i}} : 1 \leq i \leq u\} \subseteq D$ such that $T_{j} <_{h} x_{h_{1}} \ldots x_{h_{u}} T_{k}$, and there exists $x_{b}$ such that $x_{a} <_{D} x_{b}$, where $x_{b} \in \{x_{h_{i}} : 1 \leq i \leq u\}$, then each data item $x_{y'}$ in $\{x_{h_{i}} : 1 \leq i \leq u\}$ will be such that $x_{a} <_{D} x_{y'}$. This will be used in the following Theorem 4.7.

**Theorem 4.7.** Let $h$ be a history over transactions $T = \{T_{i} : i \in N_{1}\}$, where each $T_{i} \in T$ accesses the data items $D_{i} \subseteq D$. Then, if $G(h)$ has a cycle of length $n$ and $n \geq 3$, there are two transactions $T_{i_{1}}, T_{i_{2}}$ such that $G(h)$ has the cycle $(T_{i_{1}}, T_{i_{2}}), (T_{i_{2}}, T_{i_{1}})$.

**Proof.** Assume that $G(h)$ has a cycle

$$(T_{1}, T_{2}), (T_{2}, T_{3}), \ldots, (T_{n}, T_{1})$$

(4.8)

but no such cycle between two transactions. Choose $x_{a}, x_{b}, x_{c}, \ldots, x_{d}$ such that:

$$T_{1} <_{h} x_{a} T_{2}, T_{2} <_{h} x_{b} T_{3}, T_{3} <_{h} x_{a} T_{4}, \ldots, T_{n} <_{h} x_{d} T_{1}$$

(see Figure 4.7). Put:

$$x_{a''} = \min \{x_{a'} : D : T_{1} <_{D} x_{a'} T_{2}\}$$
$x_{y'} = \min\{x_{y'_1} \in D : T_2 <^D T'_3\}$

$x_{d'} = \min\{x_{d'_1} \in D : T_3 <^D T'_4\}$

... 

$x_{d''} = \min\{x_{d'_1} \in D : T_n <^D T'_1\}$

(see Figure 4.8), where \(\min\) is with respect to \(<_D\). By Lemma 4.5, there exist \(x_{a'}\) and \(x_{y'}\) such that:

\[x_{a'} <_D x_{y'}, T_1 <^h T_2, T_2 <^h T_3\]

By Lemma 4.6, as \(T_1 <^h T_2, T_2 <^h T_3\) and \(x_{a'} <_D x_{y'}\), we have that \(x_{a'} <_D x_{y'}\). As, by the definition of \(x_{a''}, x_{a''} \leq_D x_{a'}\), we have that \(x_{a''} \leq_D x_{a'} <_D x_{y''}\) and so

\[x_{a''} <_D x_{y''}\]  \hspace{1cm} (4.9)

In a similar fashion, we can prove that:

\[x_{y''} <_D x_{o''}, \ldots, x_{d''} <_D x_{a''}\]  \hspace{1cm} (4.10)

From (4.9) and (4.10) we have that

\[x_{a''} <_D x_{y''}, \ldots, x_{d''} <_D x_{a''}\]

The contradiction \(x_{a''} <_D x_{a''}\) completes the proof.

Theorem 4.7 shows that if we have a cycle in the conflict graph \(G(h)\) of length \(n\), where \(n > 2\), then there is a cycle of length two in \(G(h)\). This result is the basis of the serializability condition for these kinds of transactions, given in Theorem 4.8, below.

Figure 4.7: Cycle in \(G(h)\) of length \(n\)
4.4 The serializability condition

Theorem 4.8 gives a condition for testing for cycles between two transactions in a history and hence, by Theorem 3.4 and Theorem 4.7, a test for serializability. The condition can easily be expressed in either of the temporal logics CTL or LTL.

**Theorem 4.8.** A history \( h \) of transactions \( T = \{ T_i : i \in N_1 \} \) is serializable iff for any two transactions \( T_i, T_i' \in T \) \((i, i' \geq 1, i \neq i')\) one of them, \( T_i \) say, is such that, for all \( x \in D_i \cap D_{i'} \), \( w_i(x) <_h r_{i'}(x) \).

**Proof.** Only if part
Assume that \( h \) is a history where the condition is not satisfied. One possibility is that there exists a data item \( x_1 \) such that \( r_{i'}(x_1) <_h w_i(x_1) \), giving an arc \((T_{i'}, T_i)\) in \( G(h) \), and another different data item \( x_2 \) such that \( w_i(x_2) <_h r_{i'}(x_2) \), giving an arc \((T_i, T_{i'})\) in \( G(h) \) thereby completing a cycle. Hence, in this case, by Theorem 3.4, \( h \) is not serializable. The other possibility is that the condition is breached on a single data item, i.e. one of the following cases:

\[
\ldots r_i(x) \ldots r_{i'}(x) \ldots w_i(x) \ldots w_{i'}(x) \ldots \quad (4.11)
\]
\[
\ldots r_{i'}(x) \ldots r_i(x) \ldots w_i(x) \ldots w_{i'}(x) \ldots \quad (4.12)
\]
\[
\ldots r_i(x) \ldots r_{i'}(x) \ldots w_{i'}(x) \ldots w_i(x) \ldots \quad (4.13)
\]
\[
\ldots r_{i'}(x) \ldots r_i(x) \ldots w_{i'}(x) \ldots w_i(x) \ldots \quad (4.14)
\]

In cases (4.11)-(4.14), it is clear that the conflict graphs of the histories are cyclic and, by Theorem 3.4, not serializable.

If part
Assume that the history \( h \) is not serializable. We show that the condition does not hold. To say that \( h \) is not serializable means, by Theorem 3.4, that there exists a cycle in \( G(h) \). From Theorem 4.7, \( G(h) \) has a cycle \((T_i, T_{i'}), (T_{i'}, T_i)\), where \( i, i' \geq 1 \)
and $i \neq i'$. Then, $h$ is one of the following forms:

\begin{align*}
\cdots r_i(x) \cdots r_i'(x) \cdots w_i(x) \cdots w_i'(x) \cdots & \quad (4.15) \\
\cdots r_i(x) \cdots r_i'(x) \cdots w_i(x) \cdots w_i'(x) \cdots & \quad (4.16) \\
\cdots r_i'(x) \cdots r_i(x) \cdots w_i'(x) \cdots w_i(x) \cdots & \quad (4.17) \\
\cdots r_i'(x) \cdots r_i(x) \cdots w_i'(x) \cdots w_i(x) \cdots & \quad (4.18) \\
\cdots r_i(x) \cdots w_i'(x) \cdots r_i'(y) \cdots w_i(y) \cdots & \quad (4.19) \\
\cdots r_i'(x) \cdots w_i'(x) \cdots r_i(y) \cdots w_i(y) \cdots & \quad (4.20)
\end{align*}

In (4.15)-(4.20) steps are underlined if they cause the condition to be breached. Cases (4.15)-(4.18) are when one data item $x$ causes a cycle, and cases (4.19) and (4.20) are when 2 data items $x$ and $y$ cause a cycle.

To explain how the serializability condition of Theorem 4.8 is used to verify whether a history $h$ is serializable or not, we shall give the following example:

Let $T_1, T_2, T_3$ and $T_4$ be multi-step transactions as follows

$T_1 = r_1(x_2)w_1(x_2)r_1(x_3)w_1(x_3)r_1(x_4)w_1(x_4)$

$T_2 = r_2(x_2)w_2(x_2)r_2(x_2)w_2(x_2)$

$T_3 = r_3(x_2)w_3(x_2)r_3(x_3)w_3(x_3)r_3(x_4)w_3(x_4)r_3(x_5)w_3(x_5)$

$T_4 = r_4(x_2)w_4(x_2)r_4(x_2)w_4(x_2)$

Let $D$ be the set of all data items as follows

$D = \{x_1, x_2, x_3, x_4, x_5\}$

Also, let $h_{T'}$ be the history of $T' = \{T_i : 1 \leq i \leq 4\}$, where $T' \subseteq T$,

$h_{T'} = r_1(x_2)r_2(x_1)w_1(x_2)r_2(x_2)r_3(x_2)w_3(x_2)w_2(x_2)r_4(x_2)w_4(x_2)r_1(x_3)$

$w_1(x_3)r_3(x_3)w_3(x_3)r_4(x_3)w_4(x_3)r_1(x_4)w_1(x_4)r_3(x_4)w_3(x_4)w_2(x_4)r_3(x_5)w_3(x_5)$

Firstly, we shall chop the history $h_{T'}$ up into sets of histories each containing two different transactions. Then, we shall check whether the serializability condition is satisfied for each set, to see if the history $h_{T'}$ is serializable. If $h_{T'}$ is not serializable, the main history $h$ will not be serializable. Consider:

$h_{\{T_1, T_2\}} = r_1(x_2)r_2(x_1)w_1(x_2)r_2(x_2)r_1(x_3)w_1(x_3)r_1(x_4)w_1(x_4)$

We notice, from $h_{\{T_1, T_3\}}$, that $D_1 \cap D_2 = \{x_2\}$. According to the serializability condition in Theorem 4.8, if $h$ is serializable then, we should have either, for all
Chapter 4. Multi-step Transactions Accessing Ordered Sets Of Data Items  

For all \( x \in D_1 \cap D_2, w_1(x) <_h r_2(x) \) or, for all \( x \in D_1 \cap D_2, w_2(x) <_h r_1(x) \). From \( h_{(T_1, T_2)} \), we have \( w_1(x_2) <_h r_2(x_2) \). This means that \( T_1 \) and \( T_2 \) satisfy the condition. Next, consider:

\[
h_{(T_1, T_3)} = r_1(x_2)w_1(x_2)r_3(x_2)w_3(x_2)r_1(x_3)w_1(x_3)r_3(x_3)w_3(x_3)r_1(x_4)w_1(x_4)r_3(x_4)w_3(x_4).
\]

From \( h_{(T_1, T_3)} \), we notice that \( D_1 \cap D_3 = \{x_2, x_3, x_4\} \). Also, we have for all \( x \in D_1 \cap D_3, w_1(x) <_h r_3(x) \). This means that \( T_1 \) and \( T_3 \) satisfy the condition. Next, consider:

\[
h_{(T_1, T_4)} = r_1(x_2)w_1(x_2)r_4(x_2)w_4(x_2)r_1(x_3)w_1(x_3)r_4(x_3)w_4(x_3)r_1(x_4)w_1(x_4).
\]

In \( h_{(T_1, T_4)} \), we have, for all \( x \in D_1 \cap D_4 = \{x_2, x_3\}, w_1(x) <_h r_4(x) \). This means that \( T_1 \) and \( T_4 \) satisfy the condition. Next, consider:

\[
h_{(T_2, T_3)} = r_2(x_1)w_2(x_1)r_3(x_2)w_3(x_2)r_3(x_3)w_3(x_3)r_3(x_4)w_3(x_4).
\]

In \( h_{(T_2, T_3)} \), we have \( r_2(x_2) <_h w_3(x_2) \) and \( r_3(x_2) <_h w_2(x_2) \). This breaches the condition and therefore history \( h \) is not serializable; see Figure 4.9. As \( h \) is not serializable, there is no point in checking the serializability condition for the remaining \( h_{(T_2, T_1)} \) and \( h_{(T_3, T_4)} \).

\[
h_{(T_2, T_1)} = r_2(x_1)w_2(x_1)r_2(x_2)w_2(x_2)r_4(x_2)w_4(x_2)r_4(x_3)w_4(x_3).
\]

\[
h_{(T_3, T_4)} = r_3(x_2)w_3(x_2)r_4(x_2)w_4(x_2)r_3(x_3)w_3(x_3)r_4(x_3)w_4(x_3)r_3(x_4)w_3(x_4).
\]

\[
r_3(x_5)w_3(x_5).
\]

Figure 4.9: \( G(h_{TV}) \) is a subgraph of \( G(h) \).
4.5 Applications

Recently, internet use has become widespread in business. This has caused travel agencies to invest in e-tickets. Ticket booking from the web (or e-ticketing for events, amusements, bus or flight tickets) is now one of the widely available services in E-commerce. Customers can access the database and book a ticket at any time in any location. Booking e-tickets from travel agencies involves executing multi-step transactions. The scenario of booking tickets is interesting in that the list of destinations are naturally ordered. It involves browsing the list of destinations, then checking the availability of seats, and booking one or more of them consecutively. For example, assume a passenger intends to book a ticket from location A to E. Firstly, they have to browse the list of available destinations from A. Secondly, choose the itinerary (the set of destinations may transit during the journey). Finally, book the itinerary. This scenario can be implemented as a multi-step transaction accessing ordered data, where a read step corresponds to browsing journey times from a destination, and the write step represents the booking phase of a chosen time to the next destination in the order. Assume that $D$ represents the set of available destinations in the travel agency starting from A and ending at E, and $x_i \in D$ represents the next leg of the journey from $i$, such that $x_i \in D$. The ordered set of data items is depicted in Figure 4.10. The widespread use of e-tickets makes the number of incoming and outgoing transactions unknown even though, at any point in time, the number of active transactions in the web server is finite.

$$A \rightarrow B \rightarrow C \rightarrow \cdots \rightarrow x_i \rightarrow E$$

Figure 4.10: Ordered set
Chapter 5

A Timestamp-based Protocol For Multi-step Transactions Accessing Ordered Sets Of Data Items

5.1 Introduction

In the previous chapter, we introduced the scenario of multi-step transactions accessing an ordered set of data items $D$. Each transaction accessed a subset of data items of a set of $m$ data items $D$ and each such subset accessed data items in the same order as they occurred in $D$. We proved a serializability condition, see Theorem 4.8, that represented the correctness criterion to decide whether an infinite history, composed of such transactions, is serializable. In this chapter, we shall define a protocol, based on timestamps, as a concurrency control criterion suitable for such transactions. We shall use both LTL and CTL to encode the specifications for the protocol and the serializability condition. We shall prove that the histories produced by the timestamp-based protocol are serializable, using the NuSMV model checker. After that, we shall compare LTL and CTL for their suitability for these kinds of problems.

5.2 A timestamp-based protocol

Assume that we have an ordered set of data items $D_i$, where $D_i \subseteq D$, accessed by transaction $T_i \in T$ such that $D_i = \{x_{i}, x_{i+1}, \ldots, x_p\}$. At any point in time, let $F(T_i)$ equal the first data item in $D_i$ that is still to be accessed (the value of $F(T_i)$ will keep changing during the execution time). For any transaction $T_i$ and data item $x_a$, we shall denote by $TS(T_i, x_a)$ the timestamp when $T_i$ accesses the data item $x_a$ (for the read operation). We assume that every timestamp value is
unique and accurately represents an instant in time. No two timestamps can be the same. A higher-valued timestamp occurs later in time than a lower-valued timestamp. Initially, for all \( x_a \in D_i \) and \( T_j \in T \), \( TS(T_i, x_a) = 0 \). Thereafter, when \( T_i \) accesses the data item \( x_a \), \( TS(T_i, x_a) = \text{System TimeStamp} \). Then, the value of \( TS(T_i, x_a) \) remains unchanged until the last operation in the transaction \( T_i \) (\( w_i(x_p) \)) has executed. Finally, when \( w_i(x_p) \) has executed, the value of \( TS(T_i, x_a) \) is reset to zero. Formally, we define \( TS(T_i, x_a) \) as follows:

\[
TS(T_i, x_a) = \begin{cases} 
0, & x_a \notin D_i; \\
0, & \text{when } T_i \text{ has executed } w_i(x_p); \\
\text{STS}, & \text{when } T_i \text{ accesses } x_a; \\
TS(T_i, x_a), & \text{if } T_i \text{ has not executed } w_i(x_p) \text{ yet}; 
\end{cases}
\]

where STS denotes the System TimeStamp. In order to execute read and write operations of transaction \( T_i \) on data item \( x_a \), we shall compare \( TS(T_i, x_{a-1}) \) with the remaining transactions (on the same data item \( x_{a-1} \)). If \( TS(T_i, x_{a-1}) \) has the minimum positive time stamp among all transactions that have executed \( x_{a-1} \) and are waiting to execute \( x_a \), then \( T_i \) can access \( x_a \). Otherwise, the transaction should suspend until it satisfies the condition (as we will see in the next subsection).

### 5.2.1 Accessing rules

1. There is no transaction has read \( x_a \) and has not written to \( x_a \) iff, for all \( T_j \in T \),
   
   (a) \( TS(T_j, x_a) = 0 \) or
   
   (b) \( TS(T_j, x_a) \neq 0 \) and \( w_j(x_a) \) has been executed.

2. \( T_i \) may access the first data item \( x_l \) in \( D_i \), if \( x_l \) satisfies rule 1.

3. \( T_i \) may access a data item \( x_l \), \( 2 \leq l \leq p \), if \( F(T_i) = x_l \). Otherwise, the transaction \( T_i \) will be suspended.

4. \( F(T_i) = x_l \) iff
   
   (a) \( x_l \) satisfies rule 1
   
   (b) \( TS(T_i, x_{l-1}) = \min\{TS(T_j, x_{l-1}) : 1 \leq j \leq n \text{ and } TS(T_j, x_{l-1}) > 0 \text{ and } TS(T_j, x_l) = 0\} \)
   
   (c) \( x_l \in D_i \)
For example, assume that we have 4 ordered sets of data items:

\[ D_1 = \{x_2, x_3, x_4\} \]
\[ D_2 = \{x_1, x_2\} \]
\[ D_3 = \{x_2, x_3, x_4, x_5\} \]
\[ D_4 = \{x_2, x_3\} \]

accessed by their corresponding transactions \( T_1, T_2, T_3 \) and \( T_4 \) respectively, so that the transactions are as follows:

\[ T_1 = r_1(x_2)w_1(x_2)r_1(x_3)w_1(x_3)r_1(x_4)w_1(x_4) \]
\[ T_2 = r_2(x_1)w_2(x_1)r_2(x_2)w_2(x_2) \]
\[ T_3 = r_3(x_2)w_3(x_2)r_3(x_3)w_3(x_3)r_3(x_4)w_3(x_4)r_3(x_5)w_3(x_5) \]
\[ T_4 = r_4(x_2)w_4(x_2)r_4(x_3)w_4(x_3) \]

and suppose that \( T_1 \) precedes \( T_2 \) and \( T_2 \) precedes \( T_3 \) and \( T_3 \) precedes \( T_4 \) in arriving at scheduler \( S \). Also, suppose that \( S \) makes use of the above protocol to schedule the incoming transactions. The following matrix represents the Time Stamp Matrix (TSM) for all transactions in \( T \) versus all data items in \( D \):

\[
TS(T, D) = \begin{pmatrix}
TS(T_1, x_1) & TS(T_1, x_2) & \cdots & TS(T_1, x_m) \\
TS(T_2, x_1) & TS(T_2, x_2) & \cdots & TS(T_2, x_m) \\
& \ddots & \ddots & \ddots \\
TS(T_n, x_1) & TS(T_n, x_2) & \cdots & TS(T_n, x_m)
\end{pmatrix}
\]

The entry that lies in the \( i^{th} \) row and the \( j^{th} \) column of the matrix (TMS) is typically referred to as \( TS(T_i, x_j) \), and it represents the value of the timestamp for transaction \( T_i \) when it has accessed \( x_j \). The matrix entries keep changing during the execution time. This change depends on the transactions nature (number of consecutive data items they access) and number of active transactions in any point in time. Initially, TSM will be as follows

\[
TS(T, D) = \begin{pmatrix}
0 & 0 & \cdots & 0 \\
0 & 0 & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
0 & 0 & \cdots & 0
\end{pmatrix}
\]
Now, assume that the schedule (history) \( h \) and TSM at some point in time are such that

\[
h = r_1(x_2)r_2(x_1)w_2(x_1), \quad TS(T, D) = \begin{pmatrix} 0 & 1 & ... & 0 \\ 2 & 0 & ... & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & ... & 0 \end{pmatrix}
\] (5.1)

and that the transaction \( T_3 \) tries to access its first data item \( x_2 \). Then, \( T_3 \) can not access the data item \( x_2 \) because it has been read by \( T_1 \) but not written yet, as by accessing rule 1. Thus, we can know whether a data item \( x_a \) has been accessed by any transaction \( T_j \in T \), and also which transactions, by applying accessing rule 1. Subsequently, assume that the history \( h \) is such that

\[
h = r_1(x_2)r_2(x_1)w_2(x_1)w_1(x_2),
\]

the TSM is as in (5.1) and transactions \( T_2 \) and \( T_3 \) try to access data item \( x_2 \). As \( x_2 \) is the first data item in the set \( D_3 \), we apply accessing rule 2. \( T_3 \) will find \( x_2 \) satisfies rule 1. So, \( T_3 \) can access \( x_2 \). Also, \( T_2 \) will find \( x_2 \) satisfies accessing rule 1 and rule 4 and can be accessed by \( T_2 \) itself. Consequently, whichever one of them \( (T_2 \text{ or } T_3) \) comes to the scheduler \( S \) first, can access \( x_2 \) immediately. Now, assume that the history \( h \) and TSM, at any point in time, are such that:

\[
h = r_1(x_2)r_2(x_1)w_2(x_1)w_1(x_2)r_2(x_2)w_2(x_2)r_3(x_2)w_3(x_2)r_4(x_2)w_4(x_2)
\]

\[
TS(T, D) = \begin{pmatrix} 0 & 1 & 0 & ... & 0 \\ 0 & 0 & 0 & ... & 0 \\ 0 & 4 & 0 & ... & 0 \\ 0 & 5 & 0 & ... & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & ... & 0 \end{pmatrix}
\]

and transactions \( T_1, T_3 \) and \( T_4 \) are contending to access \( x_3 \). According to accessing rule 3, any one of them that satisfies the conditions of accessing rule 4 can access \( x_3 \) (as \( x_3 \) is not the first data item in \( T_1, T_3 \) and \( T_4 \)). Hence, all of them satisfy conditions (a) and (c) of rule 4. But, only \( T_1 \) satisfies also condition (b) because it has the minimum timestamp of \( x_2 \). This means that \( T_1 \) accessed \( x_2 \) first, and it should also access \( x_3 \) first. It can be easily seen that for transaction \( T_2 \), timestamps \( TS(T_2, x_1) \) and \( TS(T_2, x_2) \) are reset to zero. This occurs when any transaction finishes its execution on all its data items.
5.3 Linear temporal logic specifications

In this section, we present LTL as a logic that can be used to specify and verify infinite histories composed of \( n \) transactions each accessing contiguous subsets of \( m \) ordered data items, and repeating infinitely often. The aggregate of all the repetitions of the \( n \) transactions gives an infinite number of transactions \( T = \{ T_i : i \in \mathbb{N}_1 \} \). Infinite histories are produced by executing the accessing protocol of Section 5.2, on an unlimited number of these kinds of transactions. The reason for using LTL as a specification language in this context, is that LTL formulae are interpreted over both finite and infinite sequence of states [1], as we will see in the next sections, which is useful for the histories that are produced from executing the accessing protocol. Also, LTL is used for specifying general reactive and concurrent systems [4, 3]. We will encode the specifications of the accessing protocol, which is timestamp-based, and the serializability condition into LTL. Then, we will build a model using the NuSMV model checker to check automatically if the specification of the protocol implies the specification of the serializability condition. This will correspond to verifying whether the histories produced by the protocol are serializable or not.

5.3.1 Syntax of LTL

The alphabet of LTL consists of a set of propositions symbols \( p_0, p_1, \ldots \), distinguished read/write step propositional symbols \( r_i(x_j), w_i(x_j) \), where \( i \geq 1 \) and \( j \geq 1 \), booleans \( \neg, \lor, \land, \top, \bot \), and temporal operators \( X, F, G, U, O, Y, H, \) and \( S \). Formulae in LTL are those generated by:

\[
\phi ::= p_i \mid r_i(x_j) \mid w_i(x_j) \mid \neg \phi \mid \phi_1 \lor \phi_2 \mid \phi_1 \land \phi_2 \mid X\phi \mid F\phi \mid G\phi \mid \phi_1 U \phi_2 \mid O\phi \mid H\phi \mid Y\phi \mid \phi_1 S \phi_2
\]

The symbols \( \bot \) and \( \top \) will also be used to denote the truth values false and true respectively and the abbreviations \( \Rightarrow \) and \( \Leftrightarrow \) will have their usual logical meaning.

5.3.2 Semantics of LTL

An interpretation for LTL, \( I(s_a) \), at a given state \( s_a \in S \), where \( S \) is a set of states, assigns truth values \( p_i^{I(s_a)}, r_i(x_j)^{I(s_a)} \) and \( w_i(x_j)^{I(s_a)} \in \{ \bot, \top \} \) to propositional symbol \( p_i \), \( r_i(x_j) \) and \( w_i(x_j) \), respectively. A Kripke structure \( M \), as we defined in chapter 3, is a triple \( \langle S, R, I \rangle \), where \( S \) is a set of states, \( R \subseteq S \times S \) a transition relation such that, for all \( s \in S \), there exists \( s' \in S \) with \( (s, s') \in R \). A path in \( M \) is an infinite sequence of states, \( \pi = s_a, s_{a+1}, \ldots \), such that, for every
b ≥ a, (s_b, s_{b+1}) ∈ R. We use π^a to denote the suffix of π starting at s_a. As each state in a Kripke structure is required to have at least one successor, it follows that π^a ≠ {} for any state s_a. The semantics of a LTL formula φ is given by the truth relation $M, s_a ⊨ φ$ which means that φ holds at state s_a in the Kripke structure M. Similarly, if φ is a path formula, $M, π ⊨ φ$ means that φ holds along path π in the Kripke structure M. The relation ⊨ is defined inductively as follows:

$M, s_a ⊨ p_i$ iff $p_i^{(s_a)} = T$

$M, s_a ⊨ r_i(x_j)$ iff $r_i(x_j)^{l(s_a)} = T$

$M, s_a ⊨ w_i(x_j)$ iff $w_i(x_j)^{l(s_a)} = T$

$M, s_a ⊨ \neg \phi$ iff $M, s_a \not\models \phi$

$M, s_a ⊨ \phi_1 \lor \phi_2$ iff $M, s_a \models \phi_1$ or $M, s_a \models \phi_2$

$M, s_a ⊨ \phi_1 \land \phi_2$ iff $M, s_a \models \phi_1$ and $M, s_a \models \phi_2$

$M, s_a \models X\phi$ iff $M, s_{a+1} \models \phi$

$M, s_a \models F\phi$ iff there exists k ≥ a such that $M, s_k \models \phi$

$M, s_a \models G\phi$ iff for all k ≥ a such that $M, s_k \models \phi$

$M, s_a \models \phi_1 U \phi_2$ iff there exists c ≥ a, $M, s_c \models \phi_2$ and, for all a ≤ b < c, $M, s_b \models \phi_1$

Moreover, LTL can have operators expressing properties over the past evolution of the states. These past-time temporal operators allow properties of the path that lead to the current situation to be expressed. It is well-known that temporal logics combining past and future modalities make some specifications easier to write and more natural [6] and can be used together to describe complex properties of the systems. Also, past-time LTL can sometimes express temporal properties more succinctly than future-time LTL [5]. The semantics of past-time temporal operators as follows:

$M, s_a \models O\phi$ iff there exists k ≤ a such that $M, s_k \models \phi$

$M, s_a \models H\phi$ iff, for all k ≤ a, $M, s_k \models \phi$

$M, s_a \models Y\phi$ iff $M, s_{a-1} \models \phi$

$M, s_a \models \phi_1 S \phi_2$ iff there exists c < a, such that $M, s_c \models \phi_2$ and, for all c < b ≤ a, $M, s_b \models \phi_1$. 

\( \text{Chapter 5. A Timestamp-based Protocol For Multi-step Transactions} \) \( \text{62} \)
Usually, past-time operators do not add expressive power. This means that any LTL formula with past-time operators can be rewritten by only using future-time operators [7]. But, these past-time operators are very useful in keeping specifications simple and easy to understand [7].

5.4 Properties of read and write propositions

Assume that we have a Kripke structure $M$ and that the following properties, relating to $r_i(x_j)$ and $w_i(x_j)$ propositions, hold in $M$:

(P1) Read/write alternation
A transaction $T_i$ cannot have read two distinct data items (in $D_i$) without having written to one of them, i.e. if $x_j <_D x_j'$, $r_i(x_j')$ cannot be executed until $w_i(x_j)$ has been executed.

(P2) Write implies read
A transaction $T_i$ can only have written to $x_j$ if it has read $x_j$, i.e. if $w_i(x_j)$ executes, then $r_i(x_j)$ must have executed before.

(P3) Read/write step proposition remains true until the next operation, belonging to the same transaction, becomes true
If a read/write step has taken place, the corresponding proposition remains true until the next operation in $T_i$ becomes true, i.e. $r_i(x_j)/w_i(x_j)$ is true, remains true until the next step $w_i(x_j)/r_i(x_j)$, where $x_j <_D x_j'$, becomes true.

(P4) At most one step occurs at each successive state
No two distinct steps can both be false in a state, and then both true in a next state.

(P5) A transaction $T_i$ accesses each data item $x \in D_i$ exactly once for both read and write operations
For all $x \in D_i$, a transaction $T_i$ can have exactly one read operation ($r_i(x)$) and exactly one write operation ($w_i(x)$) for the data item $x$.

The semantics of formula $\phi$ is now given by a truth relation $M, s_a \models \phi$, where $M$ is a structure for LTL satisfying the additional properties (P1)-(P4). Given a state $s_a$ and a path $\pi$, there corresponds a sequence of read and write step propositions that become true in $s_a, s_{a+1}, \ldots$. In this way, $\pi$ yields a history of the transactions $\{T_1, T_2, \ldots\}$ produced by the protocol and starting their execution at $s_a$. We illustrate this correspondence between paths and histories as follows:
In the depiction above, we have

\[ D_1 = \{x_1, x_2, x_3\} \]
\[ D_2 = \{x_2, x_3, x_4\} \]
\[ D_3 = \{x_1, x_2, x_3, x_4, x_5\} \]

and the corresponding transactions as follows

\[ T_1 = r_1(x_1)w_1(x_1)r_1(x_2)w_1(x_2)r_1(x_3)w_1(x_3) \]
\[ T_2 = r_2(x_2)w_2(x_2)r_2(x_3)w_2(x_3)r_2(x_4)w_2(x_4) \]
\[ T_3 = r_3(x_1)w_3(x_1)r_3(x_2)w_3(x_2)r_3(x_3)w_3(x_3)r_3(x_4)w_3(x_4)r_3(x_5)w_3(x_5) \]

The read and write propositions that are given for each successive state represent the propositions that are true in those states. The top of each column displays the unique proposition that becomes true in the particular state. This represents the read and write operations that has been scheduled by the protocol. In order
to make it easier to follow the structure which has a large number of propositions which are false, only the values of propositions that are true in the states of the trace are shown. The corresponding history $h$ is:

$$h = r_1(x_1)r_2(x_2)w_1(x_1)w_2(x_2)r_3(x_1)w_3(x_1)r_1(x_2)w_1(x_2)r_3(x_2)w_3(x_2)r_2(x_3)w_2(x_3)$$
$$r_1(x_3)w_1(x_3)r_3(x_3)w_3(x_3)r_2(x_4)w_2(x_4)r_3(x_4)w_3(x_4)r_3(x_5)w_3(x_5).$$

We make use of additional propositions $\text{begin}_i$ and $\text{end}_i$ to refer to the begin and the end of each transaction. Notice that the interpretation of the read and write propositions is different from the interpretation defined in Chapter 3. The main difference is the read/write step proposition remains true until the next operation, belonging to the same transaction, becomes true. The reason for this is to make the specifications more concise. In the next section, we will illustrate this.

### 5.5 Encoding the accessing protocol and serializability condition into LTL

Firstly, we encode, using past-time and future-time temporal operators, the properties $(P1)-(P5)$ of the read and write propositions in the LTL structures (as in Section 5.4) as $\sigma_0$, $\sigma_1$, $\sigma_2$, $\sigma_3$ and $\sigma_4$ respectively, as follows

(P1) Read/write alternation

A transaction $T_i$ cannot have read two distinct data items (in $D_i$) without having written to one of them, i.e. if $x <_D y$, $r_i(y)$ cannot be executed until $w_i(x)$ has been executed.

$$\sigma_0 = \bigwedge_{i \geq 1} \bigwedge_{x,y \in D_i, x <_D y} G[(r_i(y) \land O(r_i(x))) \Rightarrow O(w_i(x))]$$

or

$$\sigma_0 = \bigwedge_{i \geq 1} \bigwedge_{x,y \in D_i, x <_D y} G[(r_i(x) \Rightarrow F(w_i(x) \land F(r_i(y)))]$$

(P2) Write implies read

A transaction $T_i$ can only have written to $x$ if it has read $x$, i.e. if $w_i(x)$ executes, then $r_i(x)$ must have executed before.

$$\sigma_1 = \bigwedge_{i \geq 1} \bigwedge_{x \in D_i} G[w_i(x) \Rightarrow O(r_i(x))]$$

or

$$\sigma_1 = \bigwedge_{i \geq 1} \bigwedge_{x \in D_i} G[r_i(x) \Rightarrow F(w_i(x))]$$
(P3) Read/write step proposition remains true until the next operation, belonging to the same transaction, becomes true.

If a read/write step has taken place, the corresponding proposition remains true until the next operation in $T_i$ becomes true, i.e. if $r_i(x)/w_i(x)$ is true, it remains true until the next step $w_i(x)/r_i(y)$, where $x <_D y$, becomes true.

\[
\sigma_2 = \bigwedge_{i \geq 1} \bigwedge_{x \in D_i} G[w_i(x) \Rightarrow O(r_i(x) \cup w_i(x))] \land \\
\bigwedge_{i \geq 1} \bigwedge_{x,y \in D_i, x <_D y} G[(r_i(y) \Rightarrow O(w_i(x) \cup r_i(y))]
\]
or

\[
\sigma_2 = \bigwedge_{i \geq 1} \bigwedge_{x \in D_i} G[w_i(x) \Rightarrow \neg r_i(x)] \land \\
\bigwedge_{i \geq 1} \bigwedge_{x,y \in D_i, x <_D y} G[r_i(y) \Rightarrow \neg w_i(x)]
\]

(P4) At most one step occurs at each successive state.

No two distinct steps can both be false in a state, and then both true in a next state.

\[
\sigma_3 = \bigwedge_{i,i', \geq 1} \bigwedge_{1 \leq j, j' \leq m} G[-((-r_i(x_j) \land \neg r_{i'}(x_{j'})) \land X(r_i(x_j) \land r_{i'}(x_{j'}))) \\
\land -((-r_i(x_j) \land \neg w_{i'}(x_{j'})) \land X(r_i(x_j) \land w_{i'}(x_{j'}))) \\
\land -((-w_i(x_j) \land \neg w_{i'}(x_{j'})) \land X(w_i(x_j) \land w_{i'}(x_{j'})))]
\]
or

\[
\sigma_3 = \bigwedge_{i,i', \geq 1} \bigwedge_{1 \leq j, j' \leq m} G[-((r_i(x_j) \land r_{i'}(x_{j'})) \land Y(-r_i(x_j) \land \neg r_{i'}(x_{j'}))) \\
\land -((r_i(x_j) \land w_{i'}(x_{j'})) \land Y(r_i(x_j) \land \neg w_{i'}(x_{j'}))) \\
\land -((w_i(x_j) \land w_{i'}(x_{j'})) \land Y(-w_i(x_j) \land \neg w_{i'}(x_{j'})))]
\]

(P5) A transaction $T_i$ accesses each data item $x \in D_i$ exactly once for both read and write operations.

For all $x \in D_i$, a transaction $T_i$ can have exactly one read operation ($r_i(x)$) and exactly one write operation ($w_i(x)$) for the data item $x$, i.e. we can not have $T_i$
such that
\[ T_i = r_i(x_1)w_i(x_1) \ldots r_i(x_l)w_i(x_l) \]
or
\[ T_i = r_i(x_1)w_i(x_1) \ldots w_i(x_{l})w_i(x_{l}) \]
where \( D_i = \{ x_1, x_2, \ldots, x_p \} \). This is defined in \( \sigma_5 \) as follows

\[
\sigma_4 = \left( \bigwedge_{i \geq 1} \bigwedge_{x \in D_i} \neg[r_i(x) \land F(\neg r_i(x) \land F r_i(x))]) \right) \land \\
\left( \bigwedge_{i \geq 1} \bigwedge_{x \in D_i} \neg[w_i(x) \land F(\neg w_i(x) \land F w_i(x))] \right)
\]
or

\[
\sigma_4 = \left( \bigwedge_{i \geq 1} \bigwedge_{x \in D_i} \neg[r_i(x) \land O(\neg r_i(x) \land O r_i(x))]) \right) \land \\
\left( \bigwedge_{i \geq 1} \bigwedge_{x \in D_i} \neg[w_i(x) \land O(\neg w_i(x) \land O w_i(x))] \right)
\]

Next, we encode the serializability condition of Theorem 4.8. This is defined in terms of \( \sigma_5 \) and \( \sigma_6 \):

\[
\sigma_5 = \bigwedge_{i,i' \geq 1, i \neq i'} \bigwedge_{x,y \in D_i \cap D_{i'}, x < y} \neg[(w_i(y) \land O(w_{i'}(y) \land O(w_{i'}(x) \land O(w_i(x)))) \lor \\
(w_{i'}(y) \land O(w_i(y) \land O(w_i(x) \land O(w_{i'}(x))))])
\]
or

\[
\sigma_5 = \bigwedge_{i,i' \geq 1, i \neq i'} \bigwedge_{x,y \in D_i \cap D_{i'}, x < y} \neg[(w_i(x) \land F(w_{i'}(x) \land F(w_{i'}(y) \land F(w_i(y)))) \lor \\
(w_{i'}(x) \land F(w_i(x) \land F(w_i(y) \land F(w_{i'}(y))))])
\]

\[
\sigma_6 = \bigwedge_{i,i' \geq 1, i \neq i'} \bigwedge_{x \in D_i \cap D_{i'}} \neg[r_i(x) \land r_{i'}(x)]
\]

The serializability condition of Theorem 4.8 says if we choose any two transactions \( (T_i, T_{i'}) \) participating in a history \( h\{(x_i, x_{i'})\} \), where \( D_i \cap D_{i'} = \{ x_1, \ldots, x_p \} \), of the form

\[
h\{(x_i, x_{i'})\} = \ldots r_i(x_1) \ldots w_i(x_l) \ldots r_{i'}(x_{l'}) \ldots w_{i'}(x_1) \ldots r_i(x_p) \ldots w_i(x_p) \ldots r_{i'}(x_p) \ldots w_{i'}(x_{l'}) \ldots
\]
then the history $h$ is serializable. In $\sigma_6$, we encode that if a transaction $T_i$ begins executing a read operation on data item $x$, no read operation on data item $x$ by any other transactions occurs (executes) until the write operation of $T_i$ completes its execution on data item $x$, i.e.:

$$\ldots r_i(x) \ldots w_i(x) \ldots$$

Therefore, if we avoid the situation above, there is no cycle between $T_i$ and any other transaction $T_{i'}$ on the same data item; see Figure 5.1(a). Now, the serializability condition, of Theorem 4.8, is to hold for each $x \in D_i \cap D_{i'}$. It is possible to make a cycle in $G(h)$ on different data items, i.e. if $T_i$ precedes $T_{i'}$ in accessing data item $x$ and $T_{i'}$ precedes $T_i$ in accessing data item $y$, or $T_i$ precedes $T_{i'}$ in accessing data item $x$ and $T_{i'}$ precedes $T_i$ in accessing data item $y$. Therefore, we encode in $\sigma_5$ that these two situations do not occur. The formulae $\sigma_5$ and $\sigma_6$ together represent the encoding of the serializability condition into LTL denoted by $\sigma_7$:

$$\sigma_7 = \sigma_5 \land \sigma_6$$

Figure 5.1: Cycle of length two

Alternatively, assume that the read/write step propositions were interpreted as remaining true until the end of the transaction (as in the Chapter 3 interpretation). This would be depicted as follows:

$$\square \quad \square \quad \square$$

Then, the encoding of the condition of $\sigma_6$ into LTL, which states that if a transaction $T_i$ begins executing a read operation on data item $x$ no read operation on
Chapter 5. A Timestamp-based Protocol For Multi-step Transactions

data item $x$ by any other transactions occurs (executes) until the write operation of $T_i$ completes its execution on data item $x$, would be as follows:

$$\sigma_6 = \bigwedge_{i', \exists i \neq i'} \bigwedge_{x \in D_i \cap D_{i'}} \mathbf{G}[-r_i(x) \land r_{i'}(x) \land (-w_i(x) \lor -w_{i'}(x))]$$

Comparing $\sigma_6'$ with $\sigma_6$, we find that $\sigma_6'$ is a longer formula. As mentioned in Chapter 3, the time complexity of verifying LTL formula is $O(|S| + |R|).2^{|f|}$, where $|S|$ is the number of states in the finite state machine, $|R|$ the number of transitions, and $|f|$ is the length of the specification formula. This means that when the length of the specification formula $|f|$ increases, the time complexity of verifying LTL formula increases exponentially. This is the reason for changing the interpretation of the read and write propositions here.

Next, we encode the accessing protocol, which defines in section 5.2, in $\sigma_{10}$ as follows:

$$\sigma_8 = \bigwedge_{i,j,k \geq 1 \atop i \neq j \neq k \neq i} \mathbf{G}[(\bigwedge_{x \in D_i \cap D_j \cap D_k} (w_k(x) \land \mathbf{O}(w_j(x) \land \mathbf{O}(w_i(x)))) \lor \\
(\bigwedge_{x \in D_i \cap D_j \cap D_k} (w_j(x) \land \mathbf{O}(w_k(x) \land \mathbf{O}(w_i(x)))) \lor \\
(\bigwedge_{x \in D_i \cap D_j \cap D_k} (w_i(x) \land \mathbf{O}(w_k(x) \land \mathbf{O}(w_j(x)))))]$$

or

$$\sigma_8 = \bigwedge_{i,j,k \geq 1 \atop i \neq j \neq k \neq i} \mathbf{G}[(\bigwedge_{x \in D_i \cap D_j \cap D_k} (w_i(x) \land \mathbf{F}(w_j(x) \land \mathbf{F}(w_k(x)))) \lor \\
(\bigwedge_{x \in D_i \cap D_j \cap D_k} (w_j(x) \land \mathbf{F}(w_k(x) \land \mathbf{F}(w_j(x)))) \lor \\
(\bigwedge_{x \in D_i \cap D_j \cap D_k} (w_k(x) \land \mathbf{F}(w_i(x) \land \mathbf{F}(w_j(x))))].$$

In $\sigma_8$, we model an unbounded number of transactions, that may come to the scheduler $S$, which uses the protocol that is defined in section 5.2, by three transactions $T_i, T_j$ and $T_k$. Transactions $T_i$ and $T_j$ represent any two particular transactions satisfying the accessing rules of the protocol and $T_k$ represents any other transaction in the schedule $S$. Transaction $T_k$ could execute any data item $x \in D_i \cap D_j \cap D_k$ as follows

Case 1: $T_k$ could execute $x$ after $T_i$ and $T_j$
Case 2: $T_k$ could execute $x$ after $T_i$ and before $T_j$

\[
\ldots \bullet \ldots \bullet \ldots \bullet \ldots \bullet \ldots
\]

$w_i(x) \quad w_j(x) \quad w_k(x)$

Case 3: $T_k$ could execute $x$ before $T_i$ and $T_j$

\[
\ldots \bullet \ldots \bullet \ldots \bullet \ldots \bullet \ldots
\]

$w_k(x) \quad w_i(x) \quad w_j(x)$

In the depiction above, we assume that $T_i$ executes the data item $x$, where $x \in D_i \cap D_j$, before $T_j$ does. Therefore, we can say that, $T_i$ and $T_j$ satisfy the accessing rules of the protocol if and only if for all $x \in D_i \cap D_j$, $TS(T_i, x) < TS(T_i, x)$ or $TS(T_j, x) < TS(T_i, x)$ regardless of when the transaction $T_k$ could access the data item $x$. We can illustrate this in the LTL structure as follows:

\[
\begin{align*}
&\frac{\sqrt{\sqrt{\sqrt{\sqrt{\frac{\sqrt{\sqrt{\sqrt{w_i(x_i) \quad r_j(x_i) \quad w_j(x_i) \quad w_i(x_{i+1}) \quad r_j(x_{i+1})}}}}}}}}
&\frac{\sqrt{\sqrt{\sqrt{\sqrt{\frac{\sqrt{\sqrt{\sqrt{w_i(x_i) \quad w_i(x_i) \quad w_i(x_i) \quad w_i(x_{i+1}) \quad w_i(x_{i+1}) \quad r_j(x_i) \quad w_j(x_i) \quad w_j(x_i) \quad r_j(x_{i+1})}}}}}}}}
&\frac{\sqrt{\sqrt{\sqrt{\sqrt{\frac{\sqrt{\sqrt{\sqrt{w_j(x_{i+1}) \quad w_i(x_{p}) \quad r_j(x_{p}) \quad w_j(x_{p})}}}}}}}}
&\frac{\sqrt{\sqrt{\sqrt{\sqrt{\frac{\sqrt{\sqrt{\sqrt{w_i(x_{i+1}) \quad w_i(x_{i+1}) \quad w_i(x_{i+1}) \quad w_i(x_{i+1}) \quad w_i(x_{i+1}) \quad r_j(x_{i+1}) \quad w_j(x_{i+1}) \quad r_j(x_{i+1})}}}}}}}}
\end{align*}
\]

where $D_i \cap D_j = \{x_i, x_{i+1}, \ldots, x_p\}$. We notice, from the depiction above, that transaction $T_i$ has executed $w_i(x_i)$ and $T_j$ has not executed both $r_j(x_i)$ and $w_j(x_i)$ yet, regardless of when/where the transaction $T_k$ executes read and write operations on the data item $x_i$ and similarly for $x_{i+1}$ up to $x_p$. Transaction $T_j$ will execute both $r_j(x_i)$ and $w_j(x_i)$ after $T_i$ does and this will keep $T_i$ and $T_j$ serializable. In other words, execution of any operations belonging to other transactions will not affect the serializability of the transactions $T_i$ and $T_j$. 
Chapter 5. A Timestamp-based Protocol For Multi-step Transactions

The access rules of the protocol (see subsection 5.2.1) say that if a transaction $T_i$ has executed $r_i(x)$ but not $w_i(x)$, no other transaction $T_j$ can execute $r_j(x)$. This can be encoded into LTL as follows

$$\sigma_9 = \bigwedge_{i,j \geq 1, i \neq j} \bigwedge_{x \in D_i \cap D_j} G[r_i(x) \implies \neg r_j(x)]$$

Therefore $\sigma_{10}$, which represents the access protocol, is defined as follows

$$\sigma_{10} = \sigma_8 \land \sigma_9.$$

This means that any history $h$ produced by the access protocol should satisfy both $\sigma_8$ and $\sigma_9$, if we model the read and write operations in the history $h$ by an LTL structure.

Now, the future in LTL is seen as a sequence of states, so the future is a path. Therefore, if we consider the states in the LTL path as instants of time, we can assign a truth value to each proposition at each time instant so that the interpretation maps to each instant of time a set of propositions that hold at that instant. As a consequence of this, we can assign a truth value to each read/write proposition belonging to any active transaction, at any given time, that is scheduled by the protocol. As we are dealing with a protocol based on timestamps, our interpretation will, therefore, map each timestamp to a set of propositions that hold at that timestamp. Thus, the interpretation $I$ is a function

$$I : N \rightarrow 2^{prop}$$

where $N$ is a set of timestamps (natural number) and $prop$ is a set of all propositions. This means that we can specify infinite histories generated by a protocol based on timestamps using LTL specifications. Now, as $\sigma_{10}$ represents the histories produced (or generated) by the protocol and $\sigma_7$ represents the serializability condition then, we can prove that the histories are serializable by proving the following formula:

$$\sigma_{10} \implies \sigma_7$$

5.6 NuSMV model

As mentioned in Section 1.4, model checking is an automated technique that, given a finite-state model of a system and its required formal properties (specifications) written in logic, systematically checks whether the properties hold for that model. We shall use this technique to verify the timestamps-based protocol using the
NuSMV model checker. NuSMV is a reimplementation and extension of SMV, the first model checker based on BDDs. NuSMV was developed by a joint project between Carnegie Mellon University (CMU) and Istituto per la Ricerca Scientifica e Tecnologica (IRST) [63]. NuSMV has been designed to be an open architecture for model checking, which can be reliably used for the verification of industrial designs, as a core for custom verification tools [10]. The input language of NuSMV is designed to allow for the description of Finite State Machines (FSMs) whose transition relation describes the evolutions of the states of the FSM. FSMs range from synchronous to asynchronous, and from the detailed to the abstract. Because it is intended to describe finite state machines, the only data types in the language are finite ones - Booleans, scalars and fixed arrays. Static data types can also be constructed [11]. Specifications in NuSMV can be written in CTL or LTL. These temporal logics allow us to express, in a concise syntax, the most important temporal properties of a system such as safety, liveness, fairness and deadlock freedom.

5.6.1 Modelling

Recall, from Subsection 1.4.1, the three main parts of the process of model checking - modelling, specification and verification. The modelling of the behaviour of the protocol is given as a finite state machine in the input language of NuSMV shown as follows, in which the following keywords appear:

- **MODULE**: Either the main module or a subroutine.
- **VAR**: Define types of variables as boolean or sets of symbols.
- **SPEC**: Refers to specification in CTL.
- **LTLSPEC**: Refers to specification in LTL.
- **ASSIGN**: Defines transition relations for variables.
- **init**: Defines the initial values of the variables.
- **next**: Defines a relationship between values of variables in a particular state and its successor state.
- **Case**: Returns the value of the first expression whose corresponding condition evaluates to TRUE.
- **Process**: Defines a collection of parallel processes, whose actions are interleaved asynchronously.
Chapter 5. A Timestamp-based Protocol For Multi-step Transactions

- - - : Indicates a comment.

The symbols !, &, |, -> and <-> represent logical not, and, or, implies and equivalence respectively. Some of the administrative commands offered by the interactive shell of NuSMV are described as follows

- `read_model`: Reads a NUSMV file.
- `go`: This command initializes the system for verification.
- `pick_state`: Chooses an element from the set of initial states and makes it the current state.
- `check_ltlspec`: Performs model checking of LTL formulae.
- `check_ctlspec`: Performs model checking of CTL formulae.
- `time`: Prints the processor time used since the last invocation of the time command, and the total processor time used since NUSMV was started.
- `check fsm`: Checks if the transition relation is total. If the transition relation is not total then a potential deadlock state is shown.
- `print_reachable_states`: Prints the number of reachable states of the given model.

5.6.1.1 The Model Variables

We define, in the NuSMV model, three transactions contending to access five data items. The reason for defining three transactions was explained in Section 5.5 above. The reason for defining five data items is that the proposed application (see Section 4.5) was to use these kinds of transactions to book e-tickets from travel agencies. Each data item, in the proposed application, represents 'stops' (or intermediate destinations). Typically, any travel agency does not have an itinerary from location A to B of more than five stops. So, it suffices to assume that the number of data items is five. The operations of the transactions are declared as values in the sets $T_1$, $T_2$ and $Toth$ in the main module (lines 39-41 of Appendix A), where $T_1$, $T_2$ and $Toth$ represent three different transactions operations, e.g. transaction $T_1$ contains six steps (operations). We denote $r_1(x_1)$ and $w_1(x_1)$ by $r1x1$ and $w1x1$, respectively. The set variable $T1$ can have one of the following values: begin_1, r1x1, w1x1, r1x2, w1x2, r1x3, w1x3 and end_1,
as in line 39. The value begin.1 indicates that the transaction wants to execute its first operation. The value end.1 indicates that the transaction has executed all of its operations. The expression $T_1 = \text{write} x_1$ means that the transaction $T_1$ is currently executing the write operation on the data item $x_1$ and has executed the read operation on the same data item $x_1$ in the past. The initial values of the sets $T_1, T_2$ and $T_{oth}$ are begin.1, begin.2 and begin.o, respectively, as in lines 47-49 of Appendix A. The values of the variables $T_1$, $T_2$ and $T_{oth}$ in the next state are given by the case statement in lines 4-12, lines 13-21 and lines 22-34 of Appendix A, respectively. The value of a case statement is determined by evaluating the clauses within the statement in sequence. Each clause consists of a condition and an expression which are separated by a colon (:). If the condition in the first clause holds, the value of the corresponding expression determines the value of the case statement. Otherwise the next clause is evaluated. The $\text{VAR}$ statement can also be used to instantiate a module as shown in lines 43-45. In our model, the module move is instantiated three times (lines 43-45), first with the name $L_1$, second with the name $L_2$ and finally with the name $L_3$. These three instantiations of a module move represent three transactions which are contending to execute under the accessing rule of the protocol. Because the keyword process is used in all cases, the global model is constructed by interleaving steps from $L_1$, $L_2$ and $L_3$. These interleaving steps, from different transactions, represent the history or the schedule that we want to generate and check against the specifications. However, the global model is not forced to eventually choose a given process (or transaction) to execute. As a consequence, the history generated may not contain operations from some transaction. In order to force a given process to execute infinitely often, we can use a fairness constraint. A fairness constraint restricts the attention of the model checker to only those execution paths along which a given formula is true infinitely often. Each process has a special variable called $\text{running}$ which is 1 if and only if that process is currently executing, see line 391.

### 5.6.2 Specifications

The FSM of the protocol is defined by instantiating three times the module type move in the module main, with the names $T_1$, $T_2$ and $T_3$ respectively. The move module has formal parameters $T.c, T_0, T_1$ and $n$ (line 1). The variables $T.c, T_0$ and $T_1$ receive $T_1$, $T_2$ and $T_{oth}$. The formal parameter $n$ receives three numbers 1, 2 and 3, where 1 means that the process $T_1$ is currently executing, 2 means process $T_2$ is currently executing and 3 means process $T_3$ is currently executing. The formal parameter $T.c$ receives the variables $T_1$, $T_2$ and $T_{oth}$ which represent the operations of the transactions. For example, assume that $T.c$ receives $T_1$ and
Chapter 5. A Timestamp-based Protocol For Multi-step Transactions

T1=begin1. This means that the current process (or transaction) that has been chosen by the model checker to execute is T.1 and T.1 wants to execute its first operation. The formal parameters TO and T1 receive the current operations values of the other transactions that has been executed. Therefore, the module move represents the movement of each transaction from one operation to the next one. To make sure that a transaction executes its operations sequentially (one by one) as in Figure 5.2, we have lines 4-12, 13-21 and 22-34. For example, we notice that if the current transaction is T.1 (we know that from n=1) and the current operation of T.1 is r1x1, the next operation that T.1 can execute is w1x1 and so on. But, by accessing rule 1 (Subsection 5.2.1), a transaction can not execute read operation on a data item x if and only if there is another transaction currently reading x. This is specified by adding a condition to be satisfied in order to execute any read operation, and this condition represents accessing rule 1. For example, as in line 16 and 17,

16 T_c=w2x2 & n=2 & (! (T0=r1x3) & (! (T_c=r2x3))
17 & (! (T1=rox3)) : r2x3;

Figure 5.2: Transaction executes its operations sequentially.

the transaction (or the process) T.2 can execute the read operation on the third data item r2x3 if the condition ( (! (T0=r1x3)) & (! (T_c=r2x3)) & (! (T1=rox3)) ) is satisfied. Because we have three transactions, we check to see if no transaction of all the transactions is currently executing a read operation on x3. Then, the transaction T.2 can execute it if T.2 satisfies the other accessing rules as well. This makes sure that the global model (or history) will not have the situations, where two transactions, at any point in time t_s, are executing their read operations on the same data item; see Figure 5.3.

Figure 5.3: No two transactions are reading the same data item simultaneously.
In order to make sure the our NuSMV model meets the model that is defined in 4.4, we specify the properties (P1)-(P5) of the read and write propositions using LTL specifications as follows.

(P1) Read/write alternation
This property is defined in \( \sigma_0 \) using past-time and future time operators such that

\[
\sigma_0 = \bigwedge_{i \geq 1} \bigwedge_{x,y \in D_i, x < y} \mathbf{G}[r_i(y) \land O(r_i(x)) \Rightarrow O(w_i(x))]
\]
or

\[
\sigma_0 = \bigwedge_{i \geq 1} \bigwedge_{x,y \in D_i, x < y} \mathbf{G}[r_i(x) \Rightarrow F(w_i(x) \land F(r_i(y)))]
\]

We notice that if we expand \( \sigma_0 \) to every transaction and every data item accessed by the transaction, we have specifications as in lines 57-94 i.e. letting \( i = 1 \), we can expand \( \sigma_0 \) as follows

\[
\mathbf{G}[r_1(x_2) \land O(r_1(x_1)) \Rightarrow O(w_1(x_1))]
\]
\[
\mathbf{G}[r_1(x_3) \land O(r_1(x_2)) \Rightarrow O(w_1(x_2))]
\]
\[
\mathbf{G}[r_1(x_3) \land O(r_1(x_1)) \Rightarrow O(w_1(x_1))]
\]

The corresponding specifications, as in lines 57-59, in the description language of the model checker (NuSMV) are

57  LTLSPEC G ( ((T1=r1x2) & O(T1=r1x1)) \rightarrow O(T1=w1x1) )
58  LTLSPEC G ( ((T1=r1x3) & O(T1=r1x1)) \rightarrow O(T1=w1x1) )
59  LTLSPEC G ( ((T1=r1x3) & O(T1=r1x2)) \rightarrow O(T1=w1x2) )

Also, we have the following specifications (that use future-time operators), if we expand \( \sigma_0 \) to every transaction and every data item accessed by the transaction,

\[
\mathbf{G}[r_1(x_1) \land F(w_1(x_1)) \Rightarrow F(r_1(x_2))]
\]
\[
\mathbf{G}[r_1(x_2) \land F(w_1(x_2)) \Rightarrow F(r_1(x_3))]\]
Chapter 5. A Timestamp-based Protocol For Multi-step Transactions

(P2) Write implies read
This property is defined in $\sigma_1$ as follows

$$\sigma_1 = \bigwedge_{i \geq 1} \bigwedge_{x \in D_i} G[w_i(x) \Rightarrow O(r_i(x))]$$
or

$$\sigma_1 = \bigwedge_{i \geq 1} \bigwedge_{x \in D_i} G[r_i(x) \Rightarrow F(w_i(x))]$$

Similar to $\sigma_0$, after expanding $\sigma_1$ to every transaction and every data item accessed by the transaction, we have the corresponding specifications in lines 97-124.

(P3) Read/write step proposition remains true until the next operation, belonging to the same transaction, becomes true

$$\sigma_2 = \bigwedge_{i \geq 1} \bigwedge_{x \in D_i} G[w_i(x) \Rightarrow O(r_i(x) \cup w_i(x))] \land$$
$$\bigwedge_{i \geq 1} \bigwedge_{x \in D_i} x \prec y \rightarrow G[r_i(y) \Rightarrow O(w_i(x) \cup r_i(y))]$$
or

$$\sigma_2 = \bigwedge_{i \geq 1} \bigwedge_{x \in D_i} G[w_i(x) \Rightarrow \lnot r_i(x)] \land$$
$$\bigwedge_{i \geq 1} \bigwedge_{x \in D_i} x \prec y \rightarrow G[r_i(y) \Rightarrow \lnot w_i(x)]$$

$\sigma_2$ is encoded in lines 127-176.

(P4) At most one step occurs at each successive state

$$\sigma_3 = \bigwedge_{i, i' \geq 1} \bigwedge_{1 \leq j, j' \leq m, \ i \neq i'} G \left[ \lnot (\lnot r_i(x_j) \land \lnot w_i'(x_{j'})) \land X(r_i(x_j) \land w_i'(x_{j'})) \right] \land$$
$$\lnot (\lnot r_i(x_j) \land \lnot w_i'(x_{j'})) \land X(r_i(x_j) \land w_i'(x_{j'})) \land$$
$$\lnot (\lnot w_i(x_j) \land \lnot w_i'(x_{j'})) \land X(w_i(x_j) \land w_i'(x_{j'}))).$$
This property is defined in $\sigma_3$ and the corresponding code in NuSMV description language is in lines 179-274.

(P5) A transaction $T_i$ accesses each data item $x \in D_i$ exactly once for both read and write operations

$$
\sigma_4 = \left( \bigwedge_{i \geq 1} \bigwedge_{x \in D_i} G \neg [r_i(x) \land F(\neg r_i(x) \land F_r_i(x))] \land 
\bigwedge_{i \geq 1} \bigwedge_{x \in D_i} G \neg [w_i(x) \land F(\neg w_i(x) \land Fw_i(x))] \right)
$$

or

$$
\sigma_4 = \left( \bigwedge_{i \geq 1} \bigwedge_{x \in D_i} G \neg [r_i(x) \land O(\neg r_i(x) \land O_r_i(x))] \land 
\bigwedge_{i \geq 1} \bigwedge_{x \in D_i} G \neg [w_i(x) \land O(\neg w_i(x) \land Ow_i(x))] \right)
$$

This is defined in $\sigma_4$ and $\sigma_4$ is encoded in lines 277-325. Finally, as $\sigma_{10}$ represents the histories produced (or generated) by the protocol as follows

$$
\sigma_{10} = \sigma_8 \land \sigma_9
$$

where

$$
\sigma_8 = \bigwedge_{i,j,k \geq 1 \atop i \neq j, j \neq k, i \neq k} G(\bigwedge_{x \in D_i \cap D_j \cap D_k} (w_k(x) \land O(w_j(x) \land O(w_i(x)))) \lor 
(\bigwedge_{x \in D_i \cap D_j \cap D_k} (w_j(x) \land O(w_k(x) \land O(w_i(x)))) \lor 
(\bigwedge_{x \in D_i \cap D_j \cap D_k} (w_j(x) \land O(w_i(x) \land O(w_k(x))))).)
$$
or

\[ \sigma_8 = \bigwedge_{i,j,k \geq 1} \bigwedge_{i \neq j \neq k, i \neq k} G[(\bigwedge_{x \in D_i \cap D_j \cap D_k} (w_i(x) \land F(w_j(x) \land F(w_k(x))))) \lor \left( \bigwedge_{x \in D_i \cap D_j \cap D_k} (w_i(x) \land F(w_j(x) \land F(w_j(x)))) \right) \lor \left( \bigwedge_{x \in D_i \cap D_j \cap D_k} (w_k(x) \land F(w_i(x) \land F(w_j(x)))) \right)]. \]

and \( \sigma_9 \)

\[ \sigma_9 = \bigwedge_{i \neq i'} \bigwedge_{x \in D_i \cap D_{i'}} G[r_i(x) \implies \neg r_{i'}(x)]. \]

and, \( \sigma_7 \) represents the serializability condition such that

\[ \sigma_7 = \sigma_5 \land \sigma_6 \]

where \( \sigma_5 \)

\[ \sigma_5 = \bigwedge_{i,i' \geq 1, i \neq i'} \bigwedge_{x,y \in D_i \cap D_{i'}, x < y} G\neg[(w_i(y) \land O(w_{i'}(y) \land O(w_i(x) \land O(w_i(x)))) \lor (w_{i'}(y) \land O(w_i(y) \land O(w_i(x) \land O(w_i(x)))) \})] \]

or

\[ \sigma_5 = \bigwedge_{i,i' \geq 1, i \neq i'} \bigwedge_{x,y \in D_i \cap D_{i'}, x < y} G\neg[(w_i(x) \land F(w_{i'}(x) \land F(w_{i'}(y) \land F(w_i(y)))) \lor (w_{i'}(y) \land F(w_i(x) \land F(w_i(x) \land F(w_i(y)))) \})] \]

and \( \sigma_6 \)

\[ \sigma_6 = \bigwedge_{i,i' \geq 1, i \neq i'} \bigwedge_{x \in D_i \cap D_{i'}} G\neg[r_i(x) \land r_{i'}(x)]. \]

then we can prove that the histories are serializable if and only if the following formula holds:

\[ G(\sigma_{8'} \implies \sigma_{5'}) \]

where \( \sigma_{8'} \) and \( \sigma_{5'} \) are the same as \( \sigma_8 \) and \( \sigma_5 \), respectively, without the LTL operator \( G \). This means that, at any point in time, if a history satisfies the access rules of the protocol, then the serializability condition is satisfied on that history. This is encoded in lines 343-390. The serializability condition and the protocol assert that no two transactions read the same data item simultaneously, as in \( \sigma_8 \) and \( \sigma_9 \).
respectively. This is specified in lines 329-338.

5.6.3 Verification

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. Then, by exploring the state space of the state transition system, it is possible to check automatically if the design 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 [10]. We verify our transactions model which is defined in the input language of NuSMV, in appendix A. The LTL 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. The run of the NuSMV model checker in Appendix A produces true for all specifications that are given in Section 5.5 and 5.6. This means that the serializability condition is true on the histories produced by the accessing protocol. We notice that the LTL specifications in 5.5 and 5.6 are encoded in past-time and future-time operators. The reason for using past-time operators, is that if the intended situation occurs, we can express the properties of the path that lead to it. For example, one of the model properties (P2), in Section 5.5, says that if \( w_i(x) \) is executed, then \( r_i(x) \) must have executed before and the corresponding LTL specification (using past-time operator) is

\[
\sigma_1 = \bigwedge_{i \geq 1} \bigwedge_{x \in D_i} G[w_i(x) \Rightarrow O(r_i(x))] 
\]

We notice, from \( \sigma_1 \), that if \( w_i(x) \) (intended situation) takes place, we check whether \( r_i(x) \) has executed in the path that lead to \( w_i(x) \). We can encode \( \sigma_1 \) (using future-time operators):

\[
\sigma_1 = \bigwedge_{i \geq 1} \bigwedge_{x \in D_i} G[r_i(x) \Rightarrow F(w_i(x))]
\]

This states that if the properties of the path that lead to the intended situation occur, eventually the intended situation will occur. In our model, when NuSMV is run on these two specifications, the output in Figure 5.4 is produced.
Chapter 5. A Timestamp-based Protocol For Multi-step Transactions

-- specification G (T1 = w1x1 -> 0 T1 = r1x1) is true
-- specification G (T1 = r1x1 -> F T1 = w1x1) is false

as demonstrated by the following execution sequence

Trace Type: Counterexample

-> State: 1.1 <-
  T1 = begin_1
  T2 = begin_2
  Toth = begin_o

-> Input: 1.2 <-
  _process_selector_ = T_1
  running = 0
  T_3.running = 0
  T_2.running = 0
  T_1.running = 1

-> State: 1.2 <-
  T1 = r1x1

-> Input: 1.3 <-
  _process_selector_ = main
  running = 1
  T_3.running = 0
  T_2.running = 0
  T_1.running = 0

-> State: 1.3 <-

-> Input: 1.4 <-
  _process_selector_ = T_2
  running = 0
  T_3.running = 0
  T_2.running = 1
  T_1.running = 0

-- Loop starts here

-> State: 1.4 <-
  T2 = r2x2

-> Input: 1.5 <-
  _process_selector_ = main
  running = 1
  T_3.running = 0
  T_2.running = 0
  T_1.running = 0

Figure 5.4: Output generated by NuSMV for σ₁
We notice that the specification that uses future-time operators is false. The counterexample, in Figure 5.4, demonstrates that the transaction T1 executes r1x1 and does not execute w1x1 because transaction T2 can execute r2x2 after that and stay in r2x2 forever. This is denoted by -- loop starts here in the NuSMV counterexample. However, the specification that uses past-time operators is true because if we assume that w1x1 is done, we check whether r1x1 has executed before or not. In order to make the specifications that use future-time operators true, we can add fairness constraints as in lines 51-53. These constraints assert that no transaction remains in any of its operations forever. Also, the counterexample in Figure 5.4 shows why it can happen that T1 never enters its operation w1x1. This can be illustrated, practically, when the transaction executes some of its operations and then aborts for any reason. But, our concern, in these specifications, is to verify the protocol. So, we need to make sure that all transactions, participating in a history generated by the protocol, execute their operations and satisfy the access rules, in order to verify whether the corresponding history satisfies the serializability condition or not.

5.7 Encoding into CTL

In this section, we use CTL to specify and verify the protocol and the serializability condition. As in chapter 3, 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. Also, CTL achieves polynomial-time model checking [8]. It is worthwhile using CTL to specify transactions to gain polynomial-time model checking. In section 5.5, we encoded the protocol and the serializability condition using LTL and we considered the interpretation $I$ as a function

$$I : \mathbb{N} \rightarrow 2^{\text{prop}}$$

where $\mathbb{N}$ is a set of timestamps (natural numbers) and prop is the set of all propositions. However, in the case of CTL, time is branching and so at each moment we have several different possible futures. Therefore, we can not model states as representing timestamps, because, at any point in time $t$, we may have more than one possible transition and all have to have the same timestamp; see Figure 5.5. In our model, each transition corresponds to a transaction executing its next operation. Hence, in CTL, we have to define timestamps explicitly for each active transaction $T_i$ and data item $x_a \in D_i$, as we will see in the next subsections.
5.7.1 The Model Variables

The model variables are exactly the same as the model variables that have been given in Section 5.6 with additional variables to represent timestamps. In Section 5.2, we defined the matrix $TSM$ to represent the timestamps for all active transactions in the scheduler $S$, which applies the access rules of the protocol, for all data items in $D$. In our model, we represent this by the variable $TSM$ as in line 149 of Appendix B. Because we have 3 transactions and 5 data items, we define $TSM$ to be a matrix of 3 rows and 5 columns. Each entry in the matrix can have only the values between 0 to 11. The reason is that we have 3 data items for transaction $T_1$, 3 data items for transaction $T_2$ and 5 data items for transaction $T_3$ (lines 144-146 respectively). Therefore, we need 11 different timestamps to deal with the interleaving of the operations of the transactions. In our model, we assume that the System TimeStamp (STS) is a variable, called $p_c$, specified as a set of consecutive integer numbers between 0 to 11 as in line 151. The initial value is set to zero as in line 161. The next value, for the variable $p_c$, is issued if any transaction accesses any data item for a read operation as in lines 64-75. Otherwise, the value remains unchanged as in line 76. If the value of $p_c$ reaches 11, this means that all transactions have completed their executions. As in Section 5.2, the initial value for each entry in the matrix $TSM$, which represents $TS(T_i, x_a)$, where $x_a \in D_i$ and $T_i \in T$, is 0. This is in lines 163-177. The next value of $TS(T_i, x_a)$ equals the next value of $p_c$, which means that $T_i$ accesses $x_a$ (for a read operation) as in line 80. This means that, at the same point in time, if a transaction executes the read operation on any data item, the value of $p_c$ should be incremented by one and the value of $TS(T_i, x_a)$ should be updated by the new value of $p_c$. When
Chapter 5. A Timestamp-based Protocol For Multi-step Transactions

the transaction completes its execution, $TS(T_i, x_a)$ should reset to 0 as in line 81. Otherwise, the value of $TS(T_i, x_a)$ remains unchanged as in line 82, as follows:

79 $\text{next}(TSM[1][1]) := \text{case}$
80 $T_c != r1x1 \& \text{next}(T_c) = r1x1 \& n=1: \text{next}(p_c);$;
81 $\text{next}(T_c) = \text{end}_1 : 0;$
82 $1 : TSM[1][1];$
83 esac;

5.7.2 Specifications

The FSM of the protocol is defined to be the same as in Subsection 5.6.2. The formal parameters in the module move, of Appendix B, are $T_c,n,T0,T1,TSM$ and $p_c$, where $T_c,n,T0$ and $T1$ have the same meaning as the formal parameters in the module move, of Appendix A. We add the parameters $TSM$ and $p_c$ to receive the current values of the timestamps for all active transactions and all data items in $D$, defined in the module main in array $TSM$, and the current value of the system timestamp, respectively. Now, as in accessing rule 1 of the protocol (Subsection 5.2.1), a transaction can not execute a read operation on a data item $x$ if and only if there is another transaction currently reading $x$. This is specified by adding a condition to be satisfied for any transaction wanting to execute a read operation. For example, in lines 13-17:

13 $T_c = w1x2 \& n=1 \& ((TSM[1][3]=0) \&(TSM[1][3]!=0 \&(T_c=r1x3)))$
14 \hspace{1cm} & ((TSM[2][3]=0)\&(TSM[2][3]!=0 \&(T0=r2x3)))$
15 \hspace{1cm} & ((TSM[3][3]=0)\&(TSM[3][3]!=0 \&(T1=ro3x3))) \&
16 \hspace{1cm} !(TSM[2][2]>0 \&(TSM[2][2]<TSM[1][2] \& TSM[2][3]=0)) \&
17 \hspace{1cm} !(TSM[3][2]>0 \&(TSM[3][2]<TSM[1][2] \& TSM[3][3]=0)) : r1x3;

the transaction $T_1$ may execute the read operation on the third data item $r1x3$ if the condition

$(((TSM[1][3]=0)|(TSM[1][3]!=0 \&(T_c=r1x3)))\&$

$((TSM[2][3]=0)|(TSM[2][3]!=0 \&(T0=r2x3)))\&$

$((TSM[3][3]=0)|(TSM[3][3]!=0 \&(T1=ro3x3)))$

is satisfied along with the condition in lines 16 and 17. Because we have three transactions, we check that if any transaction $T_i$ such that $T_i$ is not currently executing the read operation on $x_3$ (i.e $TSM[1][3]=0$) or has executed the write operation on $x_3$ (i.e $TSM[1][3]!=0 \&(T_c=r1x3)$), then there is no transaction reading $x_3$ at this point in time. This makes sure that the global model (or history) will not have the situation, where two transactions, at any point in time,
are executing their read operations on the same data item simultaneously. This is specified in the NuSMV model in lines 293-302 and also represents access rules 1 and 2 of the protocol. As \( x_2 \) is not the first data item in the set \( D_1 \), \( T_1 \) should satisfy access rules 3 and 4. We specify this in the condition

\[
\]

This condition asserts that if we do not have any transaction \( T_i \) such that \( T_i \) had executed the data item \( x_2 \) (i.e. \( TSM[2][2] > 0 \)) before \( T_1 \) (i.e. \( TSM[2][2] < TSM[1][2] \)), where \( x_2 \) is the data item that precedes \( x_3 \) in \( D \), and has not executed the read operation on \( x_2 \) (\( TSM[2][3] = 0 \)), then transaction \( T_1 \) can execute the read operation on \( x_3 \).

Now, to make sure the our NuSMV model, in Appendix B, meets the model that is defined in Section 5.4, we specify the properties (P1)-(P5) of the read and write propositions using CTL specifications as follows:

(P1) Read/write alternation

A transaction \( T_i \) cannot have read two distinct data items (in \( D_i \)) without having written to one of them, i.e. let \( x <_D y \), \( r_i(y) \) cannot be executed until \( w_i(x) \) has been executed. This property is defined in \( \sigma_{\nu} \):

\[
\sigma_{\nu} = \bigwedge_{i \geq 1} \bigwedge_{x,y \in D_i, x <_D y} AG[r_i(x) \implies AF(w_i(x) \land AFr_i(y))]
\]

The corresponding specifications in the NuSMV model are in lines 187-196.

(P2) Write implies read

A transaction \( T_i \) can only have written to \( x \) if it has read \( x \), i.e. if \( w_i(x) \) is executed, then \( r_i(x) \) must have executed before.

\[
\sigma_{\nu} = \bigwedge_{i \geq 1} \bigwedge_{x \in D_i} AG[r_i(x) \implies AFw_i(x)]
\]

This formula is specified in the NuSMV model in lines 199-211.

(P3) Read/write step proposition remains true until the next operation, belonging to the same transaction, becomes true

If a read/write step has taken place, the corresponding proposition remains true until the next operation in \( T_i \) becomes true, i.e. \( r_i(x)/w_i(x) \) is true, remains true until the next step \( w_i(x)/r_i(y) \), where \( x <_D y \), becomes true.
Chapter 5. A Timestamp-based Protocol For Multi-step Transactions

\[ \sigma_{y'} = \bigwedge_{i \geq 1} \bigwedge_{x \in D_i} AG[w_i(x) \Rightarrow \neg r_i(x)] \land \\
\bigwedge_{i \geq 1} \bigwedge_{x,y \in D_i \land x < D_y} AG[(r_i(y) \Rightarrow \neg w_i(x))]
\]

We specify \( \sigma_{y'} \) in lines 214-234 of Appendix B.

(P4) At most one step occurs at each successive state

No two distinct steps can both be false in a state, and then both true in a next state.

\[ \sigma_{y'} = \bigwedge_{1 \leq i,j \leq m} \bigwedge_{i \neq i' \text{ or } j \neq j'} AG\left[ \neg (\neg r_i(x_j) \land \neg r_{i'}(x_{j'})) \land \text{EX}(r_i(x_j) \land r_{i'}(x_{j'})) \right] \\
\land \neg (\neg w_i(x_j) \land \neg w_{i'}(x_{j'})) \land \text{EX}(w_i(x_j) \land w_{i'}(x_{j'})))
\]

The corresponding specifications of \( \sigma_{y'} \) in NuSMV are in lines 238-263.

(P5) A transaction \( T_i \) accesses each data item \( x \in D_i \) exactly once for both read and write operations

\[ \sigma_{y'} = \left( \bigwedge_{i \geq 1} \bigwedge_{x \in D_i} AF[\neg r_i(x) \land AF(\neg r_i(x) \land \text{EF}r_i(x))] \right) \land \\
\left( \bigwedge_{i \geq 1} \bigwedge_{x \in D_i} AF[\neg w_i(x) \land AF(\neg w_i(x) \land \text{EF}w_i(x))] \right)
\]

\( \sigma_{y'} \) is specified in the NuSMV model in lines 266-290. Next, we encode the serializability condition of Theorem 4.8. This is defined by \( \sigma_{y'} \) as follows:

\[ \sigma_{y'} = \bigwedge_{i,i' \geq 1, i \neq i'} \bigwedge_{x,y \in D_{i''} \land x \neq D_{i''}} AF[-[(w_{i''}(x) \land AF(w_{i''}(x) \land \text{EF}(w_{i''}(y) \land AF(w_{i''}(y)))) \lor \\
(w_{i''}(x) \land AF(w_{i''}(x) \land \text{EF}(w_{i''}(y) \land AF(w_{i''}(y))))])
\]

The serializability condition is specified in the NuSMV model in lines 305-306.

In the NuSMV model that is discussed in Subsection 5.6.2, we encode the histories produced (or generated) by the protocol explicitly, as in \( \sigma_{10} \). The reason is that we consider the states in a path as instants of time and these instants of time represent the set of timestamps that are generated by the system. Therefore, we can specify histories produced by the protocol, using LTL operators, regardless
of the histories that produced by the model. In other words, the NuSMV model that is discussed in Subsection 5.6.2 generates histories satisfying access rules 1 and 2. But, it does not satisfy access rules 3 and 4. This means that we can generate histories that do not satisfy the protocol. But, we can specify the histories generated by the protocol, as a subset of the histories generated by the NuSMV model. It is these histories that should imply the serializability condition, to prove that any history satisfying the access rules of the protocol is serializable. Now, in case of the NuSMV model that is discussed in Subsection 5.7.1, we explicitly define the timestamps by means of a variable in the model, see line 149, and therefore we can directly implement the access rules of the protocol, based on the explicit definition of timestamps in the model. This means that the NuSMV model that is discussed in Subsection 5.7.1 generates histories conforming to the access rules of the protocol. Therefore, there is no need to encode the histories produced by the protocol explicitly. All we need is to encode the serializability condition, as we do in $\sigma_s$, to check if the access rules imply the serializability condition.

5.7.3 Verification

As in subsection 5.6.3, we add fairness constraints (see line 307 and lines 179-181) to make sure that all transactions will participate in any history generated by the NuSMV model, to verify that such a history satisfies the serializability condition. The CTL specifications are run by NuSMV in order to determine their truth or falsity in the finite state machine model. In NuSMV a CTL specification is given as a CTL formula introduced by the keyword SPEC (lines 187-306). For all specifications, the run of NuSMV model produces true which means that the serializability condition is true on the histories produced by the protocol and the NuSMV model matches the model that is defined in 5.4.

5.8 Comparison of LTL and CTL

In this section, we shall compare CTL and LTL for specifying and verifying such protocols based on timestamps. Actually, CTL and LTL have incomparable expressive power [12]. Many CTL formulas cannot be expressed in LTL, especially those containing paths quantified existentially. For example, the following formula

$$\text{AG}[\phi \implies EF\varphi]$$

can not be expressed in LTL. Also, the following formula in LTL

$$\text{GF}\phi \implies \text{GF}\varphi$$
can not be expressed in CTL. Now, given a transition system $M$ and a linear temporal logic formula $\phi$, the model-checking problem for $M$ and $\phi$ is to decide whether $\phi$ holds in all the computations (paths) of $M$. When $\phi$ is a CTL formula, the problem is to decide whether it holds in the computation tree of $M$. As mentioned before, given a transition system of size $|S| + |R|$, where $|S|$ is the number of states in the finite state machine, $|R|$ the number of transitions, and $|f|$ the length of the formula, CTL model-checking algorithms run in time $O(|f| \cdot (|S| + |R|))$, while LTL model-checking algorithms run in time $O(|S| + |R|) \cdot 2^{O(|f|)}$. Thus, model checking for CTL takes time linear in the size of the specification, and for LTL takes time exponential in the size of the specification. This seems to suggest that model checking for CTL is more efficient than for LTL. But, we conduct our own comparison of the verification of the protocol based on timestamps using LTL and CTL, given in Appendix A and Appendix B respectively. The results are in Figure 5.6 and Figure 5.7, respectively. The model that uses LTL specifications executes in much less time than the model that uses CTL specifications.
Chapter 5. A Timestamp-based Protocol For Multi-step Transactions

Figure 5.6: Time and space for the model of LTL specifications.
Chapter 5. A Timestamp-based Protocol For Multi-step Transactions

NuSMV >read_model -i ctl_finite.smv
NuSMV >go
NuSMV >pick_state
NuSMV >time
elapsed: 0.1 seconds, total: 0.1 seconds
NuSMV >check_ctlspec -o ctl_finite.txt
Output to file: ctl_finite.txt
NuSMV >time
elapsed: 2538.9 seconds, total: 2538.9 seconds
NuSMV >print_usage
BDD statistics
------------------------
BDD nodes allocated: 2448420
------------------------
Statistics on BDD FSM machine.
BDD nodes representing init set of states: 75
BDD nodes representing state constraints: 1
BDD nodes representing input constraints: 1
Forward Partitioning Schedule BDD cluster size (#nodes):
cluster 1 : size 1124
cluster 2 : size 1259
cluster 3 : size 1732
cluster 4 : size 1149
cluster 5 : size 65
Backward Partitioning Schedule BDD cluster size (#nodes):
cluster 1 : size 1124
cluster 2 : size 1259
cluster 3 : size 1732
cluster 4 : size 1149
cluster 5 : size 65
NuSMV > check_fsm
#########################################################
The transition relation is total: No deadlock state exists
#########################################################
NuSMV > print_reachable_states
#########################################################
system diameter: 26
reachable states: 110697 (2^16.7563) out of 1.41991e+020 (2^66.9444)
#########################################################

Figure 5.7: Time and space for the model of CTL specifications.
The reason is that the model which uses CTL specifications (with explicit timestamps variables) produces a large number of states in comparison with the model which uses LTL specifications (without timestamps variables); see Figure 5.7 and 5.6. The large number of states is due to the explicit definition of the timestamps as variables in the model. The large number of variables in the model makes the size of the transition system \(|S| + |R|\) grow and affects the model checking time. In our problem, the size of the transition system \(|S| + |R|\) is much larger than the size of the specification \(|f|\). This makes the effect of \(|S| + |R|\) (the size of the transition system) dominant over \(|f|\) (the size of the specification) in the time complexity formula for both CTL and LTL. Therefore, we see that LTL verification is not particularly inefficient and is even better than CTL verification for such problems. Note that, Figures 5.6 and 5.7 show that no deadlock state exists in both models.

Next, we explain how we can modify the model in order to encode transactions iterating infinitely many times, see Figure 5.8.

In the NuSMV model that uses LTL specifications, we can do this in a simple way by adding a statement which says that once a transaction has executed all operations then it can iterate again. The NuSMV statement, see lines 14, 25 and 40 of Appendix C is:

\begin{verbatim}
14 T_c=end_1 & n=1 :begin_1;
\end{verbatim}

The above statement asserts that when a transaction which has executed all operations (T_c=end_1) chooses to execute again, it can take, at the next point in time, the value begin_1. This means that the transaction is ready to iterate (T_c=begin_1) in the future. All LTL specifications, which are encoded in Section 5.5, remains unchanged except the specification which represents (P5). The new specification for (P5) is as follows

\begin{align*}
\left( \bigwedge_{i \geq 1} \bigwedge_{x \in D_i} G[begin_i \implies F(\neg r_i(x) \land F(\neg r_i(x) \land Fr_i(x)) \cup end_i)] \right) \land \\
\left( \bigwedge_{i \geq 1} \bigwedge_{x \in D_i} G[begin_i \implies F(\neg w_i(x) \land F(\neg w_i(x) \land Fr_i(x)) \cup end_i)] \right)
\end{align*}
The corresponding specifications in the NuSMV description language are in lines 312-405 of Appendix C.

In the NuSMV model that uses CTL specifications, we have to make some modifications. Firstly, as we have 11 different operations in all transactions (as we discussed in 5.7.1), we define the variable \( p_c \), which represents the system timestamps (STS), as a set of consecutive integer numbers between 0 and 11. Each number represents a different timestamp. But, in the case when transactions iterate infinitely many times, this assumption will not be correct anymore, because we do not know how many times each transaction will iterate. Assume that (at any point in time) we have 3 (or less) active transactions contending to execute under the accessing rule of the protocol. Then, the maximum number of the different timestamps we need is 11. Once \( p_c \) reaches 11 (the maximum value), it is reset to 1 again and so on, as in line 82 of Appendix D. As in the NuSMV model that uses LTL specifications, we add statements which say that once the transaction has executed all operations, it can iterate again (lines 21, 39 and 66 of Appendix D). But, after simulating the corresponding model (Appendix D) for our assumptions in the NuSMV model checker, we find that the assumption is not correct, as in the following example:

```bash
NuSMV > read_model -i ctl_infinite.smv
NuSMV > go
NuSMV > pick_state
NuSMV > simulate -r 100

******* Simulation Starting From State 1.1  *******
NuSMV > goto_state 1.1

The current state for new trace is:

\[
\text{-> State 2.1 <-}
\]

...process_selector_ = main
T1 = begin_1
T2 = begin_2
Toth = begin_o
TSM[1][1] = 0
TSM[1][2] = 0
TSM[1][3] = 0
TSM[1][4] = 0
TSM[1][5] = 0
TSM[2][1] = 0
TSM[2][2] = 0
TSM[2][3] = 0
TSM[2][4] = 0
```
Chapter 5. A Timestamp-based Protocol For Multi-step Transactions

\[
\begin{align*}
TSM[2][5] &= 0 \\
TSM[3][1] &= 0 \\
TSM[3][2] &= 0 \\
TSM[3][3] &= 0 \\
TSM[3][4] &= 0 \\
TSM[3][5] &= 0 \\
p_c &= 0
\end{align*}
\]

NuSMV \texttt{>goto\_state 1.22}

The current state for new trace is:

\[
\begin{align*}
\text{\_process\_selector\_} &= \text{main} \\
T1 &= \text{wlx3} \\
T2 &= \text{begin\_2} \\
Toth &= \text{rox5} \\
TSM[1][1] &= 3 \\
TSM[1][2] &= 4 \\
TSM[1][3] &= 7 \\
TSM[1][4] &= 0 \\
TSM[1][5] &= 0 \\
TSM[2][1] &= 0 \\
TSM[2][2] &= 0 \\
TSM[2][3] &= 0 \\
TSM[2][4] &= 0 \\
TSM[2][5] &= 0 \\
TSM[3][1] &= 1 \\
TSM[3][2] &= 2 \\
TSM[3][3] &= 5 \\
TSM[3][4] &= 6 \\
TSM[3][5] &= 8 \\
p_c &= 8
\end{align*}
\]

NuSMV \texttt{>goto\_state 1.23}

The current state for new trace is:

\[
\begin{align*}
\text{\_process\_selector\_} &= \text{main} \\
T1 &= \text{end\_1} \\
T2 &= \text{begin\_2} \\
Toth &= \text{rox5} \\
TSM[1][1] &= 0 \\
TSM[1][2] &= 0
\end{align*}
\]
Chapter 5. A Timestamp-based Protocol For Multi-step Transactions

TSM[1] [3] = 0
TSM[1] [4] = 0
TSM[1] [5] = 0
TSM[2] [1] = 0
TSM[2] [2] = 0
TSM[2] [3] = 0
TSM[2] [4] = 0
TSM[2] [5] = 0
TSM[3] [1] = 1
TSM[3] [2] = 2
TSM[3] [3] = 5
TSM[3] [5] = 8
p_c = 8
NuSMV > goto_state 1.24

The current state for new trace is:

\[ \text{State 7.24} \leftarrow \]

\[ \text{process_selector} = \text{main} \]
\[ T1 = \text{begin}_1 \]
\[ T2 = \text{begin}_2 \]
\[ Toth = \text{rox5} \]
\[ TSM[1] [1] = 0 \]
\[ TSM[1] [2] = 0 \]
\[ TSM[1] [3] = 0 \]
\[ TSM[1] [4] = 0 \]
\[ TSM[1] [5] = 0 \]
\[ TSM[2] [1] = 0 \]
\[ TSM[2] [2] = 0 \]
\[ TSM[2] [3] = 0 \]
\[ TSM[2] [4] = 0 \]
\[ TSM[2] [5] = 0 \]
\[ TSM[3] [1] = 1 \]
\[ TSM[3] [2] = 2 \]
\[ TSM[3] [3] = 5 \]
\[ TSM[3] [4] = 6 \]
\[ TSM[3] [5] = 8 \]
\[ p_c = 8 \]

NuSMV > goto_state 1.26

The current state for new trace is:
Chapter 5. A Timestamp-based Protocol For Multi-step Transactions

-> State 8.26 <-

_process_selector_ = main
T1 = r1x1
T2 = begin_2
Toth = w0x5
TSM[1][1] = 9
TSM[1][2] = 0
TSM[1][3] = 0
TSM[1][4] = 0
TSM[1][5] = 0
TSM[2][1] = 0
TSM[2][2] = 0
TSM[2][3] = 0
TSM[2][4] = 0
TSM[2][5] = 0
TSM[3][1] = 1
TSM[3][2] = 2
TSM[3][3] = 5
TSM[3][4] = 6
TSM[3][5] = 8
p_c = 9
NuSMV > goto_state 1.35

The current state for new trace is:
-> State 14.35 <-

_process_selector_ = main
T1 = w1x2
T2 = w2x3
Toth = end_0
TSM[1][1] = 9
TSM[1][2] = 1
TSM[1][3] = 0
TSM[1][4] = 0
TSM[1][5] = 0
TSM[2][1] = 0
TSM[2][2] = 10
TSM[2][3] = 11
TSM[2][4] = 0
TSM[2][5] = 0
TSM[3][1] = 0
We notice from the example above that transaction $T_1$ ends the execution of its operations in State 6.23 and $T_2$ has not begun yet. Then, $T_1$ begins the execution again (second iteration) in State 8.26 and $T_2$ has still not begun. Meanwhile, $T_3$ has executed some of its operations and is waiting to resume ($Toth = \text{wox5}$). In State 14.35 transaction $T_3$ ends its execution and $T_2$ and the second iteration of $T_1$ have executed some of their operations. At the same state, the system timestamp ($p_c$) is reset to 1, because it reaches 11 (the maximum value) in the previous state. Also, in State 14.35, $TSM[1][2] = 1$, which represents the timestamp when the second iteration of transaction $T_1$ accesses the data item $x_2$, and $TSM[2][2] = 10$. This means that the second iteration of $T_1$ executes its operations on $x_2$ before $T_2$ does. But, we know, from the trace above, that this is incorrect. The reason is that the value of system timestamps ($p_c$) is reset to 1 and there are transactions which have not completed their execution. These transactions have values of timestamps for their data items greater than 1, and the new read operation has the value 1. This means that, according to the protocol, the new operation, which has the smallest timestamp, precedes the other data items in the history. To solve this problem, we have to define $p_c$ to be infinite in order to keep issuing a new timestamp in order to avoid the previous situation. It is impossible in any model checker to define a variable with an infinite number of values.

Therefore, we can say that from the comparison above of the use of CTL and LTL in the context of protocols based on timestamps, the tradeoff between CTL and LTL is not a simple tradeoff between complexity and expressiveness [13]. It is a more complicated matter of suitability for the problem.
Chapter 6

Specification and Verification of Mobile Multi-step Transactions with Priority

6.1 Introduction

In the previous chapter, we introduced a protocol, based on timestamps, to be a correctness criterion for multi-step transactions accessing ordered sets of data items $D$. Also, we proved that the protocol satisfies the serializability condition. A further observation was that the verification of the LTL specifications was easier than the CTL specifications that we gave. In this chapter, we shall modify the protocol in order to deal with mobile environments where there are two types of multi-step transactions, 'local' and 'mobile' transactions, iterating infinitely many times. These two types of transactions (mobile and local) are introduced in the context of mobile environments in [17, 16]. We shall specify the modified protocol using LTL specifications and verify using the NuSMV model checker to show that this modification of the protocol keeps the transactions (mobile and local) serializable. We shall also discuss other properties such as starvation.

6.2 Mobile Transactions

As in [16], we assume two kinds of transactions executing concurrently on any mobile database participating in a MDBC (Mobile Database Community), mobile transactions and local transactions. A MDBC is a dynamic collection of distributed, autonomous, heterogeneous and mobile databases, interconnected through a wireless communication infrastructure of a MANET (mobile ad hoc network). A local transaction is submitted directly to a mobile database on the same host. In
contrast, a mobile transaction (MT), is submitted to one or more different mobile databases in different hosts. At any instant in time, a new participant may join this community or transiently disconnect from it. The number of mobile transactions that may be initiated by participants is unbounded. Most of the mobile computing environment consists of fixed (FH) and mobile hosts (MH). Mobile hosts may comprise PDAs (Personal Digital Assistants), cellular phones, laptops and other mobile-enabled technologies devices. A MH can retain its network connections to FHs (wired networks) through base stations (BS) via a wireless channel; see Figure 6.1. A mobile transaction, in this context, is generated by a MH and can be executed at its generating host and/or some fixed hosts. A local transaction (LT) is generated by a host (fixed or mobile) and can be executed on the same host [17]. In this chapter, we shall assume that a mobile transaction is a transaction initiated (or generated) in a MH and executing at different hosts (fixed or mobile). This means that the MH does not have any relevant data or enough server capabilities to execute its transactions. In the next section, we shall introduce an application that reflects these definitions of MTs and LTs.

6.3 Application

Recently, mobile-enabled devices have become common, cheap and widely used. The mobile services introduced by different cooperations are developing at an exponential rate. Moreover, the number of mobile customers that use digital mobile networks for voice and data transfer has already exceeded one billion and probably will exceed two billions in a few years [18]. The winners, in this context, will be those who have built a flexible infrastructure that delivers services easily and cost-efficiently, as the number of subscribers they serve rapidly grows beyond bounds.
Chapter 6. Specification and Verification of Mobile Transactions with Priority 99

As universities are cooperations, many universities and colleges have started to invest heavily in gaining maximum benefit from this increased development in mobile computing [19].

For example, universities can introduce mobile services for students, using their own mobile-enabled devices, for various needs such as learning, reading course content, revising for exams, meeting course deadlines and registering on courses [20]. In this chapter, we shall concentrate on MTs which are used to register on courses. In Figure 6.2, students can register on courses using either LTs or MTs. By LTs, in this context, we mean the transactions that are generated by students to register on one or more courses from the available courses, using wired communications. For example, students can make use of the existing infrastructure of the wired network in university buildings to access the registration system. By MTs we mean the transactions that are generated by students to register on one or more courses, from the available courses, using wireless communications. For example, students can make use of the infrastructure of the wireless network on the university campus to access the registration system. They may also use mobile-enabled devices off-campus to access the university registration system using the infrastructure of the wireless network of their service provider. This means that students can access the website of the registration system of the university from their homes.

Figure 6.2: Mobile and local transactions in a registration system.
We assume that the set of available courses, for each subject, is a totally ordered set as in Definition 4.1. We can interpret the relationship among courses as a total order, where each course has a prerequisite course (or courses). Also, some courses can be taken at the same time (co-requisites). Therefore, if any student wishes to register on a course, they should meet the requirements of a prerequisite course, or they can register on the course and the prerequisite course in the same semester. For example, let the set of available courses in the computer science department be AC={discrete maths, logic design, computer systems, computer architecture, ..., graduation project} and AC1, AC2 and AC3 be the sets of the intended courses to be registered by students S1, S2 and S3, respectively, as follows:

AC1={logic design, computer systems}
AC2={discrete math, logic design}
AC3={logic design, computer systems, computer architecture}

Here, for example, S1 wishes to register on logic design and computer systems in the same semester as co-requisite courses, and S1 took the prerequisite course discrete math in the previous semester. Also, if S2 intends to register, for example, then the equivalent transaction will be as follows:

Read(discrete math) Write(discrete math) Read(logic design) Write(logic design)

where the data item discrete math represents the number of students who have registered on the course. A student can not be registered on any course (write operation) without knowing, beforehand, if the number of students who have registered is less than the maximum number of students allowed to register on that course. Also, a student can not register on the logic design course without having completed registering on the prerequisite course (discrete math) beforehand. As mentioned previously, students generate their transactions either using the wired infrastructure network of the university (LTS) or using the wireless infrastructure network of the university or the service provider (MTs); see Figure 6.2. In fact, a local transaction scheduler in any mobile or fixed host (which has the 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 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 [21]. This gives the higher priority transactions (MTs) a chance to execute their operations as soon as possible to avoid any disconnection that may occur.


6.4 The modified protocol

In the timestamp-based protocol, which is defined in Subsection 5.2.1, we assume that all transactions have the same priority to execute their operations in the scheduler. In this chapter, we give mobile transactions higher priority over local transactions. This means that we have to modify the accessing rules of the protocol, to new ones that are in line with our assumption. The question is - what do we mean by higher priority? The answer is that once a MT comes to the scheduler, all LTs are suspended until the MT finishes its execution. Also, if we have more than one MT in the schedule, we should schedule MTs together according to the accessing rules of the timestamp-based protocol. In other words, if the scheduler contains only LTs or only MTs, then we apply the accessing rules ordinarily. But, if there is a mixture of LTs and MTs, then the MTs apply the accessing rules until they finish their operations. Meanwhile, all LTs are suspended and wait for the MTs to finish their operations in order to resume again. For example, assume that we have 4 ordered sets of data items:

\[ D_{l_1} = \{x_2, x_3, x_4\} \]
\[ D_{l_2} = \{x_1, x_2\} \]
\[ D_{m_1} = \{x_2, x_3, x_4, x_5\} \]
\[ D_{m_2} = \{x_2, x_3\} \]

accessed by corresponding transactions \(T_{l_1}, T_{l_2}, T_{m_1}\), and \(T_{m_2}\) respectively. Then, the transactions will be as follows:

\[ T_{l_1} = r_1(x_2)w_1(x_2)r_1(x_3)w_1(x_3)r_1(x_4)w_1(x_4) \]
\[ T_{l_2} = r_2(x_1)w_2(x_1)r_2(x_2)w_2(x_2) \]
\[ T_{m_1} = r_3(x_2)w_3(x_2)r_3(x_3)w_3(x_3)r_3(x_4)w_3(x_4)r_3(x_5)w_3(x_5) \]
\[ T_{m_2} = r_4(x_2)w_4(x_2)r_4(x_3)w_4(x_3) \]

where \(T_{l_1}\) and \(T_{l_2}\) represent local transactions and \(T_{m_1}\) and \(T_{m_2}\) represent mobile transactions. Suppose that \(T_{l_1}\) precedes \(T_{m_1}\), \(T_{m_1}\) precedes \(T_{m_2}\), and \(T_{m_2}\) precedes \(T_{l_1}\) in arriving at scheduler \(S\). Also, suppose that \(S\) makes use of the priority assumption above to schedule the incoming transactions. Initially, the TSM will be as follows:

\[ TS(T, D) = \begin{pmatrix} 0 & 0 & \ldots & 0 \\ 0 & 0 & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \ldots & 0 \end{pmatrix} \]
Chapter 6. Specification and Verification of Mobile Transactions with Priority

Now, assume that history $h$ and TSM at some point in time are such that:

$$h = r_l(x_1)w_l(x_1), \quad TS(T, D) = \begin{pmatrix}
0 & 0 & \ldots & 0 \\
1 & 0 & \ldots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
0 & 0 & \ldots & 0
\end{pmatrix}$$

and that the transaction $T_{m_1}$ comes to the scheduler to access its first data item $x_2$. According to the assumption above, $T_1$ and $T_2$ should suspend until $T_{m_1}$ finishes its execution. This means that, at the next point in time, TSM and $h$ will be such that:

$$h = r_l(x_1)w_l(x_1)r_{m_1}(x_2), \quad TS(T, D) = \begin{pmatrix}
0 & 0 & \ldots & 0 \\
1 & 0 & \ldots & 0 \\
0 & 0 & \ldots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
0 & 0 & \ldots & 0
\end{pmatrix}$$

Subsequently, assume that the history $h$ and TSM, at point in time, are such that:

$$h = r_l(x_1)w_l(x_1)r_{m_1}(x_2)w_{m_1}(x_2), \quad TS(T, D) = \begin{pmatrix}
0 & 0 & \ldots & 0 \\
1 & 0 & \ldots & 0 \\
0 & 0 & \ldots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
0 & 0 & \ldots & 0
\end{pmatrix}$$

and that the transaction $T_{m_2}$ comes to the scheduler to access its first data item $x_2$. According to the assumption above and accessing rule 2, $T_{m_2}$ can access $x_2$. Therefore, at the next point in time, the TSM and $h$ will be such that:

$$h = r_l(x_1)w_l(x_1)r_{m_1}(x_2)w_{m_1}(x_2)r_{m_2}(x_2), \quad TS(T, D) = \begin{pmatrix}
0 & 0 & \ldots & 0 \\
1 & 0 & \ldots & 0 \\
0 & 0 & \ldots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
0 & 0 & \ldots & 0
\end{pmatrix}$$

Now, assume that the history $h$ is:

$$h = r_l(x_1)w_l(x_1)r_{m_1}(x_2)w_{m_1}(x_2)r_{m_2}(x_2)w_{m_2}(x_2),$$

the TSM is as in 6.1, and transactions $T_{m_1}$ and $T_{m_2}$ try to access data item $x_3$. According to accessing rule 3, any one of them satisfying the conditions of
accessing rule 4 can access $x_3$ first. Now, both of them satisfy conditions (a) and (c) of rule 4. But, only $T_{m_1}$ also satisfies condition (b) because it has the minimum timestamp of $x_2$. This means that $T_{m_1}$ preceded $T_{m_2}$ in accessing $x_2$ and it should also precede $T_{m_2}$ in accessing $x_3$. Subsequently, when $T_{m_1}$ and $T_{m_2}$ have finished their execution, the history $h$ and TSM will be as follows:

$$h = r_{i_2}(x_1)w_{i_2}(x_1)r_{m_1}(x_2)w_{m_1}(x_2)r_{m_2}(x_2)w_{m_2}(x_2)r_{m_1}(x_3)w_{m_1}(x_3)$$

$$r_{m_2}(x_3)w_{m_2}(x_3)r_{m_1}(x_4)w_{m_1}(x_4)r_{m_1}(x_5)w_{m_1}(x_5)$$

$$TS(T,D) = \begin{pmatrix}
0 & 0 & \ldots & 0 \\
1 & 0 & \ldots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
0 & 0 & \ldots & 0
\end{pmatrix}$$

We notice that the transactions $T_{m_1}$ and $T_{m_2}$ timestamps entries are reset to zero. This occurs when any transaction finishes its execution on all its data items. Now, $T_{i_1}$ and $T_{i_2}$ can resume their executions according to the accessing rules of the protocol. In the previous chapter, we proved that the scheduler, which makes use of the accessing rules of the protocol, generates serializable histories. In the next subsections, we shall specify and verify the modified protocol to see whether with the new additional assumption that we have added about the transactions (MTs have higher priority than LTs), the scheduler still generates serializable histories.

### 6.5 Transactions model

In this section, we shall define properties of the read and write propositions in a Kripke structure $M$. Also, we shall illustrate the correspondence between a path in a Kripke structure $M$ and a history $h$.

#### 6.5.1 Properties of read and write propositions

Assume that we have a Kripke structure $M$, with the following properties (P1-P6) relating to $r_i(x_j)$ and $w_i(x_j)$ propositions, holding for $M$. We have the same meaning as in Section 5.4 for properties (P1-P5):

- **(P1)** Read/write alternation
- **(P2)** Write implies read
- **(P3)** Read/write step proposition remains true until the next operation, belonging to the same transaction, becomes true
Chapter 6. Specification and Verification of Mobile Transactions with Priority

(P4) At most one step occurs at each successive state

(P5) A transaction $T_i$ accesses each data item $x \in D_i$ exactly once for both read and write operations

(P6) Once a mobile transaction comes to the scheduler then, all local transactions suspend until there is no mobile transactions in the scheduler.

For all $T_i \in T_l$, where $T_l$ is a set of local transactions, $T_i$ is suspended if there is an active mobile transaction $T_m \in T_m$, where $T_m$ is a set of mobile transactions, in the scheduler.

Property (P6) represents the modification on the protocol for the assumption that says that mobile transactions have higher priority than local transactions.

6.5.2 The correspondence between paths and histories

In this subsection, we illustrate the correspondence between a path in a Kripke structure $M$ and a history $h$ generated by the modified protocol, as follows:

\[
\begin{array}{cccccccc}
 & r_{i_1}(x_1) & w_{i_1}(x_1) & r_{m_1}(x_2) & w_{m_1}(x_2) & r_{m_1}(x_3) & w_{m_1}(x_3) & \text{end}_{m_1} \\
\text{s}_0 & \text{s}_1 & \text{s}_2 & \text{s}_3 & \text{s}_4 & \text{s}_5 & \text{s}_6 & \text{s}_7 \\
\text{begin}_{l_1} & r_{l_1}(x_1) & w_{l_1}(x_1) & w_{l_1}(x_1) & w_{l_1}(x_1) & w_{l_1}(x_1) & w_{l_1}(x_1) & w_{l_1}(x_1) \\
\text{begin}_{l_2} & \text{begin}_{l_2} & \text{begin}_{l_2} & \text{begin}_{l_2} & \text{begin}_{l_2} & \text{begin}_{l_2} & \text{begin}_{l_2} & \text{begin}_{l_2} \\
\text{begin}_{m_1} & \text{begin}_{m_1} & \text{begin}_{m_1} & \text{begin}_{m_1} & \text{begin}_{m_1} & \text{begin}_{m_1} & \text{begin}_{m_1} & \text{begin}_{m_1} \\
\text{begin}_{m_2} & \text{begin}_{m_2} & \text{begin}_{m_2} & \text{begin}_{m_2} & \text{begin}_{m_2} & \text{begin}_{m_2} & \text{begin}_{m_2} & \text{begin}_{m_2} \\
\text{begin}_{m_3} & \text{begin}_{m_3} & \text{begin}_{m_3} & \text{begin}_{m_3} & \text{begin}_{m_3} & \text{begin}_{m_3} & \text{begin}_{m_3} & \text{begin}_{m_3} \\
\text{begin}_{m_4} & \text{begin}_{m_4} & \text{begin}_{m_4} & \text{begin}_{m_4} & \text{begin}_{m_4} & \text{begin}_{m_4} & \text{begin}_{m_4} & \text{begin}_{m_4} \\
\text{begin}_{m_5} & \text{begin}_{m_5} & \text{begin}_{m_5} & \text{begin}_{m_5} & \text{begin}_{m_5} & \text{begin}_{m_5} & \text{begin}_{m_5} & \text{begin}_{m_5} \\
\text{begin}_{m_6} & \text{begin}_{m_6} & \text{begin}_{m_6} & \text{begin}_{m_6} & \text{begin}_{m_6} & \text{begin}_{m_6} & \text{begin}_{m_6} & \text{begin}_{m_6} \\
\text{begin}_{m_7} & \text{begin}_{m_7} & \text{begin}_{m_7} & \text{begin}_{m_7} & \text{begin}_{m_7} & \text{begin}_{m_7} & \text{begin}_{m_7} & \text{begin}_{m_7} \\
\text{begin}_{m_8} & \text{begin}_{m_8} & \text{begin}_{m_8} & \text{begin}_{m_8} & \text{begin}_{m_8} & \text{begin}_{m_8} & \text{begin}_{m_8} & \text{begin}_{m_8} \\
\text{begin}_{m_9} & \text{begin}_{m_9} & \text{begin}_{m_9} & \text{begin}_{m_9} & \text{begin}_{m_9} & \text{begin}_{m_9} & \text{begin}_{m_9} & \text{begin}_{m_9} \\
\text{begin}_{m_{10}} & \text{begin}_{m_{10}} & \text{begin}_{m_{10}} & \text{begin}_{m_{10}} & \text{begin}_{m_{10}} & \text{begin}_{m_{10}} & \text{begin}_{m_{10}} & \text{begin}_{m_{10}} \\
\text{begin}_{m_{11}} & \text{begin}_{m_{11}} & \text{begin}_{m_{11}} & \text{begin}_{m_{11}} & \text{begin}_{m_{11}} & \text{begin}_{m_{11}} & \text{begin}_{m_{11}} & \text{begin}_{m_{11}} \\
\text{begin}_{m_{12}} & \text{begin}_{m_{12}} & \text{begin}_{m_{12}} & \text{begin}_{m_{12}} & \text{begin}_{m_{12}} & \text{begin}_{m_{12}} & \text{begin}_{m_{12}} & \text{begin}_{m_{12}} \\
\text{begin}_{m_{13}} & \text{begin}_{m_{13}} & \text{begin}_{m_{13}} & \text{begin}_{m_{13}} & \text{begin}_{m_{13}} & \text{begin}_{m_{13}} & \text{begin}_{m_{13}} & \text{begin}_{m_{13}} \\
\text{begin}_{m_{14}} & \text{begin}_{m_{14}} & \text{begin}_{m_{14}} & \text{begin}_{m_{14}} & \text{begin}_{m_{14}} & \text{begin}_{m_{14}} & \text{begin}_{m_{14}} & \text{begin}_{m_{14}} \\
\text{begin}_{m_{15}} & \text{begin}_{m_{15}} & \text{begin}_{m_{15}} & \text{begin}_{m_{15}} & \text{begin}_{m_{15}} & \text{begin}_{m_{15}} & \text{begin}_{m_{15}} & \text{begin}_{m_{15}} \\
\text{begin}_{m_{16}} & \text{begin}_{m_{16}} & \text{begin}_{m_{16}} & \text{begin}_{m_{16}} & \text{begin}_{m_{16}} & \text{begin}_{m_{16}} & \text{begin}_{m_{16}} & \text{begin}_{m_{16}} \\
\text{begin}_{m_{17}} & \text{begin}_{m_{17}} & \text{begin}_{m_{17}} & \text{begin}_{m_{17}} & \text{begin}_{m_{17}} & \text{begin}_{m_{17}} & \text{begin}_{m_{17}} & \text{begin}_{m_{17}} \\
\text{begin}_{m_{18}} & \text{begin}_{m_{18}} & \text{begin}_{m_{18}} & \text{begin}_{m_{18}} & \text{begin}_{m_{18}} & \text{begin}_{m_{18}} & \text{begin}_{m_{18}} & \text{begin}_{m_{18}} \\
\text{begin}_{m_{19}} & \text{begin}_{m_{19}} & \text{begin}_{m_{19}} & \text{begin}_{m_{19}} & \text{begin}_{m_{19}} & \text{begin}_{m_{19}} & \text{begin}_{m_{19}} & \text{begin}_{m_{19}} \\
\text{begin}_{m_{20}} & \text{begin}_{m_{20}} & \text{begin}_{m_{20}} & \text{begin}_{m_{20}} & \text{begin}_{m_{20}} & \text{begin}_{m_{20}} & \text{begin}_{m_{20}} & \text{begin}_{m_{20}} \\
\text{end}_{l_1} & \text{end}_{l_1} & \text{end}_{l_1} & \text{end}_{l_1} & \text{end}_{l_1} & \text{end}_{l_1} & \text{end}_{l_1} & \text{end}_{l_1} \\
\text{s}_{16} & \text{s}_{17} & \text{s}_{18} & \text{s}_{19} & \text{s}_{20} & \text{s}_{21} & \text{s}_{22} & \text{s}_{23} \\
\text{end}_{l_1} & \text{end}_{l_1} & \text{end}_{l_1} & \text{end}_{l_1} & \text{end}_{l_1} & \text{end}_{l_1} & \text{end}_{l_1} & \text{end}_{l_1} \\
\text{end}_{l_1} & \text{end}_{l_1} & \text{end}_{l_1} & \text{end}_{l_1} & \text{end}_{l_1} & \text{end}_{l_1} & \text{end}_{l_1} & \text{end}_{l_1} \\
\text{end}_{l_1} & \text{end}_{l_1} & \text{end}_{l_1} & \text{end}_{l_1} & \text{end}_{l_1} & \text{end}_{l_1} & \text{end}_{l_1} & \text{end}_{l_1} \\
\text{end}_{l_1} & \text{end}_{l_1} & \text{end}_{l_1} & \text{end}_{l_1} & \text{end}_{l_1} & \text{end}_{l_1} & \text{end}_{l_1} & \text{end}_{l_1} \\
\text{end}_{l_1} & \text{end}_{l_1} & \text{end}_{l_1} & \text{end}_{l_1} & \text{end}_{l_1} & \text{end}_{l_1} & \text{end}_{l_1} & \text{end}_{l_1} \\
\end{array}
\]
In the depiction above, we have:

\[ D_{t_1} = \{x_1, x_2, x_3\} \]
\[ D_{t_2} = \{x_2, x_3, x_4\} \]
\[ D_{m_1} = \{x_2, x_3\} \]

and the corresponding transactions as follows:

\[ T_{t_1} = r_1(x_1)w_1(x_1)r_1(x_2)w_1(x_2)r_1(x_3)w_1(x_3) \]
\[ T_{t_2} = r_2(x_2)w_2(x_2)r_2(x_3)w_2(x_3)r_2(x_4)w_2(x_4) \]
\[ T_{m_1} = r_3(x_2)w_3(x_2)r_3(x_3)w_3(x_3) \]

The read and write propositions that are given for each successive state represent the propositions that are true in that state. The top of each column displays the unique proposition that becomes true in the particular state. This represents the read and write operations that have been scheduled by the modified protocol. In order to make it easier to follow the structure with a large number of propositions which are false, only the values of propositions that are true in the states of the trace are shown. The corresponding history \( h \) is:

\[ h = r_{t_1}(x_1)w_{t_1}(x_1)r_{m_1}(x_2)w_{m_1}(x_2)r_{m_1}(x_3)w_{m_1}(x_3)r_{t_2}(x_2)w_{t_2}(x_2)r_{t_1}(x_2)w_{t_1}(x_2)r_{t_2}(x_3) \]
\[ w_{t_2}(x_3)r_{t_1}(x_3)w_{t_1}(x_3)r_{t_2}(x_4)w_{t_2}(x_4) \ldots r_{t_2}(x_2) \ldots r_{t_1}(x_1) \ldots r_{m_1}(x_2) \]

Notice, from the depiction above, that once the mobile transaction \( T_{m_1} \) starts to execute its operations in the scheduler, all local transactions suspend (see states \( s_3-s_7 \)) until the mobile transaction ends, and then they can resume again (see states \( s_8-s_{18} \)). Also, notice that each transaction (mobile or local transaction) can iterate (or repeat) in the history infinitely many times; see state \( s_k \). In \( s_{16} \) and \( s_{19} \), local transactions \( T_{t_1} \) and \( T_{t_2} \) end their execution, respectively, and may iterate again at any point in time in the future; see state \( s_k \). In state \( s_k \), we denote by \( - \) an arbitrary operation of a local transaction that has executed. This means that the local transactions can iterate in the interval between \( s_{20} \) and \( s_k \) and once the second iteration of a mobile transaction has started (or it could be a new mobile transaction), then all local transactions (new or iterated transactions) suspend. Furthermore, this means that the mobile transaction \( T_{m_1} \) may iterate again in the future regardless of the progress of the local transactions. The phenomenon of iterating transactions, in the context of students registration systems, occurs widely.
6.6 Encoding into LTL

In this section, we shall encode the properties of the read and write propositions (P1-P6) of the structure that has been explained in Subsection 6.5.2, the serializability condition and some other liveness properties into LTL. Firstly, the properties of the read and write propositions (P1-P5) are exactly the same properties that were explained and encoded in Sections 5.4 and 5.5 respectively. The additional property (P6) will be encoded as follows.

(P6) Once a mobile transaction comes to the scheduler then, all local transactions suspend until there are no mobile transactions in the scheduler.

For all $T_l, E_T$, where $T_l$ is a set of local transactions, $T_m$ is a set of mobile transactions in the scheduler.

$\sigma_{P6} = \bigwedge_{j \geq 1, T_j \in T_m} \bigwedge_{D_j = \{x_1, \ldots, x_u\}} G[(r_j(x_1) \implies X(( \bigvee_{i \geq 1, T_i \in T_l} \neg C_i) \cup \text{end}_j))]$

In $\sigma_{P6}$, we make use of additional propositions $C_i$ ($i \geq 1, T_i \in T_l$), each of which is true in a state if and only if the local transaction $T_i$ progresses (or $T_i$ does not remain on the same operation of the previous state). This can be specified as $\sigma_e$ as follows

$\sigma_e = \bigwedge_{i \geq 1, T_i \in T_l} \bigwedge_{x \in D_i} G[(r_i(x) \land \neg r_i(x)) \implies X C_i] \land \bigwedge_{i \geq 1, T_i \in T_l} \bigwedge_{x \in D_i} G[(w_i(x) \land \neg w_i(x)) \implies X C_i]$

Next, we encode the serializability condition of Theorem 4.8 as $\sigma_{sc}$ as follows

$\sigma_{sc1} = \bigwedge_{i \geq 1, i \neq j} \bigwedge_{x \in D_i \cap D_j} G[\neg r_i(x) \land r_j(x)]$

$\sigma_{sc2} = \bigwedge_{i \geq 1, i \neq j} \bigwedge_{x \in D_i \cap D_j, x_1 < x} G[P_{ij} \implies (F(w_i(x) \land F(w_j(x)))) \cup \neg P_{ij}]$

$\sigma_{sc} = \sigma_{sc1} \land \sigma_{sc2}$

As in 5.5, $\sigma_{sc1}$ represents the specification to avoid any cycle of length two on the same data item between any two transactions (see Figure 5.1(a)) and $\sigma_{sc2}$ represents the specification to avoid any cycle of length two on different data items between any two transactions (see Figures 5.1(b) and 5.1(c)). In $\sigma_{sc2}$, we
make use of additional propositions $P_{ij}$ ($i, j \geq 1$, $i \neq j$), each of which is true in a state if and only if $T_i$ precedes $T_j$ on the first data item $x_1 \in D_i \cap D_j$. Therefore, $\sigma_{scs}$ means that if $T_i$ precedes $T_j$ on the first data item $x_1 \in D_i \cap D_j$, eventually $T_i$ will precede $T_j$ on every data item $x \in D_i \cap D_j$, where $x$ is such that $x_1 <_D x$, until $T_j$ precedes $T_i$ on the first data item in the future. This means that the value of $P_{ij}$ should become false. Because we assume that the transactions may iterate infinitely many times, the value of the propositions $P_{ij}$ may also change in a path infinitely many times. For example, assume we have two transactions $T_i$ and $T_j$, where $T_i$ precedes $T_j$ on the first data item $x_1 \in D_i \cap D_j$ in the first iteration. Then, at some point in time in the future, transaction $T_i$ ends and $T_j$ has not ended yet. After that, the second iteration of $T_i$ starts its execution and $T_j$ has not yet ended the first iteration. This means that $T_j$ precedes the second iteration of $T_i$ on the first data item $x_1$, and $P_{ij}$ should change its value; see the depiction below.

\[
\begin{array}{cccccccc}
\vdots & r_i(x_1) & \vdots & w_i(x_1) & \vdots & r_j(x_1) & \vdots & \checkmark & \checkmark \\
\vdots & s_i & \vdots & s_{i+r} & \vdots & s_{i+u} & \vdots & s_{i+p} & \checkmark \\
\vdots & \bullet & \vdots & \bullet & \vdots & \bullet & \checkmark & \checkmark \\
\vdots & w_i(x_0) & \vdots & w_i(x_0) & \vdots & r_j(x_1) & \vdots & r_j(x_u) & \checkmark \\
\end{array}
\]

The following specifies that the transactions (local or mobile) will keep iterating infinitely many times in a given path:

\[
\sigma_{inf} = \bigwedge_{i \geq 1} G[(begin_i \implies F end_i) \land (end_i \implies F begin_i)].
\]

where $\sigma_{inf}$ expresses that whenever a transaction begins it eventually ends, and whenever a transaction ends it eventually begins again. The corresponding specifications of the properties $P_1$ and $P_2$ ($\sigma_{P1}$ and $\sigma_{P2}$) together with $\sigma_{inf}$ make sure that the transactions will keep iterating infinitely many times. This means that in a path of the scheduler, there must never be a point at which the condition of the iteration ($(begin_i \implies F end_i) \land (end_i \implies F begin_i)$) becomes false. Therefore, we can consider the concept of 'infinitely many times' as a loop; see Figure 6.3. The reason is that we build an infinite path using a finite number of states, so the path must contain a loop.
Chapter 6. Specification and Verification of Mobile Transactions with Priority

Figure 6.3: ‘Infinitely many times’ means a loop.

In LTL, time is linear so that at each moment in time we have only one possible future. Figure 6.3 does not represent a case that may happen in a LTL path as there are two possible futures in the state \( s_{i+n} \). But, if the loop is unwound (as in Figure 6.4), then we can look at the path as a sequence of states \( \{s_i, \ldots, s_{i+n}\} \) comprising infinitely many iterations.

Figure 6.4: Set of states iterating infinitely often

6.7 The corresponding NuSMV model

In model checking, it is crucial to ensure that the model which has been defined in the model checker description language matches the model that has been defined by the user, in order to perform correct model checking. Therefore, in this section, we shall explain how we can build a NuSMV model that matches the transactions model which has been discussed in Section 6.5.

6.7.1 The model variables

In this subsection, we explain the NuSMV model variables which are defined in the NuSMV description language in Appendix E. Firstly, we define three transactions operations \( T_1, T_2 \) and \( T_m \) in lines 77-79 of Appendix E. \( T_1 \) and \( T_2 \) represent two local transactions operations and \( T_m \) represents mobile transaction operations. These transactions (local and mobile) are contending to access five data items. The reason for defining three transactions is explained in Section 5.5. The reason for defining five data items comes from the proposed application in Section 6.3. Each available course in the registration system is represented by a data item. Typically, students can register on up to five co-requisite courses at the same time but not more. If we increase the number of available courses in each semester, the possibility that the students transactions intersect (or contend) on the data items
Chapter 6. Specification and Verification of Mobile Transactions with Priority

(available courses) will reduce. Therefore, we choose a few data items to verify the system behaviour where there is a high probability of transactions intersecting on the same data items. The additional propositions, $C_i$ which are introduced in Section 6.6, are defined in lines 84 and 85 of Appendix E. Also, the additional propositions $P_{ij}$ are defined in lines 81-83. The formal variables of the module `move` (see line 1), $T_c, T0$ and $T1$ receive the variables $T1$, $T2$ and $Tm$, as in Subsection 5.6.2. The formal parameter $n$ receives a number between 1-3 (i.e. 1 means that process (or transaction) $T1$ is currently executing), exactly as in Subsection 5.6.2. Moreover, The formal parameters $p12, p13, p23, c1$ and $c2$ receive the variables $p12, p13, p23, c1$ and $c2$ respectively.

### 6.7.2 Specifications

The behaviour of the modified protocol is specified as a finite state machine in the input language of NuSMV. The transition relation of the finite state machine is defined by a relationship between the values of boolean expressions in a state and their values in the next state. The FSM of the modified protocol is defined by instantiating the module type `move` three times in the module `main`, with the names $T1, T2$ and $T3$ respectively, see lines 87-89. The module `move` represents the movement of each transaction from one operation to the next one, as we mentioned before in Subsection 5.6.2. Now, we explain how we can specify the accessing rules of the modified protocol in the NuSMV model. Firstly, the accessing rule 1 is specified as follows:

```plaintext
3 next(T_c):=case
4 T_c=begin_1&T1=begin_m&n=1&(!(T_c=r1x1)&(!T1=rmx1)) : r1x1;
5 T_c=r1x1 &T1=begin_m&n=1 : w1x1;
6 T_c=w1x1 &T1=begin_m&n=1&(!T_c=r1x2)&(!T0=r2x2)&(!T1=rmx2) : r1x2;
7 (T_c=r1x2 &T1=begin_m&n=1 : w1x2;
8 T_c=w1x2 &T1=begin_m&n=1&(!T_c=r1x3)&(!T0=r2x3)&(!T1=rmx3) & p12 : r1x3;
```

In line 4, the current transaction $T_c$ (or the transaction being executed) can not execute or move to the read operation on a data item $x1$ without satisfying the conditions: $T1=begin_n$ and $(!T_c=r1x1)&(!T1=rmx1))$. The condition $(!T_c=r1x1)&(!T1=rmx1))$ means that if there does not exist another transaction (i.e. $T1$) currently executing the read operation on the same data item $x1$, the transaction $T_c$ can execute the read operation on $x1$. This condition represents accessing rule 1 of the modified protocol. In line 6, the same idea applies
Chapter 6. Specification and Verification of Mobile Transactions with Priority

The condition $T_1=\text{begin}_m$ represents our assumption which says that mobile transactions have higher priority over local transactions. For example, as in line 8, the current transaction $T_c$ can execute the next operation ($w_1x_2$) if the mobile transaction $T_1$ has not started yet ($T_1=\text{begin}_m$), where $T_1$ is the formal parameter of the mobile transaction $T_m$. Otherwise, assume that the mobile transaction $T_1$ is executing an operation on any data item. This means that the condition $T_1=\text{begin}_m$ is not satisfied and therefore the current transaction $T_c$ should suspend until the mobile transaction finishes its execution. Further, in lines 9 and 10 the current transaction $T_c$, where $T_c$ is the formal parameter of the transaction $T_1$, can execute the next operation ($w_1x_3$) if the variable $p_12$ is true. This means that the transaction $T_c$ can execute the read operation on the data item $x_3$ if $T_c$ has preceded the other transaction in accessing the first data item $x_2$, where $x_2$ is the first data item in $D_1 \cap D_2$. Actually, $p_12$ represents accessing rule 4 which says that the transaction $T_i$ can execute any data item $x$, where $x$ is such that $x_1 < D_i$ and $x_1$ is the first data item, if $T_i$ is the first transaction accessing $x_1$ among all other transactions that access $x_1$. The additional propositions $p_{ij}$, which have been introduced in Section 6.6, are given by the model variables $p_{12}$, $p_{13}$ and $p_{23}$. For example, the variable $p_{12}$ is true if transaction $T_1$ precedes $T_2$ on the first data item $x_2$ in $D_1 \cap D_2$. This is specified as follows:

```plaintext
42 next(p12) := case
43 (next(T_c)=r1x2 | next(T_c)=w2x2 | next(T_c)=r1x3 | next(T_c)=w1x3 |
44 next(T_c)=end_1) & T0=begin_2 : 1;
45 (next(T_c)=r2x2 | next(T_c)=w2x2 | next(T_c)=r2x3 | next(T_c)=w2x3 |
46 next(T_c)=r2x4 | next(T_c)=w2x4 | next(T_c)=end_2) & (T0=begin_1 |
47 T0=r1x1 | T0=w1x1 ) : 0;
48 1 : p12;
49 esac;
```

The initial value of the variable $p_{12}$ is set to be false (see line 97 of Appendix E). As given in line 42 above, the relationship between the value of variable $p_{12}$ in a particular state and its successor state is represented by $\text{next}(p_{12})$. The value of the variable $p_{12}$ equals true (or 1) in the next state (successor state) if the value of the transaction $T_c$ (the formal parameter of the transaction $T_1$) equals one of the following values $r1x2$, $w2x2$, $r1x3$, $w2x3$ or $\text{end}_1$, and the value of $T0$ (the formal parameter of the transaction $T_2$) equals $\text{begin}_2$. This means that $p_{12}$ is true if we have a state in the FSM where the transaction $T_1$ is executing one of its operations on the data items in $D_1 \cap D_2$ ($x_2$ and $x_3$) and the transaction $T_2$ has not started yet ($T0=\text{begin}_2$). This specifies that $T_1$ precedes $T_2$ in the first data item in $D_1 \cap D_2$. The propositions $p_{13}$ and $p_{23}$ are specified in the NuSMV...
Next, the read and write properties (P1-P6) of the model, which were introduced in Subsection 6.5.1, are specified as follows:

\((P1)\) Read/write alternation

\[ \sigma_{P1} = \bigwedge_{i \geq 1} \bigwedge_{x,y \in D_i, x < y} G[(r_i(x) \Rightarrow F(w_i(x) \land F(r_i(y)))] \]

The corresponding specifications of the property P1 in the NuSMV description language are in lines 137-139, 141-143 and 145-154 for transactions \( T_1, T_2 \) and \( T_m \) respectively.

\((P2)\) Write implies read

\[ \sigma_{P2} = \bigwedge_{i \geq 1} \bigwedge_{x \in D_i} G[r_i(x) \Rightarrow \neg r_i(x)] \land \bigwedge_{i \geq 1} \bigwedge_{x,y \in D_i, x < y} G[r_i(y) \Rightarrow \neg w_i(x)] \]

We specify the property P2 in NuSMV in lines 157-159, 162-164 and 166-170 for transactions \( T_1, T_2 \) and \( T_m \) respectively.

\((P3)\) Read/write step proposition remains true until the next operation, belonging to the same transaction, becomes true

\[ \sigma_{P3} = \bigwedge_{i \geq 1} \bigwedge_{x \in D_i} G[w_i(x) \Rightarrow \neg r_i(x)] \land \bigwedge_{i \geq 1} \bigwedge_{x,y \in D_i, x < y} G[r_i(y) \Rightarrow \neg w_i(x)] \]

Lines 173-178, 180-185 and 187-196 give the corresponding NuSMV specifications of the property P3 for the transactions \( T_1, T_2 \) and \( T_m \) respectively.

\((P4)\) At most one step occurs at each successive state

\[ \sigma_{P4} = \bigwedge_{1 \leq j, j' \leq m, i,j \neq i', j' \neq i} G[\neg((\neg r_i(x_j) \land \neg r_{i'}(x_{j'})) \land X(r_i(x_j) \land r_{i'}(x_{j'})))] \land \neg((\neg r_i(x_j) \land \neg w_{i'}(x_{j'})) \land X(r_i(x_j) \land w_{i'}(x_{j'}))) \land \neg((\neg w_i(x_j) \land \neg w_{i'}(x_{j'})) \land X(w_i(x_j) \land w_{i'}(x_{j'}))). \]

\(\sigma_{P4}\) is specified in lines 199-229 of Appendix E.

\((P5)\) A transaction \( T_i \) accesses each data item \( x \in D_i \) exactly once for both read
Chapter 6. Specification and Verification of Mobile Transactions with Priority

and write operations

\[
\sigma_{P5} = (\bigwedge_{i \geq 1} \bigwedge_{x \in D_i} G[\text{begin}_i \implies F(\neg \text{r}_i(x) \land F(\neg \text{r}_i(x) \land F\text{r}_i(x))) \lor \text{end}_i]) \land \\
(\bigwedge_{i \geq 1} \bigwedge_{x \in D_i} G[\text{begin}_i \implies F(\neg \text{w}_i(x) \land F(\neg \text{w}_i(x) \land F\text{w}_i(x))) \lor \text{end}_i])
\]

The property \(\sigma_{P5}\) is specified in the NuSMV description language in lines 232-278.

(P6) Once a mobile transaction comes to the scheduler then, all local transactions suspend until there is no mobile transactions in the scheduler.

\[
\sigma_{P6} = \bigwedge_{j \geq 1, T_j \in T_m} G[\text{r}_j(x) \implies X((\bigvee_{i \geq 1, T_i \in T_l} \neg C_i) \lor \text{end}_j)]
\]

In \(\sigma_{P6}\), we make use of additional propositions \(C_i\) \((i \geq 1, T_i \in T_l)\), which are defined in lines 84 and 85. Proposition \(C_1 \in C_i\), for example, is true if and only if the local transaction \(T_1\) progresses. This is specified (for \(C_1\)) as follows:

64 next(c1):=case
65 n=1 & T_c!=next(T_c) :1;
66 1 :0;
67 esac;

The value of \(C_1\) in the initial state is set to be false, as in line 100. At any state thereafter, the value of \(C_1\) is determined by \text{next}(c1), as in line 64 above.

The following serializability condition specifications

\[
\sigma_c = \bigwedge_{i \geq 1, T_i \in T_l} \bigwedge_{x \in D_i} G[(\text{r}_i(x) \land X^! \text{r}_i(x)) \implies X C_i] \land \\
(\bigwedge_{i \geq 1, T_i \in T_l} \bigwedge_{x \in D_i} G[(\text{w}_i(x) \land X^! \text{w}_i(x)) \implies X C_i]
\]

The corresponding specifications of \(\sigma_c\) in the NuSMV are in lines 282-299. Line 280 represents the corresponding NuSMV specification of the property P6.

The following serializability condition specifications

\[
\sigma_{sc} = \bigwedge_{i,j \geq 1, i \neq j} \bigwedge_{x \in D_i \cap D_j} G[\neg (\text{r}_i(x) \land \text{r}_j(x))]
\]
Chapter 6. Specification and Verification of Mobile Transactions with Priority

$$\sigma_{sc2} = \bigwedge_{i,j \geq 1, i \neq j} \bigwedge_{x \in D_i \cap D_j, x_1 < x_2} G[P_{ij} \implies (F(w_i(x)) \land F(w_j(x)))) \cup \neg P_{ij}]$$

are specified in NuSMV in lines 302-313 and 315-318 for $\sigma_{sc1}$ and $\sigma_{sc2}$ respectively. The specification that asserts that the transactions (local and mobile) will keep iterating infinitely many times

$$\sigma_{inf} = \bigwedge_{i \geq 1} G[(begin_i \implies F end_i) \land (end_i \implies F begin_i)]$$

is specified in the NuSMV description language in lines 320-325 for each transaction: $T_2, T_1$ and $T_m$ respectively. Also, in lines 341-355, we add specifications to make sure that $\sigma_{inf}$ is equivalent to the following specification

$$\sigma_{inf1} = \bigwedge_{i \geq 1} GF[(begin_i \implies F end_i) \land (end_i \implies F begin_i)].$$

The reason is that the ‘infinitely many times’ or ‘infinitely often’ event occurrence (called weak fairness), is usually expressed in LTL using the temporal operators $GF$ [12]. In our NuSMV model, we assume that both specifications $\sigma_{inf}$ and $\sigma_{inf1}$, which represent iteration of the transactions infinitely many times in the model, are equivalent. Moreover, as in subsection 5.6.3, we add fairness constraints (see lines 103-131) to make sure that all operations of the transactions will participate in any history generated by the NuSMV model.

6.8 Verification of the transactions model

In this section, we verify the truth or falsity of the all specifications that we have introduced in the previous section, in the finite state machine which represents the model of the modified protocol. This is performed by exploring the state space of the state transition system of the modified protocol. If any specification is true, this means that the modified protocol satisfies this specification. Otherwise, NuSMV will produce a counterexample. The run of the NuSMV model is documented in Appendix E. It produces the following outputs:

1- The corresponding specifications of the properties (P1-P6) of the model, which were introduced in Subsection 6.5.1, are true. This means that the NuSMV model matches the model given in Subsection 6.5.2. Moreover, the corresponding specifications of the specifications of the additional propositions $C_i$, which are represented in $\sigma_c$, are also true. This means that $C_i$'s
behave in the NuSMV model in the way intended.

2- The corresponding specifications of the serializability conditions $\sigma_{sc_1}$ and $\sigma_{sc_2}$ are true if the transactions that participate in $\sigma_{sc_1}$ or $\sigma_{sc_2}$, are only local or only mobile transactions, as in Figure 6.5. But, if the transactions that participate in $\sigma_{sc_1}$ or $\sigma_{sc_2}$ are a mixture of local and mobile transactions, the specifications are false; see Figure 6.6 and Figure 6.7.

In Figure 6.6, the counterexample shows that if a local transaction begins its execution ($T_1 = r_{1x_1}$), it is possible, at the next point in time, that a mobile transaction begins its execution ($T_m = r_{mx_1}$). This corresponds to the following history $h$:

\[
h = \ldots r_i(x_1)r_{i'}(x_1)w_{i'}(x_1)\ldots r_{i'}(x_p)w_{i'}(x_p)\ldots w_i(x_1)\ldots
\]

where $T_i$ is a local transaction, $T_{i'}$ is a mobile transaction and $D_i \cap D_{i'} = \{x_1, \ldots, x_p\}$. In $h$, once the write operation of the transaction $T_i$ on the data item $x_1$ ($w_i(x_1)$) occurs, we will have a cycle between $T_i$ and $T_{i'}$ on the same data item as in Figure 5.1(a). This makes the history generated by the modified protocol not serializable any more. Also, in Figure 6.7, the counterexample shows that if a local transaction ($T_2$) precedes a mobile transaction ($T_m$) on a data item at some point in time ($T_2 = w_{2x_2}$ and $T_m = begin_m$), it is possible for the mobile transaction to precede the local transaction on another data item in $D_2 \cap D_3$. This corresponds to the following history $h_1$:

\[
h_1 = \ldots r_i(x_2)w_i(x_2)\ldots r_{i'}(x_1)\ldots r_{i'}(x_3)w_{i'}(x_3)\ldots w_{i'}(x_p)\ldots r_i(x_3)w_i(x_3)\ldots
\]

where $T_i$ is a local transaction, $T_{i'}$ is a mobile transaction and $D_i \cap D_{i'} = \{x_2, \ldots, x_p\}$. In $h_1$, once $T_i$ executes the read (or write) operation on the data item $x_3$ ($r_i(x_3)$ or $w_i(x_3)$), we will have a cycle between $T_i$ and $T_{i'}$ on the different data items, as in Figures 5.1(b) and 5.1(c), and therefore the history generated by the modified protocol is not serializable any more. The reason that non-serializable histories such as $h$ and $h_1$ above are produced, is that our assumption says that once a mobile transaction comes to the scheduler, all local transactions should suspend until there are no mobile transactions in the scheduler.
Chapter 6. Specification and Verification of Mobile Transactions with Priority

Figure 6.5: $\sigma_{sc1}$ and $\sigma_{sc2}$ hold, if both transactions are of the same type.

3- The corresponding specifications of $\sigma_{inf}$, specifying iterations of the transactions infinitely many times, are true. This means that the transactions in the model will indeed keep iterating infinitely many times. Moreover, the specification which asserts the equivalence of $\sigma_{inf}$ and $\sigma_{inf1}$ is also true.

6.9 Verification of starvation freedom

From the results of the verifications in the previous section, we notice that it is possible, according to the modified protocol, to have a situation where local transactions execute some read and write operations and then a mobile transaction starts its execution while the local transactions wait for the mobile transaction to finish. This can be depicted as follows:

$$h = \ldots r_{i1}(x_1) \ldots w_{i2}(x_3) \ldots r_{q1}(x_1) \ldots w_{i1}(x_p) \ldots w_{i2}(x_1) \ldots r_{i2}(x_4) \ldots$$

where $T_{i1}, T_{i2}$ are local transactions and $T_{q1}$ is a mobile transaction. Also, the verifications above show that transactions can iterate infinitely many times in the model. Hence, we can conceive of a situation where local transactions execute some read and write operations, and then a set of mobile transactions come to the
scheduler and keep iterating infinitely many times. This means that the local transactions cannot proceed an infinite period of time while the mobile transactions continue iterating normally in the system infinitely many times. This situation can be depicted as follows:

\[ h = \ldots \tau_1(x_1) \ldots w_1(x_3) \ldots (\tau_1(x_1) \ldots w_2(x_2) \ldots w_1(x_p) \ldots w_2(x_u))_1 \]

\[ (\tau_2(x_1) \ldots w_2(x_u) \ldots w_1(x_p))_2 \]

where \( \tau_1, \tau_2 \) are local transactions and \( \tau_1', \tau_2' \) are mobile transactions. If this situation sometimes occurs in the model, we will have what is called starvation. In [26], starvation may refer to a situation where a process waits for an event that may never occur, starvation or indefinite blocking refers to a situation where processes wait indefinitely, or starvation refers to a situation where processes continue to run indefinitely but fail to make any progress. The definition of starvation in the context of transactions processing systems, is that a transaction cannot continue for an indefinite amount of time while others proceed normally [22]. We specify (in our NuSMV model) that the above situation will not happen in a history produced by the modified protocol, to make sure that the protocol is starvation-free. If not, the model checker will produce a counterexample where this situation will happen eventually, supporting our suspicion that starvation does occur. This is specified in \( \sigma_s \) as follows:

\[
\sigma_s = \bigwedge_{i,j \geq 1, i \neq j, x \in D_i \atop T_i \in L, T_j \in T_m} (G \neg [w_i(x)] \land X (GF((begin_j \implies Fend_j) \land (end_j \implies Fbegin_j))) \land w_i(x))).
\]

The corresponding specification of \( \sigma_s \) in NuSMV model is in lines 339 and 340. We run the NuSMV model, without imposing fairness constraints, to verify the truth or falsity of the specification and the output is in Figure 6.8. Actually, we remove the fairness constraints, that are specified in lines 103-118, to make sure that the NuSMV model checker will produce a relevant counterexample. Otherwise (with fairness constraints imposed), NuSMV will produce a counterexample which contains all operations of the local transactions \( T_1 \) and \( T_2 \). The reason is that the fairness constraints restrict attention of the model checker to only those execution paths that every operation in \( T_1 \) and \( T_2 \) is true infinitely often. But, we want the model checker to consider all possible paths, as one or more of them may represent starvation. The counterexample in Figure 6.8 shows that the starvation
can happen. Also, the counterexample in Figure 6.8 shows a case where the mobile transaction \( T_3 \) keep iterating in the system (State: \( T_4 - State: T_2 \)) and the local transaction \( T_1 \) is waiting to resume. Note that in the printout of an infinite counterexample in NuSMV, the starting point of the loop is marked by -- loop starts here [11]. This results asserts that the modified protocol is not starvation free.

Now, assume that we add new accessing rules to the modified protocol to make sure that the produced histories are serializable. For example, let us add accessing rules which say that when a mobile transaction comes to the scheduler and there is a local transaction which has executed some of its operations on data items, and these operations may conflict with the incoming operations of the mobile transaction, then the mobile transaction should suspend until the local transaction finishes its execution. After that, the mobile transaction starts its execution again. Meanwhile, all other local transactions should be suspended. This can be explained as follows. Let \( D_1, D_2 \) and \( D_m \) be the sets of data items:

\[
D_1 = \{x_1, x_2\} \\
D_2 = \{x_3, x_4, x_5\} \\
D_m = \{x_1, x_2, x_3\}
\]

accessed by \( T_1, T_2 \) and \( T_m \) respectively. Suppose that \( T_1 \) and \( T_2 \) are local transactions and \( T_m \) is a mobile transaction. Consider the following serializable history \( h \) which shows the effect of the new accessing rules:

\[
h = \ldots w_1(x_1)r_2(x_3)w_2(x_3)r_1(x_2)w_1(x_2)r_m(x_1)w_m(x_1)\ldots r_m(x_3)w_m(x_3)r_2(x_4)\ldots \\
\quad \uparrow \\
\quad t_a \quad T_m \text{ waiting for } T_1 \text{ to finish} \\
\quad \uparrow \\
\quad t_b \quad \text{no local transaction can proceed} \\
\quad \uparrow \\
\quad t_c
\]

In the above history \( h \), mobile transaction \( T_m \) comes to the scheduler at time \( t_a \). Local transaction \( T_1 \) has executed on the data item \( x_1 \) beforehand. At that point in time, assume that we allow the mobile transaction \( T_m \) to execute its operations. This will make a cycle of length 2 between \( T_1 \) and \( T_m \) when \( T_1 \) executes on the data item \( x_2 \). So, we suspend mobile transaction \( T_m \) until time \( t_b \) when local transaction \( T_1 \) has finished its execution. Meanwhile, between the times \( t_a \) and \( t_b \), another local transaction \( T_2 \) comes to the scheduler and executes its read and write operations on the data item \( x_3 \). At time \( t_b \), the mobile transaction \( T_m \) starts its execution because there is no local transaction that has executed some of
its operations on data items and these operations may conflict with its operations. Between the times $t_b$ and $t_c$, no local transaction can proceed. After time $t_c$, local transaction $T_2$ resumes again.

Even if these new accessing rules produce serializable histories, they will not make the protocol starvation free. The reason is that the protocol allows mobile transactions to keep iterating infinitely many times in the scheduler while local transactions are waiting to resume. This reason is the same as the one given in Figure 6.8. This means that the new accessing rules do not remove the situation which produces the starvation. For example, in the above history $h$, assume that, between times $t_b$ and $t_c$, more mobile transactions come to the scheduler and keep iterating infinitely many times. This means that local transaction $T_2$ should be suspended for an indefinite amount of time thereby producing starvation. Finally, we notice that the little modification made on the protocol, produces both non-serializable histories and also causes starvation.
Figure 6.6: $\sigma_{sq1}$ does not hold.
Chapter 6. Specification and Verification of Mobile Transactions with Priority

-- specification G (p23 -> (( F (T2 = w2x3 & F Tm = wmx3))
    U !p23)) is false

-- as demonstrated by the following execution sequence

Trace Description: LTL Counterexample
Trace Type: Counterexample

-> State: 2.1 <-
  T1 = begin_1
  T2 = begin_2
  Tm = begin_m
  p12 = 0
  p13 = 0
  p23 = 0
  c1 = 0
  c2 = 0

-> Input: 2.16 <-
  _process_selector_ = T_2
  -> State: 2.16 <-
  T2 = w2x2
  c2 = 1

-> Input: 2.32 <-
  _process_selector_ = T_3
  -> State: 2.32 <-
  Tm = rmx1
  c1 = 0

-> Input: 2.42 <-
  _process_selector_ = T_3
  -> State: 2.42 <-
  Tm = end_m

-> Input: 2.53 <-
  _process_selector_ = T_2
  -> State: 2.53 <-
  T2 = w2x3

Figure 6.7: $\sigma_{sc_2}$ does not hold.
Figure 6.8: \( \sigma_z \) does not hold.
Chapter 7

Conclusion

Most database concurrency control techniques deal with a finite number of transactions executing concurrently in schedulers. Recent advances in portable devices and wireless communications (mobile computing) allow for unlimited numbers of concurrent users executing transactions. The verification techniques used to determine the correctness of such traditional database systems are mathematical proofs. These have several disadvantages such as the possibility of human error, they may not cover all system behaviours in a mobile setting, and require special expertise in mathematics. An automatic verification approach such as ours, overcomes these disadvantages. Our approach models an unlimited number of transactions incoming and outgoing from a system. We use a model checking approach that is based on temporal logic. This approach allows for the systems and their properties to be specified using temporal logics. The verification part is performed by checking exhaustively the state space of the system behaviours against the specifications. If the system does not satisfy its properties, counterexamples to pinpoint the errors are produced.

In this thesis, we assume serializability to be the correctness condition for concurrent transactions executing in a transactions processing system. We prove that the acyclicity of conflict graphs correspond to serializability in the case of an infinite number of transactions accessing data items in some finite set. We give a simpler correspondence in the case where transactions access the same set of data items. We define and prove conditions for multi-step transactions, that iterate infinitely many times in a system and access the same set of data items in possibly different orders, to ensure serializability. We find that serializability holds for a schedule of such multi-step transactions, if and only if any two transactions, participating in the schedule, satisfy the serializability conditions. The significance of this is that we can specify and verify serializability conditions in CTL or LTL in an easy and efficient way. The process is one of specifying a scheduler by a transition system and the serializability condition by temporal logic. Then, an
automatic verification is performed using a model checker, to see whether the scheduler satisfies the serializability condition.

We consider a new case of multi-step transactions accessing different sets of data items, where each transaction accesses a contiguous subset of data items of an ordered set of all data items. We prove that if the corresponding conflict graph of a schedule has a cycle, there is a cycle of length two. We make use of this result to prove that serializability for a schedule of such multi-step transactions can be achieved, if and only if any two transactions, participating in the schedule, satisfy a serializability condition.

In order to apply the aforementioned ideas, we give a protocol, based on timestamps, to be a concurrency control criterion suitable for multi-step transactions accessing ordered sets of data items. The behaviour of the protocol is specified as a finite state machine in the input language of the NuSMV model checker. Then, we use LTL to encode the serializability condition for the multi-step transactions. We find that LTL is a good choice to specify a protocol based on timestamps because any state in LTL path could implicitly represent a timestamp. Actually, we encode the behaviour of the protocol in terms of three transactions. Two of them represent any two transactions satisfying the accessing rules of the protocol, and the remaining one represents any other transaction in the schedule. We find that the proposed protocol satisfies the serializability condition. This means that we prove that the schedules, produced by the protocol, are serializable.

Furthermore, we encode the specifications for the protocol and serializability condition of the multi-step transactions into CTL in order to compare between LTL and CTL in terms of suitability for such protocols. Because CTL considers time to be branching, at each moment there are several different possible futures. Therefore, we have to define timestamps explicitly for each active transaction and data item. We find that LTL verification is more efficient for the proposed protocol. The reason is that the explicit timestamps defined in CTL, greatly increase the number of states in the state space and this affects model checking efficiency.

We extend the LTL and CTL models to include transactions iterating infinitely many times to represent a real situation. We find that the LTL model needs a simple modification to achieve this. The CTL model needs much effort to achieve the same. Moreover, we find that CTL verification does not achieve our goal of modelling transactions iterating infinitely many times. The reason is that we define timestamps as finite and the real situation requires timestamps to be infinite and it is impossible in any model checker to define a variable with an infinite number of values. The comparison between specifications and verifications using CTL and LTL, in the context of protocols based on timestamps, is not a simple tradeoff between complexity and expressiveness. It is a more complex matter of suitability.
for the problem.

In the culminating study of this thesis, we modify the proposed protocol to satisfy real-world requirements which occur in mobile computing environments. The modified protocol considers two types of transactions mobile and local transactions, and gives mobile transactions higher priority than for local transactions. This modification is justified in that mobile transactions may transiently disconnect from the network due to communication disruption or to save power. We build a model for the modified protocol and transactions using the \textit{NuSMV} input language. Also, we encode the serializability condition and the properties of the model into LTL. We find, after performing the verification, that the modified protocol is not serializable. The counterexamples, that are produced by model checkers, show that giving mobile transactions higher priority over local transactions makes the histories produced by the protocol not serializable and may also lead to starvation in the local transactions.

**The main contributions to knowledge:**

1. We prove that acyclicity of conflict graphs correspond to serializability in the case of an infinite number of multi-step transactions accessing data items in some finite set.

2. We prove a simpler correspondence, in the case where transactions access the same set of data items, such that if the corresponding conflict graph of a schedule has a cycle, there is a cycle of length two.

3. We make use of the previous results to define and prove the main result which gives the conditions for serializability that will form the basis of the specification in CTL or LTL. This result is very useful for improving the performance of model checking i.e. we reduce the time complexity of checking $n$ transactions ($O(n!)$) to two transactions ($O(n^2)$). Moreover, encoding the serializability conditions using two transactions make the length of the specification formula $|f|$ much less than encoding serializability conditions using $n$ transactions. This means that the model checker needs much less time to verify the specification formula for two transactions than to verify the specification formula for $n$ transactions.

4. We consider a new case of multi-step transactions accessing different sets of data items, where each transaction accesses a contiguous subset of data items of an ordered set of all data items. We prove that it is sufficient to check the serializability condition for every two transactions, participating in a schedule, to make sure that the schedule is serializable.
5. We give a protocol, based on timestamps, to be a concurrency control criterion suitable for multi-step transactions accessing ordered sets of data items.

6. We compare the suitability of LTL and CTL for specifying and verifying such protocols.

7. We modify the proposed protocol to satisfy real-world requirements which occur in mobile computing environments.

8. We consider the specification and verification of (the absence of) starvation for this kind of problem.

7.1 Future work

Further work will look into other different assumptions on transactions that have wide applicability in the real-world. Moreover, we look to specify and verify a broader range of conditions based on serializability, for example cases where non-serializability can be tolerated in certain states in order to improve the performance of transactions in mobile environments.
List of publications:


References


REFERENCES


REFERENCES


REFERENCES


Appendix A

NuSMV model for multi-step transactions with LTL specification

1 MODULE move(T_c,n,T0,T1)
2 ASSIGN
3 next(T_c):=case
4 T_c=begin_1 &n=1&(!T_c=r1x1) &(!(T1=roxi)) : r1x1;
5 T_c=r1x1 &n=1 : w1x1;
6 T_c=w1x1 &n=1&((!(T_c=r1x2)) &(!(T0=r2x2)) &&(T1=roxi)) : r1x2;
7 T_c=r1x2 &n=1 : w1x2;
8 T_c=w1x2 &n=1&((!(T_c=r1x3)) &(!(T0=r2x3)) &&(T1=roxi)) : r1x3;
9 T_c=r1x3 &n=1 : w1x3;
10 T_c=w1x3 &n=1&end_1;
11 T_c=begin_2 &n=2&(!(T0=r1x2)) &(!(T_c=r2x2)) &&(T1=roxi)) : r2x2;
12 T_c=r2x2 &n=2 : w2x2;
13 T_c=w2x2 &n=2&(!(T0=r1x3)) &(!(T_c=r2x3)) &&(T1=roxi)) : r2x3;
14 T_c=r2x3 &n=2 : w2x3;
15 T_c=w2x3 &n=2&(!(T_c=r2x4)) &(!(T1=roxi)) : r2x4;
16 T_c=r2x4 &n=2 : w2x4;
17 T_c=w2x4 &n=2&end_2;
18 T_c=begin_0 &n=3&(!(T0=r1x1)) &(!(T_c=roxi)) : roxi;
19 T_c=roxi &n=3 : woxi;
Chapter A. NuSMV model for multi-step transactions with LTL specification 135

24 T_c=wox1  \&n=3&( (!(T0=r1x2)) &(!(T1=r2x2)) &(!(T_c=rox2)): rox2;
25 T_c=rox2  \&n=3 : wox2;
26 T_c=wox2  \&n=3&( (!(T0=r1x3)) &(!(T1=r2x3)) &(!(T_c=rox3)): rox3;
27 T_c=rox3  \&n=3 : wox3;
28 T_c=wox3  \&n=3&( (!(T0=r2x4)) &(!(T_c=rox4)): rox4;
29 T_c=rox4  \&n=3 : wox4;
30 T_c=wox4  \&n=3&( (!(T0=r2x4)) &(!(T_c=rox5)): rox5;
31 T_c=rox5  \&n=3 : wox5;
32 T_c=wox5  \&n=3 : end_o;
33 Lc=rox5  \&n=3 wox5;
34 1 T_c: end_o;
36 esac;
37 MODULE main
38 VAR
39 T1 : {begin_1,r1x1,w1x1,r1x2,w1x2,r1x3,w1x3,end_1};
40 T2 : {begin_2,r2x2,w2x2,r2x3,w2x3,r2x4,w2x4,end_2};
41 Toth: {begin_o,rox1,wox1,rox2,wox2,rox3,wox3,rox4,wox4,
42 rox5,wox5,end_o};
43 T_1 : process move(T1,1,T2,Toth);
44 T_2 : process move(T2,2,T1,Toth);
45 T_3 : process move(Toth,3,T1,T2);
46 ASSIGN
47 init(T1) :=begin_1;
48 init(T2) :=begin_2;
49 init(Toth) :=begin_o;
50 ----------------------------------------------
51 FAIRNESS (T1=end_1)
52 FAIRNESS (T2=end_2)
53 FAIRNESS (Toth=end_o)
54 --SPECIFICATIONS
55 ------read/write alternation ------
56 --For T1
57 LTLSPEC G (((T1=r1x2) & 0(T1=r1x1)) -> 0(T1=w1x1))
58 LTLSPEC G (((T1=r1x3) & 0(T1=r1x1)) -> 0(T1=w1x1))
59 LTLSPEC G (((T1=r1x3) & 0(T1=r1x2)) -> 0(T1=w1x2))
60 --For T2
61 LTLSPEC G (((T2=r2x3) & 0(T2=r2x2)) -> 0(T2=w2x2))
62 LTLSPEC G (((T2=r2x4) & 0(T2=r2x2)) -> 0(T2=w2x2))
Chapter A. NuSMV model for multi-step transactions with LTL specification

63 LTLSPEC G (((T2=r2x4) & 0(T2=r2x3)) -> 0(T2=w2x3))
64 --For Toth
65 LTLSPEC G (((Toth=rox2) & 0(Toth=rox1)) -> 0(Toth=wox1))
66 LTLSPEC G (((Toth=rox3) & 0(Toth=rox1)) -> 0(Toth=wox1))
67 LTLSPEC G (((Toth=rox4) & 0(Toth=rox1)) -> 0(Toth=wox1))
68 LTLSPEC G (((Toth=rox5) & 0(Toth=rox1)) -> 0(Toth=wox1))
69 LTLSPEC G (((Toth=rox3) & 0(Toth=rox2)) -> 0(Toth=wox2))
70 LTLSPEC G (((Toth=rox4) & 0(Toth=rox2)) -> 0(Toth=wox2))
71 LTLSPEC G (((Toth=rox5) & 0(Toth=rox2)) -> 0(Toth=wox2))
72 LTLSPEC G (((Toth=rox4) & 0(Toth=rox3)) -> 0(Toth=wox3))
73 LTLSPEC G (((Toth=rox5) & 0(Toth=rox3)) -> 0(Toth=wox3))
74 LTLSPEC G (((Toth=rox5) & 0(Toth=rox4)) -> 0(Toth=wox4))
75 ------read/write alternation in future -------
76 --For T1
77 LTLSPEC G (T1=r1x1 -> (F (T1=w1x1 & F (T1=r1x2))))
78 LTLSPEC G (T1=r1x2 -> (F (T1=w1x2 & F (T1=r1x3))))
79 LTLSPEC G (T1=r1x1 -> (F (T1=w1x1 & F (T1=r1x3))))
80 --For T2
81 LTLSPEC G (T2=r2x2 -> (F (T2=w2x2 & F (T2=r2x3))))
82 LTLSPEC G (T2=r2x3 -> (F (T2=w2x3 & F (T2=r2x4))))
83 LTLSPEC G (T2=r2x2 -> (F (T2=w2x2 & F (T2=r2x4))))
84 --For Toth
85 LTLSPEC G (Toth=rox1 ->(F (Toth=wox1 & F (Toth=rox2))))
86 LTLSPEC G (Toth=rox1 ->(F (Toth=wox1 & F (Toth=rox3))))
87 LTLSPEC G (Toth=rox1 ->(F (Toth=wox1 & F (Toth=rox4))))
88 LTLSPEC G (Toth=rox1 ->(F (Toth=wox1 & F (Toth=rox5))))
89 LTLSPEC G (Toth=rox2 ->(F (Toth=wox2 & F (Toth=rox3))))
90 LTLSPEC G (Toth=rox2 ->(F (Toth=wox2 & F (Toth=rox4))))
91 LTLSPEC G (Toth=rox2 ->(F (Toth=wox2 & F (Toth=rox5))))
92 LTLSPEC G (Toth=rox3 ->(F (Toth=wox3 & F (Toth=rox4))))
93 LTLSPEC G (Toth=rox3 ->(F (Toth=wox3 & F (Toth=rox5))))
94 LTLSPEC G (Toth=rox4 ->(F (Toth=wox4 & F (Toth=rox5))))
95 ------write implies Read -------------
96 --For T1
97 LTLSPEC G (((T1=w1x1) -> 0(T1=r1x1))
98 LTLSPEC G (((T1=w1x2) -> 0(T1=r1x2))
99 LTLSPEC G (((T1=w1x3) -> 0(T1=r1x3))
100 --For T2
101 LTLSPEC G (((T2=w2x2) -> 0(T2=r2x2))
Chapter A. NuSMV model for multi-step transactions with LTL specification

102 LTLSPEC G ((T2=w2x3) -> O((T2=r2x3)))
103 LTLSPEC G ((T2=w2x4) -> O((T2=r2x4)))
104 --For Toth
105 LTLSPEC G ((Toth=wox1) -> O((Toth=rox1)))
106 LTLSPEC G ((Toth=wox2) -> O((Toth=rox2)))
107 LTLSPEC G ((Toth=wox3) -> O((Toth=rox3)))
108 LTLSPEC G ((Toth=wox4) -> O((Toth=rox4)))
109 LTLSPEC G ((Toth=wox5) -> O((Toth=rox5)))
110 -------write implies Read future---------
111 --For T1
112 LTLSPEC G (T1=r1x1 -> (F (T1=w1x1 )))
113 LTLSPEC G (T1=r1x2 -> (F (T1=w1x2 )))
114 LTLSPEC G (T1=r1x3 -> (F (T1=w1x3 )))
115 --For T2
116 LTLSPEC G (T2=r2x2 -> (F (T2=w2x2 )))
117 LTLSPEC G (T2=r2x3 -> (F (T2=w2x3 )))
118 LTLSPEC G (T2=r2x4 -> (F (T2=w2x4 )))
119 --For Toth
120 LTLSPEC G (Toth=rox1 -> (F (Toth=wox1 )))
121 LTLSPEC G (Toth=rox2 -> (F (Toth=wox2 )))
122 LTLSPEC G (Toth=rox3 -> (F (Toth=wox3 )))
123 LTLSPEC G (Toth=rox4 -> (F (Toth=wox4 )))
124 LTLSPEC G (Toth=rox5 -> (F (Toth=wox5 )))
125 --proposition remains true until the next operation
126 --For T1
127 LTLSPEC G ((T1=w1x1) -> O((T1=r1x1) U (T1=w1x1)))
128 LTLSPEC G ((T1=w1x2) -> O((T1=r1x2) U (T1=w1x2)))
129 LTLSPEC G ((T1=w1x3) -> O((T1=r1x3) U (T1=w1x3)))
130 --
131 LTLSPEC G ((T1=r1x2) -> O((T1=w1x1) U (T1=r1x2)))
132 LTLSPEC G ((T1=r1x3) -> O((T1=w1x2) U (T1=r1x3)))
133 --For T2
134 LTLSPEC G ((T2=w2x2) -> O((T2=r2x2) U (T2=w2x2)))
135 LTLSPEC G ((T2=w2x3) -> O((T2=r2x3) U (T2=w2x3)))
136 LTLSPEC G ((T2=w2x4) -> O((T2=r2x4) U (T2=w2x4)))
137 --
138 LTLSPEC G ((T2=r2x3) -> O((T2=w2x2) U (T2=r2x3)))
139 LTLSPEC G ((T2=r2x4) -> O((T2=w2x3) U (T2=r2x4)))
140 --For Toth
Chapter A. NuSMV model for multi-step transactions with LTL specification

LTLSPEC G ((Toth=wox1) -> 0((Toth=rox1) U (Toth=wox1)))
LTLSPEC G ((Toth=wox2) -> 0((Toth=rox2) U (Toth=wox2)))
LTLSPEC G ((Toth=wox3) -> 0((Toth=rox3) U (Toth=wox3)))
LTLSPEC G ((Toth=wox4) -> 0((Toth=rox4) U (Toth=wox4)))
LTLSPEC G ((Toth=wox5) -> 0((Toth=rox5) U (Toth=wox5)))

LTLSPEC G ((Toth=rox2) -> 0((Toth=wox1) U (Toth=rox2)))
LTLSPEC G ((Toth=rox3) -> 0((Toth=wox2) U (Toth=rox3)))
LTLSPEC G ((Toth=rox4) -> 0((Toth=wox3) U (Toth=rox4)))
LTLSPEC G ((Toth=rox5) -> 0((Toth=wox4) U (Toth=rox5)))

-- proposition remains true until the next operation

LTLSPEC G ((T1=w1x1) -> !(T1=r1x1))
LTLSPEC G ((T1=w1x2) -> !(T1=r1x2))
LTLSPEC G ((T1=w1x3) -> !(T1=r1x3))

LTLSPEC G ((T1=r1x2) -> !(T1=w1x1))
LTLSPEC G ((T1=r1x3) -> !(T1=w1x2))

LTLSPEC G ((T2=w2x2) -> !(T2=r2x2))
LTLSPEC G ((T2=w2x3) -> !(T2=r2x3))
LTLSPEC G ((T2=w2x4) -> !(T2=r2x4))

LTLSPEC G ((T2=r2x3) -> !(T2=w2x2))
LTLSPEC G ((T2=r2x4) -> !(T2=w2x3))

LTLSPEC G ((Toth=wox1) -> !(Toth=rox1))
LTLSPEC G ((Toth=wox2) -> !(Toth=rox2))
LTLSPEC G ((Toth=wox3) -> !(Toth=rox3))
LTLSPEC G ((Toth=wox4) -> !(Toth=rox4))
LTLSPEC G ((Toth=wox5) -> !(Toth=rox5))

LTLSPEC G ((Toth=rox2) -> !(Toth=wox1))
LTLSPEC G ((Toth=rox3) -> !(Toth=wox2))
LTLSPEC G ((Toth=rox4) -> !(Toth=wox3))
LTLSPEC G ((Toth=rox5) -> !(Toth=wox4))

-- At most one step occurs at each successive state

LTLSPEC G (!((T1=r1x2)&&(T2=r2x2)) & X((T1=r1x2)&&(T2=r2x2)))

-- For T1

-- For T2

-- For Toth

-- At most one step occurs at each successive state

-- T1 and T2
Chapter A. NuSMV model for multi-step transactions with LTL specification

180 \[ (((T_1=r_1x_2) \& (T_2=w_2x_2)) \& X((T_1=r_1x_2) \& (T_2=w_2x_2))) \\&
181 \[ (((T_1=w_1x_2) \& (T_2=w_2x_2)) \& X((T_1=w_1x_2) \& (T_2=w_2x_2))) \]
182 LTLSPEC G (((T_1=r_1x_3) \& (T_2=r_2x_3)) \& X((T_1=r_1x_3) \& (T_2=r_2x_3))) \&
183 \[ (((T_1=r_1x_3) \& (T_2=w_2x_3)) \& X((T_1=r_1x_3) \& (T_2=w_2x_3))) \&
184 \[ (((T_1=w_1x_3) \& (T_2=w_2x_3)) \& X((T_1=w_1x_3) \& (T_2=w_2x_3))) \]
185 --T_1 and Toth
186 LTLSPEC G (((T_1=r_1x_1) \& (Toth=rox_1))) \&
187 \[ X((T_1=r_1x_1) \& (Toth=rox_1))) \&
188 \[ (((T_1=r_1x_1) \& (Toth=wox_1))) \&
189 \[ X((T_1=r_1x_1) \& (Toth=wox_1))) \&
190 \[ (((T_1=w_1x_1) \& (Toth=wox_1))) \&
191 \[ X((T_1=w_1x_1) \& (Toth=wox_1))) \]
192 LTLSPEC G (((T_1=r_1x_2) \& (Toth=rox_2))) \&
193 \[ X((T_1=r_1x_2) \& (Toth=rox_2))) \&
194 \[ (((T_1=r_1x_2) \& (Toth=wox_2))) \&
195 \[ X((T_1=r_1x_2) \& (Toth=wox_2))) \&
196 \[ (((T_1=w_1x_2) \& (Toth=wox_2))) \&
197 \[ X((T_1=w_1x_2) \& (Toth=wox_2))) \]
198 LTLSPEC G (((T_1=r_1x_3) \& (Toth=rox_3))) \&
199 \[ X((T_1=r_1x_3) \& (Toth=rox_3))) \&
200 \[ (((T_1=r_1x_3) \& (Toth=wox_3))) \&
201 \[ X((T_1=r_1x_3) \& (Toth=wox_3))) \&
202 \[ (((T_1=w_1x_3) \& (Toth=wox_3))) \&
203 \[ X((T_1=w_1x_3) \& (Toth=wox_3))) \]
204 --Toth and T_2
205 LTLSPEC G (((Toth=rox_2) \& (T_2=r_2x_2))) \&
206 \[ X((T_1=rox_2) \& (T_2=r_2x_2))) \&
207 \[ (((Toth=rox_2) \& (T_2=w_2x_2))) \&
208 \[ X((T_1=rox_2) \& (T_2=w_2x_2))) \&
209 \[ (((Toth=wox_2) \& (T_2=w_2x_2))) \&
210 \[ X((T_1=wox_2) \& (T_2=w_2x_2))) \]
211 LTLSPEC G (((Toth=rox_3) \& (T_2=r_2x_3))) \&
212 \[ X((T_1=rox_3) \& (T_2=r_2x_3))) \&
213 \[ (((Toth=rox_3) \& (T_2=w_2x_3))) \&
214 \[ X((T_1=rox_3) \& (T_2=w_2x_3))) \&
215 \[ (((Toth=wox_3) \& (T_2=w_2x_3))) \&
216 \[ X((T_1=wox_3) \& (T_2=w_2x_3))) \]
217 LTLSPEC G (((Toth=rox_4) \& (T_2=r_2x_4))) \&
218 \[ X((T_1=rox_4) \& (T_2=r_2x_4))) \&
Chapter A. NuSMV model for multi-step transactions with LTL specification

\[ !((!(T_1=\text{t1x1})\&!(T_2=\text{t2x1}))\& \]
\[ X ( !(T_1=\text{t1x1})\&(T_2=\text{t2x1}))\& \]
\[ !((!(T_1=\text{t1x2})\&(T_2=\text{t2x2}))\& X ( !(T_1=\text{t1x2})\&(T_2=\text{t2x2}))) \]

223 --At most one step occurs at each successive state(past)--

224 --T1 and T2

225 LTLSPEC G ( !((!(T_1=\text{t1x2})\&(T_2=\text{t2x2}))\&
\[ Y ( !(T_1=\text{t1x2})\&(T_2=\text{t2x2}))\& \]
\[ !((!(T_1=\text{t1x2})\&(T_2=\text{t2x2}))\&
\[ Y ( !(T_1=\text{t1x2})\&(T_2=\text{t2x2}))\& \]
\[ !((!(T_1=\text{t1x3})\&(T_2=\text{t2x3}))\&
\[ Y ( !(T_1=\text{t1x3})\&(T_2=\text{t2x3}))\& \]
\[ !((!(T_1=\text{t1x3})\&(T_2=\text{t2x3}))\&
\[ Y ( !(T_1=\text{t1x3})\&(T_2=\text{t2x3}))\& \]

237 --T1 and Toth

238 LTLSPEC G ( !((!(T_1=\text{t1x1})\&(Toth=\text{t1x1}))\&
\[ Y ( !(T_1=\text{t1x1})\&(Toth=\text{t1x1}))\& \]
\[ !((!(T_1=\text{t1x1})\&(Toth=\text{t1x1}))\&
\[ Y ( !(T_1=\text{t1x1})\&(Toth=\text{t1x1}))\& \]
\[ !((!(T_1=\text{t1x3})\&(Toth=\text{t1x3}))\&
\[ Y ( !(T_1=\text{t1x3})\&(Toth=\text{t1x3}))\& \]

244 LTLSPEC G ( !((!(T_1=\text{t1x2})\&(Toth=\text{t1x2}))\&
\[ Y ( !(T_1=\text{t1x2})\&(Toth=\text{t1x2}))\& \]
\[ !((!(T_1=\text{t1x2})\&(Toth=\text{t1x2}))\&
\[ Y ( !(T_1=\text{t1x2})\&(Toth=\text{t1x2}))\& \]
\[ !((!(T_1=\text{t1x3})\&(Toth=\text{t1x3}))\&
\[ Y ( !(T_1=\text{t1x3})\&(Toth=\text{t1x3}))\& \]

250 LTLSPEC G ( !((!(T_1=\text{t1x3})\&(Toth=\text{t1x3}))\&
\[ Y ( !(T_1=\text{t1x3})\&(Toth=\text{t1x3}))\& \]
\[ !((!(T_1=\text{t1x3})\&(Toth=\text{t1x3}))\&
\[ Y ( !(T_1=\text{t1x3})\&(Toth=\text{t1x3}))\& \]
\[ !((!(T_1=\text{t1x3})\&(Toth=\text{t1x3}))\&
\[ Y ( !(T_1=\text{t1x3})\&(Toth=\text{t1x3}))\& \]

256 --Toth and T2

257 LTLSPEC G ( !((!(T_2=\text{t2x2})&(Toth=\text{t2x2}))\&
Chapter A. NuSMV model for multi-step transactions with LTL specification 141

258 \[ Y((T2=r2x2)! (Toth=rox2))) &
259 \[ !((T2=r2x2) & (Toth=wox2))) &
260 \[ Y((T2=r2x2)! (Toth=wox2))) &
261 \[ !((T2=w2x2) & (Toth=wox2))) &
262 \[ Y((T2=w2x2)! (Toth=wox2)))

263 LTLSPEC G (!((T2=r2x3) & (Toth=rox3))) &
264 \[ Y((T2=r2x3)! (Toth=rox3))) &
265 \[ !((T2=r2x3) & (Toth=wox3))) &
266 \[ Y((T2=r2x3)! (Toth=wox3))) &
267 \[ !((T2=w2x3) & (Toth=wox3))) &
268 \[ Y((T2=w2x3)! (Toth=wox3)))

269 LTLSPEC G (!((T2=r2x4) & (Toth=rox4))) &
270 \[ Y((T2=r2x4)! (Toth=rox4))) &
271 \[ !((T2=r2x4) & (Toth=wox4))) &
272 \[ Y((T2=r2x4)! (Toth=wox4))) &
273 \[ !((T2=w2x4) & (Toth=wox4))) &
274 \[ Y((T2=w2x4)! (Toth=wox4)))

275 -- A transaction Ti accesses x in Di exactly once

276 -- For T1
277 LTLSPEC G !((T1=r1x1 & F (T1!=r1x1 & F T1=r1x1)))
278 LTLSPEC G !((T1=w1x1 & F (T1!=w1x1 & F T1=w1x1)))
279 LTLSPEC G !((T1=r1x2 & F (T1!=r1x2 & F T1=r1x2)))
280 LTLSPEC G !((T1=w1x2 & F (T1!=w1x2 & F T1=w1x2)))
281 LTLSPEC G !((T1=r1x3 & F (T1!=r1x3 & F T1=r1x3)))
282 LTLSPEC G !((T1=w1x3 & F (T1!=w1x3 & F T1=w1x3)))

283 -- Past --
284 LTLSPEC G !((T1=r1x1 & 0 (T1!=r1x1 & 0 T1=r1x1)))
285 LTLSPEC G !((T1=w1x1 & 0 (T1!=w1x1 & 0 T1=w1x1)))
286 LTLSPEC G !((T1=r1x2 & 0 (T1!=r1x2 & 0 T1=r1x2)))
287 LTLSPEC G !((T1=w1x2 & 0 (T1!=w1x2 & 0 T1=w1x2)))
288 LTLSPEC G !((T1=r1x3 & 0 (T1!=r1x3 & 0 T1=r1x3)))
289 LTLSPEC G !((T1=w1x3 & 0 (T1!=w1x3 & 0 T1=w1x3)))

290 -- For T2
291 LTLSPEC G !((T2=r2x2 & F (T2!=r2x2 & F T2=r2x2)))
292 LTLSPEC G !((T2=w2x2 & F (T2!=w2x2 & F T2=w2x2)))
293 LTLSPEC G !((T2=r2x3 & F (T2!=r2x3 & F T2=r2x3)))
294 LTLSPEC G !((T2=w2x3 & F (T2!=w2x3 & F T2=w2x3)))
295 LTLSPEC G !((T2=r2x4 & F (T2!=r2x4 & F T2=r2x4)))
296 LTLSPEC G !((T2=w2x4 & F (T2!=w2x4 & F T2=w2x4)))
Chapter A. NuSMV model for multi-step transactions with LTL specification 142

297 --past--
298 LTLSPEC G !(T2=r2x2 & 0 (T2!=r2x2 & 0 T2=r2x2))
299 LTLSPEC G !(T2=w2x2 & 0 (T2!=w2x2 & 0 T2=w2x2))
300 LTLSPEC G !(T2=r2x3 & 0 (T2!=r2x3 & 0 T2=r2x3))
301 LTLSPEC G !(T2=w2x3 & 0 (T2!=w2x3 & 0 T2=w2x3))
302 LTLSPEC G !(T2=r2x4 & 0 (T2!=r2x4 & 0 T2=r2x4))
303 LTLSPEC G !(T2=w2x4 & 0 (T2!=w2x4 & 0 T2=w2x4))
304 --For Toth
305 LTLSPEC G !(Toth=rox1& F (Toth!=rox1& F Toth=rox1))
306 LTLSPEC G !(Toth=wox1& F (Toth!=wox1& F Toth=wox1))
307 LTLSPEC G !(Toth=rox2& F (Toth!=rox2& F Toth=rox2))
308 LTLSPEC G !(Toth=wox2& F (Toth!=wox2& F Toth=wox2))
309 LTLSPEC G !(Toth=rox3& F (Toth!=rox3& F Toth=rox3))
310 LTLSPEC G !(Toth=wox3& F (Toth!=wox3& F Toth=wox3))
311 LTLSPEC G !(Toth=rox4& F (Toth!=rox4& F Toth=rox4))
312 LTLSPEC G !(Toth=wox4& F (Toth!=wox4& F Toth=wox4))
313 LTLSPEC G !(Toth=rox5& F (Toth!=rox5& F Toth=rox5))
314 LTLSPEC G !(Toth=wox5& F (Toth!=wox5& F Toth=wox5))
315 --past--
316 LTLSPEC G !(Toth=rox1& 0 (Toth!=rox1& 0 Toth=rox1))
317 LTLSPEC G !(Toth=wox1& 0 (Toth!=wox1& 0 Toth=wox1))
318 LTLSPEC G !(Toth=rox2& 0 (Toth!=rox2& 0 Toth=rox2))
319 LTLSPEC G !(Toth=wox2& 0 (Toth!=wox2& 0 Toth=wox2))
320 LTLSPEC G !(Toth=rox3& 0 (Toth!=rox3& 0 Toth=rox3))
321 LTLSPEC G !(Toth=wox3& 0 (Toth!=wox3& 0 Toth=wox3))
322 LTLSPEC G !(Toth=rox4& 0 (Toth!=rox4& 0 Toth=rox4))
323 LTLSPEC G !(Toth=wox4& 0 (Toth!=wox4& 0 Toth=wox4))
324 LTLSPEC G !(Toth=rox5& 0 (Toth!=rox5& 0 Toth=rox5))
325 LTLSPEC G !(Toth=wox5& 0 (Toth!=wox5& 0 Toth=wox5))
326 ---------------------------------------------
327 ----no r1 and r2 on x without write for one of them--
328 --T1 and Toth
329 LTLSPEC G !(Toth=rox1 & T1=r1x1)
330 LTLSPEC G !(Toth=rox2 & T1=r1x2)
331 LTLSPEC G !(Toth=rox3 & T1=r1x3)
332 --T1 and T2
333 LTLSPEC G !(T1=r1x2 & T2=r2x2)
334 LTLSPEC G !(T1=r1x3 & T2=r2x3)
335 --T2 and Toth
Chapter A. NuSMV model for multi-step transactions with LTL specification

LTLSPEC  G ! (Toth=rox2 & T2=r2x2)
LTLSPEC  G ! (Toth=rox3 & T2=r2x3)
LTLSPEC  G ! (Toth=rox4 & T2=r2x4)

---------------------------------------------
--protocol
---------------------------------------------

--T1 precedes T2
LTLSPEC G (((Toth=wox3 & O(T2=w2x3 & O(T1=w1x3))) &
(Toth=wox2 & O(T2=w2x2 & O(T1=w1x2)))))

((T2=w2x3 & O(Toth=wox3 & O(T1=w1x3))) &
(T2=w2x2 & O(Toth=wox2 & O(T1=w1x2))))

((T2=w2x3 & O(T1=w1x3 & O(Toth=wox3))) &
(T2=w2x2 & O(T1=w1x2 & O(Toth=wox3)))))

----------Future---------------------
--T1 precedes T2
LTLSPEC G (((Toth=wox3 & O(T2=w2x3 & F(Toth=wox3))) &
(T1=w1x2 & F(T2=w2x2 & F(Toth=wox2)))))

((T1=w1x3 & O(Toth=wox3 & O(T2=w2x3))) &
(T1=w1x2 & O(Toth=wox2 & O(T2=w2x2))))

((T1=w1x3 & O(T2=w2x3 & O(Toth=wox3))) &
(T1=w1x2 & O(T2=w2x2 & O(Toth=wox3)))))

----------Future---------------------
--T2 precedes T1
LTLSPEC G (((Toth=wox3 & O(T1=w1x3 & O(T2=w2x3))) &
(Toth=wox2 & O(T1=w1x2 & O(T2=w2x2))))

((T2=w2x3 & O(T1=w1x3 & O(Toth=wox3))) &
(T2=w2x2 & O(T1=w1x2 & O(Toth=wox3)))))

----------Future---------------------
--T2 precedes T1
LTLSPEC G (((Toth=wox3 & O(T1=w1x3 & O(T2=w2x3))) &
(T1=w1x2 & O(T2=w2x2 & O(Toth=wox3))))

((T1=w1x3 & F(Toth=wox3 & F(T2=w2x3))) &
(T1=w1x2 & F(Toth=wox2 & F(T2=w2x2))))

((Toth=wox3 & F(T1=w1x3 & F(T2=w2x3))) &
Chapter A. NuSMV model for multi-step transactions with LTL specification

375 \((\text{Toth}=\text{wox}2 & F(\text{T1}=\text{w1x}2 & F(\text{T2}=\text{w2x}2))))\)
376 \rightarrow
377 !( (\text{T1}=\text{w1x}2 & F(\text{T2}=\text{w2x}2 & F(\text{T2}=\text{w2x}3 & F (\text{T1}=\text{w1x}3)))) | \\
378 (\text{T2}=\text{w2x}2 & F(\text{T1}=\text{w1x}2 & F(\text{T1}=\text{w1x}3 & F (\text{T2}=\text{w2x}3)))))) \\
379 \quad \text{---T2 precedes T1} \\
380 \text{LTLSPEC G } (((\text{T2}=\text{w2x}3 & F(\text{T1}=\text{w1x}3 & F(\text{Toth}=\text{wox}3))) & \\
381 (\text{T2}=\text{w2x}2 & F(\text{T1}=\text{w1x}2 & F(\text{Toth}=\text{wox}2)))) \\
382 | \\
383 ((\text{T2}=\text{w2x}3 & F(\text{Toth}=\text{wox}3 & F(\text{T1}=\text{w1x}3))) & \\
384 (\text{T2}=\text{w2x}2 & F(\text{Toth}=\text{wox}2 & F(\text{T1}=\text{w1x}2)))) \\
385 | \\
386 ((\text{Toth}=\text{wox}3 & F(\text{T2}=\text{w2x}3 & F(\text{T1}=\text{w1x}3)))) & \\
387 (\text{Toth}=\text{wox}2 & F(\text{T2}=\text{w2x}2 & F(\text{T1}=\text{w1x}2)))) \\
388 \rightarrow \\
389 !( (\text{T1}=\text{w1x}2 & F(\text{T2}=\text{w2x}2 & F(\text{T2}=\text{w2x}3 & F (\text{T1}=\text{w1x}3)))) | \\
390 (\text{T2}=\text{w2x}2 & F(\text{T1}=\text{w1x}2 & F(\text{T1}=\text{w1x}3 & F (\text{T2}=\text{w2x}3))))))
391 \text{ FAIRNESS running}
Appendix B

NuSMV model for multi-step transactions with CTL specification

1 MODULE move(T_c,n,T0,T1,TSM,p_c)
2 ASSIGN
3 next(T_c):=case
4 T_c=begin_1 & n=1&(((TSM[1][1]=0)|(TSM[1][1]!=0&!(T_c=r1x1)))
5 &((TSM[2][1]=0)
6 &((TSM[3][1]=0)|(TSM[3][1]!=0&!(T_c=roxl1))))
7 T_c=r1x1 & n=1 : w1x1;
8 T_c=w1x1 & n=1&(((TSM[1][2]=0)|(TSM[1][2]!=0&!(T_c=r1x2)))
9 &((TSM[2][2]=0)|(TSM[2][2]!=0&!(T0=r2x2)))
10 &((TSM[3][2]=0)|(TSM[3][2]!=0&!(T1=roxl2)))&
12 T_c=r1x2 & n=1 : w1x2;
13 T_c=w1x2 & n=1&(((TSM[1][3]=0)|(TSM[1][3]!=0& ! (T_c=r1x3)))
14 &((TSM[2][3]=0)|(TSM[2][3]!=0&!(T0=r2x3)))
15 &((TSM[3][3]=0)|(TSM[3][3]!=0&!(T1=roxl3)))&
18 T_c=r1x3 & n=1 : w1x3;
19 T_c=w1x3 & n=1 : end_1;
20 T_c=begin_2 & n=2&(((TSM[1][2]=0)|(TSM[1][2]!=0&!(T0=r1x2)))
21 &((TSM[2][2]=0)|(TSM[2][2]!=0&!(T_c=r2x2)))
22 &((TSM[3][2]=0)|(TSM[3][2]!=0&!(T1=roxl2))) : r2x2;
23 T_c=r2x2 & n=2 : w2x2;
Chapter B. NuSMV model for multi-step transactions with CTL specification

24 T_c=w2x2 & n=2&((TSM[1][3]=0) | (TSM[1][3]=0 & ! (T0=r1x3)))
25 &((TSM[2][3]=0) | (TSM[2][3]=0 & ! (T_c=r2x3)))
26 &((TSM[3][3]=0) | (TSM[3][3]=0 & ! (T1=rox3)))) &
29 T_c=r2x3 & n=2: w2x3;
30 T_c=w2x3 & n=2&((TSM[1][4]=0)--does not execute x4
31 &((TSM[2][4]=0) | (TSM[2][4]=0 & ! (T_c=r2x4)))
32 &((TSM[3][4]=0) | (TSM[3][4]=0 & ! (T1=rox4)))) &
34 T_c=r2x4 & n=2: w2x4;
35 T_c=w2x4 & n=2: end_2;
36 T_c=begin_o & n=3&((TSM[1][1]=0) | (TSM[1][1]=0 & ! (T0=r1x1)))
37 &((TSM[2][1]=0)
38 &((TSM[3][1]=0) | (TSM[3][1]=0 & ! (T_c=rox1))))
39 (! (TSM[1][1]>0 & TSM[1][1]<TSM[3][1] & TSM[1][2]=0)):rox1;
40 T_c=rox1 & n=3: wox1;
41 T_c=wox1 & n=3&((TSM[1][2]=0) | (TSM[1][2]=0 & ! (T0=r1x2)))
42 &((TSM[2][2]=0) | (TSM[2][2]=0 & ! (T1=r2x2)))
43 &((TSM[3][2]=0) | (TSM[3][2]=0 & ! (T_c=rox2)))):rox2;
44 T_c=rox2 & n=3: wox2;
45 T_c=wox2 & n=3&((TSM[1][3]=0) | (TSM[1][3]=0 & ! (T0=r1x3)))
46 &((TSM[2][3]=0) | (TSM[2][3]=0 & ! (T1=r2x3) ))
47 &((TSM[3][3]=0) | (TSM[3][3]=0 & ! (T_c=rox3)))) &
50 T_c=rox3 & n=3: wox3;
51 T_c=wox3 & n=3&((TSM[1][4]=0)--does not execute x4
52 &((TSM[2][4]=0) | (TSM[2][4]=0 & ! (T1=r2x4)))
53 &((TSM[3][4]=0) | (TSM[3][4]=0 & ! (T_c=rox4)))) &
55 T_c=rox4 & n=3: wox4;
56 T_c=wox4 & n=3& (TSM[1][5]=0)--does not execute x5
57 &((TSM[2][5]=0) &
58 &((TSM[3][5]=0) | (TSM[3][5]=0 & ! (T_c=rox5)))):rox5;
59 T_c=rox5 & n=3: wox5;
60 T_c=wox5 & n=3: end_o;
61 esac;
next(p_c) := case
  p_c>=0 & p_c<11 & ((T_c=begin_1 & next(T_c)=r1x1))
    | (T_c=w1x1 & next(T_c)=r1x2)
    | (T_c=w1x2 & next(T_c)=r1x3)
    | (T_c=begin_2 & next(T_c)=r2x2)
    | (T_c=w2x2 & next(T_c)=r2x3)
    | (T_c=w2x3 & next(T_c)=r2x4)
    | (T_c=begin_0 & next(T_c)=r0x1)
    | (T_c=w0x1 & next(T_c)=r0x2)
    | (T_c=w0x2 & next(T_c)=r0x3)
    | (T_c=w0x3 & next(T_c)=r0x4)
    | (T_c=w0x4 & next(T_c)=r0x5): p_c+1;
  1 : p_c;
esac;

next(TSM[1][1]) := case
  T_c! = r1x1 & next(T_c)=r1x1 & n=1:next(p_c);
  next(T_c)= end_1 : 0;
  1 : TSM[1][1];
esac;

next(TSM[1][2]) := case
  T_c! = r1x2 & next(T_c)=r1x2 & n=1:next(p_c);
  next(T_c)= end_1 : 0;
  1 : TSM[1][2];
esac;

next(TSM[1][3]) := case
  T_c! = r1x3 & next(T_c)=r1x3 & n=1:next(p_c)
  next(T_c)= end_1 : 0;
  1 : TSM[1][3];
esac;

next(TSM[1][4]) := 0;
next(TSM[1][5]) := 0;

next(TSM[2][1]) := 0;
next(TSM[2][2]) := case
  T_c! = r2x2 & next(T_c)=r2x2 & n=2:next(p_c);
  next(T_c)= end_2 : 0;
  1 : TSM[2][2];
Chapter B. NuSMV model for multi-step transactions with CTL specification

102 esac;
103 next(TSM[2][3]) := case
104 T_c!= r2x3 & next(T_c)=r2x3 & n=2:next(p_c);
105 next(T_c)= end_2 :0;
106 1 :TSM[2][3];
107 esac;
108 next(TSM[2][4]) := case
109 T_c!= r2x4 & next(T_c)=r2x4 & n=2:next(p_c);
110 next(T_c)= end_2 :0;
111 1 :TSM[2][4];
112 esac;
113 next(TSM[2][5]) := 0;
114 ----------------------------------------------
115 next(TSM[3][1]) := case
116 T_c!= rox1 & next(T_c)=rox1 & n=3:next(p_c);
117 next(T_c)= end_o :0;
118 1 :TSM[3][1];
119 esac;
120 next(TSM[3][2]) := case
121 T_c!= rox2 & next(T_c)=rox2 & n=3:next(p_c);
122 next(T_c)= end_o :0;
123 1 :TSM[3][2];
124 esac;
125 next(TSM[3][3]) := case
126 T_c!= rox3 & next(T_c)=rox3 & n=3:next(p_c);
127 next(T_c)= end_o :0;
128 1 :TSM[3][3];
129 esac;
130 next(TSM[3][4]) := case
131 T_c!= rox4 & next(T_c)=rox4 & n=3:next(p_c);
132 next(T_c)= end_o :0;
133 1 :TSM[3][4];
134 esac;
135 next(TSM[3][5]) := case
136 T_c!= rox5 & next(T_c)=rox5 & n=3:next(p_c);
137 next(T_c)= end_o :0;
138 1 :TSM[3][5];
139 esac;
140 ----------------------------------------------
Chapter B. NuSMV model for multi-step transactions with CTL specification 149

141 MODULE main
142 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
143 VAR
144 T1 : \{begin_1,r1x1,w1x1,r1x2,w1x2,r1x3,w1x3,end_1\};
145 T2 : \{begin_2,r2x2,w2x2,r2x3,w2x3,r2x4,w2x4,end_2\};
146 Toth : \{begin_o,rox1,wox1,rox2,wox2,rox3,wox3,rox4,
147 wox4,rox5,wox5,end_o\};
148 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
149 TSM : array 1 .. 3 of array 1 .. 5 of 0..11;
150 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
151 p_c:0..11; -- System Timestamps
152 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
153 T_1 : process move(T1,1,T2,Toth,TSM,p_c);
154 T_2 : process move(T2,2,T1,Toth,TSM,p_c);
155 T_3 : process move(Toth,3,T1,T2,TSM,p_c);
156 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
157 ASSIGN
158 init(T1) :=begin_1;
159 init(T2) :=begin_2;
160 init(Toth) :=begin_o;
161 init(p_c):=0;
162 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
163 init(TSM[1][1]):= 0;
164 init(TSM[1][2]):= 0;
165 init(TSM[1][3]):= 0;
166 init(TSM[1][4]):= 0;
167 init(TSM[1][5]):= 0;
168 init(TSM[2][1]):= 0;
169 init(TSM[2][2]):= 0;
170 init(TSM[2][3]):= 0;
171 init(TSM[2][4]):= 0;
172 init(TSM[2][5]):= 0;
173 init(TSM[3][1]):= 0;
174 init(TSM[3][2]):= 0;
175 init(TSM[3][3]):= 0;
176 init(TSM[3][4]):= 0;
177 init(TSM[3][5]):= 0;
178 ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
179 FAIRNESS (T1=end_1)
Chapter B. NuSMV model for multi-step transactions with CTL specification

180 FAIRNESS (T2=end_2)
181 FAIRNESS (Toth=end_o)
182 -----------------------------------------------
183 --SPECIFICATIONS
184 -----------------------------------------------
185 ------read/write alternation -------
186 --T1
187 SPEC AG ((T1=r1x1) -> AF ((T1=w1x1) & AF (T1=r1x2)))
188 SPEC AG ((T1=r1x2) -> AF ((T1=w1x2) & AF (T1=r1x3)))
189 --T2
190 SPEC AG ((T2=r2x2) -> AF ((T2=w2x2) & AF (T2=r2x3)))
191 SPEC AG ((T2=r2x3) -> AF ((T2=w2x3) & AF (T2=r2x4)))
192 --Toth
193 SPEC AG ((Toth=roxl) -> AF ((Toth=woxl) & AF (Toth=rox2)))
194 SPEC AG ((Toth=rox2) -> AF ((Toth=wox2) & AF (Toth=rox3)))
195 SPEC AG ((Toth=rox3) -> AF ((Toth=wox3) & AF (Toth=rox4)))
196 SPEC AG ((Toth=rox4) -> AF ((Toth=wox4) & AF (Toth=rox5)))
197 --------write implies read---------
198 --T1
199 SPEC AG ((T1=r1x1) -> AF(T1=w1x1))
200 SPEC AG ((T1=r1x2) -> AF(T1=w1x2))
201 SPEC AG ((T1=r1x3) -> AF(T1=w1x3))
202 --T2
203 SPEC AG ((T2=r2x2) -> AF(T2=w2x2))
204 SPEC AG ((T2=r2x3) -> AF(T2=w2x3))
205 SPEC AG ((T2=r2x4) -> AF(T2=w2x4))
206 --Toth
207 SPEC AG ((Toth=roxl) -> AF(Toth=woxl))
208 SPEC AG ((Toth=rox2) -> AF(Toth=wox2))
209 SPEC AG ((Toth=rox3) -> AF(Toth=wox3))
210 SPEC AG ((Toth=rox4) -> AF(Toth=wox4))
211 SPEC AG ((Toth=rox5) -> AF(Toth=wox5))
212 ------p3-----------------------------
213 --T1
214 SPEC AG ((T1=w1x1) -> !(T1=r1x1))
215 SPEC AG ((T1=r1x2) -> !(T1=w1x1))
216 SPEC AG ((T1=w1x2) -> !(T1=r1x2))
217 SPEC AG ((T1=r1x3) -> !(T1=w1x2))
218 SPEC AG ((T1=w1x3) -> !(T1=r1x3))
Chapter B. NuSMV model for multi-step transactions with CTL specification 151

219 --T2
220 SPEC AG ((T2=w2x2) -> !(T2=r2x2))
221 SPEC AG ((T2=r2x3) -> !(T2=w2x2))
222 SPEC AG ((T2=w2x3) -> !(T2=r2x3))
223 SPEC AG ((T2=r2x4) -> !(T2=w2x3))
224 SPEC AG ((T2=w2x4) -> !(T2=r2x4))
225 --Toth
226 SPEC AG ((Toth=wox1) -> !(Toth=rox1))
227 SPEC AG ((Toth=rox2) -> !(Toth=wox1))
228 SPEC AG ((Toth=wox2) -> !(Toth=rox2))
229 SPEC AG ((Toth=rox3) -> !(Toth=wox2))
230 SPEC AG ((Toth=wox3) -> !(Toth=rox3))
231 SPEC AG ((Toth=rox4) -> !(Toth=wox3))
232 SPEC AG ((Toth=wox4) -> !(Toth=rox4))
233 SPEC AG ((Toth=rox5) -> !(Toth=wox4))
234 SPEC AG ((Toth=wox5) -> !(Toth=rox5))
235 --------------------------------------------------------------
236 --At most one step occurs at each successive state
237 --T1 and T2
238 SPEC AG (!((! (T1=r1x2)&!(T2=r2x2))&EX((T1=r1x2)&(T2=r2x2)))&
239 !((! (T1=r1x2)&!(T2=w2x2))&EX((T1=r1x2)&(T2=w2x2))) &
240 !((! (T1=w1x2)&!(T2=w2x2))&EX((T1=w1x2)&(T2=w2x2)))
241 SPEC AG (!((! (T1=r1x3)&!(T2=r2x3))&EX((T1=r1x3)&(T2=r2x3))) &
242 !((! (T1=r1x3)&!(T2=w2x3))&EX((T1=r1x3)&(T2=w2x3))) &
243 !((! (T1=w1x3)&!(T2=w2x3))&EX((T1=w1x3)&(T2=w2x3)))
244 --T1 and Toth
245 SPEC AG (!((! (T1=r1x1)&!(Toth=rox1))&EX((T1=r1x1)&(Toth=rox1)))&
246 !((! (T1=r1x1)&!(Toth=wox1))&EX((T1=r1x1)&(Toth=wox1))) &
247 !((! (T1=w1x1)&!(Toth=wox1))&EX((T1=w1x1)&(Toth=wox1)))
248 SPEC AG (!((! (T1=r1x2)&!(Toth=rox2))&EX((T1=r1x2)&(Toth=rox2)))&
249 !((! (T1=r1x2)&!(Toth=wox2))&EX((T1=r1x2)&(Toth=wox2))) &
250 !((! (T1=w1x2)&!(Toth=wox2))&EX((T1=w1x2)&(Toth=wox2)))
251 SPEC AG (!((! (T1=r1x3)&!(Toth=rox3))&EX((T1=r1x3)&(Toth=rox3)))&
252 !((! (T1=r1x3)&!(Toth=wox3))&EX((T1=r1x3)&(Toth=wox3))) &
253 !((! (T1=w1x3)&!(Toth=wox3))&EX((T1=w1x3)&(Toth=wox3)))
254 --Toth and T2
255 SPEC AG (!((! (Toth=rox2)&!(T2=r2x2))&EX((T1=rox2)&(T2=r2x2)))&
256 !((! (Toth=rox2)&!(T2=w2x2))&EX((T1=rox2)&(T2=w2x2))) &
257 !((! (Toth=wox2)&!(T2=w2x2))&EX((T1=wox2)&(T2=w2x2)))
Chapter B. NuSMV model for multi-step transactions with CTL specification

258 \[ \text{SPEC AG } (!!(Toth=rox3) \& !(T2=r2x3)) \& \text{EX}((T1=rox3) \& (T2=r2x3)) \]
259 \[ \text{SPEC AG } (!!(Toth=rox3) \& !(T2=w2x3)) \& \text{EX}((T1=rox3) \& (T2=w2x3)) \]
260 \[ \text{SPEC AG } (!!(Toth=wox3) \& !(T2=wox3)) \& \text{EX}((T1=wox3) \& (T2=wox3)) \]

261 \[ \text{SPEC AG } (!!(Toth=rox4) \& !(T2=r2x4)) \& \text{EX}((T1=rox4) \& (T2=r2x4)) \]
262 \[ \text{SPEC AG } (!!(Toth=rox4) \& !(T2=w2x4)) \& \text{EX}((T1=rox4) \& (T2=w2x4)) \]
263 \[ \text{SPEC AG } (!!(Toth=wox4) \& !(T2=wox4)) \& \text{EX}((T1=wox4) \& (T2=wox4)) \]

264 --p5-----------------------

265 --T1

266 \[ \text{SPEC AF } (!((T1=r1x1) \& AF (!((T1=r1x1) \& EF(T1=r1x1)))) \]
267 \[ \text{SPEC AF } (!((T1=w1x1) \& AF (!((T1=w1x1) \& EF(T1=w1x1)))) \]
268 \[ \text{SPEC AF } (!((T1=r1x2) \& AF (!((T1=r1x2) \& EF(T1=r1x2)))) \]
269 \[ \text{SPEC AF } (!((T1=w1x2) \& AF (!((T1=w1x2) \& EF(T1=w1x2)))) \]
270 \[ \text{SPEC AF } (!((T1=r1x3) \& AF (!((T1=r1x3) \& EF(T1=r1x3)))) \]
271 \[ \text{SPEC AF } (!((T1=w1x3) \& AF (!((T1=w1x3) \& EF(T1=w1x3)))) \]

272 --T2

273 \[ \text{SPEC AF } (!((T2=r2x2) \& AF (!((T2=r2x2) \& EF(T2=r2x2)))) \]
274 \[ \text{SPEC AF } (!((T2=w2x2) \& AF (!((T2=w2x2) \& EF(T2=w2x2)))) \]
275 \[ \text{SPEC AF } (!((T2=r2x3) \& AF (!((T2=r2x3) \& EF(T2=r2x3)))) \]
276 \[ \text{SPEC AF } (!((T2=w2x3) \& AF (!((T2=w2x3) \& EF(T2=w2x3)))) \]
277 \[ \text{SPEC AF } (!((T2=r2x4) \& AF (!((T2=r2x4) \& EF(T2=r2x4)))) \]
278 \[ \text{SPEC AF } (!((T2=w2x4) \& AF (!((T2=w2x4) \& EF(T2=w2x4)))) \]

279 --Toth

280 \[ \text{SPEC AF } (!((Toth=rox1) \& AF (!((Toth=rox1) \& EF(Toth=rox1)))) \]
281 \[ \text{SPEC AF } (!((Toth=wox1) \& AF (!((Toth=wox1) \& EF(Toth=wox1)))) \]
282 \[ \text{SPEC AF } (!((Toth=rox2) \& AF (!((Toth=rox2) \& EF(Toth=rox2)))) \]
283 \[ \text{SPEC AF } (!((Toth=wox2) \& AF (!((Toth=wox2) \& EF(Toth=wox2)))) \]
284 \[ \text{SPEC AF } (!((Toth=rox3) \& AF (!((Toth=rox3) \& EF(Toth=rox3)))) \]
285 \[ \text{SPEC AF } (!((Toth=wox3) \& AF (!((Toth=wox3) \& EF(Toth=wox3)))) \]
286 \[ \text{SPEC AF } (!((Toth=rox4) \& AF (!((Toth=rox4) \& EF(Toth=rox4)))) \]
287 \[ \text{SPEC AF } (!((Toth=wox4) \& AF (!((Toth=wox4) \& EF(Toth=wox4)))) \]
288 \[ \text{SPEC AF } (!((Toth=rox5) \& AF (!((Toth=rox5) \& EF(Toth=rox5)))) \]
289 \[ \text{SPEC AF } (!((Toth=wox5) \& AF (!((Toth=wox5) \& EF(Toth=wox5)))) \]

290 ---------------------------------------------------------------

291 ----no r1 and r2 on x without write for one of them

292 --T1 and Toth

293 \[ \text{SPEC } AG (Toth=rox1 \& T1=r1x1) \]
294 \[ \text{SPEC } AG (Toth=rox2 \& T1=r1x2) \]
295 \[ \text{SPEC } AG (Toth=rox3 \& T1=r1x3) \]
296 --T1 and T2
Chapter B. NuSMV model for multi-step transactions with CTL specification

297 SPEC AG ! (T1=r1x2 & T2=r2x2)
298 SPEC AG ! (T1=r1x3 & T2=r2x3)
299 --T2 and Toth
300 SPEC AG ! (Toth=rox2 & T2=r2x2)
301 SPEC AG ! (Toth=rox3 & T2=r2x3)
302 SPEC AG ! (Toth=rox4 & T2=r2x4)
303 -------------------------------
304 --protocol--the correct---------
305 SPEC AF ! (T1=w1x2 & AF(T2=w2x2 & EF(T2=w2x3 & AF(T1=w1x3))))
306 (T2=w2x2 & AF(T1=w1x2 & EF(T1=w1x3 & AF(T2=w2x3))))
307 FAIRNESS running
Appendix C

NuSMV model for iterated transactions with LTL specification

1 MODULE move(T_c,n,T0,T1)
2 ASSIGN
3 next(T_c):=case
4 T_c=begin_1 & n=1& !((T_c=r1x1)&(!(T1=rox1))) : r1x1;
5 T_c=r1x1 & n=1 : w1x1;
6 T_c=w1x1 & n=1& !(T_c=r1x2)& !(T0=r2x2) & !(T1=rox2) :
7 r1x2;
8 T_c=r1x2 & n=1 : w1x2;
9 T_c=w1x2 & n=1& !(T_c=r1x3)& !(T0=r2x3) & !(T1=rox3) :
10 r1x3;
11 T_c=r1x3 & n=1 : w1x3;
12 T_c=w1x3 & n=1 : end_1;
13 -- this to keep transaction 1 repeats
14 T_c=end_1 & n=1 : begin_1;
15 T_c=begin_2 & n=2 & !(T0=r2x2) & !(T_c=r2x2) & !(T1=rox2) :
16 r2x2;
17 T_c=r2x2 & n=2 : w2x2;
18 T_c=w2x2 & n=2 & !(T0=r2x3) & !(T_c=r2x3) & !(T1=rox3) :
19 r2x3;
20 T_c=r2x3 & n=2 : w2x3;
21 T_c=w2x3 & n=2 & !(T_c=r2x4) & !(T1=rox4) :
22 r2x4;
23 T_c=r2x4 & n=2 : w2x4;
24 T_c=w2x4 & n=2 : end_2;
Chapter C. NuSMV model for iterated transactions with LTL specification

24 -- this to keep transaction 2 repeats

25 \text{T}_c= \text{end}_2 \hspace{1em} \& \hspace{1em} n=2
\text{:begin}_2;

26 \text{T}_c= \text{begin}_0 \& \hspace{1em} n=3 \& \left( (\neg (\text{TO=} \text{rlx1})) \& (\neg (\text{T}_c=\text{rox1})) \right) \hspace{1em} :\text{rox1};

27 \text{T}_c=\text{rox1} \hspace{1em} \& \hspace{1em} n=3
\hspace{1em} :\text{wox1};

28 \text{T}_c=\text{wox1} \hspace{1em} \& \hspace{1em} n=3 \& \left( (\neg (\text{TO=} \text{rlx2})) \& (\neg (\text{T}_c=\text{rox2})) \right)
\left( (\neg (\text{T}_c=\text{rox2})) \right) \hspace{1em} :\text{rox2};

29 \text{T}_c=\text{rox2} \hspace{1em} \& \hspace{1em} n=3
\hspace{1em} :\text{wox2};

30 \text{T}_c=\text{wox2} \hspace{1em} \& \hspace{1em} n=3 \& \left( (\neg (\text{TO=} \text{rlx3})) \& (\neg (\text{T}_c=\text{rox3})) \right)
\left( (\neg (\text{T}_c=\text{rox3})) \right) \hspace{1em} :\text{rox3};

31 \text{T}_c=\text{rox3} \hspace{1em} \& \hspace{1em} n=3
\hspace{1em} :\text{wox3};

32 \text{T}_c=\text{wox3} \hspace{1em} \& \hspace{1em} n=3 \& \left( (\neg (\text{TO=} \text{rlx4})) \& (\neg (\text{T}_c=\text{rox4})) \right)
\left( (\neg (\text{T}_c=\text{rox4})) \right) \hspace{1em} :\text{rox4};

33 \text{T}_c=\text{rox4} \hspace{1em} \& \hspace{1em} n=3
\hspace{1em} :\text{wox4};

34 \text{T}_c=\text{wox4} \hspace{1em} \& \hspace{1em} n=3 \& (\neg (\text{T}_c=\text{rox5}))
\hspace{1em} :\text{rox5};

35 \text{T}_c=\text{rox5} \hspace{1em} \& \hspace{1em} n=3
\hspace{1em} :\text{wox5};

36 \text{T}_c=\text{wox5} \hspace{1em} \& \hspace{1em} n=3
\hspace{1em} :\text{end}_0;

39 -- this to keep transaction 2 repeats

40 \text{T}_c=\text{end}_0 \hspace{1em} \& \hspace{1em} n=3
\text{:begin}_0;

41 1
\text{:T}_c;

42 esac;

43 MODULE main

44 VAR

45 \text{T}_1 : \{ \text{begin}_1, \text{rlx1}, \text{wlx1}, \text{rlx2}, \text{wlx2}, \text{rlx3}, \text{wlx3}, \text{end}_1 \};

46 \text{T}_2 : \{ \text{begin}_2, \text{rlx2}, \text{wlx2}, \text{rlx3}, \text{wlx3}, \text{rlx4}, \text{wlx4}, \text{end}_2 \};

47 \text{Toth} : \{ \text{begin}_0, \text{roxl}, \text{woxl}, \text{rox2}, \text{wox2}, \text{rox3}, \text{wox3}, \text{rox4}, \text{wox4}, \text{rox5}, \text{end}_0 \};

49 \text{T}_1 \hspace{1em} \text{process move}(\text{T}_1, 1, \text{T}_2, \text{Toth});

50 \text{T}_2 \hspace{1em} \text{process move}(\text{T}_2, 2, \text{T}_1, \text{Toth});

51 \text{T}_3 \hspace{1em} \text{process move}(\text{Toth}, 3, \text{T}_1, \text{T}_2);

52 ASSIGN

53 \text{init}(\text{T}_1) := \text{begin}_1;

54 \text{init}(\text{T}_2) := \text{begin}_2;

55 \text{init}(\text{Toth}) := \text{begin}_0;

56 \text{----------------------------------------------}

57 \text{FAIRNESS} ! (\text{T}_1=\text{begin}_1)

58 \text{FAIRNESS} ! (\text{T}_1=\text{rlx1})

59 \text{FAIRNESS} ! (\text{T}_1=\text{wlx1})

60 \text{FAIRNESS} ! (\text{T}_1=\text{rlx2})

61 \text{FAIRNESS} ! (\text{T}_1=\text{wlx2})

62 \text{FAIRNESS} ! (\text{T}_1=\text{rlx3})
63 FAIRNESS !(T1=r1x3)
64 FAIRNESS !(T1=end_1)
65 FAIRNESS !(T2=begin_2)
66 FAIRNESS !(T2=r2x2)
67 FAIRNESS !(T2=w2x2)
68 FAIRNESS !(T2=r2x3)
69 FAIRNESS !(T2=w2x3)
70 FAIRNESS !(T2=r2x4)
71 FAIRNESS !(T2=w2x4)
72 FAIRNESS !(T2=end_2)
73 FAIRNESS !(Toth=begin_o)
74 FAIRNESS !(Toth=rox1)
75 FAIRNESS !(Toth=wox1)
76 FAIRNESS !(Toth=rox2)
77 FAIRNESS !(Toth=wox2)
78 FAIRNESS !(Toth=rox3)
79 FAIRNESS !(Toth=wox3)
80 FAIRNESS !(Toth=rox4)
81 FAIRNESS !(Toth=wox4)
82 FAIRNESS !(Toth=rox5)
83 FAIRNESS !(Toth=wox5)
84 FAIRNESS !(Toth=end_o)
85 ----------------------------------------------
86 --SPECIFICATIONS
87 ----------------------------------------------
88 ------read/write alternation ------
89 --For T1
90 LTLSPEC G ((T1=r1x2) & O(T1=r1x1)) -> O(T1=w1x1)
91 LTLSPEC G ((T1=r1x3) & O(T1=r1x1)) -> O(T1=w1x1)
92 LTLSPEC G ((T1=r1x3) & O(T1=r1x2)) -> O(T1=w1x2)
93 --For T2
94 LTLSPEC G ((T2=r2x3) & O(T2=r2x2)) -> O(T2=w2x2)
95 LTLSPEC G ((T2=r2x4) & O(T2=r2x2)) -> O(T2=w2x2)
96 LTLSPEC G ((T2=r2x4) & O(T2=r2x3)) -> O(T2=w2x3)
97 --For Toth
98 LTLSPEC G ((Toth=rox2) & O(Toth=rox1)) -> O(Toth=wox1)
99 LTLSPEC G ((Toth=rox3) & O(Toth=rox1)) -> O(Toth=wox1)
100 LTLSPEC G ((Toth=rox4) & O(Toth=rox1)) -> O(Toth=wox1)
101 LTLSPEC G ((Toth=rox5) & O(Toth=rox1)) -> O(Toth=wox1)
Chapter C. NuSMV model for iterated transactions with LTL specification

102 \text{LTLSPEC} G (((\text{Toth} = \text{rox3}) \& 0(\text{Toth} = \text{rox2})) \rightarrow 0(\text{Toth} = \text{wox2}))
103 \text{LTLSPEC} G (((\text{Toth} = \text{rox4}) \& 0(\text{Toth} = \text{rox2})) \rightarrow 0(\text{Toth} = \text{wox2}))
104 \text{LTLSPEC} G (((\text{Toth} = \text{rox5}) \& 0(\text{Toth} = \text{rox2})) \rightarrow 0(\text{Toth} = \text{wox2}))
105 \text{LTLSPEC} G (((\text{Toth} = \text{rox4}) \& 0(\text{Toth} = \text{rox3})) \rightarrow 0(\text{Toth} = \text{wox3}))
106 \text{LTLSPEC} G (((\text{Toth} = \text{rox5}) \& 0(\text{Toth} = \text{rox3})) \rightarrow 0(\text{Toth} = \text{wox3}))
107 \text{LTLSPEC} G (((\text{Toth} = \text{rox5}) \& 0(\text{Toth} = \text{rox4})) \rightarrow 0(\text{Toth} = \text{wox4}))
108 \text{------read/write alternation in future ------}
109 --For \text{T1}
110 \text{LTLSPEC} G (T1 = \text{r1x1} \rightarrow (F (T1 = \text{w1x1} \& F (T1 = \text{r1x2})))
111 \text{LTLSPEC} G (T1 = \text{r1x2} \rightarrow (F (T1 = \text{w1x2} \& F (T1 = \text{r1x3})))
112 \text{LTLSPEC} G (T1 = \text{r1x1} \rightarrow (F (T1 = \text{w1x1} \& F (T1 = \text{r1x3})))
113 --For \text{T2}
114 \text{LTLSPEC} G (T2 = \text{r2x2} \rightarrow (F (T2 = \text{w2x2} \& F (T2 = \text{r2x3})))
115 \text{LTLSPEC} G (T2 = \text{r2x3} \rightarrow (F (T2 = \text{w2x3} \& F (T2 = \text{r2x4})))
116 \text{LTLSPEC} G (T2 = \text{r2x2} \rightarrow (F (T2 = \text{w2x2} \& F (T2 = \text{r2x4})))
117 --For \text{Toth}
118 \text{LTLSPEC} G (\text{Toth} = \text{rox1} \rightarrow (F (\text{Toth} = \text{wox1} \& F (\text{Toth} = \text{rox2})))
119 \text{LTLSPEC} G (\text{Toth} = \text{rox1} \rightarrow (F (\text{Toth} = \text{wox1} \& F (\text{Toth} = \text{rox3})))
120 \text{LTLSPEC} G (\text{Toth} = \text{rox1} \rightarrow (F (\text{Toth} = \text{wox1} \& F (\text{Toth} = \text{rox4})))
121 \text{LTLSPEC} G (\text{Toth} = \text{rox1} \rightarrow (F (\text{Toth} = \text{wox1} \& F (\text{Toth} = \text{rox5})))
122 \text{LTLSPEC} G (\text{Toth} = \text{rox2} \rightarrow (F (\text{Toth} = \text{wox2} \& F (\text{Toth} = \text{rox3})))
123 \text{LTLSPEC} G (\text{Toth} = \text{rox2} \rightarrow (F (\text{Toth} = \text{wox2} \& F (\text{Toth} = \text{rox4})))
124 \text{LTLSPEC} G (\text{Toth} = \text{rox2} \rightarrow (F (\text{Toth} = \text{wox2} \& F (\text{Toth} = \text{rox5})))
125 \text{LTLSPEC} G (\text{Toth} = \text{rox3} \rightarrow (F (\text{Toth} = \text{wox3} \& F (\text{Toth} = \text{rox4})))
126 \text{LTLSPEC} G (\text{Toth} = \text{rox3} \rightarrow (F (\text{Toth} = \text{wox3} \& F (\text{Toth} = \text{rox5})))
127 \text{LTLSPEC} G (\text{Toth} = \text{rox4} \rightarrow (F (\text{Toth} = \text{wox4} \& F (\text{Toth} = \text{rox5})))
128 \text{-------------------------------}
129 \text{------write implies Read ------}
130 --For \text{T1}
131 \text{LTLSPEC} G ((T1 = \text{w1x1}) \rightarrow 0(T1 = \text{r1x1}))
132 \text{LTLSPEC} G ((T1 = \text{w1x2}) \rightarrow 0(T1 = \text{r1x2}))
133 \text{LTLSPEC} G ((T1 = \text{w1x3}) \rightarrow 0(T1 = \text{r1x3}))
134 --For \text{T2}
135 \text{LTLSPEC} G ((T2 = \text{w2x2}) \rightarrow 0(T2 = \text{r2x2}))
136 \text{LTLSPEC} G ((T2 = \text{w2x3}) \rightarrow 0(T2 = \text{r2x3}))
137 \text{LTLSPEC} G ((T2 = \text{w2x4}) \rightarrow 0(T2 = \text{r2x4}))
138 --For \text{Toth}
139 \text{LTLSPEC} G ((\text{Toth} = \text{wox1}) \rightarrow 0(\text{Toth} = \text{rox1}))
140 \text{LTLSPEC} G ((\text{Toth} = \text{wox2}) \rightarrow 0(\text{Toth} = \text{rox2}))
Chapter C. NuSMV model for iterated transactions with LTL specification 158

141 \text{LTLSPEC } G ((\text{Toth}=\text{wox3}) \rightarrow O(\text{Toth}=\text{rox3}))
142 \text{LTLSPEC } G ((\text{Toth}=\text{wox4}) \rightarrow O(\text{Toth}=\text{rox4}))
143 \text{LTLSPEC } G ((\text{Toth}=\text{wox5}) \rightarrow O(\text{Toth}=\text{rox5}))
144 \text{------write implies Read future -----------}
145 \text{--For T1}
146 \text{LTLSPEC } G (T1=r1x1 \rightarrow (F (T1=w1x1 )))
147 \text{LTLSPEC } G (T1=r1x2 \rightarrow (F (T1=w1x2 )))
148 \text{LTLSPEC } G (T1=r1x3 \rightarrow (F (T1=w1x3 )))
149 \text{--For T2}
150 \text{LTLSPEC } G (T2=r2x2 \rightarrow (F (T2=w2x2 )))
151 \text{LTLSPEC } G (T2=r2x3 \rightarrow (F (T2=w2x3 )))
152 \text{LTLSPEC } G (T2=r2x4 \rightarrow (F (T2=w2x4 )))
153 \text{--For Toth}
154 \text{LTLSPEC } G (\text{Toth}=\text{rox1} \rightarrow (F (\text{Toth}=\text{wox1 })))
155 \text{LTLSPEC } G (\text{Toth}=\text{rox2} \rightarrow (F (\text{Toth}=\text{wox2 })))
156 \text{LTLSPEC } G (\text{Toth}=\text{rox3} \rightarrow (F (\text{Toth}=\text{wox3 })))
157 \text{LTLSPEC } G (\text{Toth}=\text{rox4} \rightarrow (F (\text{Toth}=\text{wox4 })))
158 \text{LTLSPEC } G (\text{Toth}=\text{rox5} \rightarrow (F (\text{Toth}=\text{wox5 })))
159 \text{--proposition remains true until the next operation}
160 \text{--For T1}
161 \text{LTLSPEC } G ((T1=w1x1)-->) 0((T1=r1x1) U (T1=w1x1))
162 \text{LTLSPEC } G ((T1=w1x2)-->) 0((T1=r1x2) U (T1=w1x2))
163 \text{LTLSPEC } G ((T1=w1x3)-->) 0((T1=r1x3) U (T1=w1x3))
164 --
165 \text{LTLSPEC } G ((T1=r1x2)-->) 0((T1=w1x1) U (T1=r1x2))
166 \text{LTLSPEC } G ((T1=r1x3)-->) 0((T1=w1x2) U (T1=r1x3))
167 --For T2
168 \text{LTLSPEC } G ((T2=w2x2)-->) 0((T2=r2x2) U (T2=w2x2))
169 \text{LTLSPEC } G ((T2=w2x3)-->) 0((T2=r2x3) U (T2=w2x3))
170 \text{LTLSPEC } G ((T2=w2x4)-->) 0((T2=r2x4) U (T2=w2x4))
171 --
172 \text{LTLSPEC } G ((T2=r2x3)-->) 0((T2=w2x2) U (T2=r2x3))
173 \text{LTLSPEC } G ((T2=r2x4)-->) 0((T2=w2x3) U (T2=r2x4))
174 --For Toth
175 \text{LTLSPEC } G ((\text{Toth}=\text{wox1})-->) 0((\text{Toth}=\text{rox1}) U (\text{Toth}=\text{wox1}))
176 \text{LTLSPEC } G ((\text{Toth}=\text{wox2})-->) 0((\text{Toth}=\text{rox2}) U (\text{Toth}=\text{wox2}))
177 \text{LTLSPEC } G ((\text{Toth}=\text{wox3})-->) 0((\text{Toth}=\text{rox3}) U (\text{Toth}=\text{wox3}))
178 \text{LTLSPEC } G ((\text{Toth}=\text{wox4})-->) 0((\text{Toth}=\text{rox4}) U (\text{Toth}=\text{wox4}))
179 \text{LTLSPEC } G ((\text{Toth}=\text{wox5})-->) 0((\text{Toth}=\text{rox5}) U (\text{Toth}=\text{wox5}))
Chapter C. NuSMV model for iterated transactions with LTL specification

180 --
181 LTLSPEC G ((Toth=rox2) -> 0((Toth=wox1) U (Toth=rox2)))
182 LTLSPEC G ((Toth=rox3) -> 0((Toth=wox2) U (Toth=rox3)))
183 LTLSPEC G ((Toth=rox4) -> 0((Toth=wox3) U (Toth=rox4)))
184 LTLSPEC G ((Toth=rox5) -> 0((Toth=wox4) U (Toth=rox5)))
185 --proposition remains true until the next operation (Future)
186 -- For T1
187 LTLSPEC G ((T1=w1x1) -> !(T1=r1x1))
188 LTLSPEC G ((T1=w1x2) -> !(T1=r1x2))
189 LTLSPEC G ((T1=w1x3) -> !(T1=r1x3))
190 --
191 LTLSPEC G ((T1=r1x2) -> !(T1=w1x1))
192 LTLSPEC G ((T1=r1x3) -> !(T1=w1x2))
193 -- For T2
194 LTLSPEC G ((T2=w2x2) -> !(T2=r2x2))
195 LTLSPEC G ((T2=w2x3) -> !(T2=r2x3))
196 LTLSPEC G ((T2=w2x4) -> !(T2=r2x4))
197 --
198 LTLSPEC G ((T2=r2x3) -> !(T2=w2x2))
199 LTLSPEC G ((T2=r2x4) -> !(T2=w2x3))
200 -- For Toth
201 LTLSPEC G ((Toth=wox1) -> !(Toth=rox1))
202 LTLSPEC G ((Toth=wox2) -> !(Toth=rox2))
203 LTLSPEC G ((Toth=wox3) -> !(Toth=rox3))
204 LTLSPEC G ((Toth=wox4) -> !(Toth=rox4))
205 LTLSPEC G ((Toth=wox5) -> !(Toth=rox5))
206 --
207 LTLSPEC G ((Toth=rox2) -> !(Toth=wox1))
208 LTLSPEC G ((Toth=rox3) -> !(Toth=wox2))
209 LTLSPEC G ((Toth=rox4) -> !(Toth=wox3))
210 LTLSPEC G ((Toth=rox5) -> !(Toth=wox4))
211 -- At most one step occurs at each successive state (Future) --
212 -- T1 and T2
213 LTLSPEC G (!(!((T1=r1x2) & (T2=r2x2)) & X((T1=r1x2) & (T2=r2x2))) &
214 !(!((T1=r1x2) & (T2=w2x2)) & X((T1=r1x2) & (T2=w2x2))) &
215 !(!((T1=w1x2) & (T2=w2x2)) & X((T1=w1x2) & (T2=w2x2)))
216 LTLSPEC G (!(!((T1=r1x3) & (T2=r2x3)) & X((T1=r1x3) & (T2=r2x3))) &
217 !(!((T1=r1x3) & (T2=w2x3)) & X((T1=r1x3) & (T2=w2x3))) &
218 !(!((T1=w1x3) & (T2=w2x3)) & X((T1=w1x3) & (T2=w2x3))))
Chapter C. NuSMV model for iterated transactions with LTL specification

219 --T1 and Toth
220 LTLSPEC G ( !((!T1=r1x1)&(!Toth=rox1))&
221 X((!T1=r1x1)&(Toth=rox1)))&
222 !((!T1=r1x1)&(!Toth=rox1))&
223 X((!T1=r1x1)&(Toth=rox1)))&
224 !((!T1=w1x1)&(!Toth=rox1))&
225 X((!T1=w1x1)&(Toth=rox1)))
226 LTLSPEC G ( !((!T1=r1x2)&(!Toth=rox2))&
227 X((!T1=r1x2)&(Toth=rox2)))&
228 !((!T1=r1x2)&(!Toth=rox2))&
229 X((!T1=r1x2)&(Toth=rox2)))&
230 !((!T1=w1x2)&(!Toth=rox2))&
231 X((!T1=w1x2)&(Toth=rox2)))
232 LTLSPEC G ( !((!T1=r1x3)&(!Toth=rox3))&
233 X((!T1=r1x3)&(Toth=rox3)))&
234 !((!T1=r1x3)&(!Toth=rox3))&
235 X((!T1=r1x3)&(Toth=rox3)))&
236 !((!T1=w1x3)&(!Toth=rox3))&
237 X((!T1=w1x3)&(Toth=rox3)))
238 --Toth and T2
239 LTLSPEC G ( !((!Toth=rox2)&(!T2=r2x2))&
240 X((!Toth=rox2)&(T2=r2x2)))&
241 !((!Toth=rox2)&(!T2=r2x2))&
242 X((!Toth=rox2)&(T2=r2x2))+
243 !((!Toth=wox2)&(!T2=r2x2))&
244 X((!Toth=wox2)&(T2=r2x2))
245 LTLSPEC G ( !((!Toth=rox3)&(!T2=r2x3))&
246 X((!Toth=rox3)&(T2=r2x3)))&
247 !((!Toth=rox3)&(!T2=r2x3))&
248 X((!Toth=rox3)&(T2=r2x3))&
249 !((!Toth=wox3)&(!T2=r2x3))&
250 X((!Toth=wox3)&(T2=r2x3))
251 LTLSPEC G ( !((!Toth=rox4)&(!T2=r2x4))&
252 X((!Toth=rox4)&(T2=r2x4))&
253 !((!Toth=rox4)&(!T2=r2x4))&
254 X((!Toth=rox4)&(T2=r2x4))&
255 !((!Toth=wox4)&(!T2=r2x4))&
256 X((!Toth=wox4)&(T2=r2x4))
257 --At most one step occurs at each successive state(past)--
Chapter C. NuSMV model for iterated transactions with LTL specification

258 --T1 and T2
259 LTLSPEC G (!(((T1=r1x2) & (T2=r2x2))) &
260   Y (!((T1=r1x2) & (T2=r2x2))) &
261 !(((T1=r1x2) & (T2=w2x2))) &
262   Y (!((T1=r1x2) & (T2=w2x2))) &
263 !(((T1=w1x2) & (T2=w2x2))) &
264   Y (!((T1=w1x2) & (T2=w2x2))) &
265 LTLSPEC G (!(((T1=r1x3) & (T2=r2x3))) &
266   Y (!((T1=r1x3) & (T2=r2x3))) &
267 !(((T1=r1x3) & (T2=w2x3))) &
268   Y (!((T1=r1x3) & (T2=w2x3))) &
269 !(((T1=w1x3) & (T2=w2x3))) &
270   Y (!((T1=w1x3) & (T2=w2x3))) &
271 --T1 and Toth
272 LTLSPEC G (!(((T1=r1x1) & (Toth=rox1))) &
273   Y (!((T1=r1x1) & (Toth=rox1))) &
274 !(((T1=r1x1) & (Toth=wox1))) &
275   Y (!((T1=r1x1) & (Toth=wox1))) &
276 !(((T1=w1x1) & (Toth=wox1))) &
277   Y (!((T1=w1x1) & (Toth=wox1))) &
278 LTLSPEC G (!(((T1=r1x2) & (Toth=r2x2))) &
279   Y (!((T1=r1x2) & (Toth=r2x2))) &
280 !(((T1=r1x2) & (Toth=w2x2))) &
281   Y (!((T1=r1x2) & (Toth=w2x2))) &
282 !(((T1=w1x2) & (Toth=w2x2))) &
283   Y (!((T1=w1x2) & (Toth=w2x2))) &
284 LTLSPEC G (!(((T1=r1x3) & (Toth=rox3))) &
285   Y (!((T1=r1x3) & (Toth=rox3))) &
286 !(((T1=r1x3) & (Toth=wox3))) &
287   Y (!((T1=r1x3) & (Toth=wox3))) &
288 !(((T1=w1x3) & (Toth=wox3))) &
289   Y (!((T1=w1x3) & (Toth=wox3))) &
290 --Toth and T2
291 LTLSPEC G (!(((T2=r2x2) & (Toth=rox2))) &
292   Y (!((T2=r2x2) & (Toth=rox2))) &
293 !(((T2=r2x2) & (Toth=wox2))) &
294   Y (!((T2=r2x2) & (Toth=wox2))) &
295 !(((T2=w2x2) & (Toth=wox2))) &
296   Y (!((T2=w2x2) & (Toth=wox2))) &
Chapter C. NuSMV model for iterated transactions with LTL specification

297 LTLSPEC G ( !( (T2=r2x3) & (Toth=rox3)) )
298 \ Y(! (T2=r2x3) & ! (Toth=rox3)) )
299 ! ( (T2=r2x3) & (Toth=wox3) )
300 \ Y(! (T2=r2x3) & ! (Toth=wox3)) )
301 ! ( (T2=w2x3) & (Toth=wox3) )
302 \ Y(! (T2=w2x3) & ! (Toth=wox3)) )
303 LTLSPEC G ( !( (T2=r2x4) & (Toth=rox4)) )
304 \ Y(! (T2=r2x4) & ! (Toth=rox4)) )
305 ! ( (T2=r2x4) & (Toth=wox4) )
306 \ Y(! (T2=r2x4) & ! (Toth=wox4)) )
307 ! ( (T2=w2x4) & (Toth=wox4) )
308 \ Y(! (T2=w2x4) & ! (Toth=wox4)) )
309
310 --p5-ini--
311 --T1
312 LTLSPEC G( (T1=begin_1) \rightarrow ((F! (T1=rlx1 ) &
313 \ F \ (T1=rlx1 \& F \ T1=rlx1 ))) ) \cup (T1=\text{end}_1))
314 LTLSPEC G( (T1=begin_1) \rightarrow ((F! (T1=w1x1 ) &
315 \ F \ (T1=w1x1 \& F \ T1=w1x1 ))) ) \cup (T1=\text{end}_1))
316 LTLSPEC G( (T1=begin_1) \rightarrow ((F! (T1=r1x2 ) &
317 \ F \ (T1=r1x2 \& F \ T1=r1x2 ))) ) \cup (T1=\text{end}_1))
318 LTLSPEC G( (T1=begin_1) \rightarrow ((F! (T1=w1x2 ) &
319 \ F \ (T1=w1x2 \& F \ T1=w1x2 ))) ) \cup (T1=\text{end}_1))
320 LTLSPEC G( (T1=begin_1) \rightarrow ((F! (T1=r1x3 ) &
321 \ F \ (T1=r1x3 \& F \ T1=r1x3 ))) ) \cup (T1=\text{end}_1))
322 LTLSPEC G( (T1=begin_1) \rightarrow ((F! (T1=w1x3 ) &
323 \ F \ (T1=w1x3 \& F \ T1=w1x3 ))) ) \cup (T1=\text{end}_1))
324 --T2
325 LTLSPEC G( (T2=begin_2) \rightarrow ((F! (T2=r2x2 ) &
326 \ F \ (T2=r2x2 \& F \ T2=r2x2 ))) ) \cup (T2=\text{end}_2))
327 LTLSPEC G( (T2=begin_2) \rightarrow ((F! (T2=w2x2 ) &
328 \ F \ (T2=w2x2 \& F \ T2=w2x2 ))) ) \cup (T2=\text{end}_2))
329 LTLSPEC G( (T2=begin_2) \rightarrow ((F! (T2=r2x3 ) &
330 \ F \ (T2=r2x3 \& F \ T2=r2x3 ))) ) \cup (T2=\text{end}_2))
331 LTLSPEC G( (T2=begin_2) \rightarrow ((F! (T2=w2x3 ) &
332 \ F \ (T2=w2x3 \& F \ T2=w2x3 ))) ) \cup (T2=\text{end}_2))
333 LTLSPEC G( (T2=begin_2) \rightarrow ((F! (T2=r2x4 ) &
334 \ F \ (T2=r2x4 \& F \ T2=r2x4 ))) ) \cup (T2=\text{end}_2))
335 LTLSPEC G( (T2=begin_2) \rightarrow ((F! (T2=w2x4 &
Chapter C. NuSMV model for iterated transactions with LTL specification

336 \[ F (T2!=w2x4 \& F T2=w2x4)) U (T2=end_2)) \]
337 --Toth
338 LTLSPEC G( (Toth=begin_o) \rightarrow ((F!(Toth=rox1 \&
339 \ F (Toth!=rox1 \& F Toth=rox1))) U (Toth=end_o)) )
340 LTLSPEC G( (Toth=begin_o) \rightarrow ((F!(Toth=wox1 \&
341 \ F (Toth!=wox1 \& F Toth=wox1))) U (Toth=end_o)) )
342 LTLSPEC G( (Toth=begin_o) \rightarrow ((F!(Toth=rox2 \&
343 \ F (Toth!=rox2 \& F Toth=rox2))) U (Toth=end_o)) )
344 LTLSPEC G( (Toth=begin_o) \rightarrow ((F!(Toth=wox2 \&
345 \ F (Toth!=wox2 \& F Toth=wox2))) U (Toth=end_o)) )
346 LTLSPEC G( (Toth=begin_o) \rightarrow ((F!(Toth=rox3 \&
347 \ F (Toth!=rox3 \& F Toth=rox3))) U (Toth=end_o)) )
348 LTLSPEC G( (Toth=begin_o) \rightarrow ((F!(Toth=wox3 \&
349 \ F (Toth!=wox3 \& F Toth=wox3))) U (Toth=end_o)) )
350 LTLSPEC G( (Toth=begin_o) \rightarrow ((F!(Toth=rox4 \&
351 \ F (Toth!=rox4 \& F Toth=rox4))) U (Toth=end_o)) )
352 LTLSPEC G( (Toth=begin_o) \rightarrow ((F!(Toth=wox4 \&
353 \ F (Toth!=wox4 \& F Toth=wox4))) U (Toth=end_o)) )
354 LTLSPEC G( (Toth=begin_o) \rightarrow ((F!(Toth=rox5 \&
355 \ F (Toth!=rox5 \& F Toth=rox5))) U (Toth=end_o)) )
356 LTLSPEC G( (Toth=begin_o) \rightarrow ((F!(Toth=wox5 \&
357 \ F (Toth!=wox5 \& F Toth=wox5))) U (Toth=end_o)) )
358 --past--
359 --T1
360 LTLSPEC G( (T1=end_1) \rightarrow (0!(T1=r1x1 \&
361 \ 0 (T1!=r1x1 \& 0 T1=r1x1)) S (T1=begin_1)) )
362 LTLSPEC G( (T1=end_1) \rightarrow (0!(T1=w1x1 \&
363 \ 0 (T1!=w1x1 \& 0 T1=w1x1)) S (T1=begin_1)) )
364 LTLSPEC G( (T1=end_1) \rightarrow (0!(T1=r1x2 \&
365 \ 0 (T1!=r1x2 \& 0 T1=r1x2)) S (T1=begin_1)) )
366 LTLSPEC G( (T1=end_1) \rightarrow (0!(T1=w1x2 \&
367 \ 0 (T1!=w1x2 \& 0 T1=w1x2)) S (T1=begin_1)) )
368 LTLSPEC G( (T1=end_1) \rightarrow (0!(T1=r1x3 \&
369 \ 0 (T1!=r1x3 \& 0 T1=r1x3)) S (T1=begin_1)) )
370 LTLSPEC G( (T1=end_1) \rightarrow (0!(T1=w1x3 \&
371 \ 0 (T1!=w1x3 \& 0 T1=w1x3)) S (T1=begin_1)) )
372 --T2
373 LTLSPEC G( (T2=end_2) \rightarrow (0!(T2=r2x2 \&
374 \ 0 (T2!=r2x2 \& 0 T2=r2x2)) S (T2=begin_2)) )
Chapter C. NuSMV model for iterated transactions with LTL specification

375 LTLSPEC G (T2=end_2 -> (O!(T2=w2x2 &
376 0 (T2!=w2x2 & 0 T2=w2x2)) S (T2=begin_2)) )
377 LTLSPEC G (T2=end_2 -> (O!(T2=r2x3 &
378 0 (T2!=r2x3 & 0 T2=r2x3)) S (T2=begin_2)) )
379 LTLSPEC G (T2=end_2 -> (O!(T2=w2x3 &
380 0 (T2!=w2x3 & 0 T2=w2x3)) S (T2=begin_2)) )
381 LTLSPEC G (T2=end_2 -> (O!(T2=r2x4 &
382 0 (T2!=r2x4 & 0 T2=r2x4)) S (T2=begin_2)) )
383 LTLSPEC G (T2=end_2 -> (O!(T2=w2x4 &
384 0 (T2!=w2x4 & 0 T2=w2x4)) S (T2=begin_2)) )
385 --Toth
386 LTLSPEC G (Toth=end_o) -> (O!(Toth=rox1 &
387 0 (Toth!=rox1 & 0 Toth=rox1)) S (Toth=begin_o)) )
388 LTLSPEC G (Toth=end_o) -> (O!(Toth=wox1 &
389 0 (Toth!=wox1 & 0 Toth=wox1)) S (Toth=begin_o)) )
390 LTLSPEC G (Toth=end_o) -> (O!(Toth=rox2 &
391 0 (Toth!=rox2 & 0 Toth=rox2)) S (Toth=begin_o)) )
392 LTLSPEC G (Toth=end_o) -> (O!(Toth=wox2 &
393 0 (Toth!=wox2 & 0 Toth=wox2)) S (Toth=begin_o)) )
394 LTLSPEC G (Toth=end_o) -> (O!(Toth=rox3 &
395 0 (Toth!=rox3 & 0 Toth=rox3)) S (Toth=begin_o)) )
396 LTLSPEC G (Toth=end_o) -> (O!(Toth=wox3 &
397 0 (Toth!=wox3 & 0 Toth=wox3)) S (Toth=begin_o)) )
398 LTLSPEC G (Toth=end_o) -> (O!(Toth=rox4 &
399 0 (Toth!=rox4 & 0 Toth=rox4)) S (Toth=begin_o)) )
400 LTLSPEC G (Toth=end_o) -> (O!(Toth=wox4 &
401 0 (Toth!=wox4 & 0 Toth=wox4)) S (Toth=begin_o)) )
402 LTLSPEC G (Toth=end_o) -> (O!(Toth=rox5 &
403 0 (Toth!=rox5 & 0 Toth=rox5)) S (Toth=begin_o)) )
404 LTLSPEC G (Toth=end_o) -> (O!(Toth=wox5 &
405 0 (Toth!=wox5 & 0 Toth=wox5)) S (Toth=begin_o)) )
406 --------------------------------------------------
407 ----no r1 and r2 on x without write for one of them--
408 --T1 and Toth
409 LTLSPEC G !(Toth=rox1 & T1=r1x1)
410 LTLSPEC G !(Toth=rox2 & T1=r1x2)
411 LTLSPEC G !(Toth=rox3 & T1=r1x3)
412 --T1 and T2
413 LTLSPEC G !(T1=r1x2 & T2=r2x2)
Chapter C. NuSMV model for iterated transactions with LTL specification

414 LTLSPEC G !(T1=r1x3 & T2=r2x3)
415 --T2 and Toth
416 LTLSPEC G !(Toth=rox2 & T2=r2x2)
417 LTLSPEC G !(Toth=rox3 & T2=r2x3)
418 LTLSPEC G !(Toth=rox4 & T2=r2x4)
419 ---------------------------------------------------------------
420 --protocol
421 ---------------------------------------------------------------
422 --T1 precedes T2
423 LTLSPEC G (((Toth=wox3 & O(T2=w2x3 & O(T1=w1x3)))))
424 (Toth=wox2 & O(T2=w2x2 & O(T1=w1x2)))
425 |
426 ((T2=w2x3 & O(Toth=wox3 & O(T1=w1x3))) &
427 (T2=w2x2 & O(Toth=wox2 & O(T1=w1x2)))
428 |
429 ((T2=w2x3 & O(T1=w1x3 & O(Toth=wox3))) &
430 (T2=w2x2 & O(T1=w1x2 & O(Toth=wox3)))
431 ->
432 !(((T1=w1x3 & O(T2=w2x3 & O(T2=w2x2 & O(T1=w1x2)))) |
433 (T2=w2x3 & O(T1=w1x3 & O(T1=w1x2 & O(T2=w2x2))))
434 --T2 precedes T1
435 LTLSPEC G (((Toth=wox3 & O(T1=w1x3 & O(T2=w2x3))) &
436 (Toth=wox2 & O(T1=w1x2 & O(T2=w2x2)))
437 |
438 ((T1=w1x3 & O(Toth=wox3 & O(T2=w2x3))) &
439 (T1=w1x2 & O(Toth=wox2 & O(T2=w2x2)))
440 |
441 ((T1=w1x3 & O(T2=w2x3 & O(Toth=wox3))) &
442 (T1=w1x2 & O(T2=w2x2 & O(Toth=wox3)))
443 ->
444 !(((T1=w1x3 & O(T2=w2x3 & O(T2=w2x2 & O(T1=w1x2)))) |
445 (T2=w2x3 & O(T1=w1x3 & O(T1=w1x2 & O(T2=w2x2))))
446 --------------------Future------------------------------
447 --T1 precedes T2
448 LTLSPEC G (((T1=w1x3 & F(T2=w2x3 & F(Toth=wox3))) &
449 (T1=w1x2 & F(T2=w2x2 & F(Toth=wox2)))
450 |
451 ((T1=w1x3 & F(Toth=wox3 & F(T2=w2x3))) &
452 (T1=w1x2 & F(Toth=wox2 & F(T2=w2x2))))
Chapter C. NuSMV model for iterated transactions with LTL specification

453 | 
454 \((\text{Toth}=\text{wox}_3 \& \text{F}(\text{T1}=\text{w}_1 \text{x}_3 \& \text{F}(\text{T2}=\text{w}_2 \text{x}_3))) \& \) 
455 \((\text{Toth}=\text{wox}_2 \& \text{F}(\text{T1}=\text{w}_1 \text{x}_2 \& \text{F}(\text{T2}=\text{w}_2 \text{x}_2)))\) 
456 -> 
457 \!( (\text{Ti}=\text{w}_1 \text{x}_2 \& \text{F}(\text{T2}=\text{w}_2 \text{x}_2 \& \text{F}(\text{T2}=\text{w}_2 \text{x}_3 \& \text{F}(\text{T1}=\text{w}_1 \text{x}_3))) \& \) 
458 \((\text{Toth}=\text{wox}_2 \& \text{F}(\text{T1}=\text{w}_1 \text{x}_2 \& \text{F}(\text{T1}=\text{w}_1 \text{x}_3 \& \text{F}(\text{T2}=\text{w}_2 \text{x}_3))))\) 
459 -- T2 precedes T1 
460 LTLSPEC G (((\text{T2}=\text{w}_2 \text{x}_3 \& \text{F}(\text{T1}=\text{w}_1 \text{x}_3 \& \text{F}(\text{Toth}=\text{wox}_3))) \& 
461 \((\text{T2}=\text{w}_2 \text{x}_2 \& \text{F}(\text{T1}=\text{w}_1 \text{x}_2 \& \text{F}(\text{Toth}=\text{wox}_2)))\) 
462 | 
463 \((\text{T2}=\text{w}_2 \text{x}_3 \& \text{F}(\text{Toth}=\text{wox}_3 \& \text{F}(\text{Ti}=\text{w}_1 \text{x}_3))) \& 
464 \((\text{T2}=\text{w}_2 \text{x}_2 \& \text{F}(\text{Toth}=\text{wox}_2 \& \text{F}(\text{Ti}=\text{w}_1 \text{x}_2)))\) 
465 | 
466 \((\text{Toth}=\text{wox}_3 \& \text{F}(\text{T2}=\text{w}_2 \text{x}_3 \& \text{F}(\text{Ti}=\text{w}_1 \text{x}_3))) \& 
467 \((\text{Toth}=\text{wox}_2 \& \text{F}(\text{T2}=\text{w}_2 \text{x}_2 \& \text{F}(\text{Ti}=\text{w}_1 \text{x}_2)))\) 
468 -> 
469 \!( (\text{Ti}=\text{w}_1 \text{x}_2 \& \text{F}(\text{T2}=\text{w}_2 \text{x}_2 \& \text{F}(\text{T2}=\text{w}_2 \text{x}_3 \& \text{F}(\text{Ti}=\text{w}_1 \text{x}_3))) \& 
470 \((\text{T2}=\text{w}_2 \text{x}_2 \& \text{F}(\text{Ti}=\text{w}_1 \text{x}_2 \& \text{F}(\text{Ti}=\text{w}_1 \text{x}_3 \& \text{F}(\text{T2}=\text{w}_2 \text{x}_3))))\) 
470 FAIRNESS running
Appendix D

NuSMV model for iterated transactions with CTL specifications

1 MODULE move(T_c,n,T0,T1,TSM,p_c)
2 ASSIGN
3 next(T_c):=case
4 T_c=begin_1 &n=1&(((TSM[1][1]=0)|(TSM[1][1]!=0&(T_c=r1x1)))
5 &((TSM[2][1]=0)
6 &((TSM[3][1]=0)|(TSM[3][1]!=0&(T_c=rox1)))):r1x1;
7 T_c=r1x1 &n=1 :w1x1;
8 T_c=w1x1 &n=1&(((TSM[1][2]=0)|(TSM[1][2]!=0&(T_c=r1x2)))
9 &((TSM[2][2]=0)|(TSM[2][2]!=0&(T0=r2x2)))
10 &((TSM[3][2]=0)|(TSM[3][2]!=0&(T1=rox2))):r1x2;
11 T_c=r1x2 &n=1 :w1x2;
12 T_c=w1x2 &n=1&(((TSM[1][3]=0)|(TSM[1][3]!=0&(T_c=r1x3)))
13 &((TSM[2][3]=0)|(TSM[2][3]!=0&(T0=r2x3)))
14 &((TSM[3][3]=0)|(TSM[3][3]!=0&(T1=rox3))):r1x3;
15 T_c=r1x3 &n=1 :w1x3;
16 T_c=w1x3 &n=1 :end_1;
18 T_c=end_1 &n=1 :begin_1;
22 T_c=begin_2 &n=2&(((TSM[1][2]=0)|(TSM[1][2]!=0&(T0=r1x2)))
23 &((TSM[2][2]=0)|(TSM[2][2]!=0&(T_c=r2x2)))
Chapter D. NuSMV model for iterated transactions with CTL specifications

24 &((TSM[3][2]==0) | (TSM[3][2]!=0 & !(T_c=rox2))) : r2x2;
25 T_c=r2x2 & n=2 : w2x2;
26 T_c=w2x2 & n=2 & !((TSM[1][3]==0) | (TSM[1][3]!=0 & !(T_c=r2x3)))
27 &((TSM[2][3]==0) | (TSM[2][3]!=0 & !(T_c=r2x3)))
28 &((TSM[3][3]==0) | (TSM[3][3]!=0 & !(T_c=r2x3)))
31 T_c=r2x3 & n=2 : w2x3;
32 T_c=w2x3 & n=2 & !((TSM[1][4]==0) --does not execute x4
33 &((TSM[2][4]==0) | (TSM[2][4]!=0 & !(T_c=r2x4)))
34 &((TSM[3][4]==0) | (TSM[3][4]!=0 & !(T_c=r2x4)))
36 T_c=r2x4 & n=2 : w2x4;
37 T_c=w2x4 & n=2 : end_2;
38 --this to keep transaction 2 repeats
39 T_c=end_2 & n=2 : begin_2;
40 T_c=begin_0 & n=3 & !((TSM[1][1]==0) | (TSM[1][1]!=0 & !(T0=r1x1)))
41 &!(TSM[2][1]==0)
42 &!(TSM[3][1]==0) | (TSM[3][1]!=0 & !(T_c=rox1)))
43 !(TSM[1][1]>0 & TSM[1][1]<TSM[3][1] & TSM[1][2]=0 ) : rox1;
44 T_c=rox1 & n=3 : wox1;
45 T_c=wox1 & n=3 & !((TSM[1][2]==0) | (TSM[1][2]!=0 & !(T0=r1x2)))
46 &!(TSM[2][2]==0) | (TSM[2][2]!=0 & !(T1=r2x2))
47 &!(TSM[3][2]==0) | (TSM[3][2]!=0 & !(T_c=rox2))) : rox2;
48 T_c=rox2 & n=3 : wox2;
49 T_c=wox2 & n=3 & !((TSM[1][3]==0) | (TSM[1][3]!=0 & !(T0=r1x3)))
50 &!(TSM[2][3]==0) | (TSM[2][3]!=0 & !(T1=r2x3))
51 &!(TSM[3][3]==0) | (TSM[3][3]!=0 & !(T_c=rox3)))
54 T_c=rox3 & n=3 : wox3;
55 T_c=wox3 & n=3 & !((TSM[1][4]==0) --does not execute x4
56 &!(TSM[2][4]==0) | (TSM[2][4]!=0 & !(T1=r2x4))
57 &!(TSM[3][4]==0) | (TSM[3][4]!=0 & !(T_c=rox4)))
59 T_c=rox4 & n=3 : wox4;
60 T_c=wox4 & n=3 & !((TSM[1][5]==0) --does not execute x5
61 &!(TSM[2][5]=0)
62 &!(TSM[3][5]=0) | (TSM[3][5]!=0 & !(T_c=rox5))) : rox5;
Chapter D. NuSMV model for iterated transactions with CTL specifications

63 \( T_c=rox5 \) \& \( n=3 \) : \( wox5 \);
64 \( T_c=wox5 \) \& \( n=3 \) : \( end_o \);
65 \(-- \) this to keep transaction 2 repeats
66 \( T_c=end_o \) \& \( n=3 \) : \( begin_o \);
67 1 : \( T_c \);
68 esac;
69 ---------------------------------------------
70 next(p_c):=case
71 p_c>=0 \& p_c<11\&(\( T_c=begin_1 \) \& next(T_c)=r1x1)
72 | \( T_c=w1x1 \) \& next(T_c)=r1x2)
73 | \( T_c=w1x2 \) \& next(T_c)=r1x3)
74 | \( T_c=begin_2 \) \& next(T_c)=r2x2)
75 | \( T_c=w2x2 \) \& next(T_c)=r2x3)
76 | \( T_c=w2x3 \) \& next(T_c)=r2x4)
77 | \( T_c=begin_o \) \& next(T_c)=rox1)
78 | \( T_c=wox1 \) \& next(T_c)=rox2)
79 | \( T_c=wox2 \) \& next(T_c)=rox3)
80 | \( T_c=wox3 \) \& next(T_c)=rox4)
81 | \( T_c=wox4 \) \& next(T_c)=rox5))): p_c+1;
82 p_c=11 : 1;
83 1 : p_c;
84 esac;
85 next(TSM[1][1]):= case
86 T_c!= r1x1 \& next(T_c)=r1x1 \& n=1:next(p_c);
87 next(T_c)= end_1 : 0;
88 1 : TSM[1][1];
89 esac;
90 next(TSM[1][2]):= case
91 T_c!= r1x2 \& next(T_c)=r1x2 \& n=1:next(p_c);
92 next(T_c)= end_1 : 0;
93 1 : TSM[1][2];
94 esac;
95 next(TSM[1][3]):= case
96 T_c!= r1x3 \& next(T_c)=r1x3 \& n=1:next(p_c)
97 next(T_c)= end_1 : 0;
98 1 : TSM[1][3];
99 esac;
100 next(TSM[1][4]):=0;
101 next(TSM[1][5]):=0;
next(TSM[2][1]) := 0;
next(TSM[2][2]) := case
T_c != r2x2 & next(T_c) = r2x2 & n = 2 : next(p_c);
next(T_c) = end_2 : 0;
1
TSM[2][2];
esac;
next(TSM[2][3]) := case
T_c != r2x3 & next(T_c) = r2x3 & n = 2 : next(p_c);
next(T_c) = end_2 : 0;
next(TSM[2][3]);
esac;
next(TSM[2][4]) := case
T_c != r2x4 & next(T_c) = r2x4 & n = 2 : next(p_c);
next(T_c) = end_2 : 0;
next(TSM[2][4]);
esac;
next(TSM[2][5]) := 0;

next(TSM[3][1]) := case
T_c != roxl & next(T_c) = roxl & n = 3 : next(p_c);
next(T_c) = end_0 : 0;
next(TSM[3][1]);
esac;
next(TSM[3][2]) := case
T_c != rox2 & next(T_c) = rox2 & n = 3 : next(p_c);
next(T_c) = end_0 : 0;
next(TSM[3][2]);
esac;
next(TSM[3][3]) := case
T_c != rox3 & next(T_c) = rox3 & n = 3 : next(p_c);
next(T_c) = end_0 : 0;
next(TSM[3][3]);
esac;
next(TSM[3][4]) := case
T_c != rox4 & next(T_c) = rox4 & n = 3 : next(p_c);
next(T_c) = end_0 : 0;
next(TSM[3][4]);
esac;
141 next(TSM[3][5]) := case
142 T_c != rox5 & next(T_c) = rox5 & n=3 : next(p_c);
143 next(T_c) = end_o
144 1
145 esac;
146 -----------------------------------------------
147 MODULE main
148 ----------------------------------------------
149 VAR
150 T1 : {begin_1, r1x1, w1x1, r1x2, w1x2, r1x3, w1x3, end_1};
151 T2 : {begin_2, r2x2, w2x2, r2x3, w2x3, r2x4, w2x4, end_2};
152 Toth : {begin_o, rox1, wox1, rox2, wox2, rox3, wox3, rox4, wox4, rox5, wox5, end_o};
153 -----------------------------------------------
154 TSM : array 1 .. 3 of array 1 .. 5 of 0..11;
155 -----------------------------------------------
156 p_c: 0..11; -- priority counter
157 ----------------------------------------------
158 T_1 : process move(T1, 1, T2, Toth, TSM, p_c);
159 T_2 : process move(T2, 2, T1, Toth, TSM, p_c);
160 T_3 : process move(Toth, 3, T1, T2, TSM, p_c);
161 ----------------------------------------------
162 ASSIGN
163 init(T1) := begin_1;
164 init(T2) := begin_2;
165 init(Toth) := begin_o;
166 init(p_c) := 0;
167 -----------------------------------------------
168 init(TSM[1][1]) := 0;
169 init(TSM[1][2]) := 0;
170 init(TSM[1][3]) := 0;
171 init(TSM[1][4]) := 0;
172 init(TSM[1][5]) := 0;
173 init(TSM[2][1]) := 0;
174 init(TSM[2][2]) := 0;
175 init(TSM[2][3]) := 0;
176 init(TSM[2][4]) := 0;
177 init(TSM[2][5]) := 0;
178 init(TSM[3][1]) := 0;
179 init(TSM[3][2]) := 0;
Chapter D. NuSMV model for iterated transactions with CTL specifications 172

180 init(TSM[3][2]) := 0;
181 init(TSM[3][3]) := 0;
182 init(TSM[3][4]) := 0;
183 init(TSM[3][5]) := 0;
184 -------------------------------
185 -- Fairness
186 -------------------------------
187 FAIRNESS (T1=end_1)
188 FAIRNESS (T2=end_2)
189 FAIRNESS (Toth=end_o)
Appendix E

NuSMV model for modified protocol

1 MODULE move(T_c,n,T0,T1,p12,p13,p23,c1,c2)
2 ASSIGN
3 next(T_c):=case
4 T_c=begin_1&T1=begin_m&n=1&(!T_c=r1x1)&(!T1=rmx1)) :r1x1;
5 T_c=r1x1 &T1=begin_m&n=1 :w1x1;
6 T_c=w1x1 &T1=begin_m&n=1&(!T_c=r1x2)&(!T0=r2x2))&
7 (!T1=rmx2)) :r1x2;
8 T_c=r1x2 &T1=begin_m&n=1 :w1x2;
9 T_c=w1x2 &T1=begin_m&n=1&(!T_c=r1x3)&(!T0=r2x3))&
10 (!T1=rmx3))&p12& :r1x3;
11 T_c=r1x3 &T1=begin_m&n=1 :w1x3;
12 T_c=w1x3 &T1=begin_m&n=1 :end_1;
13 -- this to keep transaction 1 repeats
14 T_c=end_1 &T1=begin_m&n=1 :begin_1;
15 T_c=begin_2&T1=begin_m&n=2&(!T0=r1x2)&(!T_c=r2x2))&
16 (!T1=rmx2)) :r2x2;
17 T_c=r2x2 &T1=begin_m&n=2 :w2x2;
18 T_c=w2x2 &T1=begin_m&n=2&(!T0=r1x3)&(!T_c=r2x3))&
19 (!T1=rmx3))&p12 :r2x3;
20 T_c=r2x3 &T1=begin_m&n=2 :w2x3;
21 T_c=w2x3 &T1=begin_m&n=2&(!T_c=r2x4))&(!T1=rmx4)) :r2x4;
22 T_c=r2x4 &T1=begin_m&n=2 :w2x4;
23 T_c=w2x4 &T1=begin_m&n=2 :end_2;
24 -- this to keep transaction 2 repeats
25 T_c=end_2 &T1=begin_m&n=2 :begin_2;
NuSMV model for modified protocol

\[
\begin{align*}
26 & T_c = \text{begin}_m \land n = 3 & : \text{rmx}_1; \\
27 & T_c = \text{rmx}_1 \land n = 3 & : \text{wmx}_1; \\
28 & T_c = \text{wmx}_1 \land n = 3 & : \text{rmx}_2; \\
29 & T_c = \text{rmx}_2 \land n = 3 & : \text{wmx}_2; \\
30 & T_c = \text{wmx}_2 \land n = 3 & : \text{rmx}_3; \\
31 & T_c = \text{rmx}_3 \land n = 3 & : \text{wmx}_3; \\
32 & T_c = \text{wmx}_3 \land n = 3 & : \text{rmx}_4; \\
33 & T_c = \text{rmx}_4 \land n = 3 & : \text{wmx}_4; \\
34 & T_c = \text{wmx}_4 \land n = 3 & : \text{rmx}_5; \\
35 & T_c = \text{rmx}_5 \land n = 3 & : \text{wmx}_5; \\
36 & T_c = \text{wmx}_5 \land n = 3 & : \text{end}_m; \\
37 & \text{-- this to keep transaction } m \text{ repeats} \\
38 & T_c = \text{end}_m \land n = 3 & : \text{begin}_m; \\
39 & 1 & : T_c; \\
40 & \text{esac;} \\
41 & \text{---------------------------------------------}
\end{align*}
\]

\[
\begin{align*}
42 & \text{next}(p_{12}) := \text{case} \\
43 & (\text{next}(T_c) = \text{r}_1 x_2 \mid \text{next}(T_c) = \text{w}_2 x_2 \mid \text{next}(T_c) = \text{r}_1 x_3 \mid \text{next}(T_c) = \text{w}_1 x_3) \land T_0 = \text{begin}_2 & : 1; \\
44 & \text{next}(T_c) = \text{end}_1) \land T_0 = \text{begin}_2 & : 1; \\
45 & (\text{next}(T_c) = \text{r}_2 x_2 \mid \text{next}(T_c) = \text{w}_2 x_2 \mid \text{next}(T_c) = \text{r}_2 x_3 \mid \text{next}(T_c) = \text{w}_2 x_3) \land (T_0 = \text{begin}_1) \\
46 & \text{next}(T_c) = \text{r}_2 x_4) \land T_0 = \text{begin}_2 & : 0; \\
47 & (\text{next}(T_c) = \text{r}_1 x_1 \mid T_0 = \text{w}_1 x_1) & : p_{12}; \\
48 & 1 & : p_{12}; \\
49 & \text{esac;} \\
50 & \text{---------------------------------------------}
\end{align*}
\]

\[
\begin{align*}
51 & \text{next}(p_{13}) := \text{case} \\
52 & (\text{next}(T_c) = \text{end}_1 \mid \text{next}(T_c) = \text{begin}_1) \land n = 1 \land T_1 = \text{begin}_m & : 1; \\
53 & \text{next}(T_c) = \text{end}_m \mid \text{next}(T_c) = \text{begin}_m) \land n = 1 \land T_0 = \text{begin}_1 & : 0; \\
54 & 1 & : p_{13}; \\
55 & \text{esac;} \\
56 & \text{---------------------------------------------}
\end{align*}
\]

\[
\begin{align*}
57 & \text{next}(p_{23}) := \text{case} \\
58 & (\text{next}(T_c) = \text{end}_2 \mid \text{next}(T_c) = \text{begin}_2) \land n = 2 \land T_1 = \text{begin}_m & : 1; \\
59 & \text{next}(T_c) = \text{rmx}_1 \mid \text{next}(T_c) = \text{wmx}_1 \mid \text{next}(T_c) = \text{begin}_m) \\
60 & \text{next}(T_c) = \text{end}_m \mid n = 3 \land T_1 = \text{begin}_2 & : 0; \\
61 & 1 & : p_{23}; \\
62 & \text{esac;} \\
63 & \text{---------------------------------------------}
\end{align*}
\]

\[
\begin{align*}
64 & \text{next}(c_1) := \text{case}
\end{align*}
\]
Chapter E. NuSMV model for modified protocol

65 n=1 & T_c!=next(T_c) : 1;
66 1
67 esac;
68 ----------------------------------------------
69 next(c2):=case
70 n=2 & T_c!=next(T_c) : 1;
71 1
72 esac;
73 ----------------------------------------------
74 MODULE main
75 ----------------------------------------------
76 VAR
77 T1 : {begin_1, r1x1, w1x1, r1x2, w1x2, r1x3, w1x3, end_1};
78 T2 : {begin_2, r2x2, w2x2, r2x3, w2x3, r2x4, w2x4, end_2};
79 Tm : {begin_m, rmx1, wmx1, rmx2, wmx2, rmx3, wmx3, rmx4,
80 wmx4, rmx5, wmx5, end_m};
81 p12:boolean;
82 p13:boolean;
83 p23:boolean;
84 c1:boolean;
85 c2:boolean;
86 ----------------------------------------------
87 T_1 : process move(T1, 1, T2, Tm, p12, p13, p23, c1, c2);
88 T_2 : process move(T2, 2, T1, Tm, p12, p13, p23, c1, c2);
89 T_3 : process move(Tm, 3, T1, T2, p12, p13, p23, c1, c2);
90 ----------------------------------------------
91 ASSIGN
92 ----------------------------------------------
93 init(T1) :=begin_1;
94 init(T2) :=begin_2;
95 init(Tm) :=begin_m;
96 ----------------------------------------------
97 init(p12) :=0;
98 init(p13) :=0;
99 init(p23) :=0;
100 init(c1) :=0;
101 init(c2) :=0;
102 ----------------------------------------------
103 FAIRNESS !(T1=begin_1)
Chapter E. NuSMV model for modified protocol

104 FAIRNESS !(T1=r1x1)
105 FAIRNESS !(T1=w1x1)
106 FAIRNESS !(T1=r1x2)
107 FAIRNESS !(T1=w1x2)
108 FAIRNESS !(T1=r1x3)
109 FAIRNESS !(T1=w1x3)
110 FAIRNESS !(T1=end_1)
111 FAIRNESS !(T2=begin_2)
112 FAIRNESS !(T2=r2x2)
113 FAIRNESS !(T2=w2x2)
114 FAIRNESS !(T2=r2x3)
115 FAIRNESS !(T2=w2x3)
116 FAIRNESS !(T2=r2x4)
117 FAIRNESS !(T2=w2x4)
118 FAIRNESS !(T2=end_2)
119 FAIRNESS !(Tm=begin_m)
120 FAIRNESS !(Tm=rmx1)
121 FAIRNESS !(Tm=wmx1)
122 FAIRNESS !(Tm=rmx2)
123 FAIRNESS !(Tm=wmx2)
124 FAIRNESS !(Tm=rmx3)
125 FAIRNESS !(Tm=wmx3)
126 FAIRNESS !(Tm=rmx4)
127 FAIRNESS !(Tm=wmx4)
128 FAIRNESS !(Tm=rmx5)
129 FAIRNESS !(Tm=wmx5)
130 FAIRNESS !(Tm=end_m)
131 FAIRNESS running
132 -----------------------------------------------------------------------------
133 --SPECIFICATIONS
134 -----------------------------------------------------------------------------
135 ------read/write alternation ------
136 --For T1
137 LTLSPEC G (T1=r1x1 -> (F (T1=w1x1 & F (T1=r1x2))))
138 LTLSPEC G (T1=r1x2 -> (F (T1=w1x2 & F (T1=r1x3))))
139 LTLSPEC G (T1=r1x1 -> (F (T1=w1x1 & F (T1=r1x3))))
140 --For T2
141 LTLSPEC G (T2=r2x2 -> (F (T2=w2x2 & F (T2=r2x3))))
142 LTLSPEC G (T2=r2x3 -> (F (T2=w2x3 & F (T2=r2x4))))
143 LTLSPEC G (T2=r2x2 -> (F (T2=w2x2 & F (T2=r2x4))))
144 --For Tm
145 LTLSPEC G (Tm=rmx1 -> (F (Tm=wmx1 & F (Tm=rmx2))))
146 LTLSPEC G (Tm=rmx1 -> (F (Tm=wmx1 & F (Tm=rmx3))))
147 LTLSPEC G (Tm=rmx1 -> (F (Tm=wmx1 & F (Tm=rmx4))))
148 LTLSPEC G (Tm=rmx1 -> (F (Tm=wmx1 & F (Tm=rmx5))))
149 LTLSPEC G (Tm=rmx2 -> (F (Tm=wmx2 & F (Tm=rmx3))))
150 LTLSPEC G (Tm=rmx2 -> (F (Tm=wmx2 & F (Tm=rmx4))))
151 LTLSPEC G (Tm=rmx2 -> (F (Tm=wmx2 & F (Tm=rmx5))))
152 LTLSPEC G (Tm=rmx3 -> (F (Tm=wmx3 & F (Tm=rmx4))))
153 LTLSPEC G (Tm=rmx3 -> (F (Tm=wmx3 & F (Tm=rmx5))))
154 LTLSPEC G (Tm=rmx4 -> (F (Tm=wmx4 & F (Tm=rmx5))))
155 -------write implies Read ----------
156 --For T1
157 LTLSPEC G (T1=rlx1 -> (F (T1=w1x1 )))
158 LTLSPEC G (T1=rlx2 -> (F (T1=w1x2 )))
159 LTLSPEC G (T1=rlx3 -> (F (T1=w1x3 )))
160
161 --For T2
162 LTLSPEC G (T2=r2x2 -> (F (T2=w2x2 )))
163 LTLSPEC G (T2=r2x3 -> (F (T2=w2x3 )))
164 LTLSPEC G (T2=r2x4 -> (F (T2=w2x4 )))
165 --For Tm
166 LTLSPEC G (Tm=rmx1 -> (F (Tm=wmx1 )))
167 LTLSPEC G (Tm=rmx2 -> (F (Tm=wmx2 )))
168 LTLSPEC G (Tm=rmx3 -> (F (Tm=wmx3 )))
169 LTLSPEC G (Tm=rmx4 -> (F (Tm=wmx4 )))
170 LTLSPEC G (Tm=rmx5 -> (F (Tm=wmx5 )))
171 --Read/write step remains true until the next operation
172 --For T1
173 LTLSPEC G ((T1=w1x1)->!(T1=rlx1))
174 LTLSPEC G ((T1=w1x2)->!(T1=rlx2))
175 LTLSPEC G ((T1=w1x3)->!(T1=rlx3))
176 --
177 LTLSPEC G ((T1=rlx2)->!(T1=w1x1))
178 LTLSPEC G ((T1=rlx3)->!(T1=w1x2))
179 --For T2
180 LTLSPEC G ((T2=w2x2)->!(T2=r2x2))
181 LTLSPEC G ((T2=w2x3)->!(T2=r2x3))
Chapter E. NuSMV model for modified protocol

182 LTLSPEC G ((T2=w2x4)->!(T2=r2x4))
183 --
184 LTLSPEC G ((T2=r2x3)->!(T2=w2x2))
185 LTLSPEC G ((T2=r2x4)->!(T2=w2x3))
186 --For Tm
187 LTLSPEC G ((Tm=wmx1)->!(Tm=rmx1))
188 LTLSPEC G ((Tm=wmx2)->!(Tm=rmx2))
189 LTLSPEC G ((Tm=wmx3)->!(Tm=rmx3))
190 LTLSPEC G ((Tm=wmx4)->!(Tm=rmx4))
191 LTLSPEC G ((Tm=wmx5)->!(Tm=rmx5))
192 --
193 LTLSPEC G ((Tm=rmx2)->!(Tm=wmx1))
194 LTLSPEC G ((Tm=rmx3)->!(Tm=wmx2))
195 LTLSPEC G ((Tm=rmx4)->!(Tm=wmx3))
196 LTLSPEC G ((Tm=rmx5)->!(Tm=wmx4))
197 --At most one step occurs at each successive state
198 --Tl and T2
199 LTLSPEC G (!((!(Tl=rlx2)&!(T2=r2x2))&X((Tl=rlx2)&(T2=r2x2)))&
200 !(!!(Tl=rlx2)&!(T2=w2x2))&X((Tl=rlx2)&(T2=w2x2)))&
201 !(!!(Tl=wlx2)&!(T2=w2x2))&X((Tl=wlx2)&(T2=w2x2)))
202 LTLSPEC G (!((!(Tl=rlx3)&!(T2=r2x3))&X((Tl=rlx3)&(T2=r2x3)))&
203 !(!!(Tl=rlx3)&!(T2=w2x3))&X((Tl=rlx3)&(T2=w2x3)))
204 !(!!(Tl=wlx3)&!(T2=w2x3))&X((Tl=wlx3)&(T2=w2x3)))
205 LTLSPEC G (!((!(Tl=rlx3)&!(T2=rmx2))&X((Tl=rlx3)&(T2=rmx2)))&
206 --Tl and Tm
207 LTLSPEC G (!((!(Tl=rlx1)&!(T2=rmx1))&X((Tl=rlx1)&(T2=rmx1)))&
208 !(!!(Tl=rlx1)&!(T2=wmx1))&X((Tl=rlx1)&(T2=wmx1)))&
209 !(!!(Tl=wlx1)&!(T2=wmx1))&X((Tl=wlx1)&(T2=wmx1)))
210 LTLSPEC G (!((!(Tl=rlx2)&!(T2=rmx2))&X((Tl=rlx2)&(T2=rmx2)))&
211 !(!!(Tl=rlx2)&!(T2=wmx2))&X((Tl=rlx2)&(T2=wmx2)))
212 !(!!(Tl=wlx2)&!(T2=wmx2))&X((Tl=wlx2)&(T2=wmx2)))
213 LTLSPEC G (!((!(Tm=rmx2)&!(T2=r2x2))&X((Tm=rmx2)&(T2=r2x2)))&
214 !(!!(Tm=rmx2)&!(T2=w2x2))&X((Tm=rmx2)&(T2=w2x2)))
215 !(!!(Tm=wmx2)&!(T2=w2x2))&X((Tm=wmx2)&(T2=w2x2)))
Chapter E. NuSMV model for modified protocol

\[ \text{LTLSPEC } G((!(Tm=rmx3) & !(T2=r2x3)) \& X((T1=rmx3) & (T2=r2x3))) \& \\
\text{LTLSPEC } G((!(Tm=rmx3) & (T2=w2x3)) \& X((T1=rmx3) & (T2=w2x3))) \& \\
\text{LTLSPEC } G((!(Tm=wmx3) & !(T2=w2x3)) \& X((T1=wmx3) & (T2=w2x3))) \\
\]

\[ \text{LTLSPEC } G((!(Tm=wmx4) & !(T2=w2x4)) \& X((T1=wmx4) & (T2=w2x4))) \\
\]

\[ \text{--p5--} \]

\[ \text{--T1} \]

\[ \text{LTLSPEC } G((T1=begin_1) \rightarrow ((F!(T1=r1x1 \& \\
\text{F (T1=r1x1 \& F T1=r1x1))) U (T1=end_1))) \\
\text{LTLSPEC } G((T1=begin_1) \rightarrow ((F!(T1=w1x1 \& \\
\text{F (T1=w1x1 \& F T1=w1x1))) U (T1=end_1))) \\
\text{LTLSPEC } G((T1=begin_1) \rightarrow ((F!(T1=r1x2 \& \\
\text{F (T1=r1x2 \& F T1=r1x2))) U (T1=end_1))) \\
\text{LTLSPEC } G((T1=begin_1) \rightarrow ((F!(T1=r1x3 \& \\
\text{F (T1=r1x3 \& F T1=r1x3))) U (T1=end_1))) \\
\text{LTLSPEC } G((T1=begin_1) \rightarrow ((F!(T1=w1x3 \& \\
\text{F (T1=w1x3 \& F T1=w1x3))) U (T1=end_1))) \\
\]

\[ \text{--T2} \]

\[ \text{LTLSPEC } G((T2=begin_2) \rightarrow ((F!(T2=r2x2 \& \\
\text{F (T2=r2x2 \& F T2=r2x2))) U (T2=end_2))) \\
\text{LTLSPEC } G((T2=begin_2) \rightarrow ((F!(T2=w2x2 \& \\
\text{F (T2=w2x2 \& F T2=w2x2))) U (T2=end_2))) \\
\text{LTLSPEC } G((T2=begin_2) \rightarrow ((F!(T2=r2x3 \& \\
\text{F (T2=r2x3 \& F T2=r2x3))) U (T2=end_2))) \\
\text{LTLSPEC } G((T2=begin_2) \rightarrow ((F!(T2=w2x3 \& \\
\text{F (T2=w2x3 \& F T2=w2x3))) U (T2=end_2))) \\
\text{LTLSPEC } G((T2=begin_2) \rightarrow ((F!(T2=r2x4 \& \\
\text{F (T2=r2x4 \& F T2=r2x4))) U (T2=end_2))) \\
\text{LTLSPEC } G((T2=begin_2) \rightarrow ((F!(T2=w2x4 \& \\
\text{F (T2=w2x4 \& F T2=w2x4))) U (T2=end_2))) \\
\]

\[ \text{--Tm} \]

\[ \text{LTLSPEC } G((Tm=begin_m) \rightarrow ((F!(Tm=rmx1 \& \\
\text{F (Tm=rmx1 \& F Tm=rmx1))) U (Tm=end_m))) \\
\]
LTLSPEC $G(\text{Tm}=\text{begin}_m \rightarrow ((\neg F(\text{Tm}=\text{wm}x_1 \land 
\text{Tm}=\text{end}_m)))$)

LTLSPEC $G(\text{Tm}=\text{begin}_m \rightarrow ((\neg F(\text{Tm}=\text{rm}x_2 \land
\text{Tm}=\text{end}_m)))$)

LTLSPEC $G(\text{Tm}=\text{begin}_m \rightarrow ((\neg F(\text{Tm}=\text{wm}x_3 \land
\text{Tm}=\text{end}_m)))$)

LTLSPEC $G(\text{Tm}=\text{begin}_m \rightarrow ((\neg F(\text{Tm}=\text{rm}x_4 \land
\text{Tm}=\text{end}_m)))$)

LTLSPEC $G(\text{Tm}=\text{begin}_m \rightarrow ((\neg F(\text{Tm}=\text{wm}x_4 \land
\text{Tm}=\text{end}_m)))$)

LTLSPEC $G(\text{Tm}=\text{begin}_m \rightarrow ((\neg F(\text{Tm}=\text{rm}x_5 \land
\text{Tm}=\text{end}_m)))$)

LTLSPEC $G(\text{Tm}=\text{begin}_m \rightarrow ((\neg F(\text{Tm}=\text{wm}x_5 \land
\text{Tm}=\text{end}_m)))$)

LTLSPEC $G(\text{Tm}=\text{begin}_m \rightarrow (\text{X}(\neg \neg \neg \neg \neg \neg\text{Tm}=\text{end}_m)))$)

LTLSPEC $G(\text{Tm}=\text{begin}_m \rightarrow (\text{X}(\neg \neg \neg \neg \neg \neg\text{Tm}=\text{end}_m)))$)

LTLSPEC $G(\text{Tm}=\text{begin}_m \rightarrow (\text{X}(\neg \neg \neg \neg \neg \neg\text{Tm}=\text{end}_m)))$)

LTLSPEC $G(\text{Tm}=\text{begin}_m \rightarrow (\text{X}(\neg \neg \neg \neg \neg \neg\text{Tm}=\text{end}_m)))$)

LTLSPEC $G(\text{Tm}=\text{begin}_m \rightarrow (\text{X}(\neg \neg \neg \neg \neg \neg\text{Tm}=\text{end}_m)))$)

LTLSPEC $G(\text{Tm}=\text{begin}_m \rightarrow (\text{X}(\neg \neg \neg \neg \neg \neg\text{Tm}=\text{end}_m)))$)

LTLSPEC $G(\text{Tm}=\text{begin}_m \rightarrow (\text{X}(\neg \neg \neg \neg \neg \neg\text{Tm}=\text{end}_m)))$)

LTLSPEC $G(\text{Tm}=\text{begin}_m \rightarrow (\text{X}(\neg \neg \neg \neg \neg \neg\text{Tm}=\text{end}_m)))$)

LTLSPEC $G(\text{Tm}=\text{begin}_m \rightarrow (\text{X}(\neg \neg \neg \neg \neg \neg\text{Tm}=\text{end}_m)))$)

LTLSPEC $G(\text{Tm}=\text{begin}_m \rightarrow (\text{X}(\neg \neg \neg \neg \neg \neg\text{Tm}=\text{end}_m)))$)

LTLSPEC $G(\text{Tm}=\text{begin}_m \rightarrow (\text{X}(\neg \neg \neg \neg \neg \neg\text{Tm}=\text{end}_m)))$)

LTLSPEC $G(\text{Tm}=\text{begin}_m \rightarrow (\text{X}(\neg \neg \neg \neg \neg \neg\text{Tm}=\text{end}_m)))$)

LTLSPEC $G(\text{Tm}=\text{begin}_m \rightarrow (\text{X}(\neg \neg \neg \neg \neg \neg\text{Tm}=\text{end}_m)))$)

LTLSPEC $G(\text{Tm}=\text{begin}_m \rightarrow (\text{X}(\neg \neg \neg \neg \neg \neg\text{Tm}=\text{end}_m)))$)

LTLSPEC $G(\text{Tm}=\text{begin}_m \rightarrow (\text{X}(\neg \neg \neg \neg \neg \neg\text{Tm}=\text{end}_m)))$)

LTLSPEC $G(\text{Tm}=\text{begin}_m \rightarrow (\text{X}(\neg \neg \neg \neg \neg \neg\text{Tm}=\text{end}_m)))$)

LTLSPEC $G(\text{Tm}=\text{begin}_m \rightarrow (\text{X}(\neg \neg \neg \neg \neg \neg\text{Tm}=\text{end}_m)))$)

LTLSPEC $G(\text{Tm}=\text{begin}_m \rightarrow (\text{X}(\neg \neg \neg \neg \neg \neg\text{Tm}=\text{end}_m)))$)

LTLSPEC $G(\text{Tm}=\text{begin}_m \rightarrow (\text{X}(\neg \neg \neg \neg \neg \neg\text{Tm}=\text{end}_m)))$)

LTLSPEC $G(\text{Tm}=\text{begin}_m \rightarrow (\text{X}(\neg \neg \neg \neg \neg \neg\text{Tm}=\text{end}_m)))$)

LTLSPEC $G(\text{Tm}=\text{begin}_m \rightarrow (\text{X}(\neg \neg \neg \neg \neg \neg\text{Tm}=\text{end}_m)))$)

LTLSPEC $G(\text{Tm}=\text{begin}_m \rightarrow (\text{X}(\neg \neg \neg \neg \neg \neg\text{Tm}=\text{end}_m)))$)

LTLSPEC $G(\text{Tm}=\text{begin}_m \rightarrow (\text{X}(\neg \neg \neg \neg \neg \neg\text{Tm}=\text{end}_m)))$)

LTLSPEC $G(\text{Tm}=\text{begin}_m \rightarrow (\text{X}(\neg \neg \neg \neg \neg \neg\text{Tm}=\text{end}_m)))$)

LTLSPEC $G(\text{Tm}=\text{begin}_m \rightarrow (\text{X}(\neg \neg \neg \neg \neg \neg\text{Tm}=\text{end}_m)))$)

LTLSPEC $G(\text{Tm}=\text{begin}_m \rightarrow (\text{X}(\neg \neg \neg \neg \neg \neg\text{Tm}=\text{end}_m)))$)

LTLSPEC $G(\text{Tm}=\text{begin}_m \rightarrow (\text{X}(\neg \neg \neg \neg \neg \neg\text{Tm}=\text{end}_m)))$)

LTLSPEC $G(\text{Tm}=\text{begin}_m \rightarrow (\text{X}(\neg \neg \neg \neg \neg \neg\text{Tm}=\text{end}_m)))$)

LTLSPEC $G(\text{Tm}=\text{begin}_m \rightarrow (\text{X}(\neg \neg \neg \neg \neg \neg\text{Tm}=\text{end}_m)))$)

LTLSPEC $G(\text{Tm}=\text{begin}_m \rightarrow (\text{X}(\neg \neg \neg \neg \neg \neg\text{Tm}=\text{end}_m)))$)

LTLSPEC $G(\text{Tm}=\text{begin}_m \rightarrow (\text{X}(\neg \neg \neg \neg \neg \neg\text{Tm}=\text{end}_m)))$)

LTLSPEC $G(\text{Tm}=\text{begin}_m \rightarrow (\text{X}(\neg \neg \neg \neg \neg \neg\text{Tm}=\text{end}_m)))$)

LTLSPEC $G(\text{Tm}=\text{begin}_m \rightarrow (\text{X}(\neg \neg \neg \neg \neg \neg\text{Tm}=\text{end}_m)))$)

LTLSPEC $G(\text{Tm}=\text{begin}_m \rightarrow (\text{X}(\neg \neg \neg \neg \neg \neg\text{Tm}=\text{end}_m)))$)

LTLSPEC $G(\text{Tm}=\text{begin}_m \rightarrow (\text{X}(\neg \neg \neg \neg \neg \neg\text{Tm}=\text{end}_m)))$)

LTLSPEC $G(\text{Tm}=\text{begin}_m \rightarrow (\text{X}(\neg \neg \neg \neg \neg \neg\text{Tm}=\text{end}_m)))$)

LTLSPEC $G(\text{Tm}=\text{begin}_m \rightarrow (\text{X}(\neg \neg \neg \neg \neg \neg\text{Tm}=\text{end}_m)))$)
---no r1 and r2 on x without write for one of them---

--T1 and Tm

LTLSPEC G !((Tm=rmx1 & T1=r1x1))

LTLSPEC G !((Tm=rmx2 & T1=r1x2))

LTLSPEC G !((Tm=rmx3 & T1=r1x3))

--T1 and T2

LTLSPEC G !((T1=r1x2 & T2=r2x2))

LTLSPEC G !((T1=r1x3 & T2=r2x3))

--T2 and Tm

LTLSPEC G !((Tm=rmx2 & T2=r2x2))

LTLSPEC G !((Tm=rmx3 & T2=r2x3))

LTLSPEC G !((Tm=rmx4 & T2=r2x4))

--serializability condition for local and mobile

LTLSPEC G ( p12 -> (F (T1=w1x3 & F (T2=w2x3)) U !p12 )

LTLSPEC G ( !p12 -> (F (T2=w2x3 & F (T1=w1x3)) U p12 )

LTLSPEC G ( p13 -> (F (T1=w1x2 & F (Tm=wmx2)) U !p13 )

LTLSPEC G ( p23 -> (F (T2=w2x3 & F (Tm=wmx3)) U !p23 )

--Recurrent Sequencing (transaction iterates)

LTLSPEC G ((T2=begin_2 -> F T2=end_2) &

(T2=end_2 -> F T2=begin_2))

LTLSPEC G ((T1=begin_1 -> F T1=end_1) &

(T1=end_1 -> F T1=begin_1))

LTLSPEC G ((Tm=begin_m -> F Tm=end_m) &

(Tm=end_m -> F Tm=begin_m))

--infinitely often

LTLSPEC G F((Tm=begin_m -> F Tm=end_m) &

(Tm=end_m -> F Tm=begin_m))

LTLSPEC G F((T1=begin_1 -> F T1=end_1) &

(Tm=end_1 -> F Tm=begin_1))

LTLSPEC G F((T2=begin_2 -> F T2=end_2) &

(T2=end_2 -> F T2=begin_2))

--starvation

LTLSPEC F(T1=w1x1 ->(F G X(Tm=rmx1 & G F((Tm=begin_m ->

F Tm=end_m)&(Tm=begin_m -> F Tm=begin_m)) & T1=w1x1)))

LTLSPEC F (T1=w1x1 & X (G F((Tm=begin_m -> F Tm=end_m) &

(Tm=end_m-> F Tm=begin_m)) & T1=w1x1)))

--this show the starvation
Chapter E. NuSMV model for modified protocol

339 LTLSPEC G !(T1=w1x1 & X (G F((Tm=begin_m -> F Tm=end_m) &
440 (Tm=end_m -> F Tm=begin_m)) & T1=w1x1))
341 LTLSPEC G F((Tm=begin_m -> F Tm=end_m) & (Tm=end_m ->
342 F Tm=begin_m))
343 <->
344 G ((Tm=begin_m -> F Tm=end_m) & (Tm=end_m ->
345 F Tm=begin_m))
346 LTLSPEC G F((T1=begin_1 -> F T1=end_1) & (T1=end_1 ->
347 F T1=begin_1))
348 <->
349 G ((T1=begin_1 -> F T1=end_1) & (T1=end_1 ->
350 F T1=begin_1))
351 LTLSPEC G F((T2=begin_2 -> F T2=end_2) & (T2=end_2 ->
352 F T2=begin_2))
353 <->
354 G ((T2=begin_2 -> F T2=end_2) & (T2=end_2 ->
355 F T2=begin_2))