Flexible Verification of Transaction Timeliness and Isolation

Simin Cai, Barbara Gallina, Dag Nyström, and Cristina Seceleanu

Mälardalen Real-Time Research Centre, Mälardalen University,
Västerås, Sweden
{simin.cai,barbara.gallina,
dag.nystrom,cristina.seceleanu}@mdh.se

Abstract. Competitive real-time transaction management systems ideally must guarantee both transaction timeliness and isolation. While a high level of isolation can be achieved by the selected Concurrency Control (CC) mechanisms, the unpredictable delays introduced by such mechanisms could cause deadline misses of transactions. To avoid deadline misses, one solution is to select or design an appropriate CC mechanism that can guarantee timeliness and an acceptably relaxed level of isolation. However, trading-off isolation in favor of timeliness is not an easy task using existing analysis techniques. In this report we propose an approach to model a concurrent real time transaction system as a network of timed automata, and verify using model checking the consistency of the traded-off transaction timeliness and isolation. We propose a set of automaton patterns and skeletons as basic blocks for modeling the transactions as well as for modeling common CC mechanisms. These patterns and skeletons not only reduce the modeling efforts, but also enable easy adjustment of the CC mechanism, which can lead to the desired relaxation of isolation.

1 Introduction

In a DataBase Management System (DBMS) where concurrent transactions compete for data resources, transaction isolation, which means that transactions are not interfered by other concurrent transactions [13], is often desired to be guaranteed in order to maintain logical data consistency. Concurrency Control (CC) mechanisms are developed to regulate the execution of concurrent transactions so that the undesired interferences are prevented [10]. In a Real-Time DBMS (RTDBMS) where transactions must meet their specified deadlines, however, isolation and the entailed restrictive concurrency control may not be suitable. Long blocking, chained blocking, and arbitrary aborts and restarting introduced by concurrency control may reduce the predictability of the system and lead to deadline misses [8]. Therefore, it is common to relax isolation and exploit a less restrictive CC in RTDBMS, in order to ensure the timeliness of transactions [22].

A crucial task in developing an RTDBMS is to develop the appropriate CC that guarantees both transaction timeliness and an acceptable relaxation of isolation. To achieve this goal, the capability of reasoning about both timeliness and isolation in a unified analysis framework is necessary. Traditionally, the analyses of isolation and timeliness are separate tasks. The analysis of isolation in database community is based on dependencies exhibited in transaction execution history without incorporating timing [2]. In RTDBMS community, focuses used to be schedulability analysis [15] and experimental

studies with different CC algorithms [21], but no study have verified to which extent isolation can be achieved. In our previous work [5], we proposed a high-level process, called the DAGGERS process, for trading off the ACID (Atomicity, Consistency, Isolation and Durability) and transaction timeliness for developing a customized RTDBMS in a unified framework. In this process, we derive transaction models with conceptually relaxed ACID properties from system requirements, and model the actual transaction behaviors in timed automata. A collection of candidate run-time mechanisms, including candidate CC mechanisms, are also modeled as timed automata, and can be woven into the transaction models. The framework then model checks the desired ACID and timeliness. In case the desired properties are not satisfied, a new candidate mechanism is selected and replace the old one, and the model checking is started again.

The work of this report can be seen as part of the concretization of the DAGGERS process. Instead of covering the massive number of CC mechanisms proposed in literature, we focus on one common type of CC, called Pessimistic Concurrency Control (PCC) [10], which utilizes locking techniques to prevent interferences. The main contribution of this paper is an approach for modeling transaction behaviors under a selected PCC mechanism that allows for verification of both timeliness and isolation. This modeling approach is flexible, so that the models of different candidate PCC algorithms can be constructed with reduced efforts.

In this report we propose an approach for modeling concurrent transaction systems in timed automata, and model checking transaction timeliness and isolation using the UPPAAL model checker [19], which is the state-of-art model checker for real-time systems. We propose a set of automata skeletons modeling basic structures of transactions and common PCC mechanisms. The models of concrete systems are constructed based on these skeletons. We also propose a set of parametrized automata patterns that model finer-grained recurring activities. These patterns can be reused as basic building blocks to enrich the skeletons. In order to model and verify isolation, observer models introduced, and can be easily plugged into the model of the transaction system. Using our modeling approach, the RTDBMS designer can easily model different CC mechanisms with limited adjustments, and verify different relaxations of isolation flexibly.

The rest of the report is organized as follows. In Section 2 we discuss the related work. In Section 3 we present the background information about transactions, concurrency control, timed automata, and UPPAAL. Section 4 and 5 describe our modeling approach, the skeletons and patterns, for model checking timeliness and isolation respectively. We then describe the adjustments of models for different PCC algorithms and relaxation of isolation in Section 6. Finally, in Section 7 we conclude the paper.

2 Related Work

There have been efforts on modeling real-time transactions and verifying certain properties. Xiong et al. [23] propose the Real-Time ACTA framework to specify transactional properties, including isolation and timeliness, and verify the consistency of the specification. However, the models in Real-Time ACTA are conceptual and based on axioms. It is difficult to model different concurrency control algorithms. Our approach supports modeling of various CC algorithms, and reasoning about the low level behaviors of transactions under the modeled CC algorithm. Gallina et al. [12] propose modular

specification of advanced transaction models, and verification of isolation variants using the Alloy verification tool. There work models neither timing behaviors of transactions, nor concrete CC algorithms. Chkliaev et al. [6] propose a formal modeling and analysis approach for real-time transactions using the verification system PVS. Their work focuses on, however, the commit protocols, instead of concurrency control and isolation. Makni et al. [20] uses the verification tool SPIN to model real-time transactions with one concurrency control algorithm. Their work only tries to verify one particular CC algorithm, and they do not verify isolation.

Timed automata have been used to model real-time transactions. Lanotte et al. [18] propose a framework based on timed automata for modeling long running transactions with timing constraints. They have also proposed automata patterns for different committing protocols. Their work, different from ours, focuses on the modeling of committing protocols. The targeted properties they verify are related to transaction atomicity, instead of isolation. One similar work to ours is done by Kot [17], which models various real-time CC algorithms in UPPAAL. Despite the the similarity of the target to be modeled, this work is different from ours in several different aspects. First, the purpose of this work is to show the feasibility of modeling transactions and CC in UPPAAL, while we strive to contribute to general modeling approach for common CC mechanisms and propose skeletons and patterns to enhance the reusability of models. Second, our models are designed for flexible verification of various relaxations of isolation, while their work checks more general properties, like deadlocks, starvation, etc.

3 Preliminaries

3.1 The Concept of Transaction

A transaction is initially defined as a partially-ordered set of logically related operations that as a whole ensure the so-called ACID (Atomicity, Consistency, Isolation and Durability) properties [13]. Due to the semantic and performance restrictions of the full ACID assurance, the original "flat" transaction model is extended to allow various relaxations of the ACID properties [11]. In this paper, we assume that the isolation may be relaxed, whereas the atomicity, consistency and durability are fully ensured.

The logically related operations in a transaction may include database operations (read operations that read data from the database, and write operations that modify data in the database), and other calculations that do not interact with the database. Read and write operations are atomic operations, i.e., no interleaves occur within these operations. A read or write operation may be associated with a predicate that all tuples that satisfy the predicate are read or modified. In this paper we do not consider predicate-based operations.

A transaction is associated with transaction management primitives. The primitive *Begin* informs the transaction manager of the initiation of a transaction, whereas the primitives *Commit* and *Abort* indicate the transaction termination when all system resources possessed by the transaction are released. When a transaction commits, the changes made by this transaction are saved permanently in the database, and become visible to other transactions. When a transaction aborts, the changes made by this transaction are undone. While there may exist other primitives depending on the particular

transaction manager, *Begin*, *Commit* and *Abort* are the essential ones that define the boundary of a transaction, and thus are the primitives considered by our modeling approach. In an RTDBMS, transactions are associated with deadlines, meaning the transactions must complete within the specified time units after it begins.

Program 1.1 and 1.2 show two simple transactions T_0 and T_1 respectively. They are used as examples because they consist of read and write operations on the same data, which may lead to conflicts and expose the need for concurrency control. D_0 and D_1 are two data objects in the database. T_0 reads D_0, performs calculation, and writes the result into D_1. In case any error occurs during the calculation, T_0 will be aborted. T_1 simply updates the value of D_0 and D_1. In the following sections, we use r_i^j to denote the operation of T_i reading D_j, w_i^j denotes the operation of T_i writing D_j, c_i to denote the commit of T_i, and a_i to denote the abort of T_i. Both T_0 and T_1 have to meet their deadlines, which are 8 time units and 5 time units respectively.

Program 1.1. Transaction T_0

```
Begin
  read D0
  calculate
  if error, Abort
  write D1
Commit
```

Program 1.2. Transaction T_1

```
Begin
  write D0
  if error, Abort
  write D1
Commit
```

3.2 Concurrency Control

A transaction management system prevents data inconsistency caused by concurrent transactions accessing the same data via concurrency control. We focus on one type of concurrency control, Pessimistic Concurrency Control (PCC), which is commonly applied in modern database systems [10].

Pessimistic concurrency control protocols employ locking techniques to prevent interferences from concurrent transactions. Basically, a transaction needs to require a corresponding lock before it accesses the data, and release the lock after using the data. The CC manager receives requests for locking and unlocking data, and decides which transactions should be granted the lock, wait, or be aborted, according to the selected resolution algorithm. Among the family of proposed PCC algorithms, one of the most widely used is rigorous Two Phase Locking (2PL) [10]. According to rigorous 2PL, a transaction must acquire a write lock before writing to a data object, and must acquire a read lock or write lock before reading from a data object. If the data is already read-locked, rigorous transaction can still be granted another read lock, but they cannot acquire a write lock. If the data is write-locked, no other transactions can be granted to any lock. Transactions failing to acquire locks are put in a waiting queue. When a transaction unlocks a data, both read and write locks it has acquired are released, and the next transaction in the waiting queue will be granted the lock. Most importantly, a transaction is divided into two ordered phases, first a growing phase, and then a shrinking phase. In the growing phase the transaction can only require locks, whereas in the shrinking phase the transaction can only release locks. In rigorous 2PL, the shrinking phase occurs when the transaction commits or aborts.

Other PCC algorithms differ from rigorous 2PL on the types of locks, the decision upon lock conflicts, the point of time to acquire/release locks, etc. For examples, binary locking exploits only one type of lock, instead of read and write locks [10]. 2PL-HP [1] allows transactions with higher priority to lock a data that is already locked by another transaction with lower priority and abort it. Different PCC algorithms can also be designed to rule out different types of interferences, by adjusting the locking and unlocking time points [14].

3.3 Timed Automata and UPPAAL

A Timed Automaton (TA), proposed by Alur and Dill [4], is a finite-state automaton extended with real-valued clock variables. UPPAAL is one of the most popular and mature verification tools based on timed automata, and extends the standard framework of TA with utilization of discrete variables as well as other modeling features [19]. We use UPPAAL TA in this paper, and introduce the relevant syntax and semantics of UPPAAL TA using a simple example in this subsection. Figure 1 exhibits a network of TA, composed of timed automaton A1 and A2, that models a simple concurrent real-time system.

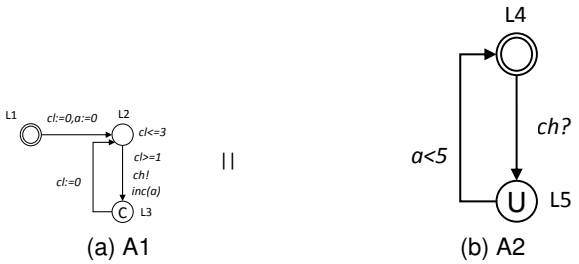


Fig. 1. A network of timed automata, composed of automaton A1 and automaton A2

A timed automaton consists of a finite set of locations connected by edges. A1 consists of locations L1, L2 and L3, among which L1 is the initial location. A clock variable *cl* is defined in A1, and progresses continuously. A discrete variable *a* is defined, and shared by A1 and A2. A1 moves from location to location along the edges. At each location, A1 may non-deterministically take a transition along an edge, or delay at the location. A location may have an **invariant**, which is a conjunction of clock constraints. The TA must leave the location before the invariant is violated. In Fig. 1a, A1 may stay at L2 until the value of *cl* reaches 3. Each edge may have a **guard**, an **action** and an **assignment**. A guard is a finite conjunction of constraints on discrete variables and/or clock variables. A transition can be taken, only if the guard over the edge is satisfied. An action is the synchronization with other automata via a **channel**. A binary channel is defined for a synchronization pair. An exclamation mark "!" following the channel name denotes the sending automaton, and a question mark "?" following the channel name denotes the receiver. If the expected sender or receiver the synchronization pair is not ready, the other automaton receiving or sending the message will be blocked. A broadcast channel is defined between one sender and an arbitrary number of receivers. The sender will not be blocked no matter how many receivers are ready. The system of A1 and A2 has a binary channel called "chan". When A1 is at location L2, A2 is at

location L4, and *cl* is greater than 1, A1 can take the transition to L3 and send the message to A2 via *chan*. Meanwhile, A2 receives the message via *chan*, and moves to L5. the An assignment resets the clock and discrete variable when a transition is taken. In UPPAAL, both guards and assignments can be user-defined functions. In our example, when A1 moves from L2 to L3, the value of *a* is incremented.

A location can be **urgent** or **committed**. When an automaton reaches an urgent location, marked as "U", it must take the next transition without any delay in time. In our example, when A2 arrives at L5, it must take the transition to L4 before the time progresses. If at this time the value of discrete variable *a* happen to be greater than or equal to 5, the transition cannot be taken, which leads to a deadlock. When A2 reaches the urgent location L5, other concurrent automata (A1 in this example) may still take transitions before A2 moves to L4, as long as the time does not evolve. When an automaton reaches a committed location, marked as "C", it must also take the next transition without delay in time, and no other transitions in other automata can be taken in between.

The UPPAAL model checker defines a subset of Timed Computation Tree Logic (TCTL) [3] as a specification language of properties, and supports verification of liveliness and safety properties [19]. For example, one can specify the safety property "A1 will never reach location L3" as "$A[] \quad not \quad A1.L3$". If a property is not satisfied, a counterexample will be provided by the model checker. Interested readers can refer to paper [19] for more information about UPPAAL.

4 Modeling Checking Transaction Timeliness

In this section we describe our approach to model a real-time concurrent transaction system that is decomposed into a set of transactions and the Concurrency Control Manager (CCManager). The entire system is modeled as a network of timed automata, in which each transaction and the CCManager are modeled as timed automata respectively. Formally, a real-time concurrent transaction system is defined as follows.

Definition 1. *A real-time concurrent transaction system N_S is defined by the following parallel composition:*

$$N_S := A_0 \parallel A_1 \parallel ... \parallel A_{n-1} \parallel A_{CCManager}$$

where A_{n-1} is the timed automaton of transaction T_{n-1}, and $A_{CCManager}$ is the timed automaton of the CCManager.

Each transaction T_i is assigned a relative deadline. The key requirement concerning real-time properties to be satisfied by the modeled system is to meet the deadlines of all transactions.

Our modeling approach comprises a set of timed automaton skeletons for modeling the basic structures A_i and $A_{CCManager}$, and a set of parametrized patterns for modeling the operations within the transactions. The skeletons are supposed to be adjusted and enriched with respect to the particular system design, such as the selected PCC algorithm. The patterns, on the contrary, can be instantiated and reused with little change, to enrich the skeleton as basic modeling units. The proposed skeletons and patterns are presented in the following subsections.

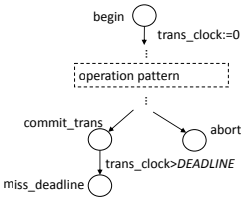


Fig. 2. Timed automaton skeleton for a basic transaction

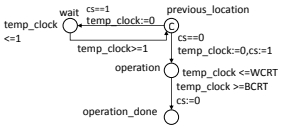


Fig. 3. Operation pattern in the transaction skeleton

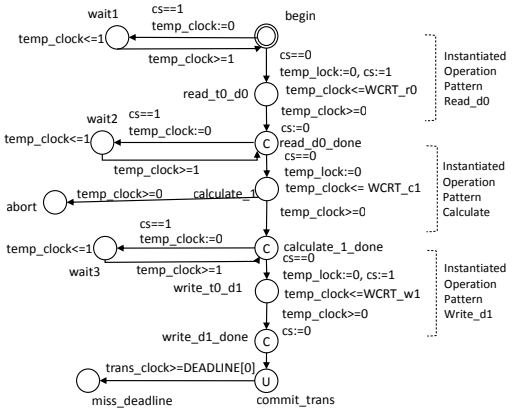


Fig. 4. Timed automaton for T_0 without CC

4.1 Basic Transaction Skeleton

The transaction management primitives (*Begin*, *Commit* and *Abort*), and the operations (*Read*, *Write* and *Calculate*), form the basis of our automaton skeleton for basic transactions. As shown in Fig. 2, in our transaction skeleton, the locations *begin*, *commit_trans* (commit is a reserved word in the UPPAAL tool) and *abort* represent the respective primitives. A transaction may contain multiple operations, each of which is modeled as an instantiation of the "operation pattern" in the automaton. Since we intend to verify the timeliness of the transaction, a clock variable *trans_clock* is defined for the automaton, and reset to zero once the transaction begins. When the transaction commits, the value of *trans_clock* is compared with the transaction's *DEADLINE*, which is a value specified by the designer. If *trans_clock* is greater than *DEADLINE*, a transition to the location *miss_deadline* will be taken.

The atomic read/write operation pattern is defined in Fig. 3. Connected to a previous location, this pattern contains locations *operation*, and *operation_done*, in which the word "operation" must be substituted by the actual operation, for instance "read_t1_d1" (T_1 reading D_1). The variable *cs* stands for the critical section that models the CPU resource and ensures the atomic behavior. A *wait* location models the behavior of waiting for the CPU. When *cs* equals 0, the automaton obtains the CPU, sets *cs* to 1, and performs the operation. Before reaching location *operation_done*, which is the end of the operation, *cs* is set back to 0. A clock variable *temp_clock* is defined to trace the time spent on the operation. *WCRT* and *BCRT* are parameters specified by the designer, representing the worst-case and best-case response time, respectively. The invariant *temp_clock* <= *WCRT* on location *operation* constrains that the execution of the operation takes at most WCRT time units. The guard *temp_clock* >= *BCRT* constrains that the execution takes at least *BCRT* time units. For non-atomic operations (calculations), the pattern is similar to the one for atomic operations, but without changing the value of *cs*.

For atomic read and write operations, the response times are equal to their execution times. For non-atomic operations, the response times could be derived by the

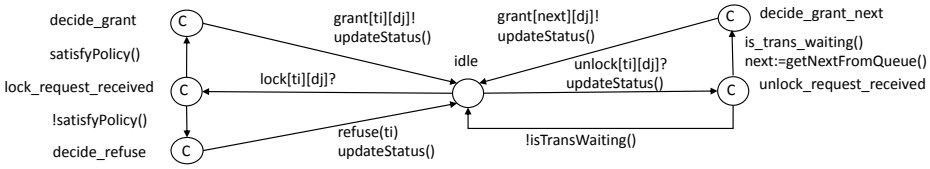


Fig. 5. Automaton skeleton for a pessimistic concurrency control manager

designer from the particular scheduling policy and the timing constraints, using for instance schedulability analysis techniques.

The modeling of a transaction is the initialization of the operation patterns, and the composition of the initialized patterns with the skeleton. For example, the automaton of transaction T_0 is shown in Fig. 4. The initialized operation patterns include **Read_d0**, **Calculate**, and **Write_d1**. These initialized patterns, as we will show in the remaining of the paper, can be reused to model a system with different CC mechanisms.

4.2 Concurrency Control Skeletons and Patterns

Pessimistic concurrency control employs locking mechanisms that require interaction between the CCManager and the transactions. We introduce skeletons for the CCManager, as well as patterns for the transaction skeleton that model such interactions.

The automaton skeleton for a PCC manager is shown in Fig. 5. When the automaton receives a locking request via the channel *lock[ti][dj]*, it takes the transition from the initial location *idle* to *lock_request_received*. A user-defined function that implements the resolution algorithm, *satisfy_policy()* is defined as a guard on the transitions from *lock_request_received*. If *satisfy_policy()* returns true, the automaton moves to *decide_grant*, and then immediately sends the signal *grant[ti][dj]!* to transaction T_i. During the transition, the automaton may need to update the status of the transactions and the locks, using a user-defined function *update_status()*.

When the PCC manager receives an unlocking request via *unlock[ti][dj]*, it updates the status of the data and the transaction, and moves to *unlock_request_received*. The guards on the transitions from this location check if any transaction is waiting for locking the data by a user-defined function *is_trans_waiting()*. If this function returns true, the automaton sends a signal via *grant[next][dj]!* to the next transaction, and updates the status accordingly.

The transaction skeleton needs to be extended to model the interaction with the PCC manager. A locking pattern and an unlocking pattern are introduced in Fig. 6 and Fig. 7 respectively. After the transaction sends a message via *lock[ti][dj]*, it waits at location *wait_for_lock_j*, until it receives the message *grant[ti][dj]*. They can be inserted into the basic transaction automata at particular positions depending on the selected PCC algorithm. For example, using rigorous 2PL write locks are released after the transaction commits or aborts. To model this PCC algorithm, the unlocking patterns for write locks must be inserted after the *commit* or *abort* location in the transaction automaton.

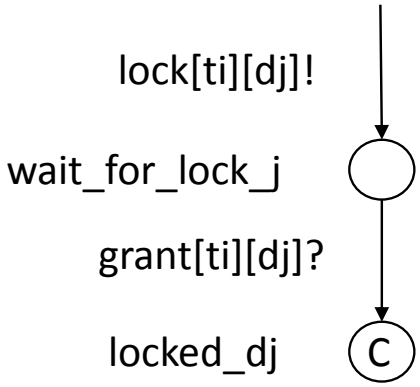


Fig. 6. Locking pattern for the transaction automaton

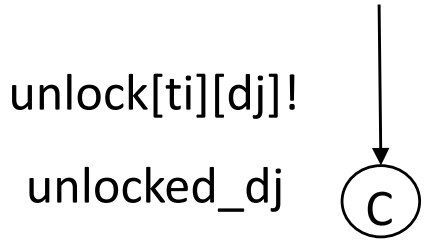


Fig. 7. Unlocking pattern for the transaction automaton

4.3 An Example: Model Checking Timeliness under Rigorous Two Phase Locking

We illustrate our modeling approach by modeling a concurrent transaction system implementing the rigorous 2PL algorithm presented in Section 3. Let us consider the two transactions in Program 1.1 and 1.2. T_0 contains operations r_0^0 and w_0^1, whereas T_1 contains operations w_1^0 and w_1^1. Assume the time spent on each individual read and write operation in the worst case is 1 time unit, and the time for the calculation in the worst case is 2 time units. The best case execution times are assumed to be 0 for simplicity. The deadlines of T_0 and T_1 are 8 and 5 time units respectively.

The model of T_0 is shown in Fig. 8. The basic transaction skeleton without CC contributes to the locations *begin*, *commit_trans*, *abort* and *miss_deadline*. The skeleton is enriched by the operations contained in the transaction, as well as locking and unlocking interactions with the CCManager. As illustrated in Fig. 8, the transaction first tries to acquire a readlock of D_0, which is modeled by an instantiated locking pattern **Readlock_d0**. The operation r_0^0 then follows, modeled by an initiated operation pattern **Read_d0**. After r_0^0, the transaction performs calculation, acquires a writelock of D_0, and performs the operation w_0^1. Before committing, the transaction releases it locks using instantiated unlocking patterns. The initialized operation patterns **Read_d0**, **Calculate** and **Write_d1** are reused from the model of T_0 without CC in Fig. 4. The timed automaton of T_1 is modeled in a similar way.

Figure 9 illustrates the timed automaton for the CCManager using the selected rigorous 2PL algorithm. The PCC skeleton is extended to incorporate two types of locks, the readlock and the writelock. In this model, the user-defined functions implement the actual algorithm. Function *satisfyPolicy()* decides whether a transaction can be granted the required lock, based on the current status of the locked data. Functions *updateGranted()*, *updateUnranted()* and *updateUnlock()* update the status of transactions and data after a locking, refusing or unlocking action is taken, respectively. Function *isTranswaiting()* checks if any transaction is waiting in the queue for locking a previously locked data. Function *getNextFromQueue()* fetches the next transaction in the waiting queue. The implementation of these functions is listed in Program 1.3.

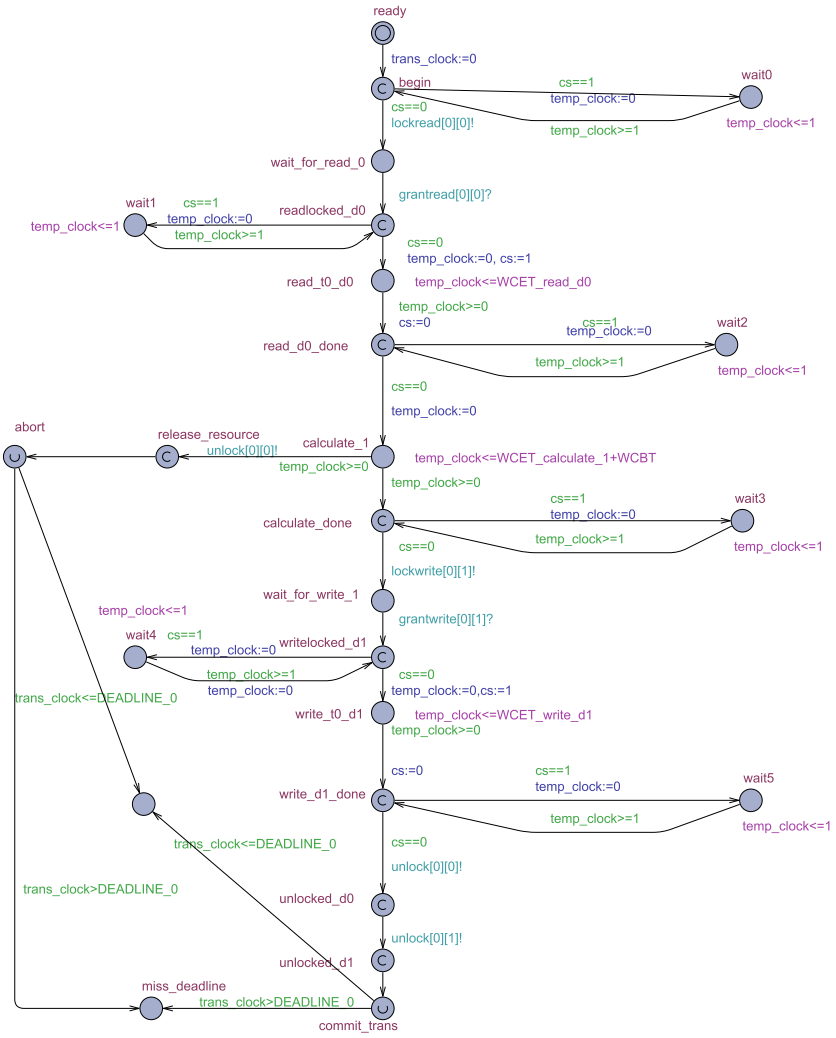


Fig. 8. The timed automaton for transaction T_0 using rigorous 2PL

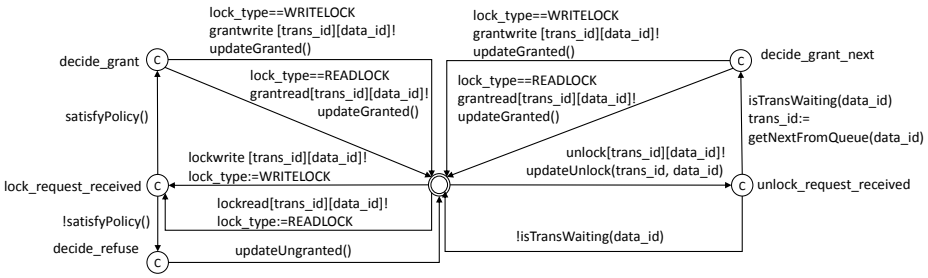


Fig. 9. The timed automaton for rigorous 2PL CCManager

Program 1.3. User-defined functions for rigorous 2PL in the CCManager model

```
void enqueue(int t, int d, int type) {
QueuedTrans qt = {t, d, type};
queue[d][len[d]]=qt;
len[d]++;}

void dequeue(int d) {
if(len[d]>0) {
int i;
for(i=0;i<len[d];i++) {
queue[d][i]=queue[d][i+1];
}
queue[d][len[d]]=null_trans;
len[d]--;}}

bool satisfyPolicy() {
//if data is not locked
if(!locked[data_id]) return true;
else if(locked[data_id]==READLOCKED){  //if data is readlocked
//Readlocks can be granted
if(lock_type==READLOCK) return true;
else if(lock_type==WRITELOCK) {
//write lock can be granted to the same transaction
if(no_of_read[data_id]==1 && readlockdata[trans_id][data_id]==1)
     return true;
else  return false;}}
//if data_id is writelocked, no locks can be granted
return false;}

void updateGranted()    {
int d = data_id, t = trans_id;
if(lock_type==READLOCK) {
locked[d]=1;
readlockdata[t][d]=1;
no_of_read[d]++;
} else if(lock_type==WRITELOCK) {
locked[d]=2;
writelockdata[t][d]=1; }}
```

```
void updateUngranted()    {
enqueue(trans_id, data_id, lock_type); }

void updateUnlock(int t, int d) {
if(locked[d]==2) {//if data d is writelocked,
locked[d]=0;
writelockdata[t][d]=0;
readlockdata[t][d]=0;
} else if(locked[d]==1) {//if data d is readlocked,
no_of_read[d]--;
readlockdata[t][d]=0;
if(no_of_read[d]==0)
locked[d]=0;}}

bool isTransWaiting(int d) {
if(no_of_read[d]==0 &&len[d]>0)
return true;
return false;}

int getNextFromQueue(int d) {
QueuedTrans t=queue[d][0];
lock_type = t.locktype;
dequeue(d);
return t.t_id;}
```

Verification of Timeliness Having a network of timed automata for the modeled system, we are able to model check the timeliness of the transactions using the UPPAAL model checker. The timeliness property, requiring each transaction meeting its deadline, can be specified as a safety property that the *miss_deadline* locations are not reachable.

The specifications are listed in Table 1. S1 and S2 specify the timeliness of T_0 and T_1, respectively. The verification by the UPPAAL model checker proves that S1 is satisfied. However, the verification of S2 fails, indicating that T_1 may miss its deadline. The model checker provides a trace that leads to the deadline miss. From the trace we realize that T_1 was trying to lock D_1 before w_1^1, while D_1 was already locked by T_0 before r_1^1 until T_0 committed. Such long blocking time introduced by the concurrency control caused the breached timeliness of T_1.

Table 1. Specification of properties and verification results

ID	Specification	Verification Time	Explored States	Result
S1	$A[]\, not\, T0.miss\_deadline$	0.001s	896	Satisfied
S2	$A[]\, not\, T1.miss\_deadline$	0.001s	439	Not Satisfied

5 Model Checking Transaction Isolation

In the previous section, transaction T_1 misses its deadline due to the blocking time introduced by the rigorous 2PL concurrency control, which aims to achieve full isolation. Alternatively, one could choose a less restrictive CC mechanism that achieves a less degree of isolation, which can hopefully improve the timeliness. In order to achieve a controlled relaxation of isolation, it is important to understand the variations of isolation, and discover means to verify the selected isolation variation. In this section, we first recall an overview of the existing isolation levels. Based on the definitions of isolation levels, we introduce an observer model and its automaton skeleton that could be composed into the automata network of the concurrent transaction system proposed in the previous section.

5.1 An Overview of Isolation

Isolation refers to the property that the execution of one transaction is not interfered by other transactions executing concurrently [2]. Since full isolation leads to performance degrades and is not always necessary, the relaxation of isolation has been introduced by both industry and academia. Most commercial DBMSs support the *isolation levels* defined by ANSI/ISO SQL92 standard [16], which are SERIALIZABILITY (the most strict isolation), REPEATABLE READS, READ COMMITTED, and READ UNCOMMITTED (the most relaxed isolation).

Adya et al. [2] have generalized these isolation levels, and provided unambiguous definitions using the concept of *phenomena*. A phenomenon is a type of behavior that can lead to inconsistent data, and can be characterized by the *direct conflicts* of two committed transactions. The isolation levels are defined in terms of the phenomena that must be avoided at each level.

The direct conflicts of two committed transactions, without considering predicate-based operations, are defined as follows [2]:

Direct read-dependency Transaction T_j directly read-depends on T_i, if T_j reads data x after T_i writes x (before other transactions write x).

Direct write-dependency Transaction T_j directly write-depends on T_i, if T_j writes data x after T_i writes x (before other transactions write x).

Direct anti-dependency Transaction T_j directly anti-depends on T_i, if T_j writes data x after T_i reads x (before other transactions write x).

Based on the direct conflicts a Direct Serialization Graph (DSG) can be constructed. Each transaction is represented by a node in a DSG. An edge $T_i \xrightarrow{wr} T_j$ represents that T_j directly read-depends on T_i. The direct write-dependency and the direct anti-dependency are denoted as $T_i \xrightarrow{ww} T_j$ and $T_i \xrightarrow{rw} T_j$ respectively. For example, one possible transaction execution involving T_0 and T_1 can be denoted as follows: <r_0^0, w_1^0, w_1^1, w_0^1>. The DSG of this transaction execution is shown in Fig. 10.

Using this representation the phenomena are defined in Table 2 [2]. According to the definition, the DSG in Fig. 10 exhibits phenomenon G2 because it contains an anti-dependency $T_0 \xrightarrow{rw} T_1$ within a cycle.

Fig. 10. The DSG of a transaction execution history <r_0^0, w_1^0, w_1^1, w_0^1>

Table 2. Phenomena defined by Adya et al. [2]

Phenomenon	Definition
G0: Write Cycles	The DSG of the transaction execution contains a directed cycle consisting entirely of write-dependency edges.
G1a: Aborted Reads	The execution includes a committed transaction T_1 and an aborted transaction T_2, and T_1 reads the data modified by T_2.
G1b: Intermediate Reads	The execution includes a committed transaction T_1 that reads a modification of T_2, and this modification is not the final modification of T_2.
G1c: Circular Information Flow	The DSG of the execution contains a directed cycle including any dependency edges (but no anti-dependency edges in the cycle
G2: Anti-dependency Cycles	The DSG of the transaction execution contains a directed cycle consisting one or more anti-dependency edges.

An isolation level can then be defined as the property of avoiding a particular subset of the defined phenomena. For example, the SERIALIZABLE level precludes all the aforementioned phenomena, whereas READ COMMITTED only precludes G0 and G1. Therefore, the execution <r_0^0, w_1^0, w_1^1, w_0^1> represented by the DSG in Fig. 10 violates SERIALIZABLE isolation. To verify a particular level of isolation is satisfied, we must verify the absence of the phenomena precluded by this level, at the presence of the underlying concurrency control mechanism.

5.2 An Observer Model for Isolation

In order to verify the satisfaction of an isolation level one needs to verify the the absence of the corresponding phenomena. We introduce IsolationObserver automata to capture the phenomena. Assuming a real-time concurrent transaction system N_S intends to achieve a selected isolation level that precludes k phenomena, then we can define N_S as follows:

$$N_S := A_0 \parallel ... \parallel A_n \parallel A_{CCManager} \parallel O_0 \parallel ... \parallel O_{k-1}$$

where A_n is the timed automaton of transaction T_n, $A_{CCManager}$ is the timed automaton of the CCManager, and O_{k-1} is the timed automaton for observing a phenomenon G_k that is disallowed by the selected isolation level.

The automaton skeleton for an isolation observer is described in Fig. 11. The automaton starts from the *idle* location, and reaches the *phenomenon_Gn* location if all transitions are taken. Each location between *idle* and *phenomenon_Gn* is a subsequence of the operation sequence defining the phenomenon Gn. Without losing generality, let as define Gn as the sequence <op_i^j, op_m^n, ...>. In Fig. 11, when transaction T_i successfully completes operation op_i^j (read or write D_j), the observer automaton is notified via the channel *notify_operation1[ti][dj]*, and takes the transition to the location *operation1_i_j*. Subsequently, when T_m successfully completes op_m^n, the automaton takes the transition from *operation1_i_j* to *operation1_i_j_operation2_m_n*. Since the observed phenomenon is started by an operation of transaction T_i, the end of T_i also means the end of the observation. Therefore, when T_i commits or aborts, the observer

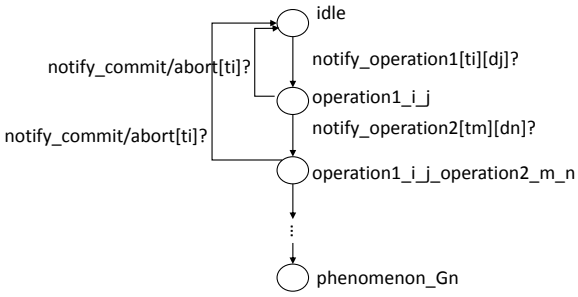


Fig. 11. Automaton pattern for an isolation observer

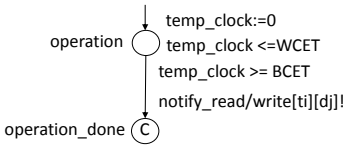
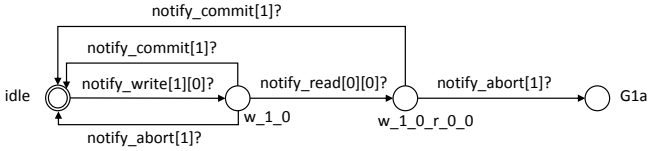
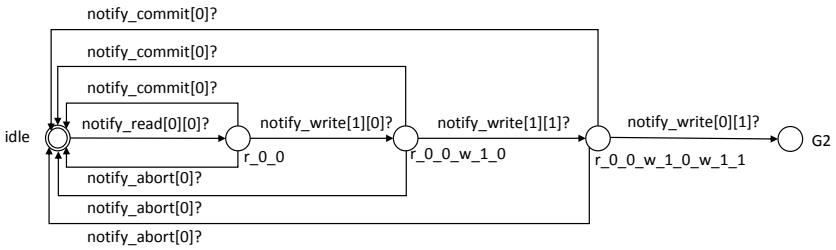


Fig. 12. Operation pattern extended for IsolationObservers in a basic transaction skeleton

automaton gets a notification via channel *notify_commit/abort[ti]*, and reset to the location *idle*. Such notification-transition behavior is repeated until the observer reaches *phenomenon_Gn*, indicating the existence of Gn and thus the violation of the desired isolation level.



(a) Observer for G1a



(b) Observer for G2

Fig. 13. IsolationObservers for G1a and G2 for the exemplary system

Accordingly, the transaction skeleton and patterns introduced in the previous section must be extended to incorporate the notifications. In the transaction skeleton in Fig. 2, *notify_commit[ti]!* and *notify_abort[ti]!* should be added to the transitions leading to *commit_trans* and *abort* respectively. The operation pattern defined in Fig. 3 needs to extend with *notify_operation[ti][dj]!* (*notify_read[ti][dj]* or *notify_write[ti][dj]*) on the transition to *operation_done*, as shown in Fig. 12.

Using the IsolationObserver skeleton and the patterns introduced above, we can verify that the example in Section 4 achieves full isolation, i.e., SERIALIZABLE isolation level. To prove this, one must prove that none of the phenomena G0, G1 (G1a,

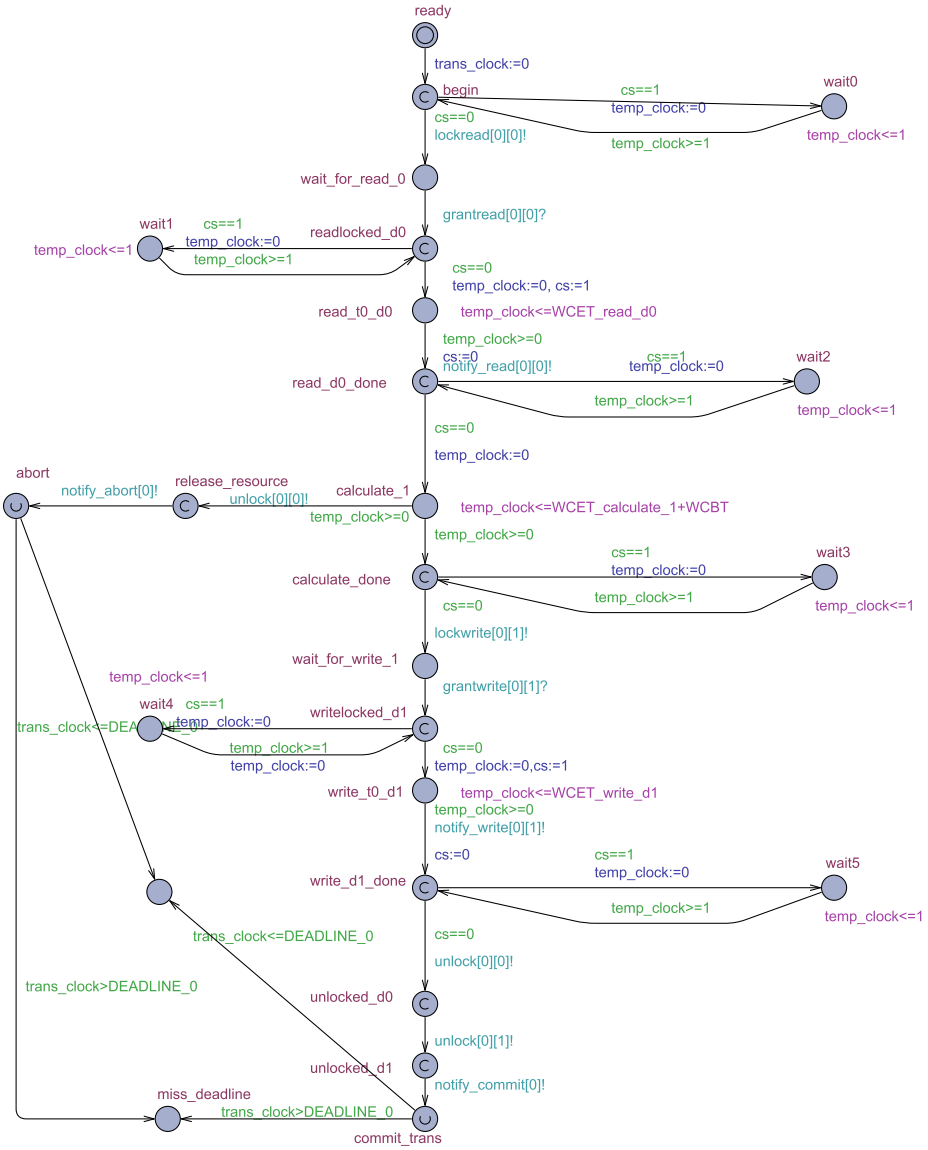


Fig. 14. UPPAAL model of T_0 using rigorous 2PL

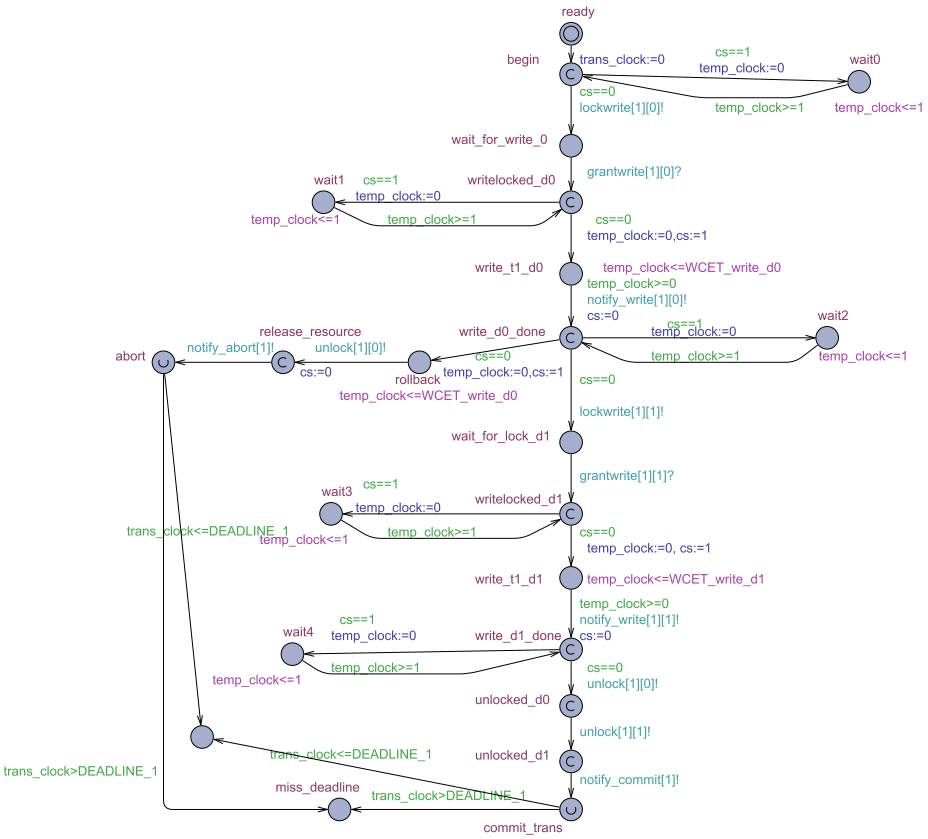


Fig. 15. UPPAAL model of T_1 using rigorous 2PL

Table 3. Verification results using rigorous 2PL

ID	Specification	Verification Time	Explored States	Result
S1	$A[]\ not\ T0.miss\_deadline$	0.001s	896	Satisfied
S2	$A[]\ not\ T1.miss\_deadline$	0.001s	896	Satisfied
S3	$A[]\ not$ $IsolationObserverG1a.G1a$	0.001s	896	Satisfied
S4	$A[]\ not$ $IsolationObserverG2.G2$	0.001s	439	Not satisfied

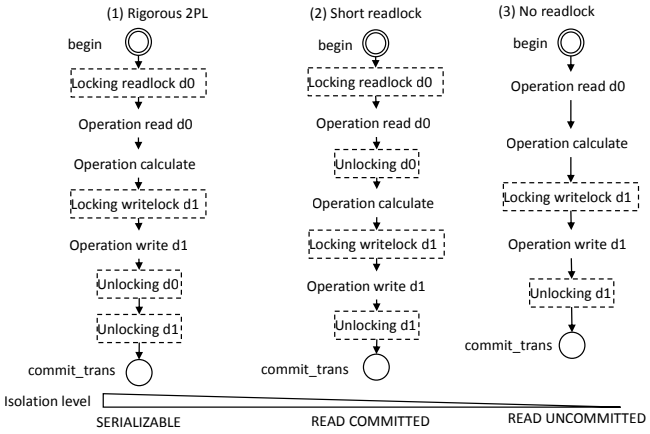


Fig. 16. Adjusting the composition positions of locking/unlocking patterns for different isolation

G1b and G1c) and G2 could occur. By definition, G0 is exhibited only if there exists a write-dependency loop between T_0 and T_1, which is not possible considering the operations of these two transactions. Similarly, G1b and G1c will not occur by definition. Therefore, in order to verify that the SERIALIZABLE isolation level is met, we only need to prove the absence of G1a and G2. Among them, G1a (Aborted Read) can be described as the sequence <w_1^0, r_0^0, a_1>, in which a_1 denotes the abort of T_1. G2 (Anti-dependency Cycles) can be described as the sequence <r_0^0, w_1^0, w_1^1, w_0^1>. The observer automata for G1a and G2 are shown in Fig. 13a and 13b. The detailed UPPAAL models of T_0 and T_1 are shown in Fig. 14 and 15. The CCManager is the same as in Fig. 9. The verification results, as listed in Table 3, show neither location *G1a* nor location *G2* is reachable, which means that the verified system achieves SERIALIZABLE isolation level.

6 Flexible Modeling of Concurrency Control for Relaxed Isolation

As demonstrated in Section 4 and Section 5, concurrency control aiming for isolation may lead to deadline miss of real-time transactions, and therefore a relaxed level of isolation could be desirable. In order to select an appropriate PCC so that both timeliness and a particular isolation level, the designer must model different PCC mechanisms together with the transactions. Our approach provides flexibility in modeling different PCC mechanisms, leading to reduced modeling efforts. The flexibility, on the one hand, lies in the easy customization of isolation level to be verified. One only needs to decide the unwanted phenomena, create the observers for these phenomena, and compose these observers into the automata network. As we have shown in Section 5.2, to verify SERIALIZABILITY, we can plug the autamata in Fig. 13a and 13b into the modeled transaction system. On the other hand, as further explained in this section, our modeling approach enables easy adjustment of existing models for a different PCC mechanism aiming for relaxed isolation levels, without major changes in the existing models.

One type of adjustments for relaxing isolation is to adjust the times and duration of locking and unlocking. By adjusting the times and durations of the locks, one can

develop different pessimistic concurrency control algorithms that achieve different isolation levels [2, 14]. For example, SERIALIZABLE can be achieved by exploiting long readlocks and long writelocks, as in rigorous 2PL. These locks are released when the transaction is committed. If the readlocks are changed to have short duration, which means the readlocks are released immediately after the read operation, a lower level of isolation such as READ COMMITTED could be achieved. The READ COMMITTED level can be further relaxed to READ UNCOMMITTED by, for instance, removing the requirement of readlocks entirely. This type of adjustment is easy to achieve in our model. Since locking and unlocking are modeled as parametrized patterns composed into the transaction skeleton, one can move them to the desired locations to achieve different durations. The adjustments of locking types and durations are illustrated in Fig. 16. The dashed rectangles represent the initiated locking and unlocking patterns in the automaton of T_0. The adjustments for different PCC algorithms can easily be accomplished by adding, removing, or moving around the locking and unlocking patterns.

We now exploit the short readlock algorithm for the concurrency control of the transaction system including T_0 and T_1. The transaction automata are adjusted according to the adjustment for short readlocks in Fig. 16. The detailed UPPAAL timed automata models of T_0 and T_1 are shown in Fig. 17 and 18 respectively. The CCManager atomaton is exactly the same as the one in Fig. 9. The IsolationObservers are exactly the same as the ones in Fig. 13a and 13b. The verification result using UPPAAL model checker is listed in Table 4. As shown by the table, both transactions can meet their deadlines. S3 is satisfied, indicating phenomenon G1a could not occur, which means the modeled system reaches READ COMMITED isolation level. However, S4, which checks the absence of G2, is not satisfied, which means the system does not meet the SERIALIZABLE isolation level.

Table 4. Verification results using the short readlock concurrency control algorithm

ID	Specification	Verification Time	Explored States	Result
S1	$A[]\ not\ T0.miss\_deadline$	0.001s	1179	Satisfied
S2	$A[]\ not\ T1.miss\_deadline$	0.001s	1179	Satisfied
S3	$A[]\ not\ IsolationObserverG1a.G1a$	0.001s	1179	Satisfied
S4	$A[]\ not\ IsolationObserverG2.G2$	0.001s	966	Not Satisfied

Another type of adjustment lies in the conflict detection and resolution policy of the PCC mechanism. This type of adjustment can be easily implemented in our model, by modifying the user-defined functions in the model. For a PCC algorithm differing from a two phase locking algorithm, the modification mainly appears in the lock granting mechanism and the queuing mechanism, that is, the *satisfyPolicy()* and *getNextFromQueue()* functions in the skeleton for CCManager in Fig. 5. For example, 2PL-HP [1] allows transactions with higher priorities to lock the data blocked by transactions with lower priorities, and aborts the lower priority transactions. To model 2PL-HP, one only needs to adjust the model by modifying the functions *satisfyPolicy()* and *getNextFromQueue()*, plus other minor extensions or adjustments in the model.

We apply our approach to modeling a set of transactions under 2PL-HP [1], a widely applied CC algorithm in real-time database systems. 2PL-HP allows transactions with

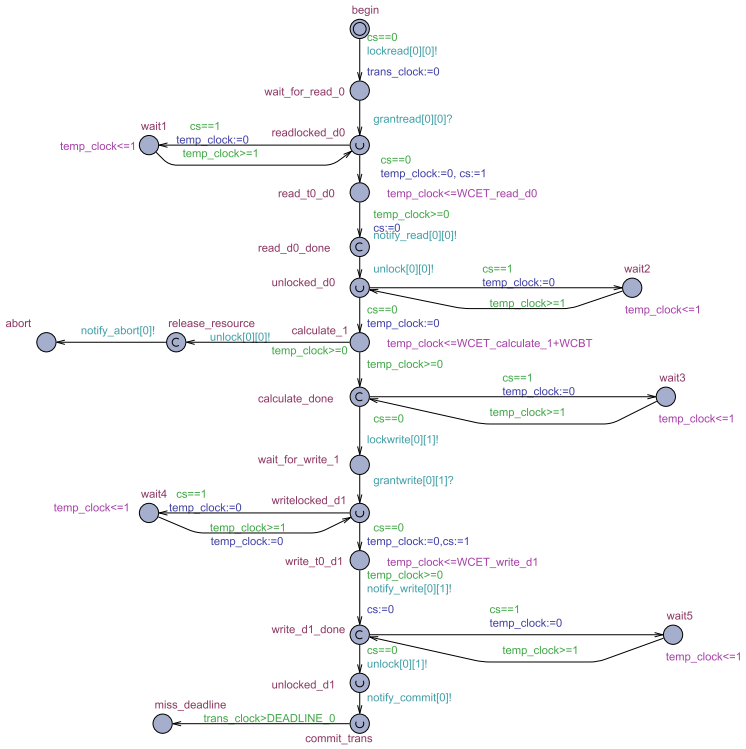


Fig. 17. UPPAAL model of T_0 using short readlocks

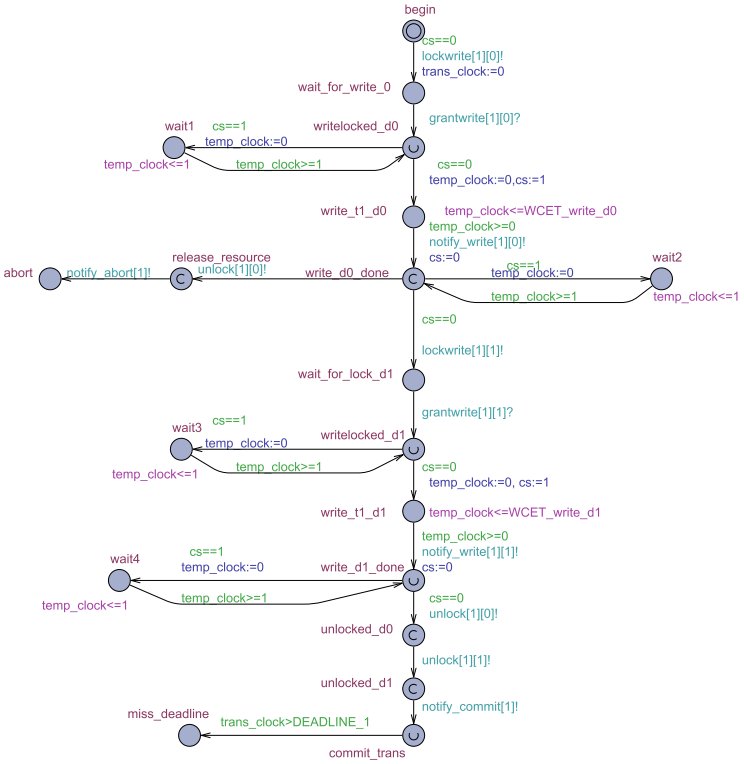


Fig. 18. UPPAAL model of T_1 using short readlocks

higher priorities to lock the data that are already locked by transactions with lower priorities. The lockers with lower priorities are aborted by the CCManager. The aborted transactions are scheduled to be restarted, according to a predefined criterion. In our case we assume that transactions are restarted if they have not missed their deadlines.

Program 1.4. Transaction T_2

```
Begin
read D0
read D1
write D2
Commit
```

Program 1.5. Transaction T_3

```
Begin
read D2
write D2
Commit
```

The transaction set consists of T_0 and T_1 listed in Programs 1.1 and 1.2 in Section 3, as well as T_2 and T_3 in Programs 1.4 and 1.5. D_0 and D_1 are shared by T_0, T_1 and T_2, while D_2 is shared by T_2 and T_3. The deadlines for T_0, T_1, T_2 and T_3 are 11, 4, 22 and 13 time units respectively. The priorities are assigned, from highest to lowest, as follows: T_1, T_0, T_3, T_2.

The UPPAAL models of the transactions using 2PL-HP and binary locks are shown in Fig. 19, 20, 21 and 21, respectively. In these models, the initialized operation patterns for modeling the read, calculate and write operations are the same as the ones in the models of rigorous 2PL (Fig. 8). Unlike rigorous 2PL using two types of locks, 2PL-HP with binary locks do not distinguish read and write locks. Figure **??** presents the model of the CCManager. A *sch()* function, as listed in Program 1.6, models the scheduling policy. The major changes, compared to the CCManager of rigorous 2PL, lie in the *satisfyPolicy()* function. The satisfyPolicy() of 2PL-HP, as listed in Program 1.7, aborts the lock holder and grants the lock to the requester, if the requester has a higher priority than the lock holder. The IsolationObservers are the same as the ones in Fig. 13a and 13b.

Program 1.6. sch() functions for 2PL-HP in the CCManager model

```
int sch() {
int next=0, prio=10;
int i;
if(cs!=-1) {return cs;}
for(i=0;i<TRNO;i++) {
if(rq[i]==1 && wq[i]==0) {//if transaction i is ready and not
    blocked
if(priorities[i]<prio) {//if i has a higher priority, i should
    be the next
next = i;
prio = priorities[i];}}}
return next;}
```

Program 1.7. satisfyPolicy() functions for 2PL-HP in the CCManager model

```
bool satisfyPolicy() {
int i;
if(!locked[data_id])
return true;
```

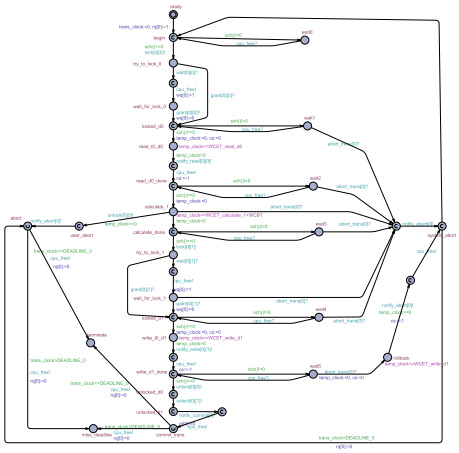


Fig. 19. UPPAAL model of T_0 using 2PL-HP

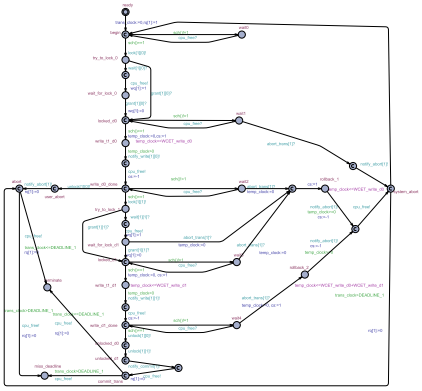


Fig. 20. UPPAAL model of T_1 using 2PL-HP

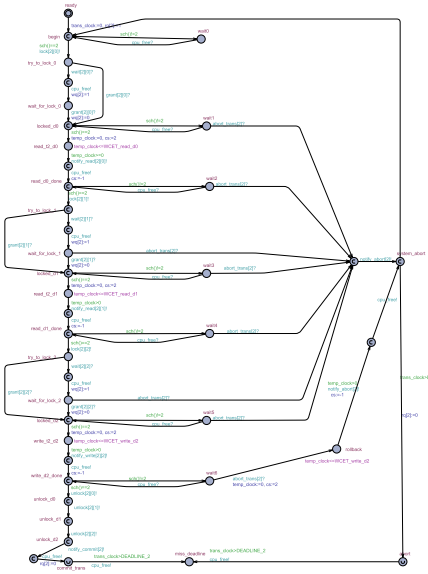


Fig. 21. UPPAAL model of T_2 using 2PL-HP

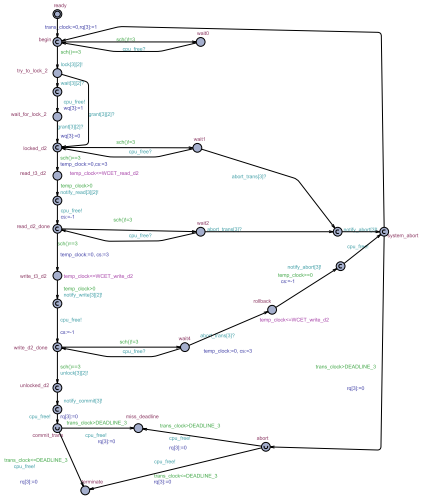


Fig. 22. UPPAAL model of T_3 using 2PL-HP

Table 5. Possible phenomena during execution

Phenomenon	Operation sequences
G1a	$<w_1^0, r_0^0, a_1><w_0^1, r_2^0, a_0><w_1^0, r_2^0, a_1><w_1^1, r_2^1, a_1><w_2^2, r_3^2, a_2>$
G2	$<r_0^0, w_1^0, w_1^1, w_0^1><w_1^0, r_2^0, r_2^1, w_1^1><r_2^0, w_1^0, w_1^1, r_2^1><r_3^2, w_2^2, w_3^2>$

```
for(i=0;i<TRNO;i++) {
if(lockdata[i][data_id]==1) {
if(priorities[trans_id]<priorities[i])
return true;}}
return false;}
```

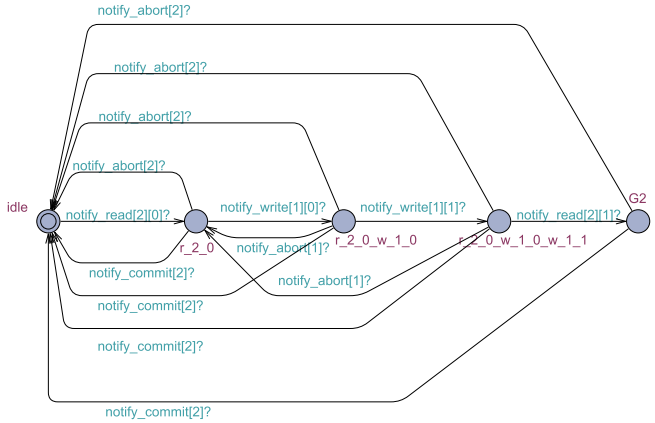


Fig. 23. IsolationObserver for $<r_2^0, w_1^0, w_1^1, r_2^1>$

The phenomena possible to occur during the execution of the example transactions are listed in Table 5. One observer automaton is constructed for each phenomenon, in a similar way as in Section 5.2. One example is the observer for $<r_2^0, w_1^0, w_1^1, r_2^1>$ in Fig.6. To verify SERIALIZABLE isolation, one needs to verify that none of the listed phenomena could actually occur. The verification results are listed in Table 6. Specification S1, S2, S3 and S4 encode timeliness of T_0, T_1, T_2, and T_3, respectively. S5 specifies that none of the locations indicating a phenomenon is reachable. All listed specifications are satisfied, which means that both timeliness and SERIALIZABLE isolation are guaranteed by 2PL-HP.

Validation of the model checking results. The authors of 2PL-HP have proved that the algorithm guarantees serializability [1], which validates the verification results regarding isolation.

We compare the model checking results of timeliness with the results of schedulability analysis. We argue that existing schedulability analysis techniques cannot be directly applied to analyze the schedulability of the transaction set. For instance, analysis of tasks in the Abort-and-Restart (AR) model assumes that higher priority tasks immediately abort lower priority tasks that are later restarted. The transaction model in

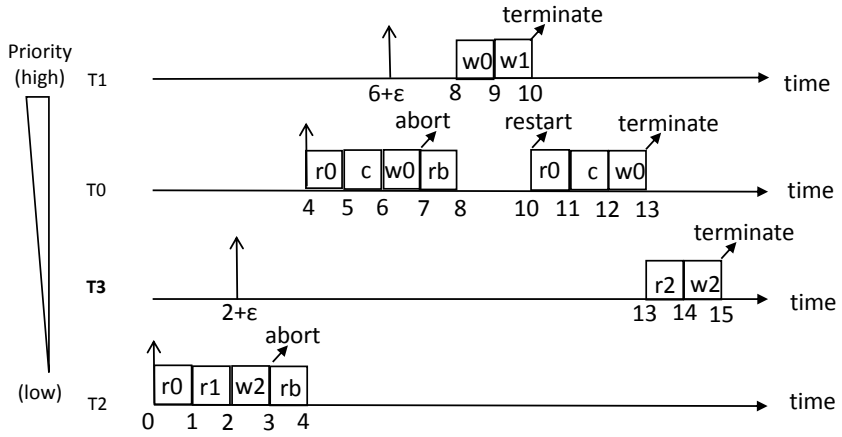


Fig. 24. Worst case for T_3

2PL-HP is more complex. A transaction may be blocked by a lower priority transaction because of the atomic operations and rollback. A transaction may be aborted (and then restarted) by a higher priority transaction if they share the same data, or be preempted if they do not share data.

Since the considered transaction set consists of only four transactions, for the purpose of validating that the model-checked transactions are indeed schedulable, we analyze the worst case for each transaction manually, assuming each transaction is modeled as a real-time task. As an example, we show the worst case for transaction T_3 in Fig 24. In this case, a lower priority transaction T_2 has read D_0 and D_1 (denoted as "r0" and "r1"), and started an atomic write operation on D_2 ("w2") at time 2. T_3 is activated at time $2 + \epsilon$, and tries to read D_2. T_2 is aborted due to conflicts, but before T_3 starts, T_2 must finish the atomic operation, and perform the rollback ("rb"). When the rollback is completed at time 4, T_0 is activated, which has a higher priority than T_3, and thus preempts T_3. However, before T_0 could complete its work, it gets aborted by T_1 at time 7, gets restarted at time 10, and terminates at time 13. T_3 is then allowed to execute, and terminates at time 15. The worst case response time of T_3 is therefore $13 - \epsilon$, smaller than its deadline 13. T_3 is indeed schedulable. Similar analysis shows that T_0, T_1 and T_2 can all meet their deadlines, whose worst case response times are 11, 4 and 22, respectively. Therefore, the model checking results with respect to timeliness are validated. For a larger transaction set under 2PL-HP, the validation via schedulability analysis should be automated, which is not trivial and out of our current scope.

Compared with schedulability analysis, our approach can perform more exact analysis for more complex transaction models. For instance, a variant of 2PL-HP conditionally aborts transactions based on their current time [1]. While existing schedulability analysis techniques can be applied but with large pessimism, our approach can easily model the conditional aborting behavior by extending the current models, and perform more exact analysis.

Table 6. Verification results using 2PL-HP

ID	Specification	Verification Time	Explored States	Result
S1	$A[]\ not\ T0.miss\_deadline$	1.592s	161126	Satisfied
S2	$A[]\ not\ T1.miss\_deadline$	1.606s	161126	Satisfied
S3	$A[]\ not\ T2.miss\_deadline$	1.623s	161126	Satisfied
S4	$A[]\ not\ T3.miss\_deadline$	1.638s	161126	Satisfied
S5	$A[]\ not\ (IsolationObserverT0T1G1a.G1a$ $or\ IsolationObserverT0T1G2.G2\ or\ IsolationObserverT0T2G1a.G1a$ $or\ IsolationObserverT1T2G1a\_0.G1a\ or\ IsolationObserverT1T2G1a\_1.G1a$ $or\ IsolationObserverT1T2G2\_0.G2\ or\ IsolationObserverT1T2G2\_1.G2$ $or\ IsolationObserverT2T3G1a.G1a\ or\ IsolationObserverT2T3G2.G2)$	1.669s	161126	Satisfied

7 Conclusion

In this paper we have proposed an approach for modeling concurrent transaction systems, with an aim of verifying transaction timeliness and isolation in a unified framework. UPPAAL timed automata are used to model the system, including transactions, the CC manager, and observers for isolation. Our approach is based on timed automata skeletons and patterns, which could reduce the modeling effort when a different CC algorithm is selected.

One concern of the proposed modeling approach is the possibility of state explosion. Since isolation is a property for concurrent transactions sharing the same data, one promising way to mitigate state explosion would be partitioning the transactions according to the data dependency. Another approach is to apply bounded model-checking, such as statistical model checking implemented in UPPAAL-SMC [9]. Although the verification result is not a guarantee using statistical model checking, it will provide valuable insights into the design, and suffices many soft real-time applications.

Besides the state explosion issue, we are going to extend our models to support predicate-based operations. We also plan to build up a framework for trading off more transactional properties, for instance atomicity and durability, that facilitates the automation of customizing an RTDBMS.

Acknowledgment

This work is funded by the Knowledge Foundation of Sweden (KK-stiftelsen) within the DAGGERS project.

References

1. Abbott, R., Garcia-Molina, H.: Scheduling real-time transactions. SIGMOD Rec. 17(1), 71–81 (Mar 1988)
2. Adya, A., Liskov, B., O'Neil, P.: Generalized isolation level definitions. In: Data Engineering, 2000. Proceedings. 16th International Conference on. pp. 67–78 (2000)
3. Alur, R., Courcoubetis, C., Dill, D.: Model-checking in dense real-time. Information and Computation 104(1), 2 – 34 (1993)
4. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical computer science 126(2), 183–235 (1994)

5. Cai, S., Gallina, B., Nyström, D., Seceleanu, C.: Trading-off data consistency for timeliness in real-time database systems. In: Proceedings of the 27th Euromicro Conference on Real-Time Systems. pp. 13–16 (2015)
6. Chkliaev, D., Hooman, J., van der Stok, P.: Formal modeling and analysis of atomic commitment protocols. In: Parallel and Distributed Systems, 2000. Proceedings. Seventh International Conference on. pp. 151–158 (2000)
7. Chrysanthis, P.K., Ramamritham, K.: Synthesis of extended transaction models using acta. ACM Trans. Database Syst. 19(3), 450–491 (Sep 1994)
8. Datta, A., Son, S.: A study of concurrency control in real-time, active database systems. Knowledge and Data Engineering, IEEE Transactions on 14(3), 465–484 (May 2002)
9. David, A., Larsen, K., Legay, A., Mikučionis, M., Poulsen, D., van Vliet, J., Wang, Z.: Statistical model checking for networks of priced timed automata. In: Formal Modeling and Analysis of Timed Systems, Lecture Notes in Computer Science, vol. 6919, pp. 80–96. Springer Berlin Heidelberg (2011)
10. Elmasri, R.A., Navathe, S.B.: Fundamentals of Database Systems. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 4th edn. (2004)
11. Gallina, B.: PRISMA: a software product line-oriented process for the requirements engineering of flexible transaction models. Ph.D. thesis, University of Luxembourg (2010)
12. Gallina, B., Guelfi, N., Kelsen, P.: Towards an alloy formal model for flexible advanced transactional model development. In: Software Engineering Workshop (SEW), 2009 33rd Annual IEEE. pp. 94–103 (Oct 2009)
13. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edn. (1992)
14. Gray, J.N., Lorie, R.A., Putzolu, G.R., Traiger, I.L.: Readings in database systems (3rd ed.). chap. Granularity of Locks and Degrees of Consistency in a Shared Data Base, pp. 175–193. Morgan Kaufmann Publishers Inc. (1998)
15. Han, S., Lam, K.Y., Wang, J., Ramamritham, K., Mok, A.: On co-scheduling of update and control transactions in real-time sensing and control systems: Algorithms, analysis, and performance. Knowledge and Data Engineering, IEEE Transactions on 25(10), 2325–2342 (Oct 2013)
16. ISO/IEC 9075:1992 Database Language SQL. Standard, International Organization for Standardization (1992)
17. Kot, M.: Modeling selected real-time database concurrency control protocols in uppaal. Innovations in Systems and Software Engineering 5(2), 129–138 (2009)
18. Lanotte, R., Maggiolo-Schettini, A., Milazzo, P., Troina, A.: Modeling longâĂŞrunning transactions with communicating hierarchical timed automata. In: Formal Methods for Open Object-Based Distributed Systems, Lecture Notes in Computer Science, vol. 4037, pp. 108–122. Springer Berlin Heidelberg (2006)
19. Larsen, K.G., Pettersson, P., Wang, Y.: Uppaal in a nutshell. International Journal on Software Tools for Technology Transfer (STTT) 1(1), 134–152 (1997)
20. Makni, A., Bouaziz, R., Gargouri, F.: Formal verification of an optimistic concurrency control algorithm using spin. In: Temporal Representation and Reasoning, 2006. TIME 2006. Thirteenth International Symposium on. pp. 160–167 (June 2006)
21. Song, X., Liu, J.: Maintaining temporal consistency: pessimistic vs. optimistic concurrency control. Knowledge and Data Engineering, IEEE Transactions on 7(5), 786–796 (Oct 1995)
22. Stankovic, J.A., Son, S.H., Hansson, J.: Misconceptions about real-time databases. Computer 32(6), 29–36 (1999)
23. Xiong, M., Ramamritham, K.: Specification and analysis of transactions in real-time active databases. In: Real-Time Database and Information Systems: Research Advances, vol. 420, pp. 327–351. Springer US (1997)