Specification and Formal Verification of Atomic Concurrent Real-Time Transactions

Simin Cai*, Barbara Gallina†, Dag Nyström‡, and Cristina Seceleanu§
Mälardalen Real-Time Research Centre
Mälardalen University, Västerås, Sweden
Email: *simin.cai@mdh.se, †barbara.gallina@mdh.se, ‡dag.nystrom@mdh.se, §cristina.seceleanu@mdh.se

***Abstract*—Although atomicity, isolation and temporal correctness are crucial to the dependability of many real-time database-centric systems, the selected assurance mechanism for one property may breach another. Trading off these properties requires to specify and analyze their dependencies, together with the selected supporting mechanisms (abort recovery, concurrency control, and scheduling), which is still insufficiently supported. In this paper, we propose a UML profile, called UTRAN, for specifying atomic concurrent real-time transactions, with explicit support for all three properties and their supporting mechanisms. We also propose a pattern-based modeling framework, called UPPCART, to formalize the transactions and the mechanisms specified in UTRAN, as UPPAAL timed automata. Various mechanisms can be modeled flexibly using our reusable patterns, after which the desired properties can be verified by the UPPAAL model checker. Our techniques facilitate systematic analysis of atomicity, isolation and temporal correctness trade-offs with guarantee, thus contributing to a dependable real-time database system.**

***Keywords*—Transaction, Atomicity, Isolation, Temporal Correctness, Unified Modeling Language, Model Checking**

I. Introduction

In many database-centric systems, critical data such as account balance and configuration parameters are stored in databases and managed by DataBase Management Systems (DBMS). To maintain the consistency of data, a DBMS organizes operations as transactions, and manages them with various Abort Recovery (AR) and Concurrency Control (CC) mechanisms [1]. Abort recovery restores the database to a consistent state when a transaction is aborted due to errors, and thus achieves atomicity. Rollback, which undoes all changes of an aborted transaction, is a common AR technique [2]. Concurrency control prevents inconsistency by regulating the concurrent access to data from different transactions. Locks are often applied to avoid arbitrary access of data, as a widely applied CC technique [1]. Together, AR and CC ensure that critical data are dependable for applications relying on them.

Many database-centric systems are also time-critical, such as industrial control systems [3] and automotive systems [4], whose configurations and states can be stored in databases. Reading an outdated sensor value, or fetching the calibration parameter too late, could result in catastrophic consequences such as loss of lives. In such real-time database systems, therefore, transactions must also be temporally correct, meaning that they must be scheduled to use fresh data, and have to meet specified deadlines [5]. The assurance of atomicity and isolation, however, may jeopardize the assurance of temporal correctness. CC may cause a transaction to be blocked for a long time. AR introduces extra workload when performing recovery. To make matters worse, some CC algorithms may directly abort transactions, while the recovery may again lock data and block other transactions further, which could lead to deadline misses. Therefore, trade offs may need to be considered during the design of Real-Time DBMS (RTDBMS) [6], with respect to deciding on “variants” [7] of atomicity and isolation, and the selection of the AR and CC mechanisms.

To achieve an appropriate trade off, it is helpful to specify all three properties, together with AR, CC, and scheduling, explicitly in a high-level language familiar to system designers. To ensure the correctness of the trade off, one should be able to analyze such specifications, and reason about whether the properties can be satisfied with the selected mechanisms. Although the specification and analysis of one or two of these properties have been targeted by the research community, existing techniques do not take all three into consideration. However, since atomicity, isolation and temporal correctness are closely inter-dependent, omitting one of them in the analysis may compromise the dependability of the whole RTDBMS.

The contribution of this paper is two-fold. First, we propose a UML (Unified Modeling Language) profile as an extension of the Activity Diagram [8], for the specification of transactions, with explicit support for atomicity, isolation and temporal correctness. We choose UML because it is a well-accepted modeling language for software systems, including real-time systems and database systems, hence it is popular with designers. Our proposed UML profile, called UTRAN, models a transaction as an activity, and includes modeling elements to express abort recovery mechanisms such as rollback and compensation, scheduling policies, as well as isolation levels and concurrency control. Time-related properties such as deadlines and periods, which are reused from the UML-MARTE (Modeling and Analysis of Real-Time Embedded systems) profile [9], can be annotated to transactions and operations.

Second, to facilitate the analysis of all three properties, we extend our UPPAAL-based analysis framework [10] that models only transactions with isolation and temporal correctness concerns in UPPAAL Timed Automata (TA) [11], to include atomicity also. The new framework, called UPPCART (UPPaal for Concurrent Atomic Real-time Transactions), models transactions with encoded timing information, as well as the selected AR, CC and scheduling mechanisms, and the inconsistency to be avoided, as a network of UPPAAL TA. To reduce the modeling effort, we propose a set of reusable basic modeling units, for describing various CC and AR mechanisms, as well as the transactions. Specifications in UTRAN can potentially be automatically transformed into UPPCART models. We also propose patterns for formalizing the atomicity, isolation and temporal correctness properties as UPPAAL specifications [11]. The formalized properties can then be verified rigorously by the model checker UPPAAL, which provides a guarantee of the correctness of the design.

The remainder of the paper is organized as follows. In Section II we present the preliminaries of the paper. In Section III and Section IV, we introduce our proposed UTRAN profile and UPPCART framework, respectively. We present an example to illustrate our approach in Section V. We discuss the related work in Section VI, after which we conclude the paper and outline future work in Section VII.

II. Background

In this section, we present the preliminaries of this paper, including the concepts of transactions, atomicity, isolation and temporal correctness (Section II-A), UML profiles (Section II-B), and UPPAAL TA (Section II-C).

A. Real-Time Transactions

In database systems, clients read and write data through a DBMS that guarantees data consistency via transaction management. A *transaction* is a partially-ordered set of logically-related operations, called a Work Unit (WU) that, as a whole, ensures the *ACID* properties [2]: *Atomicity* (a transaction either runs completely or makes no changes at all), *Consistency* (a transaction executing alone must ensure logical constraints), *Isolation* (concurrent transactions do not interfere each other), and *Durability* (committed changes are made permanent). The lifecycle of a transaction is managed by the following operations: begin (start a transaction), commit (terminate a transaction while making its changes permanent and visible), and abort (terminate a transaction and recover from its changes). Two types of aborts exist in a database system. System aborts are initiated by the DBMS due to system errors or data contentions. User aborts are issued by clients to stop the transaction deliberately according to the application semantics.

a) Atomicity: Under full atomicity, "commit" means the completion of "all" changes, and "abort" means that "nothing" is changed. In this paper, we are particularly interested in the recovery of transactions terminated by errors. Therefore, our semantics of "commit" remains "all", while "abort" could have various meanings depending on the variants of atomicity.

We refer to the "nothing" semantics of full atomicity as *failure atomicity*. Failure atomicity is achieved by *rollback*, a recovery mechanism that undoes all changes and returns to the states before the transaction starts when it gets aborted [2]. Since failure atomicity may be restricted in terms of performance and functionality, a number of variants of *relaxed atomicity* have been proposed, which allow changes to be partially undone, or recover inconsistency with extra operations [12]. The following abort recovery mechanisms that support relaxed atomicity are considered in this paper. *Immediate compensation* recovers inconsistency due to abort by immediately executing a sequence of operations, such as to update a record that represents the error state. *Deferred compensation*, in contrast, schedules an extra normal transaction to restore consistency. In both variants, the operations are designed flexibly, depending on the application semantics. An atomicity manager, which possesses the knowledge of the atomicity variants, performs the recovery at runtime.

b) Isolation: In literature, isolation has been quantified as various levels [13]. An *isolation level* is defined as the avoidance of a particular set of *phenomena*, which are interleaved transaction executions that can lead to inconsistent data. If we use r_i^j to denote that transaction T_i reads data D_j, w_i^j to denote that T_i writes D_j, the following execution is considered as a phenomenon: <r_0^0, w_1^0, w_1^1, r_0^1>, representing the execution "T_0 reads D_0, T_1 writes D_0, T_1 writes D_1, T_0 reads D_1". In this example, T_0 reads an old version of D_0 before the change of T_1, but a new version of D_1 after the change of T_1. If D_0 and D_1 are a pair of configuration parameters that should be compatible, the consequence of T_0 using these inconsistent parameters may result in unsafe system behaviors. An isolation level precludes a subset of such phenomena, thus avoiding the inconsistency. Isolation levels are also a flexible way to relax isolation, as the precluded phenomena are adjustable according to the particular semantics.

DBMS ensures isolation by associating a concurrency control manager to the managed data, which regulates the interleaved transaction executions according to a selected CC algorithm [1]. *Pessimistic Concurrency Control (PCC)*, a family of CC algorithms commonly applied in DBMS [1], is considered in this paper. PCC exploits locks to prevent unwanted interleavings. Depending on the algorithm, a transaction needs to hold a specific type of lock, before reading or writing the data. Locks are acquired at a certain time point before the operations, and are released at a certain time point afterwards. Upon receiving requests, the CC manager decides which transactions should obtain the lock, wait, or even be aborted, according to the selected algorithm. The atomicity manager, in case a transaction gets aborted by CC, performs the abort and recovery of the transaction.

c) Temporal Correctness: In a real-time database system, *temporal correctness* involves the transaction *timeliness*, and *temporal data consistency* [5]. Transactions need to meet their deadlines, which is referred to as timeliness [5]. Temporal data consistency includes two aspects. *Absolute validity* requires that data read by a transaction must not be older than a specified validity interval. *Relative validity* requires that, if a transaction reads a group of data, these data must be generated within a specified interval so that the results are temporally correct. Temporal correctness is directly influenced by the scheduling policy adopted by the RTDBMS, which schedules the operations issued clients. Commonly applied scheduling policies include First-In-First-Out (FIFO), round-robin, or based on the priorities of the transactions [1]. In addition to deadlines and validity intervals, other important time-related information includes execution times of the operations, and the arrival patterns of transactions (that is, whether a transaction is started with a period, with a bounded inter-arrival interval, or randomly) [5].

The ACID properties often need to be relaxed in order to guarantee temporal correctness [6], for instance, by using compensation rather than rollback for relaxed atomicity [14]. Real-time characteristics of transactions are incorporated in many CC algorithms for better timeliness. A widely applied real-time PCC is *Two Phase locking - High Priority (2PL-HP)* [15]. In this algorithm, a transaction acquires a readlock (write) on data before it performs a read (write) operation, and releases all locks during commitment. If two transactions try to lock the same data, and at least one of them requires a write lock, a CC conflict occurs. The transaction with higher priority will be granted with the lock, while the transaction with lower priority will be aborted by the RTDBMS. As a result, transactions with higher priorities are more likely to meet their deadlines.

B. UML Profiles and MARTE

UML is one of the most widely accepted modeling language in software development [8]. The profile mechanism is

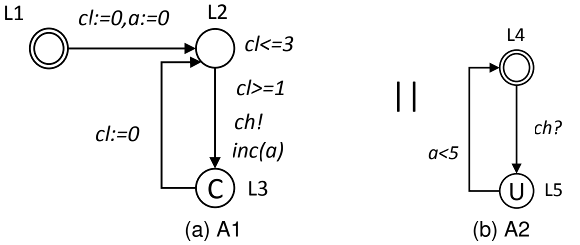


Fig. 1. A network of timed automata

designed to extend UML for languages customized for specific domains. A profile defines a package of stereotypes, which are domain-specific concepts derived from existing UML concepts, and constraints to associate them. A stereotype can have tagged values as attributes. Profiles may be used as specification languages to model systems, or adopted to add supplementary information that is used for analysis or code generation. As timing information is essential for our analysis and thus needs to be supported in the specifications, we reuse the relevant concepts from the MARTE (Modeling and Analysis of Real-Time Embedded systems) [9] profile. MARTE is a profile that defines the basic concepts to support the modeling of real-time and embedded applications, as well as to provide information for performance and schedulability analysis. In this paper, we propose a profile that encodes the information of transaction management and transactional properties for formal analysis. The following MARTE concepts are reused: (i) MARTE::NFP_Duration, a data type for time intervals; (ii) MARTE::ArrivalPattern, a data type for arrival patterns, such as periodic, sporadic and aperiodic patterns.

C. UPPAAL Timed Automata (TA)

Timed Automata are finite-state automata extended with real-valued clock variables [16]. UPPAAL TA extends TA with discrete data variables, synchronization channels, user-defined functions, among other modeling features [11]. Multiple TA can form a network via parallel composition, in which an individual automaton can perform its internal actions, while pairs of automata can perform hand-shake synchronization.

As an example to illustrate UPPAAL TA, Fig. 1 shows a network of TA modeling a simple concurrent real-time system, in which automaton A1 sporadically increments a variable *a* and synchronizes with automaton A2. A1 consists of a set of locations (L1, L2 and L3), and edges connecting them. A clock variable *cl* is defined in A1 to measure the elapse of time, and progresses continuously at rate 1. A discrete variable *a* is defined globally, and shared by A1 and A2. Semantically, a state of the network of TA consists of the current locations of the automata, together with the values of the clock and discrete variable. At each location, an automaton may stay at the location, as long as the **invariant**, which is a conjunction of clock constraints associated with the location, is satisfied. Alternatively and non-deterministically, the automaton may take a transition along an edge, if the **guard**, which is a conjunction of constraints on discrete or clock variables associated with the edge, is satisfied. In Fig. 1, A1 may delay in L2 as long as cl $\leq$ 3, or follow the edge to L3 when cl $\geq$ 1. Each edge may have an associated action, which is the synchronization with other automata via a **channel**. Binary channels are used to synchronize one sender (indicated by a mark "!") with a single receiver (indicated by a mark "?"). In Fig.1, A1 sends a message to A2 via binary channel *ch*, while taking the edge from L2 to L3. The synchronization can take place only if both the sender and the receiver are ready to traverse the edge. A broadcast channel is used to pass messages between one sender and an arbitrary number of receivers. When using broadcast channels, the sender does not block even if some of the receivers are not ready. An edge may have an *assignment*, which resets the clocks or updates discrete variables when the edge is traversed. In UPPAAL TA, both guards and assignments can be encoded as functions in a subset of the C language, which brings high flexibility and expressiveness to modeling. In our example, when A1 moves from L2 to L3, *a* is incremented using the function *inc(a)*.

A location marked as "U" is an urgent location, meaning that the automaton must leave the location without delay in time. Another automaton may fire transitions as long as time does not progress. A location marked as "C" is a committed location, which indicates no delay in time, and immediate transition. Another automaton may NOT fire any transitions, unless it is also at a committed location.

The UPPAAL model checker can verify properties specified as UPPAAL queries, in UPPAAL's property specification language [11] that is a decidable subset of Computation Tree Logic (CTL) [17]. For instance, the invariance property "A1 never reaches location L3" can be specified as "$A[\,]\, not\, A1.L3$", in which "A" is a path quantifier and reads "for all paths", whereas "[ ]" is the "always" temporal operator. If an invariance property is not satisfied, the model checker will provide a counterexample. The liveness property "If A1 reaches L2, it will eventually reach L3" can be specified, using the "leads-to ($\rightarrow$)" operator, as "$A1.L2 \rightarrow A1.L3$", which is equivalent to "$A[\,]\,(A1.L2\ imply\ A <>\ A1.L3)$", where "<>" is the "eventually" temporal operator.

In our previous, we model a concurrent transaction system as a network of UPPAAL TA, as follows [10]:

$$N' ::= A_0 \parallel ... \parallel A_{n-1} \parallel A_{CCManager} \parallel O_0 \parallel ... \parallel O_{k-1} \parallel D_0 \parallel ... \parallel D_{l-1},$$

where A_0, ..., A_{n-1} are the TA of work units of transactions T_0, ..., T_{n-1}, respectively. $A_{CCManager}$ is the automaton that models the CC algorithm. O_0, ..., O_{k-1} are the TA that observe the phenomena to be precluded by isolation, respectively. D_0, ..., D_{l-1} are the TA that monitors the time of data. Isolation and temporal correctness can be verified by UPPAAL model checker. We extend this framework in this paper to include atomicity and AR.

III. UTRAN Profile for Specification of Atomic Concurrent Real-time Transactions

In this section, we first present the domain model of real-time transactions in Section III-A, after which we introduce our proposed UTRAN profile in Section III-B.

A. Domain View

The domain model of real-time transactions is presented in Fig. 2. A transaction can be conceptually modeled as an activity in the UML activity diagram. In order for the RTDBMS to manage the life cycle of a transaction, a unique *id* is assigned to each transaction when it is started. A transaction may be assigned with a *TemporalCorrectnessSpecification* for time-related properties. This specification may specify a *priority* for the transaction, and a *relative deadline* that defines the maximum allowed time interval between the start and the

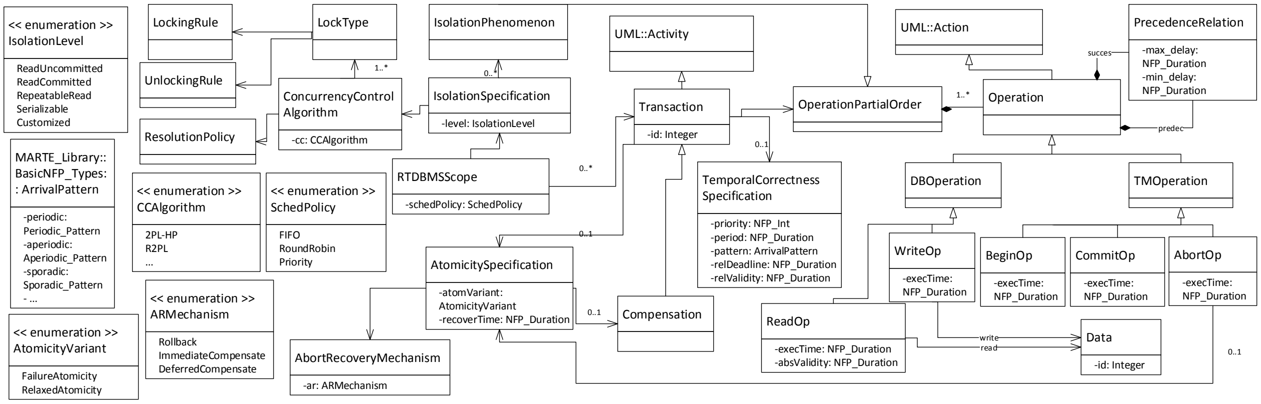


Fig. 2. Domain model of real-time transactions as a UML diagram

termination of the transaction. The arrival *pattern* can be specified for the transaction, such as periodic, sporadic and aperiodic, as well as the value of the *period* (or minimum inter-arrival time) if applicable. A transaction may also have a specified *relative validity interval*, for the validity of a group of data read by the transaction.

A transaction consists of a set of operations, represented as actions in an activity. Two types of operations are considered explicitly: *DBOperations* and *TMOperations*. DBOperations directly performs read and write of the data. Such read and write operations, denoted as *ReadOP* and *WriteOP* respectively, are atomic, whose *worst-case execution times* are also known. A ReadOP may be assigned with an *absolute validity interval* for the data it reads.

Multiple transactions managed by the same RTDBMS are related to an *RTDBMSScope*. A *scheduling policy*, which can be FIFO, RoundRobin and Priority-based, can be specified for the RTDBMSScope. An *isolation level* is specified for the RTDBMSScope, indicating the degree of isolation that should be provided for the set of transactions. Essentially, an isolation level defines a set of *IsolationPhenoma* to be precluded, which are illegal sequence of operations that lead to logical inconsistency. Therefore, an IsolationPhenomenon is basically an OperationPartialOrder. While some isolation levels have been defined in the SQL-92 standards (ReadUncommitted, ReadCommitted, RepeatableRead, Serializable) which can be selected from, customized isolation levels can also be defined with specified IsolationPhenoma. A *ConcurrencyControlAlgorithm* specifies the lock-based concurrency control algorithm selected for the specified isolation. Such an algorithm defines a set of *lock types*, each having its rules about, not only to which data it should apply, but also when a lock should be acquired and released. These rules are specified as *LockingRule* and *UnlockingRule*. A ConcurrencyControlAlgorithm also needs to specify a *resolution policy*, which describes how the conflicts are resolved when two transactions try to lock the same data.

An *AtomicitySpecification* specifies the atomicity variant to restore consistency when it gets terminated by error, as well as the desired *recovery time*. An AtomicitySpecification can either be attached to a transaction, which specifies the atomicity handling when the latter is aborted by the transaction management system, or to an abort operation, specifying the handling of abort issued by the clients. An AtomicitySpecification contains an *AtomicityVariant*, which is an enumeration of the supported atomicity variants, including *FailureAtomicity* and *RelaxedAtomicity*. An *AbortRecoveryMechanism* is associated with the atomicity variant. For FailureAtomicity, *Rollback* is the AR mechanism, by which the RTDBMS will undo all the changes in the database that have been done by aborted transaction. *ImmediateCompensate* and *DeferredCompensate* are the AR mechanisms for RelaxedAtomicity by compensation to restore the consistency of the database. The difference is that, the former allows the compensation to be executed immediately with highest priority, while in the latter case the compensation is scheduled as a separate transaction with the same priority as the aborted one. If no AtomicitySpecification is specified, atomicity is totally relaxed, and the partially changed data will not be recovered or compensated at all.

B. *UTRAN Profile*

Fig. 3 presents our UTRAN profile that contains the extensions to model the concepts in the previous domain model. The «Transaction» stereotype, extending the UML Activity metaclass, maps the Transaction domain element. Each activity stereotyped with «Transaction» may have a «TemporalCorrectnessSpecification» and an «AtomicitySpecification», which are associated comments that extends the Comment metaclass. A «TemporalCorrectnessSpecification» contains the information about the deadline, priority, arrival pattern, period, and relative validity of the transaction. An «AtomicitySpecification» specifies the selected AtomicityVariant and ARMechanism, from an enumeration of supported variants, as well as the recovery time, and the id of the compensation transaction which is a special transaction specified by the stereotype «Compensation». The actions in a «Transaction» are stereotyped as «Operation», each having the transaction id that they belong to, respectively. «DBOperation», «TMOperation» and «ClientOperation» map the DBOperation, TMOperation and ClientOperation, respectively. A «DBOperation» specifies the execution time to execute such an operation, and the id of the data it accesses. «ReadOP» and «WriteOP» extends «DBOperation», to map the ReadOP and WriteOP, respectively. A «TMOperation» specifies the execution time for the transaction management operation, which can be «BeginOP», «CommitOP», or «AbortOP».

IV. UPPCART Framework for Modeling Atomic Concurrent Real-time Transactions

In order to analyze the transactions specified in UTRAN formally, we propose a pattern-based framework, called UP-

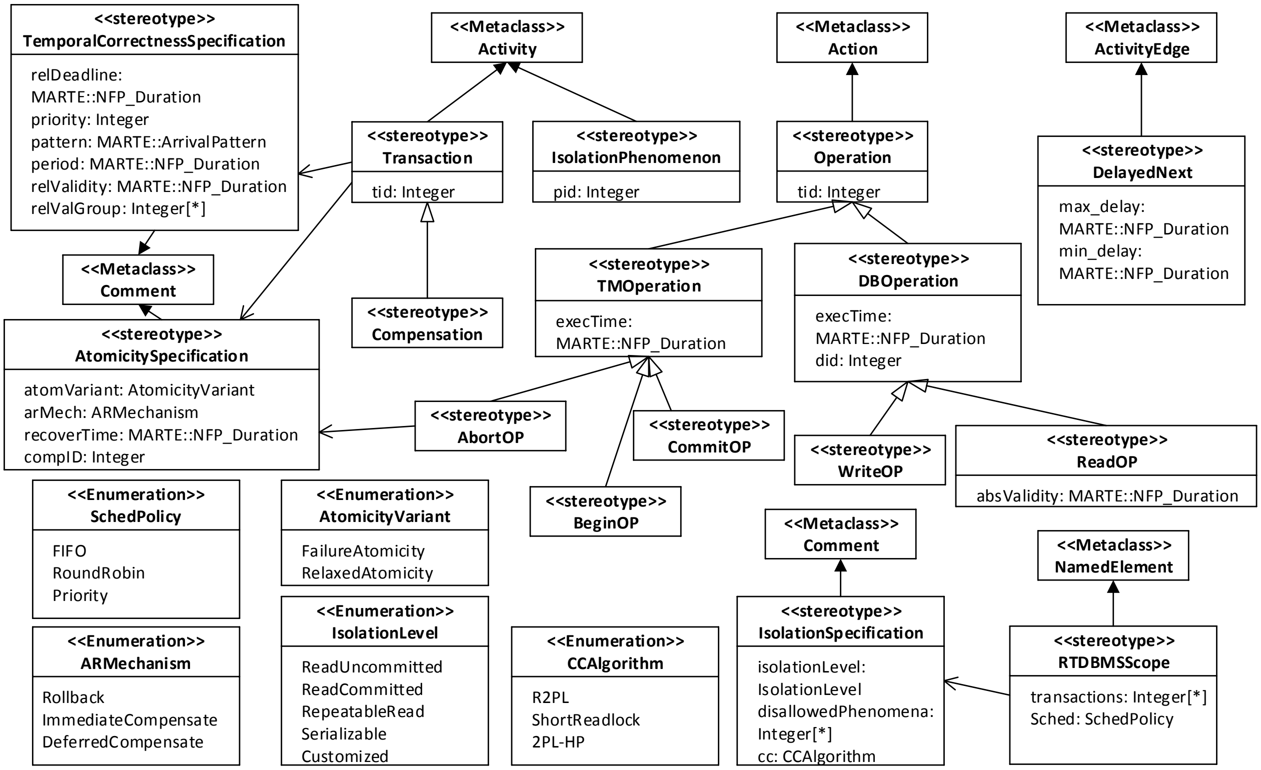


Fig. 3. UTRAN profile for real-time transactions

PCART (UPPaal for Concurrent Atomic Real-time Transactions), for modeling real-time transactions with concurrency control and abort recovery in UPPAAL TA. The proposed framework is based on our previous work [10], which focuses on isolation and temporal correctness only. In this work, we extend the previous framework with several substantial aspects. First, we extend the framework to incorporate the modeling of atomicity, via a set of reusable patterns for modeling various AR mechanisms, especially their interplays with CC. We also extend existing patterns for time-related behaviors, such as periodicity and delays. In addition, the proposed framework improves reusability of existing modeling units for transactions and CC, by proposing a unified reusable modeling unit for all `read`, `write`, `begin` and `commit` operations, as well as a more generic unit for CCManager, which is suitable for a wider range of concurrency control algorithms.

We model the transactions together with the CC and the AR as a network of UPPAAL TA, defined as follows:

$$N ::= A_0 \mid\mid ... \mid\mid A_{n-1} \mid\mid A_{CCManager} \mid\mid O_0 \mid\mid ... \mid\mid O_{k-1} \mid\mid \\ D_0 \mid\mid ... \mid\mid D_{l-1} \mid\mid A_{ATManager},$$

where A_0, ..., A_{n-1} are the TA of work units of transactions T_0, ..., T_{n-1}, respectively. They also model the WU's interaction with the transaction manager with respect to concurrency control and abort recovery. $A_{CCManager}$ is the CCManager automaton that models the CC algorithm, and interacts with the work unit TA. O_0, ..., O_{k-1} are the TA of IsolationObservers that observe the phenomena to be precluded by isolation, respectively. D_0, ..., D_{l-1} are the TA that monitor the time of data. $A_{ATManager}$ is the ATManager automaton that models the atomicity controller of recovery mechanisms upon abort.

We define two types of reusable structures for constructing the TA models. A *pattern*, consisting of a set of variables, locations, edges and even other patterns, is a parametrized structure representing the repetitive modeling unit in our framework. A pattern can be composed with the rest of the automaton after instantiation. A *skeleton* is a special type of pattern that defines the basic structure of a type of constituent automata of N, that is, a work unit skeleton, a CCManager skeleton, an ATManager skeleton, an IsolationObserver skeleton, and a Data skeleton. We also provide an algorithm to construct UPPCART models from UTRAN specification, which is included in the technical report [18] due to space limit.

In the following texts, we first introduce the details of UPPCART (Section IV-A), followed by the verification of the desired properties (Section IV-B).

A. The Proposed Modeling Framework

In the following subsections, we first introduce the skeletons and patterns for work units and concurrency control, as well as the skeleton of IsolationObserver. After this, we present the skeletons and patterns for atomicity and abort recovery mechanisms, and show how they are integrated with the work units and the CCManager.

1) Modeling Work Units:

a) ***Work Unit Skeleton:*** A WU Automaton (WUA) models the work unit of a transaction and its interaction with the CC and atomicity managers. A WU skeleton, as shown in Fig. 4, is a parametrized structure that consists of the common variables, locations and edges of a WU automaton. Starting from the *initial* location, the automaton immediately initializes the transaction with the specified id *ti* and priority *p* using function *initialize(ti, p)*, and moves to the location

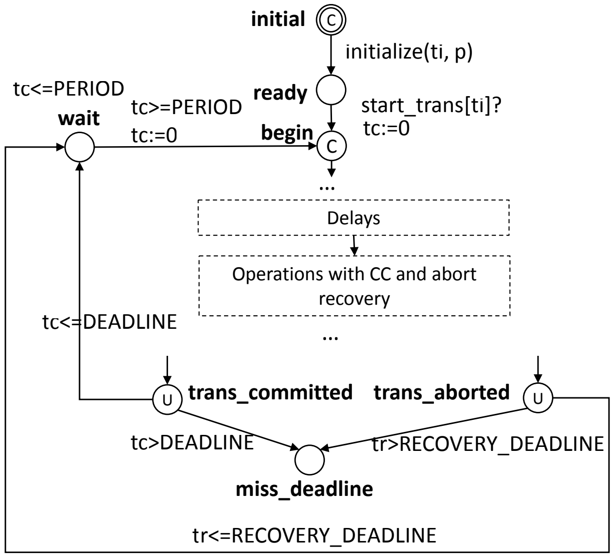


Fig. 4. TA skeleton for a work unit

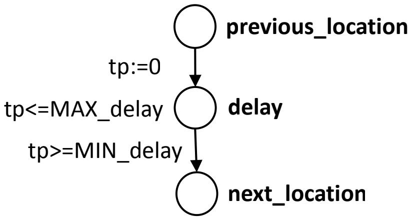


Fig. 5. Delay pattern

ready. After an arbitrary delay, it moves to the location *begin*, indicating the begin of the transaction, and sets clock variable *tc* to 0. The location *trans_committed* represents the committed state of the transaction. Between *begin* and *trans_committed* locations, there is a set of connected instantiated operation patterns that model the database and transaction management operations, and delays between the operations. If the value of *tc* is greater than the specified *DEADLINE*, the automaton moves to the location *miss_deadline*, indicating a deadline miss. Otherwise, it waits until the specified *PERIOD* has been reached, and moves to *begin* for the next activation. During the operations, the WUA may receive a message from the atomicity manager ATManager via channel *abort_trans[ti]*, and moves to the instantiated abort recovery pattern, which models the AR mechanism. The location *trans_aborted* represents the aborted state of the transaction. Similarly, if the value of *tr* is greater than a specified *RECOVERY_DEADLINE*, timeliness is breached, and the WUA moves to *miss_deadline*.

b) ***Delay Pattern****:* The pattern in Fig 5 models the delays between operations. The automaton may stay at location *delay* for at least *MIN_delay*, and at most *MAX_delay* time units, which are provided as parameters.

2) Modeling CC and Isolation Phenomena:

a) ***Operation-CC, Locking and Unlocking Patterns****:* We define a pattern to model the begin, commit, read and write operations in each work unit, respectively. Since within each operation, the work unit may interact with the CC manager according to the specific CC algorithm, our operation pattern also comprises CC-related activities such as locking and unlocking as sub-patterns. Our Operation-CC pattern is presented in Fig. 6. Scheduling is modeled by three functions, *enq_sch(ti)*, *deq_sch(ti)* and *sch()*, which specify the selected scheduling policy. After the *start_operation* location, the *enq_sch(ti)* function pushes the transaction id into the scheduling queue. On the edges from the location *check_sched*, the function *sch()* checks if the transaction is the next to be scheduled. If it is the case, the automaton moves to *do_operation*; otherwise, the automaton waits at location *wait*, until some transaction or the CCManager releases CPU via the *cpu_free* channel. The automaton may stay at *do_operation* for at most *WCRT_op* time units, and at least *BCRT_op* time units, which represent the longest and shortest allowed time to complete the operation, respectively. Upon the completion of the operation, a signal is sent to the IsolationObservers via channel *notify_op[ti]*. Before reaching *finish_operation*, the CPU is set free, and the transaction is removed from the queue

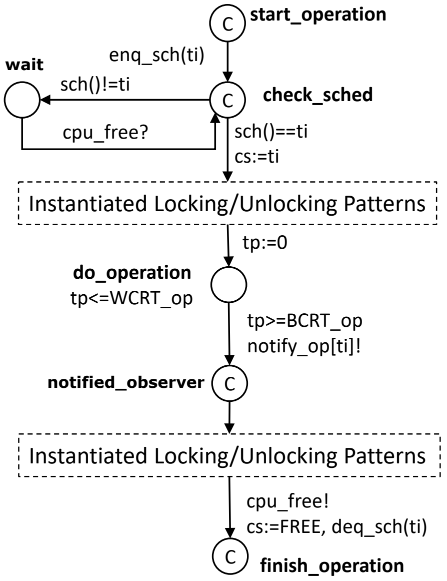
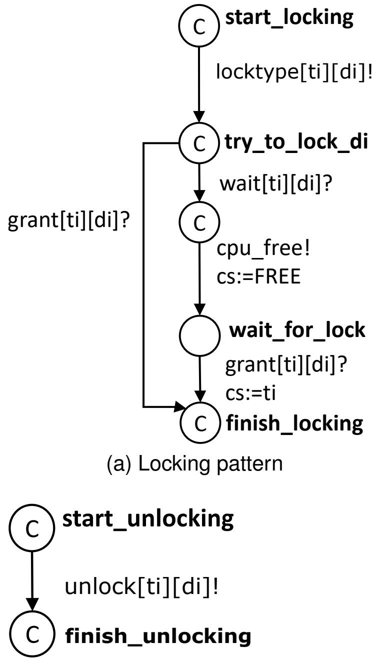


Fig. 6. Operation-CC pattern



(a) Locking pattern

(b) Unlocking pattern

Fig. 7. Locking and unlocking patterns

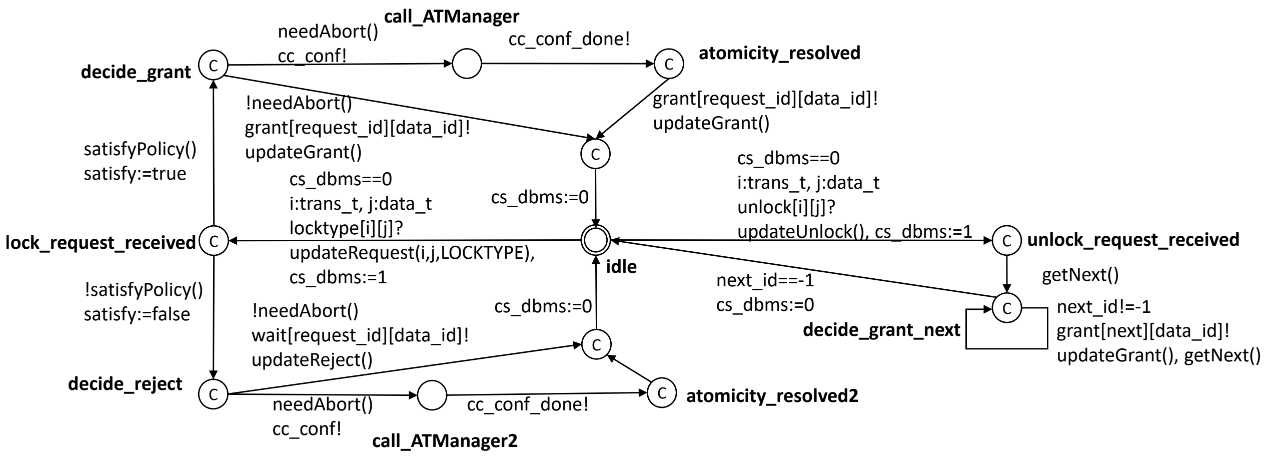


Fig. 8. TA skeleton for the CCManager

by the function *deq_sch(ti)*.

According to the selected CC algorithm, the transaction needs to lock and unlock data, before or after the operations. This is modeled by the Locking and Unlocking patterns inserted into the operation patterns, as illustrated in Fig. 6. The Locking and Unlocking patterns are presented in Fig. 7. In the Locking pattern, the automaton sends a request to the CCManager via channel *locktype[ti][di]*, in which "locktype" is parametrized for the particular type of lock, such as a readlock, specified by the CC algorithm. The automaton then either moves to location *finish_locking*, if it is granted by CCManager via channel *grant[ti][di]*, or releases the CPU and gets blocked at location *wait_for_lock*, until the CCManager grants it later. In the Unlocking pattern, the automaton sends the request via channel *unlock[ti][di]*, which is received and processed by the CCManager.

*b) **CCManager Skeleton**:* The CCManager skeleton (Fig. 8) provides a common structure for modeling various CC algorithms, as well as the interaction with the transactions and the atomicity manager. The particular resolution policy of a CC algorithm is encoded in the functions. When CCManager receives a locking request, it updates the status of the transaction and the data by calling *updateRequest()*, and judges whether the requester can obtain the lock by calling *satisfyPolicy()*. The satisfying requester is granted with the lock, if the algorithm does not abort any transactions in order to resolve conflicts. If any transactions need to be aborted due to concurrency conflicts, as suggested by *needAbort()*, CCManager sends a signal to ATManager via channel *cc_conf*, and waits until all abort and recovery are handled, before it grants the lock to the requester. On the other hand, if the requester does not satisfy the policy, it is either aborted, decided *needAbort()* according to the CC algorithm, or blocked. When CCManager receives an unlocking request, it updates the status of the transaction, and grants locks to all legitimated blocked transactions. The next granted transaction is decided using the *getNext()* function.

*c) **IsolationObserver Skeleton**:* The skeleton for an IsolationObserver is shown in Fig. 9. Each IsolationObserver observes a specified sequence of operations, by accepting the corresponding notification messages from the work unit automata via the *notify_op[ti][di]* channel when an operation is completed. If the monitored sequence indicating the phenomeon occurs, the automaton moves to the *isolation_phenomenon* location.

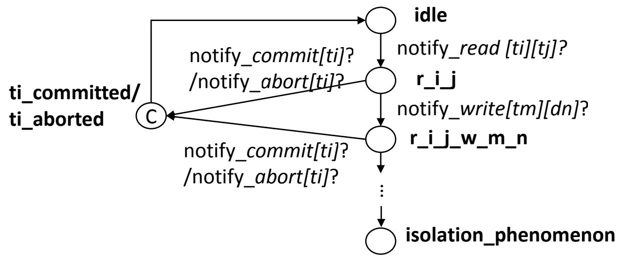


Fig. 9. IsolationObserver skeleton

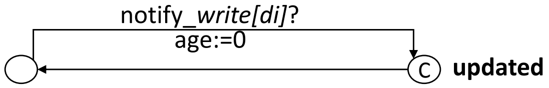


Fig. 10. TA skeleton for data

3) Modeling Data: Fig. 10 presents the skeleton of data. The clock variable *age*, which is reset every time a write operation is performed on the data, represents how old the data is since the last update.

4) Modeling Atomicity and Abort Recovery: We separate the atomicity control model into an ATManager automaton, and the abort recovery parts in work unit automata. The ATManager models the behavior of deciding the transactions to be aborted upon errors, conflicts or user's instructions. The work unit automata include the instantiated abort recovery patterns that model the selected mechanisms for the specific transactions. We distinguish two types of abort, which are *user abort* that is issued by a client using an abort operation deliberately, and *system abort* that occurs due to internal conflicts and system failures, such as CC conflicts.

*a) **ATManager Skeleton**:* Our ATManager skeleton provides a common structure for modeling the atomicity manager. As shown in Fig. 11, the ATManager may receive user abort requests via the *user_abort[i]* channel, or system abort due to CC via *cc_conf* channel from CCManager. Other types of errors, such as communication errors, can be modeled similarly. The function *getAbort()* specifies the logic to decide the transaction to be aborted. The automaton then sends the abort signal to the corresponding WU automaton via channel *abort_trans[abort_id]*, and waits until the abort is done by the WU. ATManager then updates the status and locks of transactions and data using the function *updateAbort()*, and

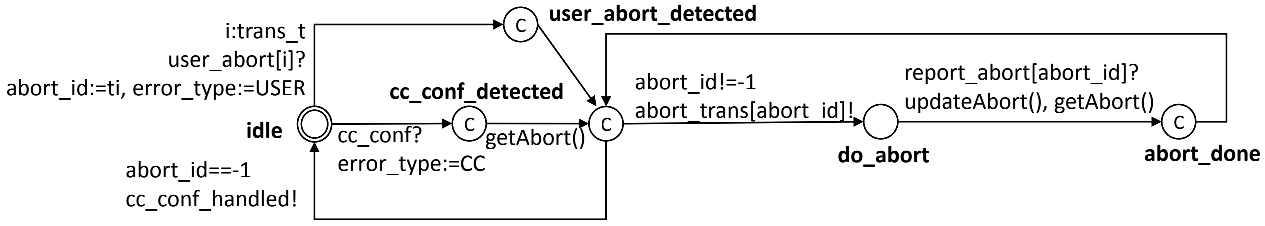


Fig. 11. TA skeleton for the ATManager

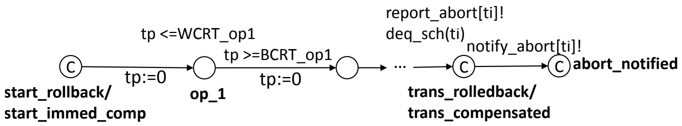


Fig. 12. RollbackImComp pattern

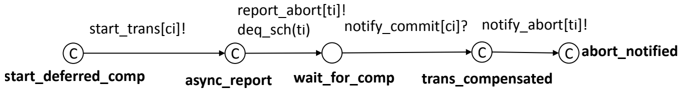


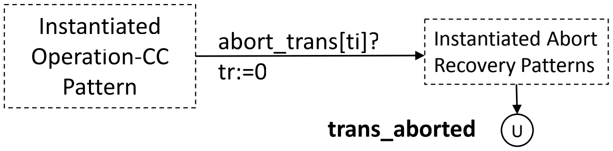
Fig. 13. DeferredComp pattern

checks if more transactions need to be aborted.

*b) **Abort Recovery Patterns: RollbackImComp and DeferredComp**:* The AR mechanisms are modeled by the RollbackImComp pattern (Fig. 12), and the DeferredComp pattern (Fig. 13), respectively, which are composed into the work unit automata. The former models the rollback and immediate compensation mechanisms, while the latter models the deferred compensation mechanism. The Rollback-ImmeComp pattern models the execution of a series of operations by the DBMS. In case of rollback, the operations are the ones completed before the abort of the transaction. In case of immediate compensation, the operations are specified for the transaction.

In the **RollbackImComp** pattern, each operation is represented by a location *op_n*, at which the automaton may stay for at most (least) *WCRT_opn* (*BCRT_opn*) time units. When all operations are completed, the work unit reports the completion of recovery to the ATManager via channel *report_abort[ti]*, removes the transaction from the scheduling queue by function *deq_sch(ti)*, and notifies the IsolationObserver via channel *notify_abort[ti]*. In case of deferred compensation, the **DeferredComp** pattern starts the compensation transaction via channel *start_trans[ci]*, where *ci* is the id of the compensating transaction, and immediately reports to ATManager and removes the aborted transaction from the scheduling queue. The compensating transaction *ci* is modeled as a separate work unit, using the work unit skeleton and the operation patterns. When *ci* commits, the DeferredComp pattern receives the notification, and notifies that the transaction is aborted and recovered via channel *notify_abort[ti]*.

*c) **SystemAbort Pattern**:* System abort is modeled as a composition of an instantiated operation pattern with a Rollback-ImmComp pattern or a DefComp pattern, as shown in Fig. 14. We refer to this compensation as the SystemAbort pattern. In this pattern, when the WU automaton receives an *abort_trans[ti]* signal from the ATManager, it moves to the corresponding abort recovery patterns.



Instantiated Operation-CC Pattern
abort_trans[ti]?
tr:=0
Instantiated Abort Recovery Patterns
trans_aborted
U

Fig. 14. SystemAbort pattern

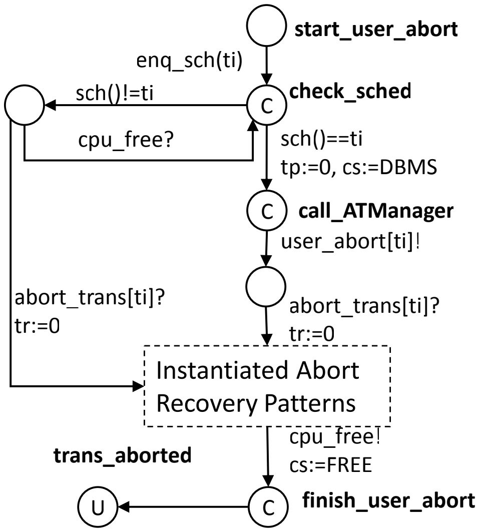


Fig. 15. UserAbort pattern

*d) **UserAbort Pattern**:* The UserAbort pattern is defined in Fig. 15. When the work unit is scheduled as the next one to execute by function *sch(ti)*, it issues the abort request to ATManager via channel *user_abort[ti]*. After it gets the permission from ATManager, the automaton moves to the corresponding abort recovery pattern. When the recovery is completed, the automaton sets the CPU to be free, and moves to location *trans_aborted*.

B. Verification

We propose an algorithm in our technical report [18] to construct UPPCART models from UTRAN using the aforementioned skeletons and patterns, which can potentially be automated. With the transactions as well as the AR and CC mechanisms modeled in UPPAAL TA, we are able to formally verify the atomicity, isolation and temporal correctness properties. Table I lists the patterns to formalize the properties in UPPAAL queries. Among them, atomicity is formalized as a liveness property, while isolation and temporal correctness are formalized as invariance properties.

TABLE I. UPPAAL QUERY PATTERNS FOR VERIFYING TRANSACTIONAL PROPERTIES

Property Type	Property Description	UPPAAL Query Pattern
Atomicity	T_i aborted due to ERRORTYPE is eventually rolled back (compensated)	$(ATManager.abort\_id == i \,\&\&\, ATManager.error\_type == ERRORTYPE) \rightarrow Ai.trans\_rolledback\,(Ai.trans\_compensated)$
Isolation	The specified isolation phenomena never occur	$A\,[\,]\ not\,(O_1.isolation\_phenomenon \vert\vert \ldots \vert\vert O_n.isolation\_phenomenon)$
Timeliness	T_i never misses its deadline	$A\,[\,]\ not\,Ai.miss\_deadline$
Absolute Validity	When read by T_i, D_j is never older than the absolute validity interval AVI(j)	$A\,[\,]\,(Ai.read\_di\_done\,imply\,Dj.age <= AVI(j))$
Relative Validity	Whenever T_i reads D_j or D_l, the age differences of D_j and D_l is smaller than or equal to the relative validity interval RVI(j,l)	$A[\,]\,((Ai.read\_dj\_done \vert\vert Ai.read\_dl\_done)\,imply\,((Dj.age - Dl.age <= RVI(j,l))\,\&\&\,(Dl.age - Dj.age <= RVI(j,l))))$

V. ILLUSTRATIVE EXAMPLE

Let us take two autonomous wheel loaders and their controller as an example. All data, including the positions of the wheel loaders, their working conditions, the work plan, and the speed configurations, are stored in the controller's database. Assume that loader A is patrolling on the location with its predefined speed, and periodically updates its position in the database. A human operator can update the configuration data, that is, the work plan, and the speed settings of the wheel loader A. These data should be updated at the same time. If the update fails, the data should be rolled back to the previous values. The controller may get a command to start a job with loader B. It updates the status of the job to "start", reads the current location of loader A, reads the work plan in the database, reads the speed configuration of loader A, and calculates the estimated speed and direction of the wheel loader B. Loader B then moves to the position and informs the controller, which updates the job to "finish". If this job fails, due to some reason, the estimated position should be updated as "unknown", and the job status as "failed".

a) ***Specification in UTRAN****:* We consider three transactions in this scenario. The first transaction (UpdateConfTrans) updates the configuration data. The second (JobTrans) controls the loader B to do the job. The third transaction (UpdateLocA) updates the location of A periodically. The temporal correctness properties are specified in their respective attached «TemporalCorrectnessSpecification», with their deadlines and validity intervals. The atomicity variant of UpdateConfTrans, which is rollback, is specified in its «AtomicitySpecification». On the contrary, JobTrans selects ImmediateCompensate, as specified in its «AtomicitySpecification», and compensates its failure with the compensation transaction LogError, which updates the estimated position and logs the error. The transactions are in the scope of the RTDBMS, stereotyped with «RTDBMSScope», whose isolation level is set to be RepeatableRead in its «IsolationSpecification», which disallow the phenomena InconsistencyConfigs1 and InconsistencyConfigs2, both stereotyped as «IsolationPhenomenon». The specification in UTRAN, due to space limit, is presented in the report [18].

b) ***Construct UPPCART models****:* We construct UPPCART models from UTRAN following the algorithm in the report [18]. The work unit skeleton, the Operation-CC pattern, the locking/unlocking patterns, as well as the abort recovery patterns, are used to construct work unit automata. The CCManager for 2PL-HP shares the same structure with the CCManager skeleton in Fig. 8. The ATManager is constructed using the ATManager skeleton in Fig. 11. The IsolationObservers and data automata are instantiated using the skeletons in Fig. 9 and Fig. 10, respectively. The functions that model the priority-based scheduling and the abort decision, as well as other TA models, are included in the technical report [18].

c) ***Verification****:* We use the patterns in Table I to formalize the properties for the system, and verify them using the UPPAAL model checker. The verification results are listed in Table II, which shows that all properties are satisfied with the selected CC, AR and scheduling mechanisms.

VI. RELATED WORK

A number of high-level description languages have been proposed for transaction-based systems. Some of them, like ours, extend UML or existing profiles. Marouane et al. [19] extends MARTE for real-time database systems. Timing properties can be specified using their profile, while atomicity and isolation are not considered. Unified Transaction Modeling Language (UTML) [20] and its extension [21] are UML-based languages for transactions that enables selection of the ACID properties. Atomicity and isolation are treated as monolithic properties respectively, rather than tunable variants [7]. Timeliness is not the authors' focus. The Business Process Execution Language (BPEL) [22] and the Business Process Model and Notation (BPMN) [23] are XML-based, high-level description languages for specifying business processes, which is a flexible transaction model with various atomicity. Rollback and compensation can be specified at transaction level and for internal activities. Charfi et al. [24] and Sun et al. [25] introduce extra concepts for transactions to BPEL, which allow transaction policies for atomicity to be specified explicitly. Compared with their work, our proposed profile can specify variants of isolation as well as timing properties. Watahiki et al. [26] introduce temporal constraints to BPMN and verify them with UPPAAL. Isolation and CC are not part of this framework. Both ASSET [27] and KALA [28] use procedural languages for flexible transaction models, in which operations and AR are specified using designated primitives. Compared to these works, ours supports specification of temporal correctness, and the selection of CC algorithms.

Much effort has been dedicated to formally model and analyze transaction properties. The ACTA framework [29] specifies transaction models in first order logic and allows for formal reasoning. Gallina [7] uses higher-order logic to specify transaction properties, which can be formally analyzed by the Alloy tool. Both frameworks are restricted in the formal specification and analysis of ACID, while temporal correctness, especially the impact of CC and abort recovery mechanisms, are not included. Derks et al. [12] propose to model and verify transactions with atomicity variants in Petri nets. Liu et al. [30] model and analyze a transaction model using Maude. Lanotte et al. [31] propose a timed-automata-based language for long running transactions with timing constraints. Committing protocols for atomicity variants can be modeled and analyzed. In contrast to these works, our work provides a formal framework for modeling transactions together with abort recovery and CC mechanisms, in which atomicity, isolation, temporal correctness, as well as their impacts on each other, can be analyzed in a unified framework.

VII. CONCLUSIONS AND FUTURE WORK

In this paper, we have proposed a UML profile called UTRAN for specifying atomic concurrent real-time transac-

TABLE II. VERIFICATION RESULTS OF THE EXAMPLE SYSTEM

Property Type	UPPAAL Query Pattern	Verification Time	Explored States	Verification Result
Atomicity	$(ATManager.abort\_id == 1 \,\&\&\, ATManager.error\_type == CC)$ $\rightarrow A1.trans\_rolledback$	5.99s	344107	Satisfied
Atomicity	$(ATManager.abort\_id == 2 \,\&\&\, ATManager.error\_type == USER)$ $\rightarrow A2.trans\_rolledback$	6.13s	344475	Satisfied
Atomicity	$(ATManager.abort\_id == 2 \,\&\&\, ATManager.error\_type == CC)$ $\rightarrow A2.trans\_compensated$	6.13s	355050	Satisfied
Isolation	$A\,[\,]\; not\,(InconsistentConfig1.isolation\_phenomenon$ $\Vert\, InconsistentConfig2.isolation\_phenomenon)$	5.60s	336405	Satisfied
Timeliness	$A\,[\,]\; not\,(A1.miss\_deadline \Vert A2.miss\_deadline \Vert A3.miss\_deadline)$	5.62s	336405	Satisfied
Absolute Validity	$A\,[\,]\,(A2.read\_d4\_done\; imply\; D4.age <= 150)$	9.45s	423960	Satisfied
Relative Validity	$A[\,]\,((A2.read\_d1\_done \Vert A2.read\_d2\_done)\; imply$ $((D1.age - D2.age <= 15) \,\&\&\, (D2.age - D1.age <= 15)))$	18.35s	547479	Satisfied

tions. UTRAN supports specification of transaction atomicity, isolation and temporal correctness, as well as the selection of AR, CC and scheduling mechanisms, in high level. Specified as UML activities with UTRAN, transactional properties can be explicitly specified, and be extracted and further analyzed by tools. We have also proposed a framework based on UPPAAL TA to formally model the UTRAN specification, which allows the specified properties to be rigorously verified by UPPAAL model checker. We provide a set of parametrized automata skeletons and patterns to model the transaction system. Via instantiation and composition, these skeletons and patterns enable flexible modeling of a wide range of abort recovery mechanisms and CC algorithms. Properties are formalized as UPPAAL queries for verification. We have also proposed an algorithm to construct the TA model from an UTRAN specification, which can potentially be automated by a tool.

Our future work will focus on a tool chain that facilitates the entire process, from high-level specification, to automatic model generation and verification. Another future work is to improve the scalability of our formal framework. In case of large systems, exhaustive model checking may not converge due to state explosion. Other formal techniques such as statistical model checking could be integrated for better scalability. The verification of the C functions encoding the mechanisms by employing a program verifier is also a future work.

Acknowledgment: The Swedish Research Council (VR) is gratefully acknowledged for supporting this research by the project "Adequacy-based Testing of Extra-Functional Properties of Embedded Systems".

REFERENCES

[1] R. A. Elmasri and S. B. Navathe, *Fundamentals of Database Systems*. Addison-Wesley Longman Publishing Co., Inc., 2004.

[2] J. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 1992.

[3] S. Han *et al.*, "On co-scheduling of update and control transactions in real-time sensing and control systems: Algorithms, analysis, and performance," *IEEE TKDE*, vol. 25, pp. 2325–2342, 2013.

[4] S. Cai *et al.*, "Customized real-time data management for automotive systems: A case study," in *43rd IECON*, 2017, pp. 8397–8404.

[5] K. Ramamritham, "Real-time databases," *Distributed and Parallel Databases*, vol. 1, no. 2, pp. 199–226, 1993.

[6] J. A. Stankovic *et al.*, "Misconceptions about real-time databases," *Computer*, vol. 32, no. 6, pp. 29–36, 1999.

[7] B. Gallina, "Prisma: a software product line-oriented process for the requirements engineering of flexible transaction models," Ph.D. dissertation, University of Luxembourg, 2010.

[8] "The unified modeling language specification version 2.5.1," OMG, Standard. [Online]. Available: https://www.omg.org/spec/UML/2.5.1/

[9] "Uml profile for marte specification version 1.1," OMG, Standard. [Online]. Available: https://www.omg.org/spec/MARTE/1.1/

[10] S. Cai *et al.*, "A formal approach for flexible modeling and analysis of transaction timeliness and isolation," in *24th RTNS*, 2016, pp. 3–12.

[11] K. Larsen *et al.*, "Uppaal in a nutshell," *International Journal on Software Tools for Technology Transfer*, vol. 1, pp. 134–152, 1997.

[12] W. a. Derks *et al.*, "Customized atomicity specification for transactional workflows," in *The Proceedings of the 3rd CODAS*, 2001, pp. 140–147.

[13] A. Adya *et al.*, "Generalized isolation level definitions," in *Proceedings of the 16th ICDE*, 2000, pp. 67–78.

[14] N. Soparkar *et al.*, "Adaptive commitment for distributed real-time transactions," in *Proceedings of the third CIKM*, 1994, pp. 187–194.

[15] R. K. Abbott and H. Garcia-Molina, "Scheduling real-time transactions: A performance evaluation," *ACM TODS*, vol. 17, pp. 513–560, 1992.

[16] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical computer science*, vol. 126, no. 2, pp. 183–235, 1994.

[17] E. M. Clarke *et al.*, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Transactions on Programming Languages and Systems*, vol. 8, no. 2, pp. 244–263, 1986.

[18] S. Cai *et al.*, "Specification and verification of transaction atomicity, isolation and temporal correctness," Tech. Rep., June 2018. [Online]. Available: http://www.es.mdh.se/publications/5154-

[19] H. Marouane *et al.*, "An uml profile for representing real-time design patterns," *Journal of King Saud University-Computer and Information Sciences*, 2017.

[20] G. Nektarios and S. Christodoulakis, "Utml: Unified transaction modeling language," in *Proceedings of the 3rd WISE*, 2002, pp. 115–126.

[21] D. Distante *et al.*, "A comprehensive design model for integrating business processes in web applications," *International Journal of Web Engineering and Technology*, vol. 3, no. 1, pp. 43–72, 2006.

[22] "Web services business process execution language version 2.0," OASIS, Standard. [Online]. Available: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html

[23] "Business process model and notation specification version 2.0," OMG, Standard. [Online]. Available: https://www.omg.org/spec/BPMN/2.0/

[24] A. Charfi *et al.*, "Transactional bpel processes with ao4bpel aspects," in *Fifth European Conference on Web Services*, 2007, pp. 149–158.

[25] C. Sun *et al.*, "Transaction management in service-oriented systems: Requirements and a proposal," *IEEE Transactions on Services Computing*, vol. 4, no. 2, pp. 167–180, 2011.

[26] K. Watahiki *et al.*, "Formal verification of business processes with temporal and resource constraints," in *IEEE International Conference on Systems, Man, and Cybernetics*, 2011, pp. 1173–1180.

[27] A. Biliris *et al.*, "Asset: A system for supporting extended transactions," in *ACM SIGMOD Record*, vol. 23, no. 2, 1994, pp. 44–54.

[28] J. Fabry and T. D'Hondt, "Kala: Kernel aspect language for advanced transactions," in *Proceedings of the 2006 SAC*, 2006, pp. 1615–1620.

[29] P. K. Chrysanthis and K. Ramamritham, "Synthesis of extended transaction models using acta," *ACM TODS*, vol. 19, pp. 450–491, 1994.

[30] S. Liu *et al.*, "Formal modeling and analysis of ramp transaction systems," in *Proceedings of the 31st SAC*, 2016, pp. 1700–1707.

[31] R. Lanotte *et al.*, "Modeling long-running transactions with communicating hierarchical timed automata," in *Formal Methods for Open Object-Based Distributed Systems*. Springer, 2006, pp. 108–122.

[32] Ï. B. Arpinar *et al.*, "Formalization of workflows and correctness issues in the presence of concurrency," *Distributed and Parallel Databases*, vol. 7, no. 2, pp. 199–248, 1999.

[33] A. Zarras and V. Issarny, "A framework for systematic synthesis of transactional middleware," in *Middleware'98*, 1998, pp. 257–272.