# Modelling Real-time Database Systems in Duration Calculus

Dang Van Hung and Ho Van Huong
United Nations University
International Institute for Software Technology
P.O.Box 3058, Macao

**ABSTRACT**
In this paper, we give a formal model for real-time database systems using Duration Calculus. Our model supports the formal reasoning about the operations in the systems. As a case study for our technique, we give a formal specification and verification of the Read/Write Priority Ceiling Protocol (R/WPCP).

**KEY WORDS**
Real-time Database, Concurrency Control, Duration Calculus.

## 1 Introduction

Real-time Database Systems (RTDBS) have been used in a wide range of applications such as air traffic control, robotics and nuclear power plants. Research on RTDBS has received a great deal of attention [1, 4], and belongs to two different important areas in Computer Science: real-time systems and database systems. Usually, transactions in RTDBS are associated with time constraints, e.g. deadline. The concurrency control in Real Time Database Systems has to ensure not only the consistency of the data in the multi-user environments like in traditional databases, but also the temporal consistency of the data, and that all transactions meet their deadline. Therefore, the concurrency control in Real Time Database Systems (RTDBS) is much more complicated than the one in traditional database systems (DBS). A formal technique for modelling and reasoning about the behaviour of the concurrency control will support the analysis and verification of concurrency control algorithms. In this paper, we focus on the development of such a mathematical model of RTDBS. In particular, we focus on the formalisation of the integration of the concurrency control with scheduling in RTDBS. We use Duration Calculus (DC) introduced by Zhou, Hoare and Ravn in [6] as a logical foundation for our approach. The reason for this choice is that DC is a simple and powerful logic for reasoning about the time durations, which is a main concept in scheduling.

Our approach is summarised as follows. We first extend the formal model of RTDBS proposed in [5] with the capacity for modelling (iterated) periodic transaction systems. Handling the periodic transaction systems with deadline is not trivial. Then, we give a formal specification of the correctness criteria for concurrency control in RTDBS which consists of the serializability, the temporal consistency and meeting timing constraints. To show the advantages of our model, we give a formal specification and verification of the Read/Write Priority Ceiling Protocol (R/WPCP).

## 2 Duration Calculus

Duration Calculus (DC) is a logical approach to formal design of real time systems. DC adopts continuous time and uses boolean-valued functions over time to model states and events of the real time systems. The duration of a state over a time interval is the accumulated presence time of the state over the interval. We refer to [3] for more comprehensive introduction to DC..

Time in DC is the set $R^+$ of non-negative real numbers. For $t, t' \in R^+, t \leq t', [t, t']$ denotes the time interval from $t$ to $t'$. Let $Intv$ denote the set of all intervals.

DC has a set $E$ of boolean state variables. $E$ includes the Boolean constants 0 and 1 denoting **false** and **true** respectively. State expressions, denoted by $P, Q, P_1, Q_1$, etc., are built from boolean state variables using boolean connectives. A state variable $P$ is interpreted as a function $I(P) : R^+ \to \{0, 1\}$ (a state). $I(P)(t) = 1$ means that state $P$ is present at time instant $t$, and $I(P)(t) = 0$ means that state $P$ is not present at time instant $t$. We assume that a state has the finite variability in a finite time interval. A state expression is interpreted as a function which is defined in the obvious way from the interpretations for the state variables and Boolean operators.

For an arbitrary state expression $P$, the duration of $P$ is denoted by $\int P$. Given an interpretation $I$ of state variables and an interval $[t, t']$, duration $\int P$ is interpreted as $\int_t^{t'} I(P)(t)dt$. $\int 1$ always gives the length of the intervals and is denoted by $\ell$. An arithmetic expression built from state durations and real constants is called a term.

We also assume a set of temporal propositional letter $X, Y, ...$ Each temporal propositional letter is interpreted by $I$ as truth-valued functions of time intervals.

A primitive duration formula is either a temporal propositional letter or a Boolean expression formed from terms by using the usual relational operations on the reals, such as equality $=$ and inequality $<$. A duration formula is either a primitive formula or an expression formed from other formulas by using the logical operators $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$, and the chop ;.

A duration formula $D$ is satisfied by an interpretation $I$ in an interval $[t', t'']$ just when it evaluates to true for that interpretation over that time interval.

Given an interpretation $I$, the chop-formula $D_1; D_2$ is true for $[t', t'']$ iff there exists a $t$ such that $t' \leq t \leq t''$ and $D_1$ and $D_2$ are true for $[t', t]$ and $[t, t'']$ respectively.

We give now shorthands for some duration formulas which are often used. For an arbitrary state variable $P$, $\lceil P \rceil$ stands for $(\int P = \ell) \wedge (\ell > 0)$. This means that interval is a non-point interval and $P$ holds almost everywhere in it. When $P$ in interpreted as a one-side continuous function, $\lceil P \rceil$ means that the interval is a non-point interval and $P$ holds everywhere in it. We use $\lceil\ \rceil$ to denote the predicate which is true only for point intervals. Modalities $\Diamond, \Box$ are defined as: $\Diamond D \widehat{=} \textbf{true}; D; \textbf{true}$, $\Box D \widehat{=} \neg\Diamond\neg D$. This means that $\Diamond D$ is true for an interval iff $D$ holds for some subinterval of it, and $\Box D$ is true for an interval iff $D$ holds for all subintervals of it. We will use the abbreviation $\lceil P \rceil^* \ \widehat{=}\ \lceil\ \rceil \vee \lceil P \rceil$

We refer the readers to [3] for a powerful proof system for DC which is complete for the abstract time domain.

# 3 Formal Model of Real Time Database Systems in DC

## 3.1 Basic model

Let a real-time database system consist of a set $\mathcal{O}$ of data objects (ranged over by $x, y, z, \ldots$), and a set $\mathcal{T}$ of $n$ transactions $T_i$, $1 \leq i \leq n$.

Each transaction $T_i$ arrives at the DB system periodically with the period $P_i$. In a period, a transaction performs some read operations on some data objects, does some local computations and then performs some write operations on some data objects. We assume the atomic commitment of transactions: if a transaction has been aborted then it's execution has no effects on the database. We also assume that each transaction can read and write to a data object at most once during its execution in one period. These assumptions are for the simplicity and well accepted in the literature. Each transaction $T_i$, besides its period $P_i$, also has its own deadline $D_i$, a priority $p_i$, an execution time $C_i$, a data read set $RO_i$ and a data write set $WO_i$ ($RO_i$ and/or $WO_i$ may be empty).

Let $x$ be a data object. For each $i \leq n$ let $T_i.wrtn(x)$ be a DC state variable expressing the behaviour of $x$. $T_i.wrtn(x)$ holds at time $t$ iff the value of $x$ at $t$ is the one written by transaction $T_i$. The view of $T_i$ on $x$ can be modelled as a state variable $T_i.read(x)$ defined as follows. $T_i.read(x)$ holds at time $t$ within a period iff $T_i$ has performed a read operation on $x$ successfully before $t$ in that period. Therefore, the read operation on $x$ in a period is performed at the time that $T_i.read(x)$ changes its value from 0 to 1 in that period. To express that a time interval $[a, b]$ is a period of $T_i$, we introduce temporal propositional letter $T_i.priod$. $T_i.period([a, b]) =$ **true** iff $[a, b]$ is a period of $T_i$. Of course,

$$T_i.period \Rightarrow \ell = P_i. \tag{1}$$

For each $i \leq n$ state variable $T_i.arrd$ is introduced to express that $T_i$ is standing in the system at time $t$: $T_i.arrd(t) = 1$ iff at time $t$ transaction $T_i$ is in the system and has not been committed or aborted since then. Because we assume that $T_i$ arrives at the beginning of any period of its, it holds:

$$T_i.period \Rightarrow \lceil T_i.arrd \rceil \frown \textbf{true} \tag{2}$$

We introduce variables $T_i.req\_rlk(x)$ and $T_i.req\_wlk(x)$ to express that $T_i$ is requesting lock for a data object $x$ at time $t$: $T_i.req\_rlk(x)(t) = 1$ iff transaction $T_i$ is requesting a read-lock on $x$ at time $t$, and $T_i.req\_wlk(x)(t) = 1$ iff transaction $T_i$ is requesting a write-lock on $x$ at time $t$.

When a transaction $T_i$ requests a lock on data object $x$, it may be granted or has to wait. Therefore, for each $i \leq n$ and for each $x$, we introduce the state variables $T_i.wt\_wlk(x)$ and $T_i.wt\_rlk(x)$ as: $T_i.wt\_rlk(x)(t) = 1$ iff transaction $T_i$ is waiting for a read-lock on data object $x$ at time $t$, $T_i.wt\_wlk(x)(t) = 1$ iff transaction $T_i$ is waiting for a write-lock on data object $x$ at time $t$, $T_i.hld\_rlk(x)(t) = 1$ iff at time $t$ transaction $T_i$ holds a read-lock on data object $x$, and $T_i.hld\_wlk(x)(t) = 1$ iff at time $t$ transaction $T_i$ holds a write-lock on data object $x$.

In a period, a transaction can commit or abort. Therefore, for each $i \leq n$ state variables $T_i.comtd$ and $T_i.abd$ are introduced to express that $T_i$ has already committed or aborted at time $t$: $T_i.comtd(t) = 1$ iff $T_i$ has committed successfully before $t$ in a period of containing $t$, and $T_i.abd(t) = 1$ iff $T_i$ has aborted before $t$ in a period of containing $t$.

At the beginning of a period, all transactions have not read anything from the database.

$$\Box(T_i.period \Rightarrow \bigwedge_{x \in RO_i} (\lceil \neg T_i.read(x) \rceil; \textbf{true})) \tag{3}$$

For any transaction $T_i$, at any time, either $T_i.arrd$ or $T_i.comtd$ or $T_i.abd$ (here we assume that at the beginning, if a transaction has not arrived, it is committed)

$$\Box\lceil T_i.arrd \vee T_i.comtd \vee T_i.abd \rceil^* \tag{4}$$

These three states are mutually exclusive:

$$\Box\lceil T_i.arrd \rceil \ \Rightarrow\ \lceil \neg(T_i.comtd \vee T_i.abd) \rceil \tag{5}$$

$$\Box\lceil T_i.comtd \rceil \ \Rightarrow\ \lceil \neg(T_i.arrd \vee T_i.abd) \rceil \tag{6}$$

$$\Box\lceil T_i.abd \rceil \ \Rightarrow\ \lceil \neg(T_i.arrd \vee T_i.comtd) \rceil \tag{7}$$

At any time the value of a data object is given by one and only one transaction (here we assume that there is a virtual transaction to write the initial value for all data):

$$\Box\lceil \bigvee_{T_i \in \mathcal{T}} T_i.wrtn(x) \rceil^* \tag{8}$$

$$\Box \bigwedge_{T_i \neq T_j \in \mathcal{T}} \lceil \neg(T_j.wrtn(x) \wedge T_i.wrtn(x)) \rceil^* \tag{9}$$

A transaction $T_i$ requests a lock for a data object $x$ iff it is in state "arrived" and it is either holding it or waiting for it:

$$\Box\lceil\!\lceil \begin{array}{l} T_i.req\_rlk(x) \iff (T_i.arrd \wedge \\ (T_i.hld\_rlk(x) \vee T_i.wt\_rlk(x))) \end{array} \rceil\!\rceil^* \quad (10)$$

$$\Box\lceil\!\lceil \begin{array}{l} T_i.req\_wlk(x) \iff (T_i.arrd \wedge \\ (T_i.hld\_wlk(x) \vee T_i.wt\_wlk(x))) \end{array} \rceil\!\rceil^* \quad (11)$$

A transaction cannot hold a lock and at the same time wait for it:

$$\Box\lceil\!\lceil \neg(T_i.hld\_rlk(x) \wedge T_i.wt\_rlk(x)) \rceil\!\rceil^* \quad (12)$$

$$\Box\lceil\!\lceil \neg(T_i.hld\_wlk(x) \wedge T_i.wt\_wlk(x)) \rceil\!\rceil^* \quad (13)$$

The conflicting locks cannot be shared by transactions. Therefore, for $T_i \neq T_j$

$$\Box\lceil\!\lceil \neg(T_i.hld\_rlk(x) \wedge T_j.hld\_wlk(x)) \rceil\!\rceil^* \quad (14)$$

$$\Box\lceil\!\lceil \neg(T_i.hld\_wlk(x) \wedge T_j.hld\_lk(x)) \rceil\!\rceil^* \quad (15)$$

A transaction can read or write on a data object $x$ only if it holds the corresponding lock on the data object $x$ at the time:

$$\Box\lceil\!\lceil \neg T_i.read(x) \rceil\!\rceil; \lceil\!\lceil T_i.read(x) \rceil\!\rceil \Rightarrow \Diamond\lceil\!\lceil T_i.hld\_rlk(x) \rceil\!\rceil \quad (16)$$

$$\Box\lceil\!\lceil \neg T_i.wrtn(x) \rceil\!\rceil; \lceil\!\lceil T_i.wrtn(x) \rceil\!\rceil \Rightarrow \Diamond\lceil\!\lceil T_i.hld\_wlk(x) \rceil\!\rceil \quad (17)$$

In any period, a transaction $T_i$ cannot hold a lock for a data objects $x$ after it has released this lock:

$$T_i.period \Rightarrow \neg\Diamond(\lceil\!\lceil T_i.hld\_rlk(x) \rceil\!\rceil; \lceil\!\lceil \neg T_i.hld\_rlk(x) \rceil\!\rceil; \lceil\!\lceil T_i.hld\_rlk(x) \rceil\!\rceil) \quad (18)$$

$$T_i.period \Rightarrow \neg\Diamond(\lceil\!\lceil T_i.hld\_wlk(x) \rceil\!\rceil; \lceil\!\lceil \neg T_i.hld\_wlk(x) \rceil\!\rceil; \lceil\!\lceil T_i.hld\_wlk(x) \rceil\!\rceil) \quad (19)$$

As mentioned earlier, for each period, for all $i$ and $x$ the state $T_i.read(x)$, $T_i.comtd$ and $T_i.abd$ can change at most once.

$$T_i.period \Rightarrow \Box(\lceil\!\lceil T_i.read(x) \rceil\!\rceil; \textbf{true} \Rightarrow \lceil\!\lceil T_i.read(x) \rceil\!\rceil) \quad (20)$$

$$T_i.period \Rightarrow \Box(\lceil\!\lceil T_i.comtd \rceil\!\rceil; \textbf{true} \Rightarrow \lceil\!\lceil T_i.comtd \rceil\!\rceil) \quad (21)$$

$$T_i.period \Rightarrow \Box(\lceil\!\lceil T_i.abd \rceil\!\rceil; \textbf{true} \Rightarrow \lceil\!\lceil T_i.abd \rceil\!\rceil) \quad (22)$$

From the assumption of atomic commitment it follows that if a transaction has written something into the database then it should commit at the end.

$$T_i.period \Rightarrow ((\Diamond\lceil\!\lceil T_i.wrtn(x) \rceil\!\rceil) \Rightarrow \textbf{true}; \lceil\!\lceil T_i.comtd \rceil\!\rceil) \quad (23)$$

Let *ENV* be the set of the formulas $(1), (2), (4), \ldots, (24)$ with $x \in \mathcal{O}$, $i \neq j$ and $i, j \leq n$.

Let state variable $T_i.run$ be defined as $T_i.run(t) = 1$ iff transaction $T_i$ is running on a processor at time $t$. When a transaction $T_i$ has arrived and got all data object locks it needs, it is ready to run on the processor. $T_i.ready(t) = 1$ iff transaction $T_i$ is ready to execute on a processor at time $t$. When a transaction $T_i$ is ready, it must not wait for a read-lock or a write-lock: $\lceil\!\lceil T_i.ready \rceil\!\rceil \Rightarrow \lceil\!\lceil \neg T_i.wt\_rlk(x) \rceil\!\rceil$ and $\lceil\!\lceil T_i.ready \rceil\!\rceil \Rightarrow \lceil\!\lceil \neg T_i.wt\_wlk(x) \rceil\!\rceil$. A transaction runs only if it is ready. Therefore for every transaction $T_i$ (A1):

$$\Box(\lceil\!\lceil T_i.run \rceil\!\rceil \Rightarrow \lceil\!\lceil T_i.ready \rceil\!\rceil)$$

In a period if a transaction is standing, the maximal required execution time has not been reached. Hence, (A2):

$$\begin{array}{l} T_i.period \Rightarrow \\ \quad (\textbf{true}; \lceil\!\lceil \neg T_i.comtd \rceil\!\rceil; \textbf{true} \Rightarrow \\ \quad (\int Ti.Run < C_i); \lceil\!\lceil \neg T_i.comtd \rceil\!\rceil; \textbf{true}) \end{array}$$

In a period if execution time of $T_i$ is equal to $C_i$, $T_i$ will commit from that time (A3):

$$\begin{array}{l} T_i.period \Rightarrow \\ \quad (\int T_i.run = C_i; \ell > 0 \Rightarrow \textbf{true}; \lceil\!\lceil T_i.comtd \rceil\!\rceil)) \end{array}$$

Assume that the transactions $T_1, \ldots, T_n$ share a single processor, and transaction priorities are assigned by the Rate Monotonic Algorithm, which assigns a higher priority to a transaction with shorter period. Without the loss of the generality, we assume $P_1 \leq P_2 \leq \cdots \leq P_n$, and priorities are in decreasing order from $p_1$ to $p_n$, where $p_i$ denote the priority of $T_i$.

At any time if one transaction is running, any other transaction cannot be running: For $i \neq j$

$$(A4) \quad \Box\lceil\!\lceil \neg(T_i.run \wedge T_j.run) \rceil\!\rceil^*$$

The processor cannot stay idle when a transaction is ready:

$$(A5) \quad \Box(\lceil\!\lceil T_i.ready \rceil\!\rceil \Rightarrow \lceil\!\lceil \bigvee_{1 \leq j \leq n} T_j.run \rceil\!\rceil)$$

A transaction with lower priority cannot be running when a transaction with higher priority is ready:

$$(A6) \quad \Box(\lceil\!\lceil T_i.ready \rceil\!\rceil \Rightarrow \bigwedge_{i < j \leq n} \lceil\!\lceil \neg T_j.run \rceil\!\rceil)$$

The conjunction of the preceding formulas (A1)–(A6) captures our uniprocessor model for the transactions.

## 3.2 Correctness Criteria of Concurrent Execution of Transaction Systems

### 3.2.1 Serializability

The serializability of an execution of the transaction system says that the relation 'before' between the executions of transactions defined by the order of the conflict operations in the execution is a partial ordering on the (infinite) set of

transaction executions. Given an execution of the transaction system, any transaction has its own period, and in each period, there is one execution of the transaction. Hence, the set of all the executions of transactions is infinite. We have to to describe a criterion for the infinite relation 'before' to be acyclic by just a formula.

There is a nice characterisation for the relation 'before' of the transaction system to be acyclic which is about the behaviour of transaction system in an interval with the length $(n+1)*P_n$ only. Namely, we prove that the relation 'before' of a system execution is acyclic if and only if for any interval which consists of exactly $(n+1)$ consecutive periods of $T_n$, its restriction on the transaction executions in this interval is acyclic. The 'only if' part of the statement is trivial. The proof of the 'if' part is by contradiction. For contradiction, assume that there is a cycle $C_1, C_2, \ldots, C_k$ in the relation 'before' over the set of transaction executions, $C_j$ is 'before' $C_{j+1}$ for $j < k$, and $C_k$ is 'before' $C_1$, where $C_j$ is an execution of a transaction $T_{i_j}$ in a period $P$. Let $I_j$ be interval of time for that period $P$. Let the cycle $C_1, C_2, \ldots, C_k$ be 'smallest' (meaning that it does not include another cycle). By the definition of the relation 'before', for each $j < k$, there is a data object $x_j$ such that $C_j$ performs an operation $c_j$ on $x_j$ before $C_{j+1}$ performs an operation on $x_j$ that is in conflict to $c_j$. That means there is a time point in $I_j$ that is earlier than a time point in $I_{j+1}$. Similarly, there is a time point in $I_k$ that is earlier than a time point in $I_1$. Since $C_1, C_2, \ldots, C_k$ is a smallest, there is not more than one execution of the same transaction in the cycle (this is because that all executions of the same transaction access the same set of data objects and performs the same set of operations). Hence, $k \leq n$. Consequently, the union of the time intervals of the executions of this cycle, $\cup_{j=1}^{k} I_j$ should be included in the time interval with the length $n * P_n$. Hence, this interval should be included in $n+1$ consecutive periods of transaction $T_n$. This is a contradiction to the fact that for any interval which consists of exactly $(n+1)$ consecutive periods of $T_n$, the restriction on the relation 'before' on the transaction executions in this interval is acyclic.

The relation 'before' between the executions of different transactions are modelled as follows. Let $a = (n+1)P_n)$. The order between conflict operations on data object $x$ in an interval with the length less than $a$ is captured by

$\mathit{WR}_{ij}(x) \widehat{=} (\Diamond(\lceil T_i.wrtn(x) \rceil \wedge \lceil \neg T_j.read(x) \rceil);$
$\ell < a; \lceil T_j.read(x) \rceil),$
$\mathit{RW}_{ij}(x) \widehat{=} (\Diamond(\lceil T_i.read(x) \rceil \wedge \lceil \neg T_j.wrtn(x) \rceil);$
$\ell < a; \lceil T_j.wrtn(x) \rceil),$
$\mathit{WW}_{ij}(x) \widehat{=} (\Diamond \lceil T_i.wrtn(x) \rceil; \ell < a; \lceil T_j.wrtn(x) \rceil).$

To express that the relation 'before' defined as above does not have a cycle longer than $n$, we first find an expression for its transitive closure. This is expresses by the following DC formula $\mathcal{C}_{ij}^n$ defined as:

$$\begin{aligned} \mathcal{C}_{ij}^1 &\widehat{=} (\mathit{RW}_{ij} \vee \mathit{WR}_{ij} \vee \mathit{WW}_{ij}) \\ \mathcal{C}_{ij}^n &\widehat{=} (\mathcal{C}_{ij}^{n-1} \vee (\mathcal{C}_{ir}^{n-1} \wedge \mathcal{C}_{rj}^{n-1})) \end{aligned}$$

**Serializability Criterion**

A concurrent execution of the set of transactions $\mathcal{T}$ is serializable iff it satisfies $(T_n.period; \ell = n * P_n) \Rightarrow \bigwedge_{i,j \leq n, i \neq j} \neg(\mathcal{C}_{ij}^n \wedge \mathcal{C}_{ji}^n)$ in any interval.

### 3.2.2 Temporal Consistency Criteria

In a RTDB, there are two kinds of data objects: continuous data objects and discrete data objects. Let $\mathcal{O}$ denote the set of all data objects in a RTDBS.

Continuous data objects are related to external objects continuously changing with time. The value of a continuous data object is obtained directly from a sensor or is computed from the values of a set of other data objects.

Discrete data objects are static in the sense that their values do not become obsolete as time passes. Let the set of discrete data objects be $Z$.

At each moment of time a continuous data object has a value represented by its current version which is valid for some time interval. Note that at the same moment of time there may be several versions of the same continuous object in database that are valid.

Let $\alpha$ be a continuous data object. For each $q \in N$ there is a state variable $\mathit{validity}_q(\alpha)$ to reflect the validity of $q$'th version for the value of $\alpha$ and a real state variable $value_q(\alpha)$ to reflect the value of $\alpha$ at time $t$ is the $q$'th version. $\mathit{validity}_q(\alpha)$ holds at time $t$ iff $q$'th version of a continuous data object $\alpha$ has been created (before time $t$) and is still valid at time $t$. For the simplicity of representation, we assume that discrete data only have 0th version ($q = 0$) with the validity interval $[0, +\infty)$.

$\mathit{validity}_q(\alpha)(t) = 1$ iff $t$ is in the valid interval of the $q$'th version of $\alpha$, $value_q(\alpha)(t) = 1$ iff the value of $\alpha$ at $t$ is the $q$'th version of $\alpha$.

There is a positive lower bound $\delta'$ for the valid interval (depending on the sampling periods), and each version may have only a single interval of validity. For a version $q$ of the data object $\alpha$, there is a predefined number $avi_q(\alpha)$ which is the maximal length of its validity interval. Namely, version $q$ of $\alpha$ is valid for $avi_q(\alpha)$ ($\geq \delta'$) time units since the time it was created. Therefore,

$$\lceil \neg \mathit{validity}_q(\alpha) \rceil; \lceil \mathit{validity}_q(\alpha) \rceil; \lceil \neg \mathit{validity}_q(\alpha) \rceil \Rightarrow \ell \geq avi_q(\alpha) \quad (24)$$

$$\lceil \mathit{validity}_q(\alpha) \rceil \Rightarrow \ell \leq avi_q(\alpha)) \quad (25)$$

$$\lceil \mathit{validity}_q(\alpha) \rceil; \mathbf{true} \Rightarrow \lceil \mathit{validity}_q(\alpha) \rceil \vee \lceil \mathit{validity}_q(\alpha) \rceil; \lceil \neg \mathit{validity}_q(\alpha) \rceil \quad (26)$$

The absolute temporal consistency at a time $t$ of a data object $\alpha$ means that there is a version $q$ of $\alpha$ which was born at time $t_{(\alpha,q)}$ that is still valid, i.e. $t - t_{(\alpha,q)} \leq avi_q(\alpha)$. The absolute temporal consistency of the data in a RTDBS means that all data objects satisfy the absolute temporal consistency at any time. Since we have assumed that at any time, there should be a version $q$ for a data object (normally, the version that was created most recently) for

which $value_q$ holds, the absolute temporal consistency is formalised simply as follows.

**Absolute Temporal Consistency Criterion** *ACONS*$(\alpha, q)$ is defined as

$\Box(\lceil value_q(\alpha)\rceil \Rightarrow \lceil$*validity*$_q(\alpha)\rceil)$

Relative consistency says that data objects from some data set should be temporally correlated. Any set $R$ of versions of continuous data objects, i.e. $R$ is a set of pairs $(\alpha, q)$, is associated with a number called length of *relative validity interval* denoted by *rvi*$(R)$.

The relative consistency of a set $R$ of versions is expressed by the following DC formula *RCONS*$(R)$, meaning that $R$ is relatively consistent iff DC formula *RCONS*$(R)$ is true for all intervals, where *RCONS*$(R)$ is defined as: For any $(\alpha_1, q), (\alpha_2, r) \in R$

$$\Box\left(\begin{array}{l}\lceil \textit{validity}_q(\alpha_1) \wedge \neg \textit{validity}_r(\alpha_2)\rceil; \\ \lceil \neg \textit{validity}_r(\alpha_2)\rceil^*; \lceil \textit{validity}_r(\alpha_2)\rceil \ \Rightarrow \\ (\ell \leq \textit{rvi}(R)); \lceil \textit{validity}_r(\alpha_2)\rceil\end{array}\right)$$

As we have said earlier, for any data object $\alpha$, at any time $t$, there is a version $q$ for which $value_q(\alpha)$ is true. Normally, when a transaction reads $\alpha$ at time $t$, it will get the version $q$ for which $value_q(\alpha)$. However, in some scheduler, they may give a different valid version. In order to be more general, we introduce the step function $T_i.readv$ to return the version number read by $T_i$ for a value of data object. $T_i.readv(\alpha)(t) = q$ iff at time $t$ transaction $T_i$ has performed a read operation on $q$'th version most recently of data object $(\alpha)$.

A transaction $T_i$ can read a set of versions of data objects in an interval. Therefore, for each $i \leq n$ we introduce a temporal variables $R\alpha_i$ to express the set of versions of data objects read by $T_i$ in an interval.

An execution of the transaction system is temporally correct iff each transaction, in each period of its, meets the deadline, reads the set of temporally valid (i.e., recent enough) data objects and is committed before any of them becomes invalid, and that all the data read by a transaction in a period are relatively temporally consistent. These conditions are specified by the DC formulas $DL_i, ATC_i, RTC_i$ as follows.

$$DL_i \widehat{=} \left(\begin{array}{l}\lceil T_i.period\rceil \Rightarrow \\ ((\Box(\lceil T_i.arrd\rceil \Rightarrow \ell \leq D_i)) \wedge \\ \int T_i.run = C_i)\end{array}\right).$$

Let

$$RDVLD \widehat{=} \left(\begin{array}{l}\bigwedge_{(\alpha)\in RO_i, q\neq 0} \lceil T_i.read(\alpha)\rceil \wedge \\ \lceil T_i.readv(\alpha) = q\rceil \Rightarrow \lceil \textit{validity}_q(\alpha)\rceil\end{array}\right).$$

Then,

$$ATC_i \widehat{=} (\lceil T_i.period\rceil \Rightarrow \Box(\lceil T_i.arrd\rceil \Rightarrow RDVLD)).$$

$$RTC_i \widehat{=} \Diamond\left(\begin{array}{l}\lceil T_i.period\rceil \wedge \\ ((\alpha, q) \in R\alpha_i \iff q \neq 0 \\ \wedge \Diamond(\lceil T_i.read(\alpha)\rceil \wedge \\ \lceil T_i.readv(\alpha) = q\rceil))\end{array}\right) \Rightarrow RCONS(R\alpha_i).$$

Let $CM \widehat{=} \bigwedge_{i\leq n} DL_i \wedge ATC_i \wedge RTC_i$.

**Correctness criterion for the execution of transactions in RTDBS:** an execution of set $\mathcal{T}$ of transactions is correct iff for any interval it satisfies the formula $SERIAL \wedge CM$.

## 4 Formalisation of Read/Write Priority Ceiling Protocol in RTDB

**Serializability of 2PL** We adapt the formalisation in [5] for the iterated transaction systems. In 2PL, a transaction execution consists of two phases. In the first phase data object locks are acquired, while in the second phase the data object locks are released and new locks can not be acquired. For each transaction $T_i$ we introduce a state variable $T_i.obphase$ to express which phase the transaction $T_i$ is in at a time. $T_i.obphase(t) = 1$ iff transaction $T_i$ is in the obtaining phase at time $t$.

Then, 2PL is formalised by the following DC formulas.

$$T_i.period \Rightarrow (\lceil T_i.obphase\rceil; \lceil \neg T_i.obphase\rceil) \quad (27)$$

$$T_i.period \Rightarrow \Box(\lceil T_i.obphase\rceil \Rightarrow \lceil \neg T_i.comtd\rceil) \quad (28)$$

$$T_i.period \Rightarrow \Box\left(\begin{array}{l}\lceil \neg T_i.hld\_rlk(x)\rceil; \lceil T_i.hld\_rlk(x)\rceil \\ \Rightarrow \lceil \neg T_i.hld\_rlk(x)\rceil; \\ \lceil T_i.obphase\rceil; \textbf{true}\end{array}\right) \quad (29)$$

$$T_i.period \Rightarrow \Box\left(\begin{array}{l}\lceil \neg T_i.hld\_wlk(x)\rceil; \lceil T_i.hld\_wlk(x)\rceil \\ \Rightarrow \lceil \neg T_i.hld\_wlk(x)\rceil; \\ \lceil T_i.obphase\rceil; \textbf{true}\end{array}\right) \quad (30)$$

$$T_i.period \Rightarrow \Box\left(\begin{array}{l}\lceil T_i.hld\_rlk(x)\rceil; \lceil \neg T_i.hld\_rlk(x)\rceil \\ \Rightarrow \lceil T_i.hld\_rlk(x)\rceil; \lceil \neg T_i.obphase\rceil\end{array}\right) \quad (31)$$

$$T_i.period \Rightarrow \Box\left(\begin{array}{l}\lceil T_i.hld\_wlk(x)\rceil; \lceil \neg T_i.hld\_wlk(x)\rceil \\ \Rightarrow \lceil T_i.hld\_wlk(x)\rceil; \lceil \neg T_i.obphase\rceil\end{array}\right) \quad (32)$$

Let $2PLC$ be the set of the DC formulas $(27), \ldots, (32)$ and *2PL* $\widehat{=} \bigwedge_{\varphi\in 2PLC} \Box\varphi$

**Theorem 1** *$SERIAL$ is provable from $ENV$ and $2PLC$.*

**Formalisation of R/WPCP** Let $WPL(x)$ and $APL(x)$ be constants denoting the write priority ceiling and the absolute priority ceiling for a data object $x$, let $PN = \{p_1, \ldots, p_n\}$ denote the set of integers for expressing the priority of transactions, and let $T_i.locked\_data$ and $T_i.sysceil$ be object-set and real state variables (defined below). The write priority ceiling $WPL(x)$ of data object $x$ is defined as the highest priority of transactions which may write $x$. $WPL(x) \ \widehat{=} \ \max\{p_j \mid x \in WO_j$ and $j \leq n\}$. The absolute priority ceiling $APL(x)$ of data object $x$ is defined as the highest priority of transactions which may

read or write $x$,
$APL(x) \widehat{=} \max\{\, p_j \mid x \in RO_j \cup WO_j \text{ and } j \leq n\}$

To express the behaviour of the R/WPCP, we introduce a real state variable $RWPL(x)$, called read write priority ceiling.

When data object $x$ is read-locked (write-locked), the read write priority ceiling $RWPL(x)$ is equal to $WPL(x)$ ($APL(x)$): $RWPL(x)(t) = 0$ if at time $t$ object $x$ is neither read-locked nor write-locked by any transaction, $RWPL(x)(t) = WPL(x)$ if at time $t$ object $x$ is read-locked (by some transaction) and is not write-locked, and $RWPL(x)(t) = APL(x)$ if at time $t$ object $x$ is write-locked (by some transaction).

$T_i.locked\_data$ is a function denoting the set of data objects locked by transactions other than $T_i$ at a time: $T_i.locked\_data \in [\mathit{Time} \rightarrow 2^{\mathcal{O}}]$, $T_i.locked\_data(t) = \{x \mid \text{ there is } j \text{ such that } T_j.hld\_lk(x)(t) \text{ and } i \neq j\}$.

$T_i.sysceil$ denotes the highest r/w priority ceiling of data objects locked by transactions other than $T_i$ at a time. $T_i.sysceil(t) = \max\{\, RWPL(x)(t) \mid x \in T_i.locked\_data(t)\}$.

When a transaction $T_i$ attempts to lock a data object $x$ at time $t$, $T_i$ will be blocked and the lock on an object $x$ will be denied, if the priority of transaction $T_i$ is not higher than $T_i.sysceil(t)$. In this case, we say $T_i$ is blocked by a transaction $T_j$ which holds a lock on $x$. We introduce a state expression $T_i.\mathit{blockedby}(T_j)$ for $T_i \neq T_j$ to express this fact:

$$\begin{array}{l}
\bigvee_{x\in\mathcal{O}}(T_j.hld_rlk(x) \wedge T_i.wt_rlk(x) \wedge \\
\quad T_i.sysceil \geq p_i) \\
\vee \bigvee_{x\in\mathcal{O}}(T_j.hld_rlk(x) \wedge T_i.wt_wlk(x) \wedge \\
\quad T_i.sysceil \geq p_i) \\
\vee \bigvee_{x\in\mathcal{O}}(T_j.hld_wlk(x) \wedge T_i.wt_rlk(x) \wedge \\
\quad T_i.sysceil \geq p_i) \\
\vee \bigvee_{x\in\mathcal{O}}(T_j.hld_wlk(x) \wedge T_i.wt_wlk(x) \wedge \\
\quad T_i.sysceil \geq p_i)
\end{array}$$

Let $\mathit{HiPri}(T_i, T_j)$ be a boolean-valued function of time (a state variable) to express which transaction between $T_i$ and $T_j$ has higher priority.

(a) As a priory, *HiPri* is a partial order:

$$\bigwedge_{T_i\neq T_j\in\mathcal{T}}(\mathit{HiPri}(T_i,T_j) \;\Rightarrow\; \neg\mathit{HiPri}(T_j,T_i))$$

$$\bigwedge_{T_i\neq T_j\neq T_k\in\mathcal{T}} \left(\begin{array}{l} (\mathit{HiPri}(T_i,T_k) \wedge \\ \mathit{HiPri}(T_k,T_j)) \Rightarrow \\ \mathit{HiPri}(T_i,T_j) \end{array}\right)$$

(b) $\mathit{HiPri}(T_i, T_j)$ has to capture the inheritance of priorities by the R/WPCP:

$$\bigwedge_{T_i\neq T_j\neq T_k\in\mathcal{T}} \left(\begin{array}{l} T_k.\mathit{blockedby}(T_i) \Rightarrow \\ (\mathit{HiPri}(T_k,T_j) \Rightarrow \\ \mathit{HiPri}(T_i,T_j)) \end{array}\right)$$

$$\bigwedge_{T_i\neq T_j\in\mathcal{T}} \left(\begin{array}{l} \bigwedge_{T_k\in\mathcal{T}}(\neg T_k.\mathit{blockedby}(T_i)) \Rightarrow \\ (\mathit{HiPri}(T_i,T_j) \Rightarrow p_i > p_j) \end{array}\right)$$

R/WPCP always allows the transaction with 'the highest priority' among the ready transactions to run (*PPS*): For $T_i \neq T_j$:
$\Box(\lceil T_i.run\rceil \wedge \lceil T_j.ready\rceil \Rightarrow \lceil \mathit{HiPri}(T_i,T_j)\rceil)$

In R/WPCP, when a transaction requests a lock, it is granted iff its priority is higher than the system ceiling for it. This is formalised by (*GrR*):

$$\Box(\lceil\neg T_i.hld\_rlk(x)\rceil; \lceil T_i.hld\_rlk(x)\rceil \Rightarrow \Diamond\lceil p_i > T_i.sysceil\rceil)$$

and (*GrW*)

$$\Box(\lceil\neg T_i.hld\_wlk(x)\rceil; \lceil T_i.hld\_wlk(x)\rceil \Rightarrow \Diamond\lceil p_i > T_i.sysceil\rceil)$$

When a lock is available for which some transactions are waiting, it will be granted to some of them. The one who gets should have the highest priority. These unblocking rules can be specified as (*UnBl1*):

$$\Box((\textstyle\bigwedge_{T_i\in\mathcal{T}}\lceil\neg T_i.hld\_lk(x)\rceil) \Rightarrow (\textstyle\bigwedge_{T_i\in\mathcal{T}}\lceil\neg T_i.wt\_lk(x)\rceil))$$

and (*UnBl2*):

$$\Box\left(\begin{array}{l} \lceil T_i.wt\_lk(x) \wedge T_j.wt\_lk(x)\rceil; \\ \lceil\neg T_i.wt\_lk(x)\rceil \Rightarrow \mathit{HiPri}_{R/WPCP}(T_i,T_j) \end{array}\right)$$

($x \in \mathcal{O}$, $i \neq j$).

$R/WPCP$ is now obtained as the conjunction *2PL* $\wedge$ *PPS* $\wedge$ *GrR* $\wedge$ *GrW* $\wedge$ *UnBl1* $\wedge$ *UnBl2*.

A number of properties of $R/WPCP$ including the correctness has been verified formally in DC. We refer the readers to [7] for details.

## References


[1] Azer Bestavros, Kwei-Jay Lin and Sang Hyuk Son. *Real-Time Database Systems: Issues and Applications*. Kluwer Academic Publishers, 1997.

[2] Philip Chan and Dang Van Hung. Duration Calculus Specification of Scheduling for Tasks with Shared Resources. LNCS 1023, Springer-Verlag 1995, pp. 365-380

[3] M.R. Hansen and Zhou Chaochen. Duration Calculus: Logical Foundations. *Formal Aspects of Computing*, 1997, 9:283-330.

[4] Kam-Yiu Lam and Tei-Wei Kuo. *Real-Time Database Systems: Architecture and Techniques*. Kluwer Academic Publishers, 2001.

[5] Ekaterina Pavlova and Dang Van Hung. A Formal Specification of the Concurrency Control in Real Time Database. The proceedins of APSEC'99, IEEE Computer Society Press 1999, pp. 94-101.

[6] Chaochen Zhou, C.A.R. Hoare, and A.P. Ravn. A Calculus of Durations. *Information Processing Letters*, 1991, 40(5):269–276.

[7] Ho Van Huong and Dang Van Hung. Modelling Real-time Database Systems in Duration Calculus. Technical Report 260, UNU-IIST, P.O. Box 3058, Macau, September 2002.