A FORMAL SPECIFICATION OF THE ABORT-ORIENTED
CONCURRENCY CONTROL FOR REAL TIME DATABASES
IN DURATION CALCULUS

HO VAN HUONG

Governmental Cipher Department HaNoi

Abstract. In this paper, we present a formal model of real time database systems using duration
calculus (DC). First, we present a formal description of the real time database model using state
variables expressing data objects and operations of period transactions. Then, we give DC formulas
to express their behavior and relationships. We also give a formal specification of the Basic Aborting
Protocol (BAP) and a formal proof for the correctness of the BAP using the DC proof system. And
then, we propose an extension of BAP.

Tóm tắt. Bài báo trình bày về một mô hình hình thức của hệ thống cơ sở dữ liệu sử dụng logic
tính toán khoa học. Phân đầu giải quyết là hệ thống của mô hình cơ sở dữ liệu thời gian thực, sử
dụng các biến trạng thái thể hiện các đối tượng dữ liệu và các thao tác của các giao tác cơ chủ yếu.
Tiếp nhận là đưa ra công thức DC (Duration Calculus) để thể hiện hành vi và quan hệ của chúng. Bài
báo còn đưa ra một đặc tả hình thức của giao thức hủy bỏ (BAP) và một chứng minh hình thức
cho điều kiện dùng của giao thức BAP sử dụng hệ thống chứng minh DC. Cuối cùng là đề xuất một
thuật toán để mô ra thông cho giao thức BAP.

1. INTRODUCTION

In the past two decades, the research in RTDBS has received a lot of attention [5, 12]. It
consists of two different important areas in computer science: real time systems and database
systems. Similar to conventional real time systems, transactions in RTDBS are usually as-
associated with time constraint, e.g., deadline. On the other hand, RTDBS must maintain a
database for useful information, support the manipulation of database, and process trans-
actions [12]. RTDBS are used in a wide range of applications such as avionic and space,
air traffic control systems, robotics, nuclear power plants, integrated manufacturing systems,
programmed stock trading systems, and network management systems.

In this paper, we concentrate on mathematical modelling of RTDBS such as the time
behaviour of the data, the integration of concurrency control with scheduling in RTDBS. We
will use real time logic for our modelling.

The main goal of this paper is to formalise some aspects of RTDBS, in particular BAP
using DC. This will allow us to verify the correctness of BAP formally using the proof system
of the DC. We also propose an extension of BAP. We make use of duration calculus because
DC is a simple and powerful logic for reasoning about real time systems, and DC has been used
successfully in many case studies, for example [6, 7, 8, 9], we will take it to be the formalism
for our specification in this paper.

Our approach is summarised as follows: We apply a formal model of RTDBS proposed
by Ho Van Huong and Dang Van Hung [9] to specify and verify the Basic Aborting Protocol.
To show the advantages of our model, we give a formal specification of the Basic Aborting
Protocol (BAP) and a formal proof for the correctness of the BAP using the DC proof system.

The paper is organized as follows. In the next section, we give an informal abstract
description of RTDBS and BAP. Section 3 presents a review of DC. Section 4 presents a
formal model of Real Time Database Systems in DC. Section 5 presents a formalization
of BAP in DC and a formal proof of correctness and certain properties of this protocol. Section
6 presents an extension of BAP.

2. PRELIMINARIES

We briefly recall in this section the main concepts of RTDBS and the integration of
concurrency control with priority scheduling, which will justify our formal model given in
later sections. We refer to [5, 9, 12] for more comprehensive introduction to RTDBS.

A real time database systems can be viewed as an amalgamation of conventional database
management system and real time system [5]. In RTDB, the transactions not only have
to meet their deadline, but also have to use the data that are valid during their execution.
Many previous studies have focused on integrating concurrency control protocols with priority
scheduling in RTDBS [5, 12].

For example, the Read/Write Priority Ceiling Protocol (R/WPCP) is an extension of the
well-known Priority Ceiling Protocol (PCP) [12] in real time concurrency control, adopts Two
Phase Locking (2PL) in preserving the serializability of transactions executions.

Although R/WPCP and its variants provide ways to bound and estimate the worst case
blocking time of a transaction, they are usually pretty conservative, and it is often unavoidable
to avoid lengthy blocking time for a transaction in many systems. Transaction aborting is
suggested by many researchers to solve problems due to lengthy blocking time. In particular,
Tei-Wei Kuo, et. al. [11] proposed a Basic Aborting Protocol (BAP). The Basic Aborting
Protocol is an integration of the Two Phase Locking Protocol, Priority Ceiling Protocol, and
a simple aborting algorithm. The basic idea of the Basic Aborting Protocol is that, when a
transaction $T_i$ attempts to lock a data object $x$, the lock request will be granted if the priority
of $T_i$ is higher than the priority ceiling of all data objects currently locked by transaction
other than $T_i$, otherwise, a rechecking procedure for the lock request is done as follows: if all
of the transactions other than $T_i$ that locked data objects with priority ceilings higher than
the priority of $T_i$ are abortable, then $T_i$ may abort all of the transactions that lock such data
objects and obtain the new lock. Otherwise, $T_i$ will be blocked. Aborted transaction are
assumed to restart immediately after their abortings. Since BAP consists of 2PL, PCP, and
a simple aborting algorithm, BAP does preserve many important properties of 2PL and PCP
such as serializable, guarantees deadlock-free and blocking at most one for every transaction.

3. DURATION CALCULUS

The Duration Calculus (DC) represents a logical approach to formal design of real time
systems. DC is proposed by Zhou, Hoare, and Ravn, which is an extension of real arithmetic
and interval temporal logic. We refer to [10] for more comprehensive introduction to Duration
Calculus.

$Time$ 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 $t'$.

We assume 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
formed by the following rules:

1. Each state variable $P \in E$ is a state expression.
2. If $P$ and $Q$ are state expressions, then so are $\neg P$, $(P \land Q)$, $(P \lor Q)$, $(P \Rightarrow Q)$, $(P \Leftrightarrow Q)$.

A state variable $P$ is interpreted as a function $I(P) : R^+ \rightarrow \{0, 1\}$ (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 finite variability in a finite time interval. A state expression is interpreted as a function which is defined by the interpretations for the state variables and Boolean operators.

For an arbitrary state expression $P$, its duration is denoted by $\int P$. Given an interpretation $I$ of state variables and an interval, duration $\int P$ is interpreted as the accumulated length of time within the interval at which $P$ is present. So for an arbitrary interval $[t, t']$, the interpretation $I(\int P)([t, t'])$ is defined as $\int_t^{t'} I(P)(t) \, dt$. Therefore, $\int 1$ 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 assume a set of temporal propositional letter $X, Y, \ldots$. 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 $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$, the chop $\cd$.

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. This is written as

$$I, [t', t''] \models D,$$

where $I$ assigns every state variable a finitely variable function from $R^+$ to $\{0, 1\}$, and $[t', t'']$ decides the observation window.

Given an interpretation $I$, the chop-formula $D_1 \cd D_2$ 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 $P$, $\|P\|$ stands for $(\int P = \ell) \wedge (\ell > 0)$. This means that interval is a non-point interval and $P$ holds almost everywhere in it. We use $\| \|$ to denote the predicate which is true only for point intervals.

Modalities $\Diamond, \Box$ are defined as: $\Diamond D \equiv \text{true} \cd \neg D \cd \text{true}, \Box D \equiv \neg \Diamond \neg D$ (we use $\equiv$ as a define). This means that $\Diamond D$ is true for an interval iff $D$ holds for some its subinterval, and $\Box D$ is true for an interval iff $D$ holds for every its subintervals.

In this paper, we will use the following abbreviation as well.

$$\|P\|^* \equiv \|P\| \lor \|P\|$$

DC with abstract duration domain is a complete calculus, which has a powerful proof system.

4. A FORMAL MODEL OF REAL TIME DATABASE SYSTEMS IN DC

4.1. Basic model

We now give a formal model of Real Time Database System (RTDBS) using DC. We will first introduce DC state variables to model the basic primitives of RTDB and to characterise the data and the transactions. We then write DC formulas on the introduced state variables to capture the essential properties of the RTDBS.

The system consists of a set $\mathcal{O}$ of data objects ranged over by $x, y, z, \ldots$, and set $T$ of $n$ transaction $T_i, 1 \leq i \leq n$.

Each transaction $T_i$ arrives at the database system at time $\lambda_i$ which is unknown in advance. After arriving 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 but well accepted in the literature. Each transaction \( T_i \) has its own deadline \( D_i \), a priority \( p_i \), an execution time \( C_i \), a period \( P_i \), a data read set \( RO_i \), a data write set \( WO_i \) (note that \( RO_i \) and \( WO_i \) may be empty).

Now we introduce DC state variables to model the behaviour of data objects and transactions. Let \( x \) be a data object. For each \( i \leq n \) a state variable let \( T_i.written(x) \) be a DC state variable expressing the behaviour of \( x \). \( T_i.written(x) \) holds at time \( t \) iff the value of \( x \) at \( t \) is the one written by transaction \( T_i \).

\[
\begin{align*}
T_i.written & \in [O \rightarrow Time \rightarrow \{0, 1\}] \\
T_i.written(x)(t) & = 1 \text{ iff at time } t \text{ object } x \text{ holds the value written by } T_i \text{ most recently}
\end{align*}
\]

For each period, a transaction \( T_i \) can read a data object \( x \) at most once, and after it reads a value of \( x \), it keeps this value until the end of the period. The view of \( T_i \) on \( x \) can be captured by a state variable \( T_i.read(x) \) defined as follows. \( T_i.read(x) \) holds at time \( t \) within a period \( i \) 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.

\[
\begin{align*}
T_i.read & \in [O \rightarrow Time \rightarrow \{0, 1\}] \\
T_i.read(x)(t) & = 1 \text{ iff } T_i \text{ has performed a read operation on } x \text{ successfully before } t \\
& \text{ in a period containing } t.
\end{align*}
\]

A transaction \( T_i \) has a period \( P_i \). Therefore, for each \( i \leq n \) temporal propositional letter \( T_i.period \) is introduced to express that a time interval \([a, b]\) is a period of \( T_i \). Let \( Inte \) denotes the set of time intervals over reals.

\[
\begin{align*}
T_i.period & \in [Inte \rightarrow \{0, 1\}] \\
T_i.period([a, b]) & = \text{true} \text{ iff } [a, b] \text{ is a period of } T_i.
\end{align*}
\]

Of course,

\[
T_i.period \Rightarrow t = P_i. \tag{1}
\]

For each \( i \leq n \) state variables \( T_i.arrived \) is introduced to express that \( T_i \) is in the system at time \( t \).

\[
\begin{align*}
T_i.arrived & \in [Time \rightarrow \{0, 1\}] \\
T_i.arrived(t) & = 1 \text{ iff at time } t \text{ transaction } T_i \text{ is in the system and} \\
& \text{has not been committed or aborted since then.}
\end{align*}
\]

Because we assume that \( T_i \) arrived at the beginning of any period, it holds:

\[
T_i.period \Rightarrow \lnot[T_i.arrived] \lor \text{true}. \tag{2}
\]

A transaction \( T_i \) can request a lock for a data object \( x \) which is either read lock or write lock. Therefore, for each \( i \leq n \) state variables \( T_i.request\_lock(x) \) and \( T_i.request\_unlock(x) \) are introduced to express that \( T_i \) is requesting lock for a data object at time \( t \).

\[
\begin{align*}
T_i.request\_lock, T_i.request\_unlock & \in [O \rightarrow Time \rightarrow \{0, 1\}] \\
T_i.request\_lock(x)(t) & = 1 \text{ iff transaction } T_i \text{ is requesting a read-lock on } x \text{ at time } t \\
T_i.request\_unlock(x)(t) & = 1 \text{ iff transaction } T_i \text{ is requesting a write-lock on } x \text{ at time } t.
\end{align*}
\]
Let $T_i request lock = T_i request wlock \lor T_i request rlock$. When a transaction $T_i$ requests a lock on data object $x$, it may be granted or may have to wait. Therefore, for each $i \leq n$ and for each $x$, we introduce the state variables $T_i wait wlock(x)$ and $T_i wait rlock(x)$ to express that $T_i$ is waiting for a lock on data object $x$ at time $t$, and state variables $T_i hold wlock(x)$ and $T_i hold rlock(x)$ to express that $T_i$ is holding a lock on data object $x$ at time $t$.

$T_i wait wlock, T_i wait rlock, T_i hold rlock, T_i hold wlock \in |O \rightarrow Time \rightarrow \{0, 1\}]$

$T_i wait rlock(x)(t) = 1$ if $T_i$ is waiting for a read-lock on data object $x$ at time $t$

$T_i wait wlock(x)(t) = 1$ if $T_i$ is waiting for a write-lock on data object $x$ at time $t$

$T_i hold rlock(x)(t) = 1$ if $T_i$ holds a read-lock on data object $x$ at time $t$

$T_i hold wlock(x)(t) = 1$ if $T_i$ holds a write-lock on data object $x$ at time $t$

Let

$T_i wait lock = T_i wait rlock(x) \lor T_i wait wlock(x)$

$T_i hold lock = T_i hold rlock(x) \lor T_i hold wlock(x)$

In a period, a transaction can commit or abort. Therefore, for each $i \leq n$ state variables $T_i committed$ and $T_i aborted$ are introduced to express that $T_i$ has already committed or aborted at time $t$.

$T_i committed, T_i aborted \in |Time \rightarrow \{0, 1\}]$

$T_i committed(t) = 1$ if $T_i$ has committed successfully before $t$ in a period of containing $t$

$T_i aborted(t) = 1$ if $T_i$ 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.

$\square(T_i period) = \bigwedge_{x \in O_i} (\neg T_i read(x) \land \text{true})$

Now, we write DC formulas to capture the properties of state variables and their relationships. Those formulas will constrain the behaviour of the state variables introduced so far that a RTDBS produces. For any transaction $T_i$, at any time, either $T_i arrived$ or $T_i committed$ or $T_i aborted$ (here we assume that at the beginning, if a transaction has not arrived, it is committed).

$\models [T_i arrived \lor T_i committed \lor T_i aborted]^{*}$ (3)

These three states are mutually exclusive:

$\models [T_i arrived \Rightarrow \neg (T_i committed \lor T_i aborted)]$ (4)

$\models [T_i committed \Rightarrow \neg (T_i arrived \lor T_i aborted)]$ (5)

$\models [T_i aborted \Rightarrow \neg (T_i arrived \lor T_i committed)]$ (6)

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):

$\models \bigvee_{T_i \in T} T_i written(x)$ (7)

$\models [T_i written(x)] = \bigwedge_{T_i \not\in T} [\neg T_j written(x)]$ (8)
A transaction $T_i$ requests a lock for a data object $x$ iff it is in arrived state and it is either holding or waiting.

$$\bigwedge_{T_i \in T} \bigwedge_{x \in O} \left[ \lnot (T_i, request\_lock(x) \iff T_i, arrived \land (T_i, hold\_lock(x) \lor T_i, wait\_lock(x))) \right]$$ \hfill (9)

$$\bigwedge_{T_i \in T} \bigwedge_{x \in O} \left[ \lnot (T_i, request\_wlock(x) \iff T_i, arrived \land (T_i, hold\_wlock(x) \lor T_i, wait\_wlock(x))) \right]$$ \hfill (10)

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

$$\bigwedge_{T_i \in T} \bigwedge_{x \in O} \left[ \lnot (T_i, hold\_lock(x) \land T_i, wait\_lock(x)) \right]$$ \hfill (11)

$$\bigwedge_{T_i \in T} \bigwedge_{x \in O} \left[ \lnot (T_i, hold\_wlock(x) \land T_i, wait\_wlock(x)) \right]$$ \hfill (12)

The conflicting locks cannot be shared by the transactions. Therefore,

$$\bigwedge_{T_i \neq T_j \in T} \bigwedge_{x \in O} \left[ \lnot (T_i, hold\_lock(x)) \Rightarrow \lnot (T_j, hold\_wlock(x)) \right]$$ \hfill (13)

$$\bigwedge_{T_i \neq T_j \in T} \bigwedge_{x \in O} \left[ \lnot (T_i, hold\_wlock(x)) \Rightarrow \lnot (T_j, hold\_lock(x)) \right]$$ \hfill (14)

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

$$\bigwedge_{T_i \in T} \bigwedge_{x \in O} \left[ \lnot (T_i, read(x)) \Rightarrow \lnot (T_i, read(x)) \Rightarrow \diamond (T_i, hold\_lock(x)) \right]$$ \hfill (15)

$$\bigwedge_{T_i \in T} \bigwedge_{x \in O} \left[ \lnot (T_i, written(x)) \Rightarrow \lnot (T_i, written(x)) \Rightarrow \diamond (T_i, hold\_wlock(x)) \right]$$ \hfill (16)

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

$$\bigwedge_{T_i \in T} \bigwedge_{x \in O} \left( T_i, period \Rightarrow \lnot (\lnot (T_i, hold\_lock(x))) \right)$$ \hfill (17)

$$\bigwedge_{T_i \in T} \bigwedge_{x \in O} \left( T_i, period \Rightarrow \lnot (\lnot (T_i, hold\_wlock(x))) \right)$$ \hfill (18)

As mentioned earlier, for each period, for every $i$ and $x$ the state $T_i, read(x)$, $T_i, committed$ and $T_i, aborted$ can change at most once.

$$T_i, period \Rightarrow \Box (\lnot (T_i, read(x)) \Rightarrow \true \Rightarrow \lnot (T_i, read(x)))$$ \hfill (19)

$$T_i, period \Rightarrow \Box (\lnot (T_i, committed) \Rightarrow \true \Rightarrow \lnot (T_i, committed))$$ \hfill (20)

$$T_i, period \Rightarrow \Box (\lnot (T_i, aborted) \Rightarrow \true \Rightarrow \lnot (T_i, aborted))$$ \hfill (21)

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.

$$T_i, period \Rightarrow \Box (\lnot (T_i, written(x)) \Rightarrow \true \Rightarrow \lnot (T_i, committed))$$ \hfill (22)
Let $ENV$ be the set of the formulas (1), (2), (3), \ldots, (22). $ENV$ capture the axioms for the state variables introduced so far.

**4.2. Execution Model**

At any time, a transaction $T_i$ is running on processor or not running on processor. Therefore, for each $i \leq n$ state variables $T_i.run$ is introduced to express that $T_i$ is running on a processor at time $t$.

$$T_i.run \in [\text{Time} \to \{0, 1\}]$$

$$T_i.run(t) = 1 \text{ iff transaction } T_i \text{ is running on a processor at time } t$$

When a transaction $T_i$ has arrived and got all data object locks it needs, it is ready to run on the processor.

$$T_i.ready \in [\text{Time} \to \{0, 1\}]$$

$$T_i.ready(t) = 1 \text{ iff transaction } T_i \text{ is ready to execute on a processor at time } t$$

$T_i.ready$ will be defined via the assumption about the behaviour of transactions as follows.

A transaction ready it must in arrived state.

$$\bigwedge_{T_i \in \mathcal{T}} [\lbrack \lbrack T_i.ready \rbrack \Rightarrow \lbrack \lbrack T_i.arrived \rbrack]$$

When a transaction $T_i$ ready then it must not wait for a read-lock or a write-lock for it.

$$\bigwedge_{T_i \in \mathcal{T}} \bigg(\bigwedge_{x \in \mathcal{O}} \lbrack \lbrack T_i.ready \rbrack \Rightarrow \lbrack \lbrack \neg T_i.wait\_rlock(x) \rbrack \bigg)$$

$$\bigwedge_{T_i \in \mathcal{T}} \bigg(\bigwedge_{x \in \mathcal{O}} \lbrack \lbrack T_i.ready \rbrack \Rightarrow \lbrack \lbrack \neg T_i.wait\_wlock(x) \rbrack \bigg)$$

A transaction runs only if it is ready and this holds for every transaction.

$$A1 \equiv \bigwedge_{i=1}^{n} \bigcirc (\lbrack \lbrack T_i.run \rbrack \Rightarrow \lbrack \lbrack T_i.ready \rbrack])$$

The accumulated run time of transaction $T_i$ over an interval is given by $\int T_i.run$. In a period if a transaction is standing, then the maximal required execution time has not been reached.

$$A2 \equiv \bigwedge_{i=1}^{n} \bigg(T_i.period \Rightarrow (\lbrack \lbrack T_i.committed \rbrack \neg \lbrack \lbrack \neg T_i.committed \rbrack \neg \lbrack \lbrack \neg T_i.committed \rbrack \neg \lbrack \lbrack true \rbrack \bigg)$$

$$= \bigwedge_{i=1}^{n} \bigg(T_i.period \Rightarrow \big(\int Ti.Run < C_i \big) \neg \lbrack \lbrack T_i.committed \rbrack \neg \lbrack \lbrack \neg T_i.committed \rbrack \neg \lbrack \lbrack true \rbrack \bigg)$$

In a period if execution time of $T_i$ is equal to $C_i$, $T_i$ will commit from that time.

$$A3 \equiv \bigwedge_{i=1}^{n} (T_i.period \Rightarrow (\int T_i.run = C_i \neg \lbrack \lbrack T_i.run \rbrack > 0 \Rightarrow \lbrack \lbrack true \rbrack \neg \lbrack \lbrack T_i.committed \rbrack \rbrack)$$

Let EXEC be $A2 \land A3$
4.2.1. Uniprocessor Model

Assume that the transactions $T_1, \ldots, T_n$ share a single processor, and transaction priorities are assigned by the Rate Monotonic Algorithm.

Since there is only one processor, at any time if one transaction is running, then any other transaction can not be running.

$$A4 \equiv \square \bigwedge_{1 \leq i \leq n} (\lbrack\lbrack T_i, run\rbrack\rbrack \Rightarrow \bigwedge_{j \neq i} \lbrack\lbrack \neg T_j, run\rbrack\rbrack)$$

The processor cannot stay idle when a transaction is ready:

$$A5 \equiv \square \left( \bigwedge_{1 \leq i \leq n} T_i, ready\rbrack\rbrack \Rightarrow \bigwedge_{1 \leq i \leq n} T_i, run\rbrack\rbrack \right)$$

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

$$A6 \equiv \bigwedge_{1 \leq i \leq n} \square (\lbrack\lbrack T_i, ready\rbrack\rbrack \Rightarrow \bigwedge_{i < j \leq n} \lbrack\lbrack \neg T_j, run\rbrack\rbrack)$$

The conjunction of the preceding formulas constitute our uniprocessor model for the transactions, we have:

$$U_{sys} \equiv A1 \land A2 \land A3 \land A4 \land A5 \land A6$$

4.2.2. Multiprocessor Model

Assume there are $n$ transaction and they share with $m$ processor.

The variables specification, some assumptions for multiprocessor are the same as that of the execution model.

In environment multiprocessor, instead of specifying that there is only one running transaction in any time interval, we specify that the number of running transactions in any time interval should be no more than the number of processors.

$$A4m \equiv \bigwedge_{1 \leq i \leq n} \square (\lbrack\lbrack T_i, run\rbrack\rbrack \Rightarrow \exists S \leq m)$$

where $S$ denotes the number of running transactions in any time interval.

The conjunction of the preceding formulas constitute our multiprocessor model for the transactions, we have:

$$M_{sys} \equiv A1 \land A2 \land A3 \land A4m$$

5. A CASE STUDY: FORMALISATION OF BASIC ABORTING PROTOCOL IN RTDB

As presented in section 2, BAP is an extension of the well-known PCP in real time concurrency control. BAP offers higher priority transaction a chance to abort lower priority transaction and BAP requires transaction to lock data object in 2PL. In this section, we show the use of our model by giving a formal specification of BAP.

5.1. Serializability of 2PL

A formal specification of 2PL can be done in the same way as in [1] and it is omitted here.
5.2. Formalisation of BAP

In order to formalise the protocol, for each $i, j \leq n$, $x \in \mathcal{O}$, we introduce the following notations. Let $PL(x)$ be constants and $PN \subseteq \mathcal{N}$ denote the set of priority numbers, $T_i.locked-data$ and $T_i.sysceil$ be temporal variables.

The priority ceiling $PL(x)$ of each data object $x$ is equal to the highest priority of transactions which may read or write $x$.

$$ PL(x) \equiv \max\{p_j | x \in RO_{j, l} \cup WO_{j, l}, j \leq n\}. $$

$T_i.locked-data$ denotes the data objects locked by transactions other than $T_i$ at time $t$.

$$ T_i.locked-data \in [\text{Time} \rightarrow 2^\mathcal{O}] $$

$$ T_i.locked-data(t) = \{x | T_j.hold.lock(x)(t), T_i \neq T_j\} $$

$T_i.sysceil$ denotes the highest priority ceiling of data objects locked by transactions other than $T_i$ at time $t$.

$$ T_i.sysceil \in [\text{Time} \rightarrow \mathcal{PN}] $$

$$ T_i.sysceil(t) = \max\{PL(x)(t) | x \in T_i.locked-data(t)\} $$

A transaction $T_j$ can abort or can not abort a lock on data object $x$. Therefore, for each $j \leq n$ state function $T_j.aborable(x)$ is introduced to express that $T_j$ can abort a lock on data object $x$ and $\neg T_j.aborable(x)$ is introduced to express that $T_j$ can not abort a lock on data object $x$.

When a transaction $T_i$ attempts to lock a data object $x$, $T_i$ will be blocked and the lock on an object $x$ will be denied, if the priority of transaction $T_i$ is not higher than $T_i.sysceil$ and transactions other than $T_i$ can not abort. Therefore, the blockedby state function is:

$$ T_i.blockedby(T_j) \equiv \bigvee_{T_i \neq T_j \in \mathcal{T} \forall x \in \mathcal{O}} (T_j.hold.lock(x) \land T_i.wait.lock(x) \land \neg T_j.aborable(x) \land T_i.sysceil \geq p_i) $$

When a transaction $T_i$ attempts to lock a data object $x$, if the priority of transaction $T_i$ is not higher than $T_i.sysceil$ and transactions other than $T_i$ are abortable then $T_i$ can abort all transactions other than $T_i$. Therefore, the abortable state function is:

$$ T_i.aborable(T_j) \equiv \bigvee_{T_i \neq T_j \in \mathcal{T} \forall x \in \mathcal{O}} (T_j.hold.lock(x) \land T_j.aborable(x) \land T_i.sysceil \geq p_i) $$

Using the framework presented above, we present DC formula schemas for specifying BAP. First, the formula schema for the preemptive priority scheduler is presented as follows:

Let $HiPri_{BAP}(T_i, T_j)$ be a boolean-valued function for denoting which transaction between $T_i$ and $T_j$ has a higher priority.

(a) $HiPri_{BAP}$ is a partial order:

$$ \bigwedge_{T_i \neq T_j \in \mathcal{T}} (HiPri_{BAP}(T_i, T_j) \Rightarrow \neg HiPri_{BAP}(T_j, T_i)) $$

$$ \bigwedge_{T_i \neq T_j \neq T_k \in \mathcal{T}} \left( HiPri_{BAP}(T_i, T_k) \land HiPri_{BAP}(T_k, T_j) \Rightarrow HiPri_{BAP}(T_i, T_j) \right) $$
(b) $HiPri_{BAP}$ depends on the priority inherited by transactions:

$$\forall_{T_i \neq T_j \neq T_k \in T} \left( T_k, \text{blockedby}(T_i) \Rightarrow (HiPri_{BAP}(T_k, T_j) \Rightarrow HiPri_{BAP}(T_i, T_j)) \right)$$

$$\forall_{T_i \neq T_j \in T} \left( \forall_{T_k \in T} (\neg T_k, \text{blockedby}(T_i)) \Rightarrow (HiPri_{BAP}(T_i, T_j) \Rightarrow p_i > p_j) \right)$$

$$\forall_{T_i \neq T_j \in T} \left( \forall_{T_k \in T} (T_k, \text{blockedby}(T_i)) \land (T_k, \text{aborted}) \Rightarrow (HiPri_{BAP}(T_i, T_j) \Rightarrow p_i > p_j) \right)$$

The first formula expresses that when a transaction $T_i$ inherits the priority of transaction $T_k$, if $HiPri_{BAP}(T_k, T_j)$ then $HiPri_{BAP}(T_i, T_j)$. The second formula shows that if a transaction $T_i$ does not inherit any priority, then the relation $HiPri_{BAP}$ is consistent with the original assigned priorities. The third formula asserts that if a transaction $T_i$ inherits any priority $T_k$ and $T_k$ is aborted, then the relation $HiPri_{BAP}$ is consistent with the original assigned priorities.

The preemptive priority scheduler can be expressed as:

$$PPS \equiv \forall_{T_i \neq T_j \in T} \Box (\left[ T_i, \text{run} \right] \land \left[ T_j, \text{ready} \right] \Rightarrow \left[ HiPri_{BAP}(T_i, T_j) \right])$$

The Granting rule for BAP can be expressed as:

**Granting Rule** used to decide if the lock data object requested is granted or not.

$$Gr \equiv \forall_{T_i \in T} \forall_{x \in \mathcal{O}} \Box (\left[ \neg T_i, \text{hold\_lock}(x) \right] \land \left[ T_i, \text{hold\_lock}(x) \right] \Rightarrow \Diamond \left[ p_i > T_i, \text{sysceil} \right])$$

The blocking rule for BAP can be expressed as:

**Blocking Rule** used to decide whether a transaction is blocked on its request for a lock data object or not.

$$Bl \equiv \forall_{T_i \in T} \forall_{x \in \mathcal{O}} \Box (\left[ p_i > T_i, \text{sysceil} \right] \Rightarrow \left[ \neg T_i, \text{wait\_lock}(x) \right])$$

Then, the unblocking rule can be specified as:

**Unblocking Rule** used for deciding which among the blocked transactions is to be granted the lock data object.

$$UnBl \equiv \forall_{T_i \neq T_j \in T} \forall_{x \in \mathcal{O}} \Box (\left[ T_i, \text{wait\_lock}(x) \land T_j, \text{wait\_lock}(x) \right] \Rightarrow HiPri_{BAP}(T_i, T_j))$$

By combining these formula schemas together, the scheduler, $BAP$, is obtained:

$$BAP \equiv (2PL \land PPS \land Gr \land Bl \land UnBl)$$

Since BAP adopts Two Phase Locking (2PL) in preserving the serializability of transactions executions. Therefore, all executions of the transactions system produced by BAP are serializable i.e $BAP \models SERIAL$.

**Properties:**
The properties for the BAP are blocked at most once and deadlock free.

As we mentioned early, when a transaction $T_i$ requests a lock on a data object $x$, $T_i$ will be blocked, if the priority of transaction $T_i$ is not higher than $T_i$.sysceil. When a transaction $T_i$ holds a lock on a data object then $T_i$ will not be blocked by any lower priority transaction until $T_i$ completes its execution.

$$BAO \equiv \bigwedge_{T_i \in T} \Box(\bigvee_{x \in O} T_i.hold\_lock(x)) \Rightarrow \bigwedge_{x \in O} \neg T_i.wait\_lock(x))$$

Deadlock free which means that no exist a situation in which some or all transactions are waiting for a lock while others are committed.

$$DLF \equiv \Box \neg (\bigwedge_{T_i \in T} \bigwedge_{x \in O} (T_i.committed \lor T_i.wait\_lock(x)) \land \bigvee_{T_i \in T} \bigvee_{x \in O} T_i.wait\_lock(x)))$$

A formal proof that $BAP \models BAO \land DLF$ can be done in the same way as in [6, 9] and is omitted here.

5.3. The schedulability condition of BAP in RTDB

Recall that in [11], we have the schedulability condition for BAP a transaction $T_i$ scheduled by BAP will always meet its deadline for all process phase if there exists a pair $(k, m) \in SP_i$ such that

$$\sum_{j \in HPC_i} (C_j[m_Pk/P_j]) + C_i + B_i + ab_i \leq m_Pk,$$

where $B_i$ and $ab_i$ are the worst case blocking cost and aborting cost of transaction $T_i$, respectively, and $HPC_i = \{T_1, T_2, \ldots, T_{i-1}\}$ be the set of transactions with a priority no less than that of $T_i$ and

$$SP_i = \{(k, m)|1 \leq k, m = 1, 2, \ldots [P_i/P_k]\}.$$

Each pair $(k, m)$ represents a scheduling time point $m_Pk$ to test the schedulability of process $T_i$.

To determine the value of $ab_i$ and $B_i$, we refer interesting readers to [11] for details.

Let $C_i^* = C_i + B_i + ab_i$. For above conditions, we can formalise the schedulability condition for BAP as:

**Theorem 1.**

$$(ENV \land Usys \land BAP \land \sum_{j \in HPC_i} (C_j[m_Pk/P_j]) + C_i^* \leq m_Pk)$$

$$\Rightarrow \left( \bigwedge_{i=1}^{n} (T_i.period \Rightarrow \int T_i.run \geq C_i^*) \right)$$

A formal proof that Theorem 1 can be done in the same way as in [9] and is omitted here.

6. EXTENSION OF BAP

Since, BAP is an integration of the 2PL, PCP, and a simple aborting algorithm. In BAP, only a priority ceiling is needed for each data object. Therefore, BAP only allows exclusive locks on data objects. We propose extension of BAP as follows: EBAP (Extension of BAP) is an integration of the R/WPCP and a simple aborting algorithm. As R/WPCP, EBAP introduces a write priority ceiling WPL($x$) and an absolute priority ceiling APL($x$) for each data object $x$ in the system to emulate share and exclusive locks, respectively.
1. The write priority ceiling \( WPL(x) \) of data object \( x \) is set equal to the highest priority transactions that may write \( x \).

2. The absolute priority ceiling \( APL(x) \) of data object \( x \) is set equal to the highest priority transactions that may read or write \( x \).

3. The read/write priority ceiling \( RWPL(x) \) of data object \( x \), that is dynamically determined at run time. When a transaction read-locks \( x \), \( RWPL(x) \) is set equal to \( WPL(x) \). When a transaction write-locks \( x \), \( RWPL(x) \) is set equal to \( APL(x) \). A transaction may lock a data object if its priority is higher than the highest read/write priority ceiling \( RWPL(x) \) of the data objects locked by other transactions.

4. Abort ceiling is a priority level associated with transaction, determined as described below. A transaction may be blocked a data object if its priority is no higher than the highest read/write priority ceiling \( RWPL(x) \) of the data objects locked by other transactions. We add an abort rule as an addition to the binary choices between preemption and blocking: Transaction \( T_i \) may abort the currently running transaction and run immediately if its priority is higher than the current abort ceiling. If this test fails, then transaction \( T_i \) must block.

We believe that with the extension of BAP, which shown the effectiveness of using read and write semantics in improving the performance of BAP.

7. CONCLUSION

In this paper, we have presented a formal model of real time database systems. We specified and verified formally the Basic Abort Protocol in Real Time Databases using the proof system of DC. We also proposed an extension of BAP. These frameworks can be used in the future for specifying many other issues of RTDBS, we easily can specify and verify for a set of the concurrency control protocols in RTDBS.

Acknowledgements. The author would like to thank Associate Professor Doan Van Ban, Associate Professor Nguyen Huu Ng, and Dr. Dang Van Hung for their kind helps and guidances.

REFERENCES


[7] Ekaterina Pavlova and Dang Van Hung, A Formal Specification of the Concurrency Control in Real Time Database, UNU/IIST Report No. 152, UNU/IIST, P.O. Box 3058,


Received October 6, 2002
Revised December 23, 2002