Databases with Deadline and Contingency Constraints

Nandit Soparkar¹ Henry F. Korth² Abraham Silberschatz³

Abstract

Real-time database systems associate the concept of deadlines with transaction executions. Previous approaches use “best effort” techniques to schedule a given set of transactions to meet the deadlines as well as to ensure the consistency of the database. However, such approaches are inadequate for target applications which have “hard” real-time deadlines that need to be met in the event of crisis situations. In such cases, it is important to obtain contingency plans that may be invoked with guaranteed execution time characteristics. This paper presents an alternative model for real-time database systems in which deadlines are associated with “contingency” constraints rather than directly with transactions. Our approach leads to a predicate-based model that intrinsically incorporates both triggering and relative timing constraints regarding the transaction executions. We exhibit that selecting contingency plans with respect to various optimality criteria has inherent computational inefficiencies. We study the issues in scheduling of the selected plans with the focus on the contention among the transactions for data resources. Our results exhibit that the data contention, by itself, has a severe adverse impact on the schedulability of the deadline-constrained transactions. We discuss some of the practical implications of our results, and we suggest some counter-measures to handle the computational complexities.
Affiliations of Authors

1 Work done while at the University of Texas at Austin, with support from an IBM Graduate Fellowship. Currently at Elect. Engin. & Comp. Sci., University of Michigan, Ann Arbor, MI 48109-2122.


3 On leave from the University of Texas at Austin. Currently at AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974.
Preferred Address

Nandit Soparkar
Electrical Engineering & Computer Science
The University of Michigan
Ann Arbor, MI 48109-2122.
Keywords

Databases, Real-time systems, Scheduling, Concurrency control, Transaction management.
1 Introduction

Real-time systems often require access to a large database of information. As a result, recent efforts have aimed at integrating real-time systems with database systems. These new types of systems, called real-time database (RTDB) systems, incorporate timing considerations into a database system (e.g., see [1, 2, 3]). In such systems, the transactions that access stored data must not only execute correctly, but also, they must complete executing within a time limit, called the deadline. Systems that incorporate strict deadlines are called hard RTDB systems while those that do not are called soft RTDB systems.

One major difficulty in accomplishing the integration of real-time systems and database systems is the issue of transaction management. Discussions in [1, 2, 4] discuss some of the important issues and approaches. Most previous work assumes a set of transactions and associated deadlines. Typically, physical resources (such as the computing units etc.) are considered with regard to availability and scheduling issues. Thereafter, experimental simulation analysis is conducted using different traditional concurrency control techniques to empirically ascertain which ones serve well to meet the imposed deadlines. These are “best effort” approaches in that given limited resources and limited knowledge regarding the transactions and their execution times, scheduling alternatives are identified such that the best performance (e.g., in terms of the least number of missed deadlines) for select workload sets is achieved. Several complicating factors usually preclude guarantees for the hard timing constraints on the executions. One such factor is the effect of contention among the transactions for access to shared data.

In this paper, we deal with situations where the deadlines are hard and must be met — which we refer to as being crisis situations. The key requirement in crisis situations may be regarded as the meeting of the hard deadlines (without affecting the logical correctness of the executions). Therefore, availability and scheduling issues for the resources become very important in such situations. In this paper, our focus is on the relatively poorly studied area of contention for data resources. In particular, we consider the effects of data conflict (which differ from other types of resource conflicts in that the order in which the common data resources are accessed is significant in terms of the logical correctness of the executions) on the scheduling of the transactions such that their deadlines are met. In a departure from other approaches, we find it useful to develop a model in which the time constraints apply directly to states of the RTDB. (Preliminary research regarding our model was reported in [3], and a portion of that material is provided in Sections 4 and 5 of this paper.)

Example 1. Consider an RTDB application in a manufacturing environment. Suppose that the state of the information maintained in the database indicates that the temperature in a furnace has risen above a particular threshold value. This state of the system may necessitate the triggering of some actions that restore the temperature to a value below the threshold. The application may enforce a maximum period of time for which the temperature is permitted to remain above the threshold — and that enforces a deadline on the triggered actions. Note that it is possible that several actions may be candidates for the restoration of the temperature. For instance, there may be actions that initiate reducing the fuel, or actions that decrease the oxygen supply, etc. Depending on the deadlines (and other factors such as the time taken by the actions), one particular action may be initiated to restore the temperature value. These actions are reflected as triggered transactions within the database.

We propose a new approach to the modeling of an RTDB. The aim is to use a semantics-based model to deal with the crisis situations. Our approach is based on a set of explicitly defined “contingency” constraints for the database. Validity of the constraints represents a “safe” state of the database (and, of course, the external environment), whereas their violation indicates that a crisis situation has arisen, and that corrective actions must be taken.

Using our model, we show that the selection of contingency plans (with regard to various optimality criteria) is typically computationally expensive. Thereafter, we consider the problems in scheduling the selected contingency plans to guarantee the necessary deadlines. In this regard, we focus attention on the problems introduced by concurrently executing transactions that contend for common data resources (and we disregard the effects of contention for other types of resource). In particular, we show that in the absence of data contention, the scheduling problem is resolved efficiently — whereas, when data contention may be present, the same problem becomes computationally intractable.
2 Database Model

We assume a standard database model with a few modifications to allow the expression and examination of triggering and hard deadline constraints germane to our research as follows.

2.1 Time

Since the notion of time is essential in an RTDB, we must provide the means to model the passage of physical time by assuming the existence of a device that measures time. We model such a device by means of a special data entity called the clock which is a read-only, positive, integer-valued counter that increases monotonically and atomically, and is accessible to all transactions. The read-only and monotonically increasing properties for a clock captures the uncontrollable passage of physical time.

Additionally, we assume a “timestamp” attribute for the data stored in the database. For example, an attribute timestamp(d) for a data item d may be regarded as a temporal attribute for d. Note that the semantics associated with timestamp(d) are entirely application-dependent in that it may be regarded as the time at which the data item was last updated, or the time at which the transaction that created it was initiated etc. (e.g., see [5]).

The concept of a clock and the timestamp attribute are introduced to explain how physical time is incorporated into our model; we do not explicitly use them for the described research.

2.2 Trigger Constraints

The transactions in an RTDB environment are often constrained to be triggered (i.e., invoked) at particular points in physical time depending on the state of the database. The inclusion of the clock in the database allows us to describe constraints (on the database states) whose violation may trigger some transaction. We refer to such constraints as “trigger constraints,” and these are not regarded as part of the database integrity constraints (e.g., unlike the description in [1] in which the uncontrolled passage of time may cause the database to become inconsistent). The trigger constraints, which are assumed to be stated explicitly for an application, are useful for triggering certain transactions, and they are liable to become invalid from time-to-time as explained below. A transaction, if triggered by a trigger constraint, is expected to restore the database state to one that satisfies the trigger constraint. Any transaction that may affect a trigger constraint is also assumed to check the validity of that constraint.

Example 2. Suppose that the relationship “timestamp(d) + c ≤ clock” is a trigger constraint, which represents the requirement that a transaction that updates the value of the entity d every c units of physical time, be invoked periodically with a period of c. In a manufacturing environment, the periodic checks on the operating parameters may be represented by such trigger constraints. Thus, in a situation where the temperature of a furnace is being monitored every 5 seconds, c may be regarded as 5 seconds, d may be regarded as the entity that represents the temperature, and the invoked transaction would be the one that makes a recording of the transducer reading into the database. □

2.3 Contingency Constraints

The nature of the crisis situations with which we are concerned must be anticipated in advance, and the transactions designed to deal with them should be triggered automatically by the RTDB. We refer to special trigger constraints whose violation represent crisis situations as “contingency constraints.” The clock is not mentioned in these constraints since crisis situations occur with no known periodicity or relationship to physical time. However, we associate a temporal deadline within which the constraint needs to be satisfied in case it does get violated. A transaction that is invoked due to a contingency constraint violation is required to complete execution, and thereby, to restore the validity of the constraint, within the time specified by the deadline.

Example 3. Suppose that the temperature in a nuclear reactor is measured and written into a data entity d in the database. A contingency constraint may be stated as d ≤ 1000 to represent a constraint that the temperature should never exceed 1000 degrees. Furthermore, a deadline value of 100 time units may
be associated with the constraint to represent that the “dangerous” temperature must be rectified within a period not exceeding 100 time units from the time that the crisis occurred. Hence, a transaction invoked as a result of a violation of the constraint must complete execution within 100 time units.

In general, we deal with transactions after they have been invoked, and are not concerned with the actual triggering mechanisms (which is investigated in other research—e.g., see [6]). However, we do examine the issues related to when and which transactions are triggered.

3 Transaction Model

We assume a standard transaction model (e.g., see [7]), with certain extensions (e.g., see [8]) to permit the examination of serializable as well as more liberal scheduling criteria within a common framework. Specifically, input and output predicates are defined for a transaction that refer to data items within the database. The input predicate defines the conditions under which its associated transaction may proceed with its execution. For the sake of simplicity, we assume that input predicates are evaluated by the corresponding transaction programs themselves. The output predicates describe the state into which an isolated execution of the associated transactions is guaranteed to place the database assuming that the input predicates were satisfied at the invocation times. The input and output predicates of each transaction must satisfy the database integrity constraints. The input and output predicates are usually obtained from the applications.

An RTDB system interacts with the external world in several ways. Events in the external world are recorded in the database. Transactions in the RTDB initiate external actions. This leads us to partition the set of transactions into three categories as follows. (1) “External-input” transactions which record in the database some events that occur in the external world. Often, such a transaction is a write-only transaction, and is of short-duration. (2) “Internal” transactions which access the database in a manner similar to any traditional database transaction except that it may be of long-duration. The purpose of this type of transaction is the restoration of a contingency constraint that may have been violated as a result of some external-input transaction. (3) “External-output” transactions which cause some events to occur in the world external to the system. These transactions are of short-duration from a system perspective, although the external actions they trigger may take a longer time to complete.

The three types of transactions differ in their atomicity and concurrency requirements. A write-only external-input transaction should never wait since it is used to record the outside world within the system as soon as possible (in order to deal with contingency constraint violations quickly).

Transactions, together with contingency constraints, facilitate dealing with situations where certain actions need to be triggered and executed within a particular interval of time—as illustrated in the following example.

**Example 4.** Consider an example adapted from airspace control (e.g., see [5]) in which an airplane moves from an airspace A to an adjacent airspace B. Different air traffic controllers are assumed to be responsible for each airspace; it is required that the aircraft never be outside the control of both controllers for more than 500 milliseconds, and be normally under the control of one. Let a data entity $O_a$ indicate whether the control of the aircraft is with airspace A or not according as its value being 1 or 0. In a similar manner, let the value of $O_b$ indicate the control of the aircraft with respect to airspace B. The requirement that at least one airspace should have control over the aircraft is captured by the contingency constraint $O_a + O_b \geq 1$. Moreover, the temporal constraint on a situation that violates the normal circumstances is defined by associating a temporal deadline of 500 milliseconds with the contingency constraint.

To implement this, four transactions may be defined as follows. Transactions $T_a$ and $T_b$ update the values of $O_a$ and $O_b$, respectively, and thereby indicate the changeover of control between the air-spaces. Transactions $T_x$ and $T_y$ read the data entities $O_a$ and $O_b$, respectively, and update the displays for their respective controllers (in case the aircraft happens to be within their airspace jurisdiction). The transactions may be described as $T_a$: $W_a(O_a)$; $T_b$: $W_b(O_b)$; $T_x$: $R_x(O_a) W_x(display_1)$; and $T_y$: $R_y(O_b) W_y(display_2)$.

Note that while any interleaved execution of the above transactions is serializable, the temporal constraint must also be obeyed in order to ensure the safety requirement for correctly controlled flight. It happens to be the case that any serializable execution in which the two operations $W_a(O_a)$ and $W_b(O_b)$ are separated in physical time by no more than 500 milliseconds, is safe. Therefore, in this example, the contingency
constraint violation (e.g., due to the execution of $T_k$) may impose a hard deadline on the transaction $T_k$ which would need to be obeyed to ensure that the constraint remained invalidated for a period not exceeding its 500 milliseconds deadline. □

Transactions in RTDBs may be submitted either by users, or by external devices. In addition, transactions may also be triggered by the state of the system. If an external-input transaction changes the database state to one that is inconsistent with the contingency constraints, an internal transaction must be run to restore the constraints. These transactions are not necessarily triggered by an external-input transaction. Rather, they may depend on both the external-input transaction and the database state. Triggered internal transactions are the focus of our attention in this paper.

Our RTDB system model may be regarded in the abstract as comprising of finite sets $T = \{T_1, T_2, \ldots, T_n\}$ of predefined transactions, and $C = \{c_1, c_2, \ldots, c_m\}$ of predefined contingency constraints in the form of conjuncts. Conjuncts are formulae consisting of a disjunction of possibly negated terms. The contingency constraint for the entire database may be represented by $\bigwedge_{i=1}^m c_i$. Thus, for the system in Example 3, the term “$d \leq 1000$” (possibly in disjunction with other such terms arising from the description of the system) may be regarded as being a particular contingency constraint. Furthermore, the associated RTDB would be regarded as being in a safe state when each of (i.e., the conjunction of) the contingency constraints is satisfied. Some instances of the transactions may be triggered by the invalidation of a constraint, and may function to restore the truth of the constraint. However, the execution of these transactions may falsify some other constraints. Thus, the system may be regarded as consisting of transactions and constraints that interact with one another.

4 A Graph-based Approach

To help describe our model and the algorithmic analyses, we define a predicate-precedence graph (PPG) which captures the relationships between the transactions and the constraints. Subsequently, we describe how certain annotations on the graph may be used to incorporate timing constraints. A PPG is a directed bipartite graph with a set of vertices $V = T \cup C$, where $T$ denotes the set of transactions, and $C$ denotes the set of constraint vertices. The edges in a PPG represent the triggering of transactions by the invalidation of the constraints, and, in turn, the invalidation of constraints by the transactions. If an instance of a transaction $T_i$ may invalidate a constraint $c_j$, then the directed edge $T_i \rightarrow c_j$ appears in the graph. If a transaction $T_k$ ensures the truth of a constraint $c_j$ upon completion, then the directed edge $c_i \rightarrow T_k$ appears in the graph. Thus, the PPG represents the transactions available to the system for restoring the contingency constraints.

Example 5. Consider the PPG shown in Figure 1 where transactions and constraints are represented by square and round vertices, respectively. The PPG specifies that the falsification of constraint $c_1$ may be resolved by executing an instance of either one of the transactions $T_1$ or $T_2$. Furthermore, the execution of a transaction $T_1$ may result in the invalidation of the constraints $c_5$, $c_6$ and $c_7$. □

4.1 Marked PPGs

If the database state does not satisfy the contingency constraints, the vertices corresponding to the invalidated constraints are marked. To restore contingency constraints, it is necessary to run an instance of the transaction associated with the head of at least one out-edge of each marked vertex. However, running these transactions may lead to side-effects beyond restoring the truth of certain previously-false constraints. Specifically, these side effects may result in other constraints becoming false resulting in further marked vertices. Given a graph with a set of marked vertices, there may exist many ways to resolve the violation of the contingency constraints.

Note that if a constraint vertex is a sink (has no out-edges), then there is no way to restore the truth of this constraint within the system; we assume that such a situation does not occur. Also note that a cycle in the PPG represents a potentially unstable situation in that a stable state may be never achieved.

A safe strategy (i.e., one that is not potentially unstable) for resolving a database state inconsistent with respect to the contingency constraints can be represented by a directed acyclic graph (i.e., a DAG) that is
a subgraph of the PPG. The DAG should contain all the marked constraint vertices of the PPG, retain all out-edges in the PPG of transaction vertices in the DAG, and retain at least one out-edge of each constraint vertex in the DAG.

**Example 6.** Consider the PPG of Figure 1 again. The subgraph shown within the dotted outline in the figure is a DAG that provides a strategy to resolve the falsified contingency constraints if \( c_1 \) and \( c_5 \) (and possibly any or all of \( c_3, c_5, \) and \( c_7 \)) are the only marked vertices. □

Consider a DAG subgraph of the PPG that can resolve the inconsistency of the database with respect to the contingency constraints: it has roots at all marked vertices, and sinks that are transactions. All out-edges in the PPG from transactions in such a DAG must be included in the DAG. When partial order on transactions induced by the DAG is observed in the execution of the transactions, the contingency constraints will be restored. We call such a subgraph an *contingency-resolution subgraph* (CRS).

We provide a formal definition of the graphs described above as follows.

**Definition 1.** A *predicate-precedence graph* is a 3-tuple \((C, T, E)\) representing a bipartite graph with vertex set \( C \cup T \) and edge set \( E \subseteq ((C \times T) \cup (T \times C)) \). A marked PPG is a PPG in which a nonempty set of vertices \( X \subseteq C \) is identified as being “marked”. □

The contingency-resolution subgraph (CRS) defined below represents a strategy for restoring contingency constraints in the database given that the marked set of constraints are false.

**Definition 2.** Let \( G = (C, T, E) \) be a PPG in which the vertices in \( X \subseteq C \) are marked. A contingency-resolution subgraph of \( G \) is a 3-tuple \( G' = (C', T', E') \) such that,
(1) \( X \subseteq C' \subseteq C, T' \subseteq T, \) and \( E' \subseteq E \),
(2) For all edges \( T_j \rightarrow c_i \in E \) such that \( T_j \in T' \), we have \( c_i \in C' \) and \( T_j \rightarrow c_i \in E' \),
(3) For all \( c_i \in C' \), there exists a path in \( G' \) from \( c_i \) to \( T_k \), where \( T_k \) is a sink in \( G \), and
(4) \( G' \) is acyclic. □

A natural question arises as to whether a CRS exists for a particular marked PPG. The following result implies that the question is easily settled.

**Theorem 1.** Let \( G = (C, T, E) \) be a marked PPG. The problem of deciding whether there is a CRS \( G' \) for \( G \) is solvable in \( O(|C| + |T| + |E|) \) time.

**Proof:** We provide a sketch of a polynomial time algorithm that manipulates the PPG, \( G \). For the ease of presentation, we introduce a (pseudo) transaction vertex, \( t' \in T \), with out-edges \( t' \rightarrow c_i \) for every \( c_i \in X \), and a (pseudo) constraint vertex, \( c' \in C \), with an out-edge \( c' \rightarrow t' \).

1. **while** \( c' \in C \) **do**
   (a) Choose a sink transaction vertex, \( T_j \). If none exists, **print** “No CRS exists”, and **stop**.
   (b) For each constraint vertex \( c_i \) such that \( c_i \rightarrow T_j \in E \), delete from \( E \) all edges (in and out) that are adjacent to \( c_i \). Hence, delete \( c_i \) from \( C \).
   (c) Delete \( T_j \).

**endwhile**

2. **print** “CRS exists”, and **stop**.

The deletion of the vertex \( c' \) from \( C \) implies that a CRS which includes all the vertices in \( X \) has been deleted. With appropriate data structures, the algorithm takes \( O(|C| + |T| + |E|) \) time. □

As a consequence of the above result, henceforth, we assume that a CRS exists for a given marked PPG.

### 4.2 Incorporating Time Costs and Deadlines

Timing constraints are represented in the PPG by associating a *time interval* with each constraint and a *time cost* with each transaction. The value associated with each constraint represents the maximum duration of a time interval during which the corresponding constraint may be false. The time cost represents an estimate of the execution time of the transaction. Although the formal model developed here is independent of the determination of the execution times, for our purpose, we consider the expected-case estimates (e.g.,
transactions are accurate. Indeed, if the database is entirely memory-resident (which is the case for most real-time applications), the differences between the worst-case and expected time estimates are likely to be negligible. We assume that the execution time estimates for the transactions are accurate.

The incorporation of time into our model is achieved by the use of the functions \( W_x \) and \( W_\tau \) which denote mappings from the constraints \( C \) and the transactions \( T \), respectively, to the set of non-negative integers. These integers represent time intervals when associated with the constraints, and execution times when associated with the transactions. Incorporating times constraints in this manner requires the annotation of the PPG. We term this new PPG as a weighted PPG, while the original PPG is termed an unweighted PPG.

**Definition 3.** A (weighted) predicate-precedence graph (PPG) is a 5-tuple 
\[
(C, T, E, W_x, W_\tau)
\]
representing a bipartite graph with vertex set \( C \cup T \) and edge set \( E \subseteq ((C \times T) \cup (T \times C)) \); and \( W_x : C \rightarrow \mathbb{R}^+ \) and \( W_\tau : T \rightarrow \mathbb{R}^+ \) are the time interval and time cost functions, respectively. □

Notice that an unweighted PPG can be represented by a PPG in which \( W_\tau \) maps all elements of \( T \) to 1, and \( W_x \) maps all elements of \( C \) to \( l \) (where \( l \) is suitably chosen). Also, we can extend the notion of a marked unweighted PPG to a marked weighted PPG in a natural manner. Note that the case where \( W_x(c_i) < W_\tau(T_j) \) for a constraint \( c_i \) and a transaction \( T_j \), it is not worthwhile including an edge \( c_i \rightarrow T_j \) in the PPG. Hence, we assume that for an edge \( c_i \rightarrow T_j \) in a PPG, it is always the case that \( W_x(c_i) \geq W_\tau(T_j) \).

We need to redefine the contingency-resolution subgraph for a weighted PPG. Again, the CRS represents a potential strategy for restoring contingency constraints in the database given that the marked set of constraints are false.

**Definition 4.** Let \( G = (C, T, E, W_x, W_\tau) \) be a weighted PPG in which the vertices \( X \subseteq C \) are marked. A contingency-resolution subgraph of \( G \) is a 5-tuple \( G' = (C', T', E', W_x', W_\tau') \) such that,
1. \( X \subseteq C' \subseteq C \), \( T' \subseteq T \), and \( E' \subseteq E \),
2. For all edges \( T_j \rightarrow c_i \in E \) such that \( T_j \in T' \), we have \( c_i \in C' \) and \( T_j \rightarrow c_i \in E' \),
3. For all \( c_i \in C' \), there exists a path in \( G' \) from \( c_i \) to \( T_k \), where \( T_k \) is a sink in \( G' \),
4. \( G' \) is acyclic,
5. \( W_x' \) is the restriction of \( W_x \) to \( C' \), and \( W_\tau' \) is the restriction of \( W_\tau \) to \( T' \), and
6. For all \( c_i \in C' \), there is an edge \( c_i \rightarrow T_j \in E' \) such that \( W_\tau(T_j) \leq W_x(c_i) \). □

There may exist several candidate CRSs for a particular marked PPG, and a decision would need to be made to select a particular one. Intuitively, the CRS that represents the “best” strategy (based on some selection criteria) to restore the constraints should be selected. Although the precise characterization for the selection criteria may be application-dependent, it is possible to identify some important ones. For example, a CRS that provides a strategy to restore consistency of the contingency constraints promptly may be regarded as being better than one that implies a slower restoration. A second selection criterion could be the choice of a CRS that invalidates the least number of contingency constraints. A third example criterion arises from the scheduling concerns for the transactions — by considering the available “slack time” which, in the case of a transaction \( T_j \) that is chosen to resolve the inconsistency of a constraint \( c_i \), may be roughly described to be \( W_x(c_i) - W_\tau(T_j) \). Availability of slack times provides more options for scheduling. Therefore, a selection criterion may be based on a function of the slack times available in a selected CRS.

Our model for RTDB transactions represented by the PPG provides the system with additional degrees of freedom in managing an RTDB. Not only can the concurrency and recovery managers take into account the constraint deadlines and time costs associated with the transactions, but also the system has some choice among the set of transactions to use in response to a particular collection of violated constraints that arise due to external events. Note that, as described, our model is more general than is required for the analyses and discussions below.

The issues related to the use of a PPG and a CRS are two-fold. First, efficient selection procedures are needed to identify a good CRS, where goodness is related to how well the chosen CRS can be used to resolve the crisis situations. Second, after the CRS has been identified, scheduling approaches are needed to execute the transactions specified by the CRS.
5 Complexity Results for Selection

In this section, we disregard the effects of the PPG-imposed partial ordering among the transactions and concurrency control issues, and instead, we focus attention on the selection of good CRSs for marked PPGs. In each case, the size of the problem is taken to be the size (of the representation) of the associated PPG.

5.1 Selection Based on Weights

Consider an unweighted, acyclic PPG, $G$. Suppose that the selection criterion for a CRS is to obtain one that includes the fewest number of transaction vertices.

The Transaction-weight Problem (TUAP — the Transaction-weight for an Unweighted, Acyclic PPG) for a marked, unweighted, acyclic PPG is: Given a marked, unweighted, acyclic PPG, $G$, and an integer $K$, is there a CRS, $G'$, such that the number of elements in $T'$ is at most $K$?

**Theorem 2.** The TUAP problem is NP-complete.

**Proof:** The proof of NP-easiness is as follows. We demonstrate how to verify in polynomial time that a non-deterministically selected graph $G'$ is a CRS with $|T'| \leq K$. Verifying that $G'$ represents a CRS is accomplished by checking that $X \subseteq C'$, and that for every $c_i \in C'$, there exists an edge $c_i \rightarrow T_j \in E'$. Checking that $|T'| \leq K$ completes the verification.

We now prove NP-hardness. An instance of the NP-complete Satisfiability problem (L01 in [9]) is reduced to the TUAP problem. Let $P$ represent the conjunction of $m$ clauses in L01, i.e., $P \equiv \bigwedge_{i=1}^{m} C_i$ where the clauses are formed over $n$ boolean variables $x_1, x_2, \ldots, x_n$. As shown in Figure 2, form an instance of a PPG, $G$, with $C = \{p, c, c_1, c_2, \ldots, c_m, x_1, x_2, \ldots, x_n\}$, $T = \{p', c', F_{x_1}, F_{x_2}, \ldots, F_{x_n}, T_{x_1}, T_{x_2}, \ldots, T_{x_n}\}$, and $X = \{p\}$. Besides the edges explicitly shown in Figure 2, $G$ includes an out-edge from a vertex $c_i$ to either $T_{x_i}$ or to $F_{x_i}$ for every positive or negative literal, respectively, formed using an $x_i$ occurring in the clause $C_i$ of the Satisfiability problem instance. We prove that $P$ is a satisfiable instance of L01 if and only if $G$ contains a CRS, $G'$, with $|T'| \leq (n + 2)$. Note that the construction guarantees the existence of a CRS.

Assume that a requisite CRS, $G'$, exists. $|T'| \geq (n + 2)$ since included in $G'$ are $p'$, $c'$, and at least one of $T_{x_i}$ or $F_{x_i}$ for every $x_i$. Since $G'$ is a requisite CRS, we have $|T'| = (n + 2)$. This implies that exactly one of the vertices reachable from a vertex $x_i$ is included in $T'$. Assign a boolean value of $T$ or $F$ to the corresponding variable $x_i$ in L01 according as $T_{x_i}$ or $F_{x_i}$ is included, respectively, in $T'$. It is clear that every clause of L01 will have one satisfied literal by this assignment.

If there is a truth assignment for every $x_i$ in the problem instance of L01 that satisfies $P$, consider a subgraph $G'$ as described next. The set $T'$ consists of $p'$, $c'$, and $T_{x_i}$ or $F_{x_i}$ according as $x_i$ is assigned $T$ or $F$, and the set $C' = C$. The subgraph $G'$ contains all possible edges of $G$. It is easy to see that $G'$ is a CRS with $|T'| \leq (n + 2)$. □

If we introduce the timing constraints in terms of the functions $W_\alpha$ and $W_\tau$, a selection criterion for a CRS could be the minimization of the sum of the time costs of the transaction vertices included in the CRS. This criterion is suggested by the need for the “fastest” contingency-resolution strategy.

The Transaction-weight Problem (TWP — the Transaction-weight for a Weighted PPG) for a marked, weighted, acyclic PPG is: Given a marked, weighted PPG, $G$, and an integer $K$, is there a CRS, $G'$, such that the sum of the weights of the elements in $T'$ is at most $K$?

**Theorem 3.** The TWP problem is NP-complete.

**Proof:** The TUAP problem is the TWP problem with unit weight assignments to the elements of $T$. □

We consider now a different selection criterion that is based on the number of constraints that may be invalidated. In the case of an marked, unweighted, acyclic PPG, a related measure of goodness would be to find a CRS which minimizes the number of contingency constraints that it may invalidate.

The Predicate-weight Problem (PUAP — the Predicate-weight for an Unweighted, Acyclic PPG) for a marked, unweighted, acyclic PPG is: Given a marked, unweighted, acyclic PPG, $G$, and an integer $K$, is there a CRS, $G'$, such that the number of elements in $C'$ is at most $K$?

**Theorem 4.** The PUAP problem is NP-complete.

**Proof:** The proof of NP-easiness is the same as that for the TUAP problem with a verification of $|C'| \leq K$ replacing $|T'| \leq K$. □
To prove NP-hardness, we exhibit a similar reduction from the problem L01 as we did for the TUAP problem. The instance of the PPG constructed is modified to have the additional subgraphs at the nodes $T_{x_i}$ and $F_{x_i}$ as shown in Figure 3. Set $K = (2n + m + 2)$. The proof is now clearly similar to the NP-hardness proof of the TUAP problem. □

The above theorems indicate that the selection procedures to find optimal CRS graphs for the PPG graphs is difficult. Therefore, we anticipate the need for heuristic approaches to find good CRS graphs in place of the “best” CRS graph.

5.2 Selection Based on Slack Times

Large slack times allow a greater flexibility in scheduling transactions, and in RTDB systems, this flexibility is valuable. To formalize the concept of slack time, consider a PPG, $G = (C, T, E, W_e, W_r)$. The (potential) slack time for a constraint vertex $c_i$ in $G$ is given by $S_e(c_i) = W_e(c_i) - \min_{T_j \in E}(W_r(T_j))$.

5.2.1 Total Slack Time

Let the sum of the slack times associated with the constraint vertices of a CRS, $G'$, be termed as the total slack time of the CRS, and be denoted by $\text{slack}(G')$. Assume that the application indicates that a selection criterion is chosen based on the maximization of the total slack time. With the $S_e$ values as provided, the CRS chosen directly would be the PPG itself — clearly an unacceptable choice. Hence, we use the method described below to limit the number of vertices chosen while retaining the criterion of total slack time maximization.

We define the “inverse” slack time associated with a constraint vertex $c_i$ to be given by $S'_e(c_i) = \eta - S_e(c_i)$ where $\eta \geq (1 + \max_{c \in C}(S_e(c)))$. The constraint on the value of $\eta$ is to ensure that $S'_e(c_i) \geq 1$ for all $c_i \in C$. Suppose that a CRS, $G'_{min}$, is chosen such that the sum of the $S'_e$ values associated with its constraint vertices is the smallest among all the CRSs, $G'$, that are possible. Using the above definition, we have $\eta|C_{min}'| - \text{slack}(G'_{min}) \leq \eta|C'| - \text{slack}(G')$. Notice that $|C'_{min}| = |C'|$ implies that $\text{slack}(G'_{min}) \geq \text{slack}(G')$, and that $\text{slack}(G'_{min}) = \text{slack}(G')$ implies that $|C'_{min}| \leq |C'|$. Thus, for two CRSs, if the number of constraint vertices in each is the same, the one with a larger total slack time is preferred by this minimization criterion. If the total slack times of the two CRSs are equal, then this criterion chooses the one with fewer constraint vertices.

The value of $\eta$ controls a trade-off in the chosen CRS in the following manner. Maximizing the total slack time without using a notion such as the inverse slack time leads to the selection of an unnecessarily large CRS with too many constraint vertices. The trade-off between increasing the total slack time, $\text{slack}(G')$, and decreasing the number of constraint vertices, $|C'|$, in the CRS is governed by the value of $\eta$ (since it determines their weightage). This is evident from the expression $\eta|C'| - \text{slack}(G')$ which is the sum of the inverse slack times of the vertices in $C'$.

Consider a modified PPG, $G'$, in which for all $c_i \in C$ and $T_j \in T$, we set $W_e(c_i) = S'_e(c_i)$ and $W_r(T_j) = 1$. By introducing inverse slack times in this manner, and choosing a desired value for $\eta$, the question of maximizing the total slack time for a CRS reduces to the following problem.

The Predicate-weight Problem (PWP — the Predicate-weight for a Weighted PPG) for a marked, weighted PPG is: Given a marked, weighted, acyclic PPG, $G$, and an integer $K$, is there a CRS, $G'$, such that the sum of the weights of the elements in $C'$ is at most $K$?

Theorem 5. The PWP problem is NP-complete.

Proof: The PUAP problem is the PWP problem with unit weight assignments to the elements in $C$. □

5.2.2 Large Individual Slack Times

It may be argued that it is more germane to use a selection criterion for a CRS based on the largeness of the slack times associated with the constraints. That is, the cost of a CRS $G' = (C', T', E', W'_e, W'_r)$ is $\max_{c \in C'}(S'_e(c_i))$. Large slack times provide the flexibility in scheduling the contingency-resolving instances of transactions which may be necessitated by concurrency control considerations.
The Individual Slack Time Problem (IST — the Individual Slack Time) for a PPG is: For a given marked, weighted, acyclic PPG, \( G \), and an integer \( K \), is there a CRS, \( G' \), such that \( \max_{v \in V(G')} S'_v(c_i) \) is at most \( K \)?

**Theorem 6.** For a PPG, \( G = (C, T, E) \), the IST problem is solvable in \( O(|C| + |T| + |E|) \) time.

**Proof:** Add a (pseudo) transaction vertex \( t' \) to \( T \) with out-edges \( t' \rightarrow c_i \) to every \( c_i \in X \). With each vertex \( v \in C \cup T \), associate two values, \( V(v) \) and \( \text{tag}(v) \). Set \( \text{tag}(T_j) = 1 \) for each sink transaction vertex \( T_j \), and set all the remaining \( V \) and \( \text{tag} \) values to 0. Following this, execute:

1. **while** \( \text{tag}(t') = 0 \) **do**
   
   (a) Choose vertex \( v \) with \( \text{tag}(v) = 0 \) and all successor vertices \( u \) with \( \text{tag}(u) = 1 \).
   
   (b) Set the value of \( V(v) \) to \( \max_{u \rightarrow v \in E}(V(u)) \) or \( \max(S'_v(v), \min_{u \rightarrow v \in E}(V(u))) \) according as \( v \in T \) or \( v \in C \), respectively.
   
   (c) \( \text{tag}(v) := 1 \).

2. If \( V(t') \leq K \) then print “Yes” else print “No”, and stop.

At the end of loop statement, a tagged vertex, \( v \), has the value \( V(v) \) that provides the cost of the subgraph of the best CRS (in the IST sense) that is rooted at that vertex. With the use of suitable data structures, the algorithm runs in \( O(|C| + |T| + |E|) \) time. \( \square \)

### 5.3 Discussion on Selection

The significance of the complexity results is only that the optimal solutions are computationally expensive to obtain. However, as in many other situations, near optimal solutions would serve almost as well, and use could be made of various heuristics available in the literature (e.g., from [9]). Since these techniques may be application-specific, we do not explore them further in this paper. In the case of PPGs that are small in size, it may be possible to select optimal CRSs by exhaustively searches. However, it is apparent that in most cases, the selection for the CRSs would need to be done “off-line.”

Once a CRS is chosen, the question arises as to how the actions that it implies should be scheduled. It may be argued that since the transactions are likely to interact, concurrency control requirements may render the selection criteria for the CRS untenable. However, note that the intractability of the problems encountered indicate that additional criteria are not likely to make the problems any easier, and heuristic methods must be used. Therefore, we separate the two issues of selection and scheduling for a CRS.

### 6 Difficulties in Scheduling

We focus attention on the scheduling of a CRS that has already been selected. In particular, we consider the data conflicts among the transactions. We describe some of the difficulties with regard to scheduling by the means of simple examples. Note that given that we seek the prompt restoration of contingency constraints in RTDBs, the need for a significant degree of concurrency among instances of the transactions in \( T' \) is desirable. Also, unless otherwise mentioned, we assume that each transaction within a CRS executes only once.

**Example 7.** Consider (a subgraph of) a CRS shown in Figure 4. The parenthesized numbers give the values for the \( W_e \) and \( W_r \) mappings. Assume that \( c_2 \) and \( c_3 \) become inconsistent immediately after the completion of \( T_1 \). The CRS does not allow any slack time for the resolution of the inconsistency in either of these constraints, and hence, \( T_2 \) and \( T_3 \) must be scheduled immediately. The constraints \( c_4 \) and \( c_5 \) may become inconsistent immediately after \( T_2 \) and \( T_3 \) complete, respectively. Notice that neither \( c_4 \) nor \( c_5 \) have any slack time, and hence, as soon as either of them becomes inconsistent, \( T_4 \) must be scheduled. In this example, \( c_4 \) and \( c_5 \) become inconsistent within \( W_r(T_4) = 3 \) time units of each other (in fact, within 1 time unit) — but not simultaneously. Thus, if the transaction \( T_4 \) is used to resolve the inconsistencies for both the constraints, irrespective of when it is scheduled, one of the two constraints will remain inconsistent for a period greater than is permissible. Furthermore, assuming that \( c_4 \) and \( c_5 \) do not become inconsistent within
$W_{c}(T_{4})$ time units of each other, it is the case that a single execution of $T_{4}$ will not suffice to resolve both the inconsistencies. □

The following example illustrates a similar situation with regard to scheduling.

**Example 8.** Figure 5 shows a constraint vertex, $c_{1}$, that may become inconsistent due to the execution of either $T_{1}$ or $T_{2}$. Suppose that $T_{1}$ makes $c_{1}$ inconsistent, and $T_{2}$ does the same within the next $W_{c}(c_{1}) = 3$ units of time. In this situation, no matter when $T_{3}$ is scheduled, the time period for which $c_{1}$ will remain inconsistent will exceed $W_{c}(c_{1})$. □

In the examples discussed above, if the constraints have larger slack times due to larger deadlines, the scheduling problems may be alleviated. For example, if it were the case that in Figure 4, $W_{c}(c_{5}) = 4$, and in Figure 5, $W_{c}(c_{1}) >> 3$, then the scheduling of the transactions may be successfully accomplished. Large slack times are useful in other contexts as well. Before transactions begin executing, it is often the case that they proceed through a time-consuming phase of resource-acquisition. If the transactions are triggered by constraints with large slack times, the initial phase of the transactions could be accommodated by scheduling the transactions early. To a certain extent, one way to accomplish this would be to identify the constraints with large slack times, and to use the concept of nested transactions (e.g., see [8, 10]) in the following manner. The constraints that are identified with large slack times serve as triggering constraints for CRSs. The constraints with small slack times are embodied within each CRS. Thus, a CRS may be regarded as a nested transaction consisting of a collection of partially ordered (sub)transactions. Most of the CRSs may be assumed to be triggered by constraints with large slack times.

**Example 9.** Consider a PPG shown in Figure 6. We represent constraints that have been identified to have large slack times by triangular vertices. In the manner explained above, some vertices of the PPG are shown to be grouped together by the dotted outlines to form nested transactions that are denoted by $nT_{1}$, $nT_{2}$, $nT_{3}$, and $nT_{4}$. The constraint vertex $c_{1}$ may trigger instances of either one of the nested transactions $nT_{1}$ or $nT_{2}$. In $nT_{1}$, the parent transaction $T_{1}$ may spawn the child transactions $T_{5}$, $T_{6}$, and $T_{7}$ by making the constraints $c_{5}$, $c_{6}$, and $c_{7}$ inconsistent. Similarly, $nT_{2}$ has a parent transaction $T_{2}$, an instance of which may spawn child transactions $T_{8}$ and $T_{9}$. Note that an instance of $nT_{2}$ could make $c_{8}$ inconsistent, and this would trigger an instance of $nT_{4}$ which consists of the single transaction $T_{8}$. The nested transaction $nT_{3}$ has a parent transaction $T_{4}$, an instance of which may spawn just a single child transaction $T_{8}$. □

It is useful to identify the potential for concurrency among the transactions that are not ordered by the partial order within a CRS. Note that although two transactions may not affect any common constraints, it may happen that they access common data items, and hence, they may not be able to execute concurrently.

**Example 10.** Consider a PPG in which there are two edges $e_{1} ightarrow T_{1}$ and $e_{2} ightarrow T_{2}$, and assume that the PPG does not require $T_{1}$ and $T_{2}$ to execute in any particular (mutual) order. If $e_{1}$ and $e_{2}$ each mention a data item “$d_{i}$” and both the transactions $T_{1}$ and $T_{2}$ access that data item, then it is possible that the concurrent execution of the two transactions may be unacceptable. Therefore, the transactions $T_{1}$ and $T_{2}$ may have to be sequenced in some (mutual) order. □

The occurrence of problems such as those illustrated in the examples above is not peculiar to our particular formulation. They will occur in general in systems with “relative” timing constraints, and the problems must be addressed if RTDBs are to be realized. Our model exhibits these problems and serves as a tool by which they may be analyzed.

### 7 Scheduling with Simplified Assumptions

We now consider whether or not the transactions within a given CRS may be scheduled such that the (invalidated) contingency constraints can be restored within their corresponding deadlines. In this section, we study this CRS scheduling problem with the simplifying assumption that there is never any contention for data among the transactions. That is, any two transactions which are not required to execute in a particular prescribed order due to the edges in the CRS, may be executed concurrently. In Section 8, we describe the study the effects of data contention for scheduling the CRSs.
7.1 Simplifying Assumptions

We assume that each transaction in a CRS will execute once; the selection criteria for CRSs implicitly make this assumption. We also assume that there are at least as many processors as the number of concurrently executing transactions in the given CRS; this assumption will allow the examination of the effects that data contention has on the scheduling (which is considered in Section 8). Furthermore, we assume that exactly one constraint vertex, the “source” vertex, in a CRS is invalidated at the time that a CRS is invoked, and the time considerations for the CRS are made relative to the time that the source vertex is invalidated; CRSs are expected to be designed to handle, at least, the violation of a single constraint at a time. Lastly, we assume that the CRSs under consideration have constraint vertices with only one outgoing edge each. We also assume that there are at least as many processors as the number of concurrently executing transactions in the given CRS; this assumption will allow the examination of the effects that data contention having currently assumed if, for all edges $c_k \rightarrow T_i$, it were the case that $T_i$ required to access all the data items mentioned in $c_k$. (On the other hand, we do consider situations with regard to the transaction vertices having more than a single outgoing edge.)

Even with the above simplifications, the issues in scheduling are not easily resolved. While this should be evident from the scheduling examples suggested in Section 6, there are additional reasons that make it a difficult problem. Among the reasons is the fact that the slack time available for a transaction to begin execution creates scheduling choices that may be difficult to determine, as exemplified below.

**Example 11.** Consider Figure 7 that depicts part of a CRS. Note that transaction $T_3$ must execute after both $T_1$ and $T_2$ complete their execution (since $T_3$ may execute at most once), and it should begin executing before it is too late to rectify the invalidation of either $c_3$ or $c_4$. Slack times (as described in Section 5) available for $T_1$ and $T_2$ provide choices as to when each may begin, however, these choices are affected by the requirements on $T_3$. □

7.2 Scheduling Terminology

To facilitate further discussions, we define a few additional terms as follows.

For a particular invocation of a CRS, the **trigger-time** and **end-trigger-time** of a contingency constraint $c_i$, denoted by $tt(c_i)$ and $ett(c_i)$, respectively, are the times (relative to the invocation time for the CRS) at which the constraint $c_i$ is first rendered invalid and last rendered invalid, respectively, for a particular invocation of a CRS.

The trigger-times are associated with the completion points in time of the transactions that invalidate the constraint in question. In the above definition, note that the first (last) invalidation requirement is stated to capture the earliest (latest) point in time that a particular contingency constraint is violated (there may be several transactions whose execution invalidates the constraint). Without loss of generality, assume that the trigger-time and end-trigger-time for the source vertex of a CRS is 0.

At some point in time after a contingency constraint is rendered invalid, the transaction that reinstates the constraint is invoked to begin execution. For a particular invocation of a CRS, the **start-time** of a transaction $T_i$, denoted by $st(T_i)$, is the time (relative to the invocation time for the CRS) at which the transaction $T_i$ is invoked to begin execution.

The question of establishing the “temporal feasibility” for a CRS is one of determining the requisite value of $st(T_i)$ for each transaction $T_i$ in the CRS. The $st(T_i)$ value for a transaction $T_i$ must be such that $T_i$ completes execution before the deadline passes for any constraint which $T_i$ reinstates.

**Definition 5.** The temporal feasibility problem for a CRS, $G'$, is to determine whether there exists an assignment of integral values to $st(T_i)$ for each $T_i \in G'$ such that for each edge $c_k \rightarrow T_i \in G'$, the inequalities $ett(c_k) \leq st(T_i)$ and $st(T_i) + W_T(T_i) \leq tt(c_k) + W_T(c_k)$ hold. □

7.3 An Efficient Technique

We now study whether the temporal feasibility problem for a given CRS can be solved efficiently (i.e., in time polynomial in the size of the CRS). Consider any pair of edges $T_i \rightarrow c_i$ and $c_i \rightarrow T_j$, with a common constraint vertex $c_i$, in the given CRS. To ensure that $T_j$ executes at most once, it should execute only after
every transaction that may invalidate \( c_e \) completes execution. Note that \( st(T_i) + W_x(T_i) \leq ctt(T_i) \) for every edge \( T_i \rightarrow c_e \) in the CRS (for some edge(s) \( T_i' \rightarrow c_e \), it is the case that \( ctt(T_i') = st(T_i'') + W_x(T_i'') \)). Therefore, the following single-execution inequality must hold, and in fact, it derives from the definition of temporal feasibility.

\[
st(T_i) + W_x(T_i) \leq st(T_j)
\]

Furthermore, to ensure temporal feasibility, \( T_j \) should begin executing sufficiently early in order that the deadline, \( W_x(c_e) \), on the constraint \( c_e \) be met. This latter consideration is captured by the deadline-meeting inequality, \( st(T_j) + W_x(T_j) \leq tt(c_e) + W_x(c_e) \). However, note that \( tt(c_e) \leq st(T_i) + W_x(T_i) \) for every edge \( T_i \rightarrow c_e \) in the CRS (for some edge(s) \( T_i' \rightarrow c_e \), it is the case that \( tt(c_e) = st(T_i'') + W_x(T_i'') \)). Therefore, the deadline-meeting inequality may be re-written as follows.

\[
st(T_j) + W_x(T_j) \leq st(T_i) + W_x(T_i) + W_x(c_e)
\]

For obvious reasons, in the specific case of the source vertex, denoted by \( c_0 \), and its outgoing edge, denoted by \( c_0 \rightarrow T_0 \), a single-execution inequality may be regarded as \( st(T_0) \geq 0 \), and a deadline-meeting inequality may be regarded as \( st(T_0) \leq W_x(c_0) \).

From the above discussions, note that each constraint vertex in a CRS, except for the source vertex, provides exactly one pair of inequalities for each incoming edge to it, and that together with the inequalities due to the source vertex, these are the only such inequalities. Also, note that the inequalities capture the requirements of the temporal feasibility problem. An assignment of values to the start times of the transactions in the CRS that satisfies the inequalities provides a “temporally feasible” solution to scheduling the transactions. Conversely, any feasible solution to scheduling the transactions in the CRS also satisfies the inequalities. Finally, note that the number of inequalities is upper bound by the size of the given CRS.

**Theorem 7.** In the absence of data contention, the temporal feasibility, and a temporally feasible schedule, if one exists, can both be determined efficiently for a given CRS.

**Proof:** Recast each single-execution or deadline-meeting inequality corresponding to a CRS in the following form for requisite values of \( k, j \) and constant \( C_i \geq 0 \) (to create the \( i \)th inequality).

\[
st(T_{ki}) - st(T_{ji}) \leq C_i
\]

Add a non-negative “slack” variable, \( x_i \), to the \( i \)th inequality to convert it to the following equality.

\[
st(T_{ki}) - st(T_{ji}) + x_i = C_i
\]

Hence, consider an “objective” function, \( \Sigma_i x_i \) (i.e., the summation over all the slack variables), to be minimized. This casts the temporal feasibility problem as a minimization problem (that is equivalent to finding a feasible solution as well) for a linear programming problem in the standard form (e.g., see [11]). Since there exists an efficient solution to the linear programming problem using the ellipsoidal method (e.g., see [12, 11]), the necessary result is established. □

The above result shows that in the absence of data contention, the temporal feasibility issue is not problematic. The manner in which the presence of data contention makes a difference is discussed below.

### 8 Scheduling with Data Contention

In this section, we consider the possibility of data contention — thereby discarding one of the simplifying assumptions of Section 7 (while keeping the other assumptions intact). The approach we follow to handle data contention is to revert from the predicate-based approach in the development of the PPG and CRS to a model based on syntactic considerations. That is, we use the simpler approaches for scheduling that involve serializability (e.g., see [7]) of the transactions. As we explain below, data contention is problematic with regard to efficient scheduling considerations.
8.1 Further Assumptions

The consideration of data contention indicates the need for further assumptions (which are justified on the basis of the intractability results to follow). We assume that each transaction in a given CRS accesses all data required by it in an exclusive mode. Thus, all data access may be regarded as being effected in the “action” model (i.e., both a read and a write access being achieved in the same atomic data access — e.g., see [15]). Furthermore, since all the transactions are pre-determined in a CRS, their data access patterns are assumed to be available at the time of scheduling. Therefore, we may use a conservative strict two-phase locking (2PL) mechanism (i.e., where all required locks are acquired by a transaction before its execution begins, and all locks are held until the transaction has committed — e.g., see [7]) to ensure serializable executions. Furthermore, we assume that all the locks for a transaction are acquired atomically (i.e., in a single step), and that the locks are released similarly — which ensures that deadlocks do not occur in the executions due to data contention. (Note that if transaction execution times and data access patterns are known in advance, actual locking need not be effected — i.e., the locks may be assumed to be acquired and released implicitly.) This concurrency control strategy ensures that any two transactions with overlapping data access sets do not execute concurrently. However, the concurrency control does not ensure temporal feasibility for scheduling a CRS.

8.2 Further Definitions and Observations

We use the term \( acc(T_i) \) to denote the set of data items accessed by a transaction \( T_i \). Similarly, by \( acc(T_i, T_j) \) we denote the data common to \( acc(T_i) \) and \( acc(T_j) \). We assume that \( acc(T_i, T_j) \) is available for every pair of transactions \( T_i \) and \( T_j \) in a CRS.

Disregarding temporal considerations, it is clear that two transactions \( T_i \) and \( T_j \) (that have no precedence constraints imposed by the CRS), can execute in true parallelism if, and only if, \( acc(T_i, T_j) \) is empty. Transactions \( T_i \) and \( T_j \) have some degree of data contention if \( acc(T_i, T_j) \) is non-empty, and then they must execute in some serial order with respect to each other. Therefore, for two transactions, \( T_i \) and \( T_j \), that have data contention, the time intervals \( (st(T_i), st(T_i) + W_r(T_i)) \) and \( (st(T_j), st(T_j) + W_r(T_j)) \), must be disjoint. That is, either \( st(T_i) + W_r(T_i) \geq st(T_j) \) or \( st(T_j) + W_r(T_j) \geq st(T_i) \), must hold. In fact, note that a sufficient criterion to meet temporal feasibility for a CRS, taking into account data contention, is to ensure that, besides the considerations described in Section 7, each pair of disjunctive inequalities described above is satisfied. Unfortunately, the problem can be shown to be computationally expensive as follows.

8.3 Provably Difficult Scheduling

In order to examine transaction scheduling under data contention restrictions, consider non-pre-emptive “shop” scheduling-theoretic problems (e.g., see [14, 15]). These problems assume a number of tasks, often with precedence constraints, that need to be executed on a set of processors. One of the restrictions placed on the schedules that has relevance to our problem is that each processor may execute at most a single task at any given point in time. Therefore, two tasks mapped to the same processor for execution must be ‘serialized’ with regard to each other. By regarding data sets as processors, and a suitable mapping of transactions to the data sets in terms of their \( acc \) sets, it is possible to use the shop scheduling theory in the context of determining temporal feasibility of CRFS. In particular, it is easy to demonstrate intractability results by reducing scheduling theory problems to CRS scheduling. To a lesser extent, few available scheduling-theoretic heuristics are applicable to the domain of CRFS.

**Lemma 1.** A job-shop scheduling problem may be reduced in polynomial time to the temporal feasibility problem for a CRS with data contention taken into account.

**Proof:** Consider the Job-Shop Scheduling problem (SS18 in [9]) instantiated with \( m \) processors \((P_1, P_2, \ldots, P_m)\), \( n \) jobs \((J_1, J_2, \ldots, J_n)\) with each job \( J_i \) requiring tasks \( t_{i1} \) through \( t_{im} \) to be executed in that precedence order, an assignment \( p(t) \) for task \( t \) to a processor \( P_i \) from among the processors, and an overall deadline \( D \in Z^+ \). The decision problem is whether there exists a legal non-preemptive schedule for the tasks that will meet the deadline \( D \) given the execution time, \( e(t) \), for each task \( t \).
The required reduction is achieved as follows. For each job $J_i$, create a portion of the associated CRS as shown in Figure 8. The weights for the vertices in the created portion are $W_x(T_{ij}) = e(t_{ij})$ and $W_x(c_{ij}) = D + c$ where $c$ is some safe, sufficiently large, number (i.e., we do not really impose any deadlines on the constraints for these portions). For transactions $T_{xy}$ and $T_{uv}$, designate $acc(T_{xy},T_{uv})$ as being non-empty (i.e., implying that the transactions have data conflicts), if, and only if, $p(t_{xy}) = p(t_{uv})$ for the corresponding two tasks among the jobs. An intuitive approach to this designation is to define data items $P_1$ through $P_m$, and to let $acc(T_{xy})$ for a transaction $T_{xy}$ include the value of $p(t_{xy})$ where $t_{xy}$ is the task corresponding to $T_{xy}$ in one of the portions created.

A CRS using the portions created above may now be formed as depicted in Figure 9. The portions labeled $J_i$ correspond to the portions of the CRS created above for a job $J_i$. Vertices $c_0$ and $T_0$ are needed to simultaneously trigger each $J_i$ portion; vertices $c_3$ and $T_3$ are used to provide a deadline of $D$ on the $J_i$ portions; vertices $c_2$ and $T_2$ are used to enforce the deadline on all the portions; and vertices $c_1$ and $T_1$ are present to permit the $J_i$ portions to finish earlier, if possible, than the deadline $D$. Set the following values for the vertex weights. $W_x(c_0) = W_x(T_0) = W_x(c_2) = W_x(T_2) = W_x(T_1) = 1$, $W_x(c_1) = D + c$, and $W_x(c_3) = W_x(T_3) = D + 1$. The CRS obtained can be verified to be the job-shop scheduling problem represented in terms of the temporal feasibility criteria. □

By restricting the number of tasks in each job in a job-shop scheduling environment to a constant value, the following result is obtained.

**Theorem 8.** Finding a temporally feasible schedule for a CRS with $n$ transaction vertices, with data contention taken into account, is NP-hard.

**Proof:** The Job-Shop Scheduling problem (see the proof for Lemma 1) remains NP-hard for the case that the number of processors is restricted to 3, and the number of tasks in each job is restricted to at most 2 (e.g., see [14], and the comments on Job-Shop Scheduling problem in [9]). Hence, by Lemma 1, the requisite NP-hardness is demonstrated. □

The key to the approach used in analyzing schedulability in the presence of data conflicts was to view the data as processors on which to execute the transactions. Given that scheduling theory has meager heuristics for multiprocessor scheduling, the scheduling of CRs also appears discouraging. On the other hand, for situations where there are available heuristics or special cases, scheduling corresponding CRs could avail of the scheduling theory approaches.

The results in this section show that despite several simplifying assumptions, it is the case that the presence of data conflicts can adversely affect the scheduling of the CRs. This result is to be contrasted with the efficient procedures identified in Section 7 in the absence of such contention. Since we have disregarded contention for other resource types, our analyses shows that data contention by itself poses severe problems in the case of hard real-time transaction scheduling (when “relative” time constraints are present).

### 9 Alternative Strategies and Extensions

There are several issues that need to be discussed regarding our RTDB model. In this section, we consider some issues such as the potential for concurrency, handling of multiple contingency constraint invalidations, etc. To do so, we consider, qualitatively, alternatives and extensions to a CRS-based approach to contingency handling. We use examples to stress the issues of concurrent and logically correct executions over those of temporal feasibility.

#### 9.1 Simple Transactions

One alternative to a CRS-based approach would be to simply trigger a particular transaction for each contingency constraint without taking into account the effects of executing the transaction on the temporal feasibility of the CRS as a whole. That is, transactions from a PPG may be invoked dynamically as and when required. Such an approach allows several invocations of the same transaction, and may handle more than a single contingency constraint invalidated simultaneously. However, this approach is simplistic in that it is unlikely to be able to guarantee temporal feasibility to a degree greater than that offered by the CRS approach. The analysis required in this new approach concerns the handling of constraints that are
invalidated dose together in time. The subsequent constraints invalidations that would occur as a result of executing the transactions invoked, would have to be regarded as a new set of invalidated constraints to be similarly restored.

The example below illustrates an approach for “coalescing” transactions in order to potentially improving the speed at which a constraint invalidation is resolved. In the extreme, the approach may be regarded as culminating in the creation of a CRS which is developed to handle the invalidation of a single source constraint vertex (as mentioned in Section 6).

**Example 12.** Consider a portion of the PPG as depicted in Figure 10. Let $T_1$ and $T_2$ be designed as a single transaction $T_{12}$ that handles the combined constraints $c_1$ and $c_2$ regarded as a single constraint $c'_{12}$. This transformed version of the PPG may take less time to execute (because $T_{12}$ is present) than the original version (because $T_1$ and $T_2$ are present) since the latter requires the serial execution of the un-combined transactions that have data contention. That is, in the transformed PPG, the transactions $T_1$ and $T_2$ may execute concurrently within $T_{12}$. □

The approach suggested in Example 12 may be expected to execute faster for a particular (source) constraint vertex invalidation, but it may reduce concurrency when more than a single (source) constraint vertex gets invalidated close together in time. The main reason is that a “large” transaction obeying conservative strict 2PL (as described in Section 8) would adversely affect concurrency as discussed in the following example.

**Example 13.** Consider transactions $T_{1x}$, $T_{2x}$, $T_{1y}$, and $T_{2y}$ (to interpret the nomenclature: $T_{iu}$ represents a subtransaction $T_i$ that accesses a data item $u$), such that $\text{acc}(T_{1x}, T_{2x})$ and $\text{acc}(T_{1y}, T_{2y})$ are each non-empty, but $\text{acc}(T_{1x}) \cup \text{acc}(T_{2x})$ does not intersect $\text{acc}(T_{1y}) \cup \text{acc}(T_{2y})$. Thus, it is possible to execute one of $T_{1x}$ or $T_{2x}$ concurrently with one of $T_{1y}$ or $T_{2y}$. A situation with two CRSs, $G'_1$ and $G'_2$, is depicted in Figure 11. Let $G'_1$ contain instances of transactions $T_x$ and $T_y$ denoted by $T_{1x}$ and $T_{1y}$, respectively, and similarly for $G'_2$. The data accessed by the transactions $T_x$ and $T_y$ are depicted in the columns, and it is assumed that $\text{acc}(T_x, T_y)$ is empty. Assume that each CRS obeys conservative strict 2PL as a whole, and that $T_{1x}$ accesses its data before any other transaction begins executing. Figure 11(a) in depicts two CRSs and their serialization and time order of execution. Figure 11(b) in depicts an additional serialization and time order for the constituent transactions of the CRSs. This implies that executing the constraint transactions of the CRSs singly allows greater concurrency. □

### 9.2 Multiple CRS Invocations

In situations where several constraints get invalidated proximally in time, several CRSs may be activated concurrently. As a result, it is necessary to determine that the ensuing executions are safe with regard to logical correctness (assuming that the temporal feasibility of the individual CRSs is already addressed). A simple way to do so is to “isolate” the executions of the individual CRSs by serialization as described below. Consider two CRSs, $G'_1$ and $G'_2$ (from the same PPG). Consider a transaction $T_{ij}$ that is common to $G'_1$ and $G'_2$. To serialize the execution of $G'_i$ before $G'_j$, it is sufficient to ensure that for all such $T_{ij}$ common to $G'_i$ and $G'_j$, it is the case that $T_{ij}$ executes for $G'_i$ prior to its execution for $G'_j$ by $W_s(T_{ij})$ units of time. A similar statement may be made with regard to serializing $G'_j$ before $G'_i$.

For more than two CRSs, consider an undirected graph whose nodes consist of the multiple invoked CRSs. An edge $(G'_i, G'_j)$ is created and labeled $T_{ij}$ if $G'_i$ and $G'_j$ have a common transaction $T_{ij}$. It is not difficult to see that if directions can be assigned to the edges to create a directed acyclic graph such that for each directed edge $(G'_i, G'_j)$ labeled $T_{ij}$, it is the case that $G'_i$ is serialized before $G'_j$ with respect to $T_{ij}$ as described above, then the CRSs are isolated as desired. We simply note that the associated problems regarding temporal feasibility would be at least as hard as in the case of a single CRS.

### 10 Discussions and Conclusions

We have presented an approach for modeling and analyzing systems designed for hard real-time database applications. Our approach is suited to situations where a crisis may occur, and triggered corrective transactions need to be executed with guaranteed execution time characteristics.
Our approach should co-exist with transaction executions for normal operating conditions, and we describe a requisite software architecture as illustrated in Figure 12. We assume that the “time-constrained concurrency control” module effects scheduling suitable for RTDB transactions using techniques described in the literature (e.g., see [16, 17, 18, 1, 2]). Furthermore, we assume that the module will abort the execution of any “normal” transactions in the event that transactions get invoked by the “activation module” — which is consistent with a high priority given to resolving crisis situations. The activated transactions module should embody the approach proposed in this paper. It must monitor the database state and trigger the necessary corrective actions. The module stores the transactions to be triggered, and their schedulability analyses. Note that the read-only access shown is to detect crisis situations rapidly, and the access is not affected by the concurrency control. Therefore, the activation module may observe inconsistent states of the database (which is not problematic since the triggered transactions are expected to check the invalidation of the contingency constraints). However, this may deteriorate normal performance since the corrective transactions may be triggered unnecessarily. By relinquishing the read-only access, this impact on performance can be avoided at the cost of detecting crisis situations less rapidly. The activation module may use technology developed for active databases, or rule-based real-time systems, to achieve the necessary activations (e.g., see [6]).

Our model described in this paper is based upon deadlines associated with states of the database characterized by the invalidation of certain “contingency” constraints. We have demonstrated that finding optimal strategies for restoring the database to states consistent with the contingency constraints, is computationally expensive in general. This negative result does not preclude the practical use of our model. Rather, it indicates that heuristics are required, and that the selections need to be done “off-line.”

Using our model, we have described some of the key problems that arise in scheduling transactions in an environment with hard real-time constraints. In particular, we have shown that the presence of relative timing constraints and data conflicts among the transactions together change the computational complexity for the deadline-constrained scheduling problem. Our results indicate that scheduling analysis would need to be done “off-line,” and that heuristics are needed even assuming that accurate execution times for the transactions are available.

While our research and results are developed relative to our specific model, the problems studied and the solutions proposed are quite general, and would need to be addressed for most hard real-time database applications.
References


Nandit Soparkar

Nandit Soparkar is an Assistant Professor in the Department of Electrical Engineering and Computer Science at the University of Michigan at Ann Arbor. He received a B.Tech degree in Computer Science & Engineering from the Indian Institute of Technology, Delhi, and the MS and PhD degrees in Computer Sciences from the University of Texas at Austin. Previously, he has worked at the AT&T Bell Laboratories at Murray Hill. His research interests include database systems, transaction management, real-time systems, distributed computing, and memory hardware architectures. He is a member of IEEE and ACM.
Hank Korth is a Vice President of Panasonic Technologies, Incorporated and the Director of the Matsushita Information Technology Laboratory in Princeton, New Jersey. Previously, he was an Associate Professor in the Department of Computer Sciences at the University of Texas at Austin and a research staff member at the IBM T.J. Watson Research Center. He received an AB degree in Mathematics from Williams College in 1977, and a PhD in Computer Science from Princeton University in 1981. His research interests include transaction processing, concurrency control, multimedia information systems, real-time database systems, relational databases, object-oriented databases, and distributed systems. He is co-author of the textbook *Database Systems Concepts.*
Abraham Silberschatz

Abraham Silberschatz is an endowed professor in the Department of Computer Sciences of the University of Texas at Austin, specializing in the area of concurrent processing. His research interests include operating systems, database systems, and distributed systems.

Silberschatz received the PhD degree in Computer Science from the State University of New York, Stony Brook, in 1976. He joined the UT Austin faculty as an Associate Professor in 1980, earned the rank of full Professor in 1984, and had received the endowed professorship in 1988. Silberschatz has been on leave at AT&T Bell Laboratories since January 1993 where he is currently serving as a senior research scientist. In addition to his academic position, Silberschatz served as an advisor for the National Science Foundation and as a consultant for several private industry companies. He is active in the field of databases and has served in the program committees and as general conference chair for a number of prestigious symposia in this area.

Silberschatz is a recognized educator, author, and researcher. He is a recipient of the IEEE Computer Society Outstanding Paper Award for the article “Capability Manager” which appeared in IEEE Transactions on Software Engineering. His writings have appeared in numerous ACM and IEEE publications and he is co-author of two well known textbooks — Operating System Concepts and Database System Concepts. He is a member of the IEEE and ACM.
List of Captions

Figure 1: An example PPG
Figure 2: A PPG to exhibit the NP-hardness of TUAP
Figure 3: Subgraphs of a PPG to exhibit the NP-hardness of PUAP
Figure 4: A scheduling example
Figure 5: Another scheduling example
Figure 6: Nesting in a PPG
Figure 7: Slack time creates choices
Figure 8: CRS portion for each job \( J_i \)
Figure 9: CRS created for all the jobs
Figure 10: Constraints and transactions considered together
Figure 11: Several CRSs executing concurrently
Figure 12: Overview of a centralized software architecture
The dotted outline shows a DAG subgraph

Figure 1: An example PPG
Figure 2: A PPG to exhibit the NP-hardness of TUAP

Figure 3: Subgraphs of a PPG to exhibit the NP-hardness of PUAP
Figure 4: A scheduling example

Figure 5: Another scheduling example
The dotted lines outline nested transactions.

Figure 6: Nesting in a PPG

Figure 7: Slack time creates choices
Figure 8: CRS portion for each job $J_i$

Figure 9: CRS created for all the jobs

Figure 10: Constraints and transactions considered together
Figure 11: Several CRSs executing concurrently

Figure 12: Overview of a centralized software architecture