Pergamon

Information Systems Vol. 23, No. 7, pp. 437–460, 1998

Printed in Great Britain
0306-4379/98 $19.00 + 0.00

ACCESSING EXTRA-DATABASE INFORMATION: CONCURRENCY CONTROL AND CORRECTNESS†

NARAIN GEHANI[1], KRITHI RAMAMRITHAM[2], JAYAVEL SHANMUGASUNDARAM[3] and ODED SHMUELI[4]

[1]Bell Laboratories, 600 Mountain Ave, Murray Hill, NJ 07974, USA
[2]Department of Computer Science and Engineering, Indian Institute of Technology, Powai Mumbai 400076, India
[3]Department of Computer Sciences, University if Wisconsin-Madison, Madison, WI 53705, USA
[4]Computer Science Department, Technion, Technion City, Haifa 32000, Israel

(Received 15 July 1996; in final revised form 6 May 1998)

Abstract — Traditional concurrency control theory views transactions in terms of read and write operations on database items. Thus, the effects of accessing non-database entities, such as the system clock or the log, on a transaction's behavior are not explicitly considered. In this paper, we are motivated by a desire to include accesses to such *extra-data* items within the purview of transaction and database correctness. We provide a formal treatment of concurrency control when transactions are allowed access to extra-data by discussing the inter-transaction dependencies that are induced when transactions access extra-data. We also develop a spectrum of correctness criteria that apply when such transactions are considered and outline mechanisms to enforce these criteria. Furthermore, we show that allowing databases to view data which has been traditionally kept hidden from users increases the database functionality and in many cases can lead to improved performance. Copyright ©1998 Elsevier Science Ltd

Key words: Transactions, Concurrency Control, Correctness Criteria

1. INTRODUCTION

Traditional concurrency control theory views transactions in terms of reads and writes on database items. Our goal in this paper is to examine the concurrency and correctness issues that arise when we take into consideration all the data that is used or can be used by transactions, not just what is in the database. Roughly speaking, "extra-data" implies information that is not part of an enterprise's database schema, but is nevertheless useful and may affect what ultimately appears in the "proper database". Such data lives in the shaded zone between the database and the application software; it is our goal to shed some light on its use and the ensuing implications.

Thus, for example, we are concerned with data that has always been accessed by the transactions but not included traditionally in serializability theory, e.g., the system clock, communication channels, and scratch-pads. We also examine information that can be useful for transactions, such as those that are maintained by lock and recovery managers, e.g., the log and information about transactions waiting for locks (some systems already allow log accesses for tuning and control purposes).

Allowing databases to view such "extra-data" can increase database functionality and can lead to improved performance. For example, if transactions were allowed to access the log, they could answer queries about the operations performed by previously committed transactions. In this case, the functionality of transactions is enhanced because they access an extra-data item (the log). Also, with access to data such as the predicted behavior of other transactions (e.g., their expected execution times or data access patterns) and transaction management information (e.g., the number of transactions waiting to perform operations on a particular object) transactions can be designed for improved performance. For example, knowing how many transactions are waiting for a particular data item may allow a transaction to consider alternatives which are likely to reduce its response time.

†Recommended by Amr EL Abbadi

We elaborate on the benefits of accessing extra-data by (1) showing that there are in fact a number of instances where extra-data can be viewed and manipulated as first-class data items; (2) examining the properties of such extra-data from the viewpoint of the transactions and their correctness; and (3) providing a formal treatment of concurrency control when transactions are allowed access to extra-data.

We use the term *database* with its traditional meaning and use the term *extended database* to refer to the database plus the extra-data objects. We introduce an important new concept, that of *extra-data independent transaction programs*. Intuitively, transactions executing these programs perform correct database state transitions, in the conventional sense, despite their extra-data accesses. We treat four main correctness requirements: (1) extra-data independence of transaction programs (2) serializability over the extended database (3) serializability only over the database and (4) other application specific correctness criteria. For each of the above, we illustrate its usefulness and flexibility in achieving user's goals. This analysis sheds light on the role, usefulness and pitfalls in using extra-data.

In many databases, in accordance with protection and security considerations, transactions are allowed access to data on a need-to-know basis. Clearly, when transactions are allowed access to information that is usually within the purview of the transaction management system, the protection and security ramifications of such accesses must also be examined. This, however, is outside the scope of this paper.

The rest of the paper is organized as follows. Section 2 contains examples of extra-data and a discussion of the characteristics of extra-data. Section 3 has a brief introduction to the formalism used to describe the correctness properties of transactions accessing extra-data. Section 4 introduces the concept of extra-data independent transaction programs. Section 5 illustrates the different correctness notions with concrete examples and Section 6 compares the correctness notions. Section 7 addresses some practical issues concerning the correctness notions. Section 8 summarizes the paper and discusses outstanding issues.

2. EXTRA-DATA: CHARACTERISTICS AND CORRECTNESS

In this section we first give an informal definition of extra-data and discuss the characteristics of extra-data. Motivation for letting transactions access extra-data is also provided via several examples which show that potential for improvement in functionality and performance exist when transactions are allowed access to extra-data.

2.1. Characteristics of Extra-Data

Extra-data can be defined as data that is typically not considered to be part of the database. It can be classified into four categories:

1. Data values modified by some entity outside the database system. The system clock is a prime example of this.

2. Data values modified by the transaction management system in response to some request initiated by a transaction. Information about waiting transactions, concurrency control information such as the serialization graph, and the log are examples here.

3. Data values pertaining to (and modified by) the transactions themselves. Estimates of a transaction's (remaining) execution time and data requirements are examples.

4. Data private to a specific set of transactions. A scratch pad used by a set of cooperating transactions to coordinate their activities is an example of this case.

Figure 1 depicts the components of an extended database. The shaded area denotes the extra data. Note that not all data in the first two categories may be considered as extra-data since only those that can be accessed by transactions are considered to be extra-data.

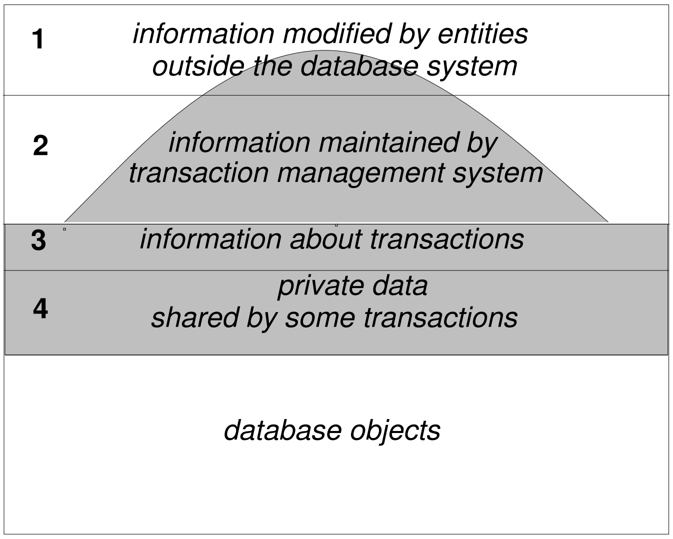

Fig. 1: The Extended Database

It is important to note that unlike database items, extra-data items need not be persistent and cannot always be *exclusively* accessed (read and updated) by user transactions.

Clearly, updates to a particular type of extra-data will be restricted based on the type. For instance, whereas no transaction can update "system updated" or "transaction-management updated" extra-data, a transaction might be allowed to update extra-data pertaining to itself and not others. These observations have certain important implications for concurrency control. For instance, while it may be possible to delay the writing of extra-data items belonging to categories (2) and (3) for concurrency control reasons, this is not possible for extra-data items in category (1). Such delays must be carefully evaluated for their effects on other transactions. For instance, if transactions' commitment or abortion could be delayed (for correctness reasons) purely because of their access to extra-data, performance can degrade. Thus, it is important to weigh the pros and cons of transactions' access to extra-data before their use is permitted in general. Our goal in this paper is to understand the concurrency control and correctness issues related to such accesses to extra-data since in reality transactions *do* access data outside the database.

Given a database system that allows transactions to access extra-data, the following additional questions become relevant:

- What are the operations defined on extra-data objects?
- What are the semantics of these operations? In particular, how do the semantics of the operations defined on an object affect the transactions that invoke these operations?
- What are the correctness requirements imposed on the transactions when they access these extra-data objects? We will focus on two types of correctness-related issues corresponding to what is referred to as safety and liveness in concurrent systems:
  - When transactions are allowed to access extra-data, what type of guarantees can be provided about the values returned by transactions and about the state of the database when they terminate?
  - When transactions access extra-data, will it affect their liveness, namely, termination, properties?

    Under normal circumstances, every transaction will terminate, i.e., abort or commit. Clearly, this property of transactions must be preserved even when they access extra-data.

These questions are answered in the following sections. In preparation for that, we now present some examples of extra-data and give a simple illustration of the correctness implications of accessing extra-data.

2.2. Examples of Extra-Data

- *The System Log.* Traditional database systems hide recovery data from the user. Eliminating this restriction, that is, storing such data as just another object that can be accessed by any user has many benefits [12, 4]. Users can pose queries on the log that cannot be specified in traditional database systems and queries that were not envisioned by the system designers. For example, (1) Find all users issuing transactions that changed item x between Jan 1, 1993 and Jan 31, 1993. (2) Which transactions are most likely to run on the last business day of the month?
- *Predicted Transaction Behavior.*
  - *Knowledge of the set of possible values a transaction will write to a data item.* If a transaction knows the set of possible values it will write to a data item, it can *proclaim* this to other transactions, which may then be in a position to proceed without waiting for this transaction to relinquish its lock on the data item [7].
  - *Estimate of execution time and data requirements of transactions.* These will be useful for dealing with transactions in real-time database systems [11]. For example, if it is possible for a transaction to realize that there is not enough time left for it to complete execution or that a feasible schedule is not possible, it could instead invoke an alternative, with a lower computational requirement.
- *Concurrency Status of Data Items.* Knowing how many transactions are waiting to access a data item will allow a transaction to consider alternatives that are likely to reduce its response time.
- *Information about Waiting Transactions.* In conjunction with the previous item, access to information about waiting transactions can avoid/prevent deadlocks. For example, a transaction can check before it performs an operation on a data item whether the operation can proceed without delay. If not, and if the transaction is willing to wait, it will check for possible deadlocks among waiting transactions and if a deadlock is possible, the transaction can take alternative actions.
- *The System Clock.* Many transactions possess functionality that depends on the time at which they execute. Examples occur in business applications and in real-time systems.
- *Limited Information Sharing.* Consider two transactions, executing on behalf of two designers who cooperate while making changes to a design object. They maintain a scratch-pad in which they keep notes about changes made by each. A designer makes an intended change only if the other has not already made it. This scratch pad is accessible only by these designers and is not part of the database, i.e., it can be seen as an extra-data object.

2.3. Accessing Extra-Data - Correctness Implications

Both the control flow within a transaction as well as a transaction's data manipulation properties may be modified by its extra-data accesses. What a transaction is allowed to do with the extra-data that it reads depends on the correctness requirements imposed on the transactions. For instance, a transaction can use it to optimize its actions – but we may require that the semantics of the transformation performed by the transaction be independent of the optimization. This may require including the extra-data items within the scope of concurrency control, as will be discussed later.

More liberal is the case where transactions are allowed to make changes to the database and return values that depend on the values of the extra data. In this case again, it may be necessary to include the extra-data items within the scope of concurrency control, as will be evident from the example given below.

Consider two transactions t_1 and t_2 which access Joe_Status (a database item) and the system clock (an extra-data item). Both t_1 and t_2 execute the following program.

```
read(Time);
read(Joe_Status);
print ("Joe is" Joe_Status "at time" Time);
```

Thus both transactions t_1 and t_2 read the current time, check Joe's status (whether he is dead or alive) at that time and report it to the user. If we now consider another transaction t_3 which changes Joe_Status from "alive" to "dead" (represented as $write_{t_3}(Joe\_Status)$ below) and a system event which periodically updates the clock (represented as $write_{sys}(Time)$ below), then the following interleaving of actions is possible.

$$read_{t_1}(Time); write_{sys}(Time); read_{t_2}(Time); read_{t_2}(Joe\_Status); write_{t_3}(Joe\_Status);$$
$$read_{t_1}(Joe\_Status); print_{t_1}; print_{t_2};$$

If t_1 read the clock at 3:00 am and the system increments the time by 1 minute during each update, then we would get the following output from transactions t_1 and t_2.

t_1: Joe is dead at time 3:00 am

t_2: Joe is alive at time 3:01 am

The above schedule is serializable over the database items (the serialization order is t_2, t_3, t_1) and it is thus apparent that the extra-data item (the clock) should be brought within the scope of concurrency control. However, the clock is not a normal data item in the sense that it is updated by a non-transaction entity. In fact, in the above example, the operations performed by transactions on the clock do not by themselves conflict with each other but conflict *indirectly* through the system update to the clock. Thus new mechanisms for concurrency control should be devised to deal with such properties of extra-data. These issues are treated in detail in subsequent sections.

3. PRELIMINARIES AND DEFINITIONS

In this section we introduce a simple formalism based on the ACTA transaction framework [2]. We also define some of the terms used in the rest of the paper and give formal definitions of certain correctness properties involved when transactions access extra data.

3.1. Transactions, Operations, Events, and Histories

A transaction accesses and manipulates objects in the extended database, i.e., objects in the database as well as extra-data objects, by invoking operations specific to individual objects. The operations invoked by a transaction t are determined by its corresponding transaction program.

Definition 1 A transaction t is an *instance* of a transaction program tp if the operations performed by t are determined by the program tp.

It is assumed that operations are atomic and that an operation always produces an output (return value), that is, it has an outcome (condition code) or a result. The result of an operation on an object depends on the current state of the object. For a given state s of an object, we use $return(s, p)$ to denote the output produced by operation p, and $state(s, p)$ to denote the state produced after the execution of p. We also assume that when an operation is performed on an object, no other object is affected.

Definition 2 Invocation of an operation on an object is termed an *object event*. The type of an object defines the object events that pertain to it. We use $p_t[ob]$ to denote the object event corresponding to the invocation of the operation p on object ob by transaction t. (For simplicity of exposition assume that a transaction does not contain multiple p's. When "what the ob is" is clear, we will simply say p_t.)

Definition 3 Committing or aborting a transaction and committing or aborting an operation performed by a transaction are all termed as *transaction management events*. $commit_{t_i}$ and $abort_{t_i}$ denote the commit and abort of transaction t_i respectively. $commit[p_{t_i}[ob]]$ and $abort[p_{t_i}[ob]]$ denote the commit and abort of operation p performed by transaction t_i on object ob, respectively. The effects of an operation p invoked by a transaction t_i on an object ob are made permanent in the database when $p_{t_i}[ob]$ is committed and are obliterated when $p_{t_i}[ob]$ is aborted.

Definition 4 A *history* is a partially ordered set of events invoked by transactions. Thus, object events and transaction management events are both part of a history $\mathcal{H}$. The set of events invoked by a transaction t is a partial order denoting the temporal order in which the related events occur in the history. The transaction's partial order is consistent with the history's partial order.

We write $(\epsilon \in \mathcal{H})$ to indicate that the event ϵ occurs in a history $\mathcal{H}$. $\rightarrow$ denotes precedence ordering in a history $\mathcal{H}$ and $\Rightarrow$ denotes logical implication.

3.2. Conflicts between Operations

Definition 5 Let $\mathcal{H}^{(ob)}$ denote the projection of the history with respect to the operations on ob[†].

Two operations p and q on an object *ob conflict* in a state produced by $\mathcal{H}^{(ob)}$, denoted by $conflict(\mathcal{H}^{(ob)}, p, q)$, if

$$\begin{array}{rcl} (state(p \circ \mathcal{H}^{(ob)},\ q) & \neq & state(q \circ \mathcal{H}^{(ob)},\ p)) \quad \vee \\ (return(\mathcal{H}^{(ob)},\ q) & \neq & (return(p \circ \mathcal{H}^{(ob)},\ q)) \ \vee \\ (return(\mathcal{H}^{(ob)},\ p) & \neq & (return(q \circ \mathcal{H}^{(ob)},\ p)) \end{array}$$

($\circ$ denotes functional composition.) Thus, two operations conflict if their effects on the state of an object are not independent of their execution order (first clause) or their return values are not independent of their execution order (second and third clauses). Operations p and q are said to *conflict* on object *ob*, denoted $conflict(ob, p, q)$ if there exists a history $\mathcal{H}$ such that $conflict(\mathcal{H}^{(ob)}, p, q)$. The notion of conflict will be used to define different types of correctness when transactions access data as well as extra-data concurrently.

3.3. Atomic and Non-Atomic Transactions

Traditional databases deal predominantly with atomic transactions. Thus, when a transaction commits, all the operations performed by the transaction commit and when a transaction aborts, all the operations performed by the transaction abort. However, as we saw in the previous section, when transactions modify the extra-data items too, we have to take into account the fact that the execution of a transaction may not be atomic over the extended database. While the failure recovery semantics of extra-data items accessed by transactions is usually comparable to that of database items (for example, updates to the page space map tables, which are extra-data items, are normally logged [9], see Section 5.1.3), the recovery of extra-data items after transaction aborts may not be possible. The reason for this is twofold. In some cases, when a transaction aborts, we may require that the operations performed by (or due to) the transaction on the extra-data items be persistent (for example, when the log is an extra-data item). In other cases, the extra-data item may also be modified by entities outside the transaction management system.

Thus, it is possible that certain operations on extra-data items are committed even when the invoking transaction aborts. This is formalized below.

Let $\mathcal{O}_t$ be the set of operations performed by a transaction t. This set is partitioned into three disjoint subsets, $\mathcal{U}_t$, $\mathcal{P}_t$ and $\mathcal{L}_t$. $\mathcal{U}_t$ consists of those operations performed by t whose effects are to be undone when t aborts. $\mathcal{P}_t$ consists of those operations performed by t whose effects are to

[†] $\mathcal{H}^{(ob)} = p_n \circ ... \circ p_2 \circ p_1$, indicates both the order of execution of the operations, (p_i precedes p_{i+1}), as well as the functional composition of operations. Thus, a state s of an object produced by a sequence of operations equals the state produced by applying the history $\mathcal{H}^{(ob)}$ corresponding to the sequence of operations on the object's initial state $s_0 (s = state(s_0, \mathcal{H}^{(ob)}))$. For brevity, we will use $\mathcal{H}^{(ob)}$ to denote the state of an object produced by $\mathcal{H}^{(ob)}$, implicitly assuming initial state s_0.

be made persistent even if t aborts. $\mathcal{L}_t$ consists of those operations performed by t on data items which exist only for the lifetime of t (e.g., operations on a scratch pad). The operations in $\mathcal{L}_t$ have no impact on the atomicity of t since the data items on which they are performed cease to exist on t's termination. The axioms relating the commit and abort of t to the commit and abort of the operations in $\mathcal{U}_t$ and $\mathcal{P}_t$ are given below.

- $\exists ob \exists p(commit[p_t[ob]] \in \mathcal{H} \wedge p_t[ob] \in \mathcal{U}_t) \Rightarrow commit_t \in \mathcal{H}$
  If an operation in $\mathcal{U}_t$ is committed, t must commit.
- $commit_t \in \mathcal{H} \Rightarrow \forall ob \forall p(p_t[ob] \in \mathcal{H} \wedge p_t[ob] \in \mathcal{U}_t \Rightarrow commit[p_t[ob]] \in \mathcal{H})$
  If t commits, all the operations in $\mathcal{U}_t$ are committed.
- $\exists ob \exists p(abort[p_t[ob]] \in \mathcal{H} \wedge p_t[ob] \in \mathcal{U}_t) \Rightarrow abort_t \in \mathcal{H}$
  If an operation in $\mathcal{U}_t$ is aborted, t must abort.
- $abort_t \in \mathcal{H} \Rightarrow \forall ob \forall p(p_t[ob] \in \mathcal{H} \wedge p_t[ob] \in \mathcal{U}_t \Rightarrow abort[p_t[ob]] \in \mathcal{H})$
  If t aborts, all operations in $\mathcal{U}_t$ are aborted.
- $\forall ob \forall p(p_t[ob] \in \mathcal{H} \wedge p_t[ob] \in \mathcal{P}_t \Rightarrow commit[p_t[ob]] \in \mathcal{H})$
  An operation in $\mathcal{P}_t$ is committed (independent of the commitment of t).

If $\mathcal{P}_t = \phi$, then these axioms reduce to those of traditional atomic transactions [2]. The above axioms take care of the case in which it may not be possible to undo certain operations performed by (or due to) a transaction even if it aborts. Such operations are added to the set $\mathcal{P}$ of the invoking transaction.

3.4. Serializability

In traditional databases, serializability and, in particular, *conflict serializability*, is the well-accepted criterion for correctness. We first define serializability formally since it forms the basis for the correctness notions discussed here. In what follows, we assume the update-in-place model.

Definition 6 An *axiom* $\mathcal{A}$ which induces (binary) $\mathcal{C}$ relationships between a set of transactions $\mathcal{T}$ in a history $\mathcal{H}$ is a first order logic expression of the form:

$$\forall t_i \forall t_j (t_i \in \mathcal{T} \wedge t_j \in \mathcal{T} \wedge \mathcal{X}(t_i, t_j) \Rightarrow t_i \mathcal{C} t_j)$$

where $\mathcal{X}$ is a first-order logic expression, possibly involving $\mathcal{H}$, in which the variables t_i and t_j are free. A relationship $t_i \mathcal{C} t_j$ is *induced* between two transactions t_i and t_j due to the axiom $\mathcal{A}$ iff $\mathcal{X}(t_i, t_j)$ is true.

Let $\mathcal{D}$ be the set of database items and let $\mathcal{C}$ be the conflict (binary) relation on transactions in $\mathcal{T}$, where $\mathcal{T}$ is the set of all transactions in the history $\mathcal{H}$.

Definition 7 $\forall t_i \forall t_j (t_i \in \mathcal{T} \wedge t_j \in \mathcal{T} \wedge t_i \neq t_j \wedge$

$$\exists ob \in \mathcal{D}\ \exists p, q\ (conflict(ob, p, q) \wedge (p_{t_i}[ob] \rightarrow q_{t_j}[ob]) \wedge (commit_{t_i} \in \mathcal{H}) \wedge (commit_{t_j} \in \mathcal{H})) \Rightarrow t_i \mathcal{C} t_j)$$

The $\mathcal{C}$ relation captures (a) the fact that two committed transactions have invoked conflicting operations on the same object and (b) the order in which they have invoked the conflicting operations. Consequently, the $\mathcal{C}$ relation captures direct conflicts between committed transactions in a history. The fact that a serialization order is acyclic is stated by requiring that there be no cycles in the $\mathcal{C}$ relation. This is formalized below.

Definition 8 $\mathcal{H}$, the history of events relating to transactions in $\mathcal{T}$, is *(conflict) serializable* iff $\forall t \in \mathcal{T}, \neg(t\ \mathcal{C}^*\ t)$ where $\mathcal{C}^*$ is the transitive-closure of $\mathcal{C}$.

Suppose t_j has done a write on an object and then t_i does a read on the same object. Then, $(t_j\ \mathcal{C}\ t_i)$. Also, if we desire failure atomicity, then if t_j aborts then t_i must also abort. Thus, abort dependencies [2] between transactions may also form due to conflicting operations.

However, when we extend the notion of serializability to the extended database, we have to take into account the fact that the execution of a transaction may not be atomic over the extended database. Thus, even the committed operations invoked by aborted transactions will be involved in determining the serialization ordering.

Let $\mathcal{E}$ be the set of extended database items and let $\mathcal{C}$ be the (binary) conflict relation on transactions in $\mathcal{T}$, where $\mathcal{T}$ is the set of all transactions in the history $\mathcal{H}$.

Definition 9 $\forall t_i \forall t_j (t_i \in \mathcal{T} \wedge t_j \in \mathcal{T} \wedge t_i \neq t_j \wedge$

$$\exists ob \in \mathcal{E}\ \exists p, q\ (conflict(ob, p_{t_i}[ob], q_{t_j}[ob]) \wedge (p_{t_i}[ob] \rightarrow q_{t_j}[ob]) \wedge$$

$$(commit[p_{t_i}[ob]] \in \mathcal{H}) \wedge (commit[q_{t_j}[ob]] \in \mathcal{H})) \Rightarrow t_i \mathcal{C} t_j)$$

The $\mathcal{C}$ relation, in this case, captures the ordering in which two transactions invoked (or caused to invoke) two conflicting committed operations on the same object. The definition of serializability remains the same, where acyclicity is ensured with respect to this $\mathcal{C}$ relation.

As mentioned earlier, an application may desire correctness properties that are weaker than serializability when an extended database is used. A discussion of such correctness properties can be found in [10] and some examples of their application to extra-data are presented in Section 5.4.

For the purposes of this paper, given an extended database, we can consider (1) serializability of accesses to data items in the extended database (2) serializability of accesses to data items in the database. That is, in (2), a cycle of $\mathcal{C}$ relationships formed by accesses to the database as well as by accesses to extra-data will be ignored since only the set of $\mathcal{C}$ relationships induced by the items in the database need be acyclic.

4. EXTRA-DATA INDEPENDENT TRANSACTION PROGRAMS

The reading and writing of extra-data may in principle affect both the operations performed by a transaction as well as its flow of control. In this section, we formalize the notion of when transaction programs are "dependent" on extra-data items. We show that by restricting the set of allowable histories, transaction programs can be made extra-data independent. As before, this restriction on allowable histories is specified by a set of axioms which induce $\mathcal{C}$ relationships between transactions.

Let $\mathcal{K}$ be a set of axioms which specify the conditions under which $\mathcal{C}$ relationships are induced between transactions. For example, the set $\mathcal{K}$ could contain the axiom which specifies that $\mathcal{C}$ relationships are introduced due to conflicts in a committed history (see Definition 7). In general, the axioms may involve both database and extra-data items and hence $\mathcal{C}$ relationships may be induced between transactions due to extra-data accesses also.

Definition 10 A history $\mathcal{H}$ relating to transactions in $\mathcal{T}$ is said to be *serializable with respect to a set of axioms* $\mathcal{K}$ if $\forall t \in \mathcal{T}, \neg(t\mathcal{C}^* t)$, where $\mathcal{C}$ is the relation which consists of all and only the relationships induced between transactions in $\mathcal{T}$ due to the axioms in $\mathcal{K}$.

Consider a set $\mathcal{K}$ which contains the axiom in Definition 7 (and possibly other axioms). Then, any history serializable with respect to the set of axioms $\mathcal{K}$ will also be conflict serializable because the set of $\mathcal{C}$ relationships induced due to the latter is a subset of the $\mathcal{C}$ relationships induced due to the former.

Definition 11 A *conventional* transaction program is one whose transaction instances do not access extra-data items.

Definition 12 A set of transaction programs $\mathcal{TP}$ is said to be *extra-data independent* with respect to a set of axioms $\mathcal{K}$ if for each transaction program tp in $\mathcal{TP}$, there exists a conventional transaction program tp' (let $\mathcal{TP}'$ be the set of all such tp') such that for all legal histories $\mathcal{H}$ serializable with respect to the set of axioms $\mathcal{K}$,

1. the transformation performed on the database is the same as that performed by some serial history $\mathcal{H}'$ consisting of instances of transaction programs in $\mathcal{TP}'$.
2. there is a function f mapping each instance of a transaction program $tp \in \mathcal{TP}$ in $\mathcal{H}$ to an instance of its corresponding tp' in $\mathcal{H}'$.
3. the values (if any) returned to the invoker by each instance of tp and its corresponding instance of tp' (as determined by the function f) are the same. The return value of a transaction t is some function of the values returned by the operations on objects (possibly extra-data objects) invoked by t.
4. tp' satisfies the integrity constraints on the database (i.e., each tp', when run by itself, performs a correct database state transformation).

Observe that if tp is extra-data independent, when applied to a database state s, it yields exactly what tp' yields when applied to s (here the history consists of a single transaction). If tp is extra-data independent, as far as the serializability of the changes to the database is concerned, we can think of the actual transaction instance of tp as a "surrogate" for an instance of tp'.

In the above definition, we have assumed that the correctness criterion for the conventional transactions (instances of tp' s) is serializability. If we desire a weaker (stronger) correctness criterion for the conventional transactions, then the axioms in $\mathcal{K}$ with respect to which extra-data independence is defined will also be weakened (strengthened).

Henceforth, we refer to any tp' according to the above definition for tp, as $equiv(tp)$. (In general there may be a number of such tp's, for our purposes they are all equivalent.) If a set of transaction programs $\mathcal{TP}$ is *not* extra-data independent with respect to a set of axioms $\mathcal{K}$, then it is said to be *extra-data dependent* with respect to the set of axioms $\mathcal{K}$[†].

Let us consider a simple example to clarify the notion of extra-data independence. Suppose a transaction is invoked to make a single car rental reservation. It chooses among car companies based on the anticipated delay due to waits. The transaction uses extra-data as control information to find the path with the shortest waiting time:

```
1.  trans {
2.      car_rental avis = get_oid("avis");
3.      car_rental hertz = get_oid("hertz");
4.      if (num_waiting(avis) > num_waiting(hertz))
5.          hertz->reserve(...);
6.      else
7.          avis->reserve(...);
8.  }
```

This is an example of a transaction program that is *not* extra-data independent (with respect to conflict serializability over the database): either an Avis car or a Hertz car is reserved.

Consider a transaction invoked on behalf of another user who intends to reserve an Avis car in Los Angeles and a Hertz car in Houston. If we replace lines 5-7 by the following, we accomplish this while attempting to minimize the waiting time. The operations to reserve the Avis car and the Hertz car are assumed to be commutative.

```
5.          hertz->reserve(...); avis->reserve(...);
6.      else
7.          avis->reserve(...); hertz->reserve(...);
```

[†]In the rest of the paper, when the set $\mathcal{K}$ is sufficiently clear from the context, we will omit the phrase "with respect to to the set of axioms $\mathcal{K}$". Also, for the sake of clarity, when the set $\mathcal{K}$ corresponds to the set of axioms for determining conflicts over the database, the phrase "with respect to the set of axioms $\mathcal{K}$" will be replaced by "with respect to conflict serializability over the database".

This transaction program *is* an extra-data independent (with respect to conflict serializability over the database) which is equivalent to the conventional transaction program obtained by replacing lines 4-7 by the following.

hertz->reserve(...); avis->reserve(...);

Consider a serializable history (with respect to some set of axioms $\mathcal{K}$) where all transactions are instances of extra-data independent transaction programs. In hypothetically running the original transaction programs serially, extra data accesses may be different or may not exist, which could lead to a change of control flow and/or values written. But, substituting $equiv(tp)$ for each tp isolates its effect on the database state, from other facets as embodied in extra-data.

Consider a transaction program tp written in a programming language with expressive power equivalent to that of Turing Machines. It is then undecidable whether an $equiv(tp)$ exists. So, in general, we cannot delegate the task of verifying the existence of $equiv(t)$ to an automatic tool. An alternative is to have the user supply $equiv(tp)$ and an equivalence proof (with respect to a set of transaction programs $\mathcal{TP}$ and a set of axioms $\mathcal{K}$). This is similar to having the user certify a transaction program as doing a "correct" state transformation in the traditional formulation of concurrency control theory. We return to the problem of proving the extra-data independence of transaction programs in Section 7.

5. CORRECTNESS NOTIONS WITH CONCRETE EXAMPLES

It should be obvious by now that different types of correctness notions can be applied in the context of extra-data. We first list possible correctness notions and then illustrate them with concrete examples of extra-data.

1. *Ensure extra-data independence of transaction programs.*

   Given our definitions in Section 4, this implies that we should define an appropriate set of axioms $\mathcal{K}$ with respect to which transaction programs are extra-data independent. Further, we have to ensure that the $\mathcal{C}$ relationships induced between transactions due to the axioms in $\mathcal{K}$ form an acyclic relation, i.e., we have to ensure serializability with respect to the set of axioms $\mathcal{K}$. Note that serializability with respect to the set of axioms $\mathcal{K}$ may, by itself, specify a correctness criterion (like, for example, serializability over the extended database). We return to this issue in Section 6.

   Extra-data independence of transaction programs assures view serializability [1] (though not necessarily conflict serializability) of instances of the conventional transaction programs which are equivalent to the transactions which access extra data. View serializability is ensured because, as per Definition 12, we require that the net transformation performed on the database by transactions which access extra-data is the same as the net transformation performed by some serial execution of instances of the conventional transaction programs.

2. *Achieve serializability of accesses to the extended database in the presence of extra-data dependent transaction programs.*

   We need to ensure that the $\mathcal{C}$ relationships due to conflicts over accesses to the extended database are acyclic. Since transaction programs need not be extra-data independent, transactions may produce results that reflect the fact that they accessed extra data.

3. *Achieve serializability of accesses to (just) the database in the presence of extra-data dependent transaction programs.*

   Changes to the database may depend on the transactions' accesses to extra-data. Intuitively, by serializability with respect to just the database data, we mean that if transactions are run serially and whenever an extra-data access is performed in this serial execution the same values as in the actual execution are "magically" supplied, then transactions will produce the same final database state as in the actual execution. Since the extra-data conflicts are not considered in determining which transactions conflict, transactions may produce database changes that are affected by extra-data accesses. Also, this correctness criterion *does not* guarantee consistency of the extended database (assuming consistent transactions) since the extra data items can be accessed in a non-serializable fashion.

4. *Achieve some application specific correctness criterion in the presence of extra-data dependent transaction programs.*

   Some applications may desire correctness criteria in which extra-data items must be brought within the scope of concurrency control but in which requiring serializability over the extended database would result in a loss of performance (through loss of concurrency). In this case, serializability is ensured over just the database and some weaker application dependent correctness criterion is ensured over the extended database. Note that in this case, we still require serializability over the database and the weaker correctness criterion is specified only for the extended database.

Note that we are using new (extra-data independence of transaction programs) and old (serializability, weaker correctness criteria) techniques to deal with the correctness of transactions with extra-database accesses. In other words, correctness of concurrent accesses to extra data can be dealt with in ways similar to traditional concurrency control, i.e., using serializability as the correctness criterion (and possibly exploiting the semantics of operations) or using a correctness criterion weaker than serializability. In addition, in many situations, the transaction programs are required to be extra data independent. The latter is a new property.

5.1. Extra-Data Independent Transaction Programs

In this section, we discuss six examples where we require the extra-data independence of transaction programs.

5.1.1. Work Queues

Consider a transaction program tp' which performs operations op_1, op_2, and op_3, in sequence, on each data item in a set D. tp' may access database items not in D, indicated by '...' in the code below.

trans tp' {
  for all $d \in D$ $\{op_1(d); \ldots \}$
  for all $d \in D$ $\{op_2(d); \ldots \}$
  for all $d \in D$ $\{op_3(d); \ldots \}$
}

We now show how tp' can be implemented as a transaction program tp which uses extra-data items in such a way that it has the potential for better performance in a parallel environment. That is, tp' will be $equiv(tp)$.

We proceed to construct tp as follows: tp is implemented as a nested transaction program which consists of three (sub)transaction programs tp_1, tp_2 and tp_3. Each tp_i performs the equivalent of the i^{th} for loop in the text for tp'. Instances of tp_1, tp_2 and tp_3 (call them t_1, t_2 and t_3 respectively) operate in a pipelined fashion with respect to the data items in D, i.e., after t_1 does op_1 on a data item d, t_2 does op_2 on d and then t_3 does op_3 on d.

Proper control of the transactions' actions is achieved via work queues q_1 and q_2, which are extra-data items of category 4 (see Figure 1). q_1 has two operations defined on it: $insert1$ and $remove1$. Similarly for q_2. A *remove* operation on a queue blocks if the queue is empty. t_1 inserts the id of the data item on which it just completed op_1 into q_1. t_2 removes the id in the front of queue q_1 and after doing op_2, it inserts the id into q_2. t_3 removes an id from q_2 and performs op_3 on the corresponding data item. When q_1 and q_2 are empty and t_1, t_2, and t_3 have completed their ongoing operations, they all commit together. If any of them aborts, all abort. The programs for the (sub)transactions are given below.

```
trans tp1 {                 trans tp2 {                 trans tp3 {
    for all d ∈ D {             P = D;                      P = D;
        op1(d);                 while P ≠ φ {               while P ≠ φ {
        insert1(d);             remove1(d);                 remove2(d);
        ...                     op2(d);                     op3(d);
    }                           insert2(d);                 P = P − {d};
}                               P = P − {d};                ...
                                ...                         }
                                }                       }
                            }
```

Since each *remove* operation blocks if the queue on which the operation is performed is empty, we have

$\forall d \in D(insert1(d) \rightarrow remove1(d))$

$\forall d \in D(insert2(d) \rightarrow remove2(d))$

Since we want the operations op_1, op_2 and op_3 to be performed in sequence on an object in D, we define the following $\mathcal{C}$ relationships (here t_i refers to an instance of the transaction program tp_i).

$(t_1\ \mathcal{C}\ t_2)$ if $\exists d(insert1_{t_1}(d) \rightarrow remove1_{t_2}(d))$.

$(t_2\ \mathcal{C}\ t_3)$ if $\exists d(insert2_{t_2}(d) \rightarrow remove2_{t_3}(d))$.

The above axioms along with the axiom for conflicts over the database items (see Definition 7) make up the set $\mathcal{K}$ with respect to which extra-data independence is achieved. Thus transaction program tp, consisting of subprograms tp_1, tp_2, and tp_3, has the same effect as tp' on D.

What this example shows is that it is possible to realize a given transaction program in such a way that even though the implementation uses extra-data items, its behavior with respect to the rest of the transactions and the effect on the database will be the same as the original transaction program. The motivation here is to rewrite the transaction programs in a way that allows us to execute components of the transactions in parallel thereby improving the performance of the system. The queues allow the transaction components to synchronize their activities so as to achieve the desired functionality.

5.1.2. A Car Reservation Example

Consider a modified version of the car rental transaction from Section 4. Instances of the transaction program tp_i first read the data items to determine the number of free Hertz cars and Avis cars respectively. They then reserve one Hertz car and one Avis car in different orders depending on which has a larger number of free cars. This is done so that reservations are made first with the rental company most likely to run out of free cars.

```
trans {
    car_rental avis = get_oid("avis");
    car_rental hertz = get_oid("hertz");
    ...
    if (num_available(avis) > num_available(hertz))
        hertz->reserve(...);
        avis->reserve(...);
    else
        avis->reserve(...);
        hertz->reserve(...);
    ...
}
```

In this case, it is not necessary to have an exact number of the number of cars available and an approximate value would do. Thus, to improve the performance of the above transaction program, we could have a transaction program tp_k which is invoked periodically to read the number of cars available and write them to extra-data items num_avis and num_hertz. Then transaction program tp_i could be written as a transaction program tp_j which reads from the extra-data items rather than accessing the corresponding database item, thus reducing conflicts. The transaction programs tp_j and tp_k are extra-data independent (with respect to serializability over the database) in which $equiv(tp_j)$ is tp_i and $equiv(tp_k)$ is the null transaction program. Thus, the functionality of the transaction programs is not changed and hence the consistency of the database is ensured.

This example illustrates how the concept of extra-data independence can be used to allow lower degrees of isolation in concurrency control without affecting the consistency of the database or the functionality of transactions in any way. If we believe that a transaction program tp_1 does not require exact values of a data item X, then we can prove it as follows: Create a hypothetical transaction program tp_3 which periodically accesses the value of X and writes it into an extra-data item x. Now tp_1 could be modified to read the value x instead of X (call this modified transaction program tp_2). Now tp_1 does not require exact values of the data item X iff tp_2 and tp_3 are extra data independent (with respect to serializability over the database).

5.1.3. Page Space Map Tables

Page space map tables [9] are commonly used in databases to indicate the location of free space in pages. These tables are normally accessed by transactions, though they are not considered as database items and brought within the scope of correctness and concurrency control. Here, we identify the correctness criterion relevant to this extra-data item and the impact this criterion has on concurrency control.

The following operations are defined on page space map tables for freeing and reserving space in pages: *reserve(page,begin_offset,end_offset)* reserves the area of the page *page* from *begin_offset* to *end_offset*. *free(page,begin_offset,end_offset)* frees the area of the page *page* from *begin_offset* to *end_offset*.

Here, we require the extra-data independence of transaction programs accessing the page space map tables because the state of the database should not be dependent on the physical location of the database items. In addition, in order to provide for transaction roll back (in case of transaction abort), we may need to specify that a transaction can reclaim the space it freed (in the event of it aborting) so that its changes can be undone easily [9]. Thus, we require that if a transaction t_i uses up the space freed by transaction t_j, then t_i should abort whenever t_j aborts (so that the space freed by t_j is now free). Thus t_i has an abort dependency [2] on t_j.

Note that there are no $\mathcal{C}$ relationships induced due to accesses to the extra-data item (because the page space map table is not used in any way which might affect the state of the database) but instead, an abort dependency is induced. In normal implementations of the page space map tables, a transaction t_j can use the space freed by a transaction t_i only if t_i has committed [9]. Thus, this induced abort dependency is taken care of and only serializability over accesses to the database items need be ensured.

5.1.4. Proclamation Locking

Consider the following example in which 2PL is used for concurrency control. Suppose a transaction t_1 holds a write lock on a data item and another transaction t_2 desires to read that data item. Under 2PL, t_2 would be made to wait. However, suppose t_1 gives t_2 (or any other transaction) an indication of all the possible values it might write, then t_2 might be able to proceed with its computations using this information, thus increasing the degree of concurrency. This is the idea underlying proclamations [7]. t_1 proclaims the set of possible values that may be written so that transactions such as t_2 may be able to proceed without waiting. These proclamation data items can be treated as extra data items with the required correctness criterion being the extra-data independence of transactions. A detailed proof of the extra-data independence of proclamation transactions appears in [5].

5.1.5. Serializability in Multi-Databases

Another example of extra-data independent transaction programs occurs in the multi-database concurrency control scheme proposed in [3]. Here, to ensure the serializability of transactions that access multiple (autonomous) database sites, the following scheme is used. Every site has a special "ticket" that all global transactions that visit the database at that site are expected to write. In this case, the ticket is an extra-data item since only transactions visiting that site can access it. Here we require the extra-data independence of transaction programs because the execution of the transactions accessing the ticket should correspond to some serial order of the equivalent transactions not accessing the ticket.

Extra-data independence of transaction programs in achieved in this case by defining a set of axioms $\mathcal{K}$ which achieve conflict serializability over the extended database. This is because when each transaction is serialized over accesses to the tickets and the database items, the effect of the transactions on the extended database is the same as the effect of running the transaction programs in some serial order. Also, since the tickets are not read in the transaction program (they are only written to), the database state is not affected by the values of the extra-data items. Thus, an execution of transactions accessing the extended-database performs the same transformation on the database as the execution of conventional transactions that do not access the extra-data items. A formal proof of the extra-data independence of transaction programs for this example is given in Section 7.1.

5.1.6. Audit Queries on Logs

Logs of operations performed by transactions are normally written for transaction recovery purposes [1]. These logs can also serve as an audit trail to track the progress of transactions in the transaction management system. In this example, we explore the use of the log as an audit trail.

Consider the case where we have two disjoint sets of transaction programs, say $\mathcal{TP}_\infty$ and $\mathcal{TP}_\in$, operating on the extended database in which the log is the extra-data item. $\mathcal{TP}_\infty$ is the set of transaction programs which operate on the database and write log records for each operation performed. $\mathcal{TP}_\in$ is the set of "audit" read-only transaction programs which access data items in the extended database.

In this case, we would require the extra-data independence of the transaction programs in $\mathcal{TP}_\infty$ because the transformations on the database should not be affected by the log records. This extra-data independence could be achieved by just ensuring serializability over the database, with no restrictions on updating the log (a formal proof of this is given in Section 7.1). However, for the read-only transactions in $\mathcal{TP}_\in$, we would have to ensure a correctness criterion like Serializability over the Extended Database in order to ensure consistent reads.

In this example, we have assumed that the set of transactions which read from the extra-database and which write to the extra-database are disjoint. Thus, it is easy to prove the extra-data independence of transaction programs. In Sections 5.2 and 5.4.1, we deal with examples in which transactions can both read and write log records and study the associated correctness.

5.2. Serializable Accesses to the Extended Database

Here we remove the extra-data independence requirement imposed on transaction programs, but require that the transactions be serializable with respect to the extended database. For concreteness, we consider the database log as an example of extra-data which can be accessed by transactions in the course of their execution [4]. Both committing and aborting transactions write log records. Transactions can also read the log to perform queries. We require serializability of all the data items in the extended database, in this case, the database plus the log.

Let us assume that the commitment or abortion[†] of a transaction results in the writing of a single log item[‡] containing all the relevant information about the transaction. The operation of

[†]Transactions can abort for one of many reasons, including unilateral aborts.

[‡]In intention list based transaction processing systems, a single log record is usually written for each transaction. The more general case in which many log records may be written for a single transaction is treated in Section 5.4.1.

writing this log item is committed irrespective of whether the invoking transaction commits or aborts. Conceptually, the log can be considered as a linear object that grows in one direction. Each item in the log has an id (its LSN; i.e., log sequence number).

$append_{t_i}(k)$ denotes the appending of a log item pertaining to transaction t_i; when the operation completes, the id of the appended log item is returned in k. The value of k is one larger than the id of the previously appended log item.

$read_{t_i}(k)$ denotes the read operation on log item k by transaction t_i; this item should already exist in the log for the read to be successful. Otherwise, the read fails and the transaction which invoked the read aborts. k is given as an input to the operation *read*.

$last(k)$ can be used to determine the id of the last item in the log. This id will be returned in k when *last* completes.

Here are the correctness requirements imposed on read and last operations:

- $read_{t_j}(k) \in H \Rightarrow \exists t_i \neq t_j\ (append_{t_i}(k) \rightarrow read_{t_j}(k))$

  This states that t_j reads log record k after the write of record k by some t_i.

- $last_{t_j}(k) \in H \Rightarrow \exists t_i \neq t_j\ (append_{t_i}(k) \rightarrow last_{t_j}(k) \wedge \nexists t_m, l(append_{t_i}(k) \rightarrow append_{t_m}(l) \wedge append_{t_m}(l) \rightarrow last_{t_j}(k)))$

  This states that if the *last* operation executed by t_j returns k, then the k^{th} record should have been written by some transaction t_i and no other transaction has written since.

Since we require conflict serializability over accesses to the log items, $\mathcal{C}$ relationships are induced due to conflicting operations on the log. Specifically, $\forall\ t_i, t_j\ t_i \neq t_j\ \ (t_i\ \mathcal{C}\ \ t_j)$ if:

1. $\exists\ k, k'\ (append_{t_i}(k) \rightarrow append_{t_j}(k'))$

   Since both aborting transactions and committing transactions write to the log, transactions append to the log in the same order in which they commit or abort. Also, every transaction, once it begins execution, will commit or abort. Thus, when a transaction t_i appends an entry into the log, a transaction t_j that has not yet committed or aborted, i.e., t_j is a transaction in progress, will write its log entry after t_i's entry. Since t_j updates the log after t_i updates it, $(t_i\ \mathcal{C}\ \ t_j)$, that is, t_j must appear after t_i in the serialization order.

2. $\exists\ k \exists l \geq 0\ (append_{t_i}(k) \rightarrow read_{t_j}(k+l))$

   This states that if t_j reads log record $k+l, l \geq 0$, then t_j should occur later in the serialization order than the transaction t_i which caused the log record k to be written.

3. $\exists\ k\ (append_{t_i}(k) \rightarrow last_{t_j}(k))$

   This states that if t_j, via the *last* operation, "knows" that the last log record to be written was the k^{th} record (written by transaction t_i) then transaction t_i should precede t_j in the serialization order.

4. $\exists k, k'\ (last_{t_i}(k)\ \rightarrow\ append_{t_j}(k'))$

   A transaction t_j that has not yet committed or aborted when a transaction t_i performs the *last* operation will write its log entry after t_i executes *last*. That is, t_i observes the queue before t_j updates it and hence t_j will have to follow t_i in the $\mathcal{C}$ ordering.

Since we require serializability over the extended database, we need to consider the $\mathcal{C}$ relationships that result not only from the invocation of the operations on the database objects but also from the invocation of operations on the log. The modifications that a transaction t performs on the log (through the *append* operations) are persistent irrespective of whether t commits or aborts (thus the *append* operation belongs to the set $\mathcal{P}$ as defined in Section 3.3). Serializability is achieved by ensuring that the $\mathcal{C}$ relation defined on committed *operations* is acyclic. This is the safety-related correctness property that must be satisfied usually, and is still the case when accesses to the log are considered. The practical implication of the above axioms on transaction management is the following: When an operation is performed on the log, the system must note the $\mathcal{C}$ relationships induced by the operation, in light of the above axioms, and must ensure that the $\mathcal{C}$ relation is acyclic.

Traditionally, if there are cycles in the $\mathcal{C}$ relation, not all transactions involved in the cycle can commit, i.e., cycles are broken by transaction aborts. But with transactions being able to access the log, we must worry about the liveness of the transactions because aborting and committing transactions that access the log form $\mathcal{C}$ relationships which may create cycles which cannot be broken. The proof of the following theorem allays any fears in this regard by giving an algorithm which the transaction management system can use to ensure that transactions will always be able to terminate (commit or abort) without creating cycles which cannot be broken.

We assume here that the changes done by a transaction can be undone upon its abort without forming further $\mathcal{C}$ relationships. This is a valid assumption if transactions perform read and write operations on data [1].

Theorem 1 *Given a set of existing $\mathcal{C}$ relationships defined on committed operations, it is always possible to force transactions' operations, including those on the log, to occur in an order such that $\mathcal{C}^*$ (defined on committed operations) will be acyclic.*

Proof. At the beginning of transaction execution, the statement of the theorem vacuously holds (since there are no committed operations). We will show that the statement is invariant by showing that the $\mathcal{C}$ relation continues to be acyclic during the termination (commit/abort) of transactions.

Let us consider a transaction t_i that desires (or is forced to) abort, i.e., append to the log and undo all other operations performed on the database items. In this case, the only committed operation of t_i is the single append to the log and hence no cycles can be introduced by t_i. Now let us consider a transaction t_j which desires to commit. If there does not exist an active transaction t_k such that $t_k\ \mathcal{C}\ t_j$ and the commitment of t_j will not create any cycle in the $\mathcal{C}$ relation defined on committed operations, then t_j can proceed to commit. Otherwise t_j can abort like in the previous case. □

It is important to realize that when transactions are allowed access to the log and serializability remains the correctness criterion, it may delay the commitment or abortion of other transactions. Specifically, because of Axiom 4, once a transaction t_i performs *last*, no other transaction can commit or abort until t_i commits. Such a delay can affect performance. However, if one were to relax the correctness criterion, say by requiring something akin to the lower degrees of isolation of traditional databases, then this negative impact can be reduced or eliminated. Specifically, one could opt for non-repeatable reads of the log. We return to this issue in Section 5.4.

When other extra-data objects are accessed by transactions and serializability remains the correctness criterion, the semantics of these other objects should be specified just as we dealt with the log and then we must show that safety and liveness properties of transactions are kept intact. It should also be noted that in this correctness criterion, as long as the history resulting from concurrent transaction executions is serializable, correctness is considered to be preserved. The extra-database is, for purposes of concurrency control, treated like any other data item and serializability is applied without differentiating between database items and extra-data items. One implication of this is that techniques for improving performance, such as exploiting the semantics of operations [14], can be applied to both database and extra-data items.

5.3. Serializable Accesses to the Database

In this section, we give some examples of cases where we require serializability over only the database in the presence of extra-data dependent transaction programs.

Consider a modified version of the first car rental transaction from Section 4. Here, the available credit is updated where the amount depends on the car rental company chosen.

```
trans {
    car_rental avis = get_oid("avis");
    car_rental hertz = get_oid("hertz");
    ...
    if (num_waiting(avis) > num_waiting(hertz))
        hertz->reserve(...);
        master_card->credit_reserve(100);
    else
        avis->reserve(...);
        master_card->credit_reserve(75);
    ...
}
```

The application writer may not care from which company the car is rented as long as exactly one car is rented (even though the credit reservation amounts, 75 or 100, depend on the extra-data). Thus, in this case, we do not require extra-data independence of transactions and are only interested in the serializable execution (with respect to database items) of transactions. Hence, we only need to consider the $\mathcal{C}$ relationships induced due to conflicts on database items and achieve acyclicity over this relation. This ensures that the database is always in a consistent state as far as applications are concerned.

For another example, consider two transactions t_i and t_j that access disjoint parts of the database but what one transaction accesses is dependent on what the other accesses. The transactions communicate via communication channels that can be modeled as extra-data items which are read and written by the transactions. That is, t_i writes into the channel the ids of the data items it has accessed and t_j reads these, and vice versa. Here, we do not require the transactions to be extra-data independent (which they are not) but just require serializability over the database.

There is a subtle difference here from serializability in that if we look at the resulting serial schedule and rerun the programs corresponding to t_i and t_j we will not necessarily get the same overall state changes because of accesses to the extra-data and the extra-data dependence of t_i and t_j. However, if we rerun t_i and t_j so that the values returned by the operations on the extra-data items are the same as when they originally ran, then the rerun will produce the same database state as the original schedule. Thus, this notion of serializability differs from that in [13] where serializability over a subset of data items (in this case, the database items) implies that if we rerun the programs according to the serial schedule, we would get the same overall state for that subset of data items.

5.4. Application Specific Correctness of Accesses to Extra-Data

Thus far, we have worked with serializability as the basic correctness criterion for transactions accessing extra-data. But, as we alluded to at several places, it might be appropriate to relax serializability, as has been suggested even for transactions accessing just the database. The added motivation for this in the context of extra-data access is that access to extra-data items, such as the log, which lie in the processing path of every transaction must be allowed with minimal or no impact on performance. Since relaxing correctness requirements is one way to achieve this, serious consideration must be given to it.

Let us now consider some weakened isolation requirements [6] that have been suggested and adopted in practice for transactions accessing the database. In the context of read/write objects, degree-2 isolation ignores conflicts resulting from a read followed by a write. Such a requirement leads to lack of repeatable reads. Degree-1 isolation ignores, in addition, conflicts resulting from a write followed by a read. This permits the read of an object, writes on which have not yet committed, without forming a $\mathcal{C}$ relationship between the writing and the reading transaction. Degree-0 isolation ignores all dependencies. Let us consider an example applying these ideas to extra-data.

- Suppose the transaction management system updates the wait-for-graph on behalf of a transaction that waits for a lock on an object. Another transaction, which desires to know the length of time it will be forced to wait under current circumstances, views the wait-for-graph. Under degree-1 isolation, it will be allowed to proceed without forming any additional $\mathcal{C}$ relationships.

In this case, the extra-data item of interest is not directly updated by the transactions. If such transactions do not require the *repeatable read* property as guaranteed by standard concurrency control mechanisms such as locking, then they can afford to allow update operations to take place following their reads without incurring serialization ordering requirements.

In the following sections, we consider in detail cases in which correctness requirements weaker than serializability suffice for extra-data items.

5.4.1. The Log - Multiple Entries per Transaction

If we consider the log as extra-data, the removing of Axiom 4 of the log semantics in Section 5.2 has the effect of foregoing the repeatable read requirement since two invocations of *last* may now identify two different log records as the last record in the log. This eliminates the performance penalty mentioned at the end of Section 5.2 while still providing a well-understood form of correctness (corresponding to degree-2 isolation).

In normal practice, the log is implemented in such a way that each operation performed by transactions is logged separately and so there may exist multiple log records for a single transaction. Let us consider such a log which can be accessed by transaction through the following operations:

$append(Info)$ writes the information in $Info$ as the next log record.

$read\_first(t_j, Info)$ returns the first log record written by t_j in $Info$.

$read\_next(t_j, Info)$ returns the next log record (i.e., the log record after the log record read by the previous $read\_next(t_j, \_)$ or $read\_first(t_j, \_)$ operation) written by t_j.

$read\_begin(t_j, Info)$ returns the first log record in the log and the transaction t_j which wrote the log record.

$read\_succ(t_j, Info)$ returns the next log record (i.e., the log record immediately after the log record read by the previous *read_begin* or *read_succ* operation) in the log and the transaction t_j which wrote the log record.

A *read* operation fails if there is no log record that can be returned.

With the operations on the log defined as above, if we were to require serializability, then it would have a negative impact on the degree of concurrency allowed since the operations performed by different transactions have to be logged without any interleaving. In this context, an acceptable correctness criterion is that all transactions which read a log record written by a transaction t should be serialized after t. This condition is formally specified below.

In a history $\mathcal{H}$, $(t_i \; \mathcal{C} \; t_j)$, $i \neq j$, if

- $\exists Info \; (read\_first_{t_j}(t_i, Info) \in \mathcal{H})$
- $\exists Info \; (read\_begin_{t_j}(t_i, Info) \in \mathcal{H})$
- $\exists Info \; (read\_succ_{t_j}(t_i, Info) \in \mathcal{H})$

Besides the $\mathcal{C}$ relationships induced by the above axioms, the $\mathcal{C}$ relationships induced due to conflicts on the database items must also be taken into account. We require the acyclicity of the relation involving these $\mathcal{C}$ relationships on the committed operations as our correctness criterion. With our correctness criterion specified, we now need to guarantee the liveness of transactions (that is, all transactions should be able to commit or abort). This could be proved in a manner similar to the proof of liveness in Section 5.2.

5.4.2. The System Clock

Suppose transaction t_i accesses the system-maintained current time. A subsequent update of the current time by the system clock will not affect t_i if degree-2 isolation is in effect. More generally, suppose we want transactions to possess the temporal causality property which requires that if two transactions read the clock in a certain order with an intervening system update to the clock, their serialization should reflect the order of the reads. We are also not concerned about the extra-data dependence of transactions. In this case, the correctness criterion can be formalized in terms of $\mathcal{C}$ relationships as follows:

$t_i \mathcal{C} t_j$, $i \neq j$, if $\exists k(read_{t_i} \rightarrow write_{sys,k}) \wedge (write_{sys,k} \rightarrow read_{t_j})$

Here the reads and writes refer to the operations on the clock, where the clock is updated by the system, a non-transactional entity. The index k in the event $write_{sys,k}$ uniquely identifies the different writes to the clock by the system. In addition to these $\mathcal{C}$ relationships, those arising due to conflict on the database items are also considered and acyclicity of this $\mathcal{C}$ relation is ensured over all committed operations. It is interesting to note that this correctness criterion rectifies the anomaly found in Section 2.3 dealing with Joe's status. However, this criterion is weaker than serializability in the sense that we allow for non-repeatable reads to the clock.

Another example in which clocks are accessed as extra-data items occurs in the context of real-time databases [11]. In this case, transactions have a correctness requirement which is stronger than traditional serializability in the sense that we have the additional requirement that if a transaction t_i commits, it should commit before its deadline, $deadline(t_i)$. This requirement is formally stated below: $commit_{t_i} \in \mathcal{H} \Rightarrow commit_{t_i} \in \mathcal{H}^{deadline(t_i)}$

where $\mathcal{H}^{deadline(t_i)}$ is the prefix of the history $\mathcal{H}$ until time $deadline(t_i)$. In order to satisfy this requirement, the transaction management system accesses the clock (an extra-data item) just before the transaction wants to commit. In case its deadline has been met, the transaction commits, otherwise it aborts. As per our definition in Section 4, such transaction programs are extra-data independent, where $equiv(t)$ of a real-time transaction program t is a transaction program which attempts to commit regardless of whether its deadline has been met or not. Thus, accessing the clock in the case of real-time transactions does not violate the consistency of the database and hence the clock need not be brought within the scope of concurrency control.

Also, in real-time databases, if we know that a transaction is highly unlikely to meet its deadline it is better to abort the transaction early. This can be achieved by examining the wait-for-graph. Under degree-1 isolation, a transaction will be allowed to access the wait-for-graph without forming any additional $\mathcal{C}$ relationships.

6. COMPARISON OF THE CORRECTNESS CRITERIA

In this section, we present a comparison of the correctness criteria proposed in Section 5. This comparison determines the conditions under which one correctness criterion is weaker[†] than another. The practical use of this comparison is that weaker correctness criteria could be chosen whenever possible so that the scheduler has more flexibility, potentially leading to an improvement in performance.

Figure 2 gives a diagrammatic representation of the sets of histories accepted by each correctness criterion. From this figure, we can see that *Application Specific Correctness Criteria*[‡] accept a set of histories which is a strict superset of the set of histories accepted by *Serializability over the Extended Database*. This is because *Application Specific Correctness Criteria* might not require serializability over the extended database, but just serializability over the database and some other weaker correctness criterion over the extra-database. The log example (Section 5.4.1) and thesystem clock example (Section 5.4.2) are examples in which the *Application Specific Correctness Criterion* is weaker than *Serializability over the Extended Database*.

[†] A correctness criterion $\mathcal{A}$ is said to be weaker than correctness criterion $\mathcal{B}$ if the set of histories accepted by $\mathcal{A}$ are a strict superset of the set of histories accepted by $\mathcal{B}$.

[‡] Correctness criteria for extra database items used in current database systems typically fall in this category. For example, using a lower degree of isolation (degree 1 or 2) over extra database items falls under application specific correctness criteria.

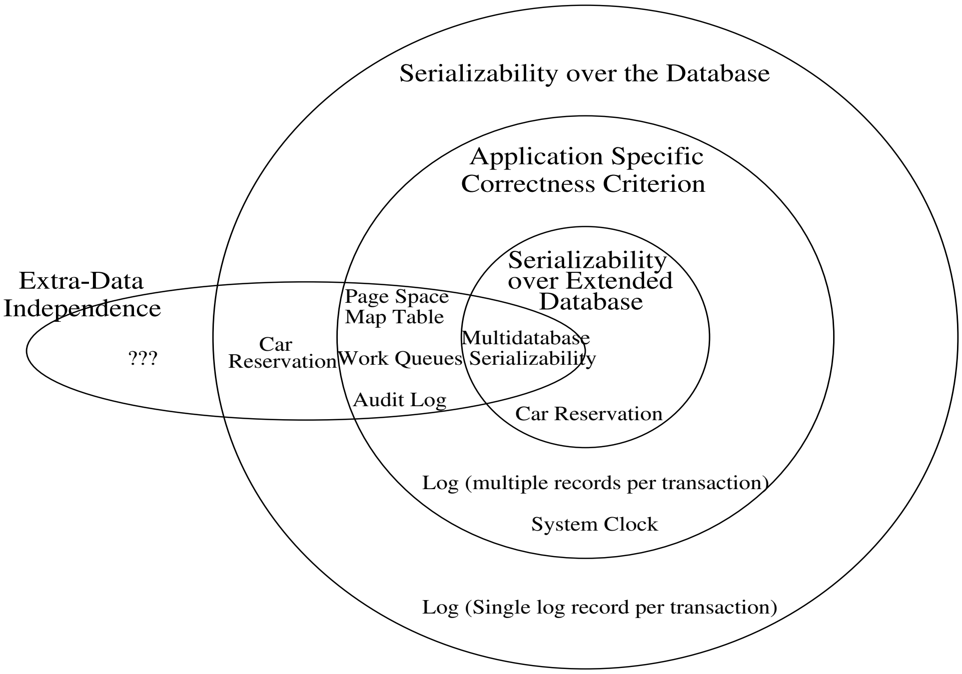

Fig. 2: Comparison of Histories Accepted by Correctness Criteria

It is also easy to see that *Serializability over the Database* is weaker than an *Application Specific Correctness Criterion* where the database is still expected to be accessed serializably. This is because, *Application Specific Correctness Criteria*, in addition to requiring serializability over the database, also require some restriction on extra-database accesses. On the other hand, *Serializability over the Database* does not impose any restrictions on extra-database accesses. The log example (Section 5.2) illustrates a case where the correctness required is just *Serializability over the Database*.

The set of histories accepted by the *Extra-Data Independence* correctness criterion is neither weaker nor stronger than the other three correctness criteria (see Figure 2). This is because in *Extra-data independence*, only the set of conventional transactions which correspond to the transactions which actually execute should be serializable, but the transactions which actually execute need only satisfy a correctness criterion specified by a set of axioms $\mathcal{K}$ (see Definition 12). The set $\mathcal{K}$ can represent a correctness criterion as strong as *Serializability over the Extended Database* (the Multi-database Serializability example in Section 5.1.5) or as weak as *Serializability over the Database* (the Car Reservation Example in Section 5.1.2). If the set of axioms $\mathcal{K}$ defines a correctness criterion stronger than *Serializability over the Database* but weaker than *Serializability over the Extended Database*, this corresponds to the *Application Specific Correctness Criteria* (see the examples on Page Space Map Tables and Work Queues in Sections 5.1.3 and 5.1.1 respectively).

It is also theoretically possible to have a set of axioms $\mathcal{K}$ weaker than *Serializability over the Database* that ensures the *Extra-data Independence* of transactions. However, this does not seem very likely in practice because this implies that one can use a weaker correctness criterion (in this case, some correctness criterion weaker than serializability over the database) for the extra-data dependent transactions in order to ensure a stronger correctness criterion (in this case, serializability over the database) for the corresponding conventional transactions.

In summary, *Serializability over the Database* is weaker than *Application Specific Correctness Criteria* which in turn is weaker than *Serializability over the Extended Database*. *Extra-Data Independence* is, however, neither weaker nor stronger than any of the above mentioned correctness criteria.

7. PRACTICAL CONSIDERATIONS

In this section, we discuss some of the practical aspects involved in ensuring the proposed correctness criteria. We first deal with the issue of constructing extra-data independent transaction programs and then address ways in which the proposed correctness criteria could be enforced.

7.1. Constructing Extra-Data Independent Transaction Programs

Constructing and proving the extra-data independence of transaction programs varies widely in difficulty. This is because, in some cases, knowing only the read and write sets of transaction programs is sufficient to prove the extra data independence of the programs. In other cases, however, the semantics of transactions have to be examined and formalized in order to prove extra-data independence. In this subsection, we investigate both possibilities.

The following theorem outlines the weakest condition under which we can determine that a set of transaction programs is extra-data independent with respect to serializability over the database, given information about only their read and write sets (note that these read and write sets are properties of transaction programs, not their instances). Intuitively, we can prove extra-data independence iff there is no "leakage" from the extra-data items into the database through transactions.

Theorem 2 *Let $\mathcal{E}$ be set of all objects in the extended database and $\mathcal{D}$ be the set of all objects in the database. Let $\mathcal{TP}_{\dashv\ddagger\ddagger}$ be the set of all transaction programs which execute in the transaction processing system, where for all $tp \in \mathcal{TP}_{\dashv\ddagger\ddagger}$, $R(tp) \subseteq \mathcal{E}$ denotes the read set of tp and $W(tp) \subseteq \mathcal{E}$ denotes the write set of tp. Given* only *the above information, a set of transaction programs $\mathcal{TP} \subseteq \mathcal{TP}_{\dashv\ddagger\ddagger}$ can be said to be extra-data independent with respect to serializability over the database iff:*

1. $(\forall tp \in \mathcal{TP}_{\dashv\ddagger\ddagger} - \mathcal{TP}) W(tp) \cap \mathcal{D} = \phi$ *and*

   Only instances of transaction programs in $\mathcal{TP}$ can write to the database.

2. *there does not exist a sequence of transaction programs of the form $(tp_1; tp_2; ...; tp_k)$, $k \geq 2$, such that*

   (a) $tp_i \in \mathcal{TP}_{\dashv\ddagger\ddagger}, 1 \leq i \leq k$

   (b) $W(tp_1) \cap R(tp_2) \cap (\mathcal{E} - \mathcal{D}) \neq \phi$

   (c) $W(tp_i) \cap R(tp_{i+1}) \neq \phi$, $2 \leq i \leq k-1$

   (d) $W(tp_k) \cap \mathcal{D} \neq \phi$

   No transaction in $\mathcal{TP}_{\dashv\ddagger\ddagger}$ (part (a)) writes to any extra-data item (part (b)) whose value can be propagated to the database (parts (c) and (d)).

Proof. (If Part) Assume that the above conditions hold. Then no transaction program which is not in $\mathcal{TP}$ writes to the database (first condition). Also, no transaction program in $\mathcal{TP}$ ever reads the value of a modified extra-data item either directly or indirectly (second condition). Thus if we replace each program $tp \in \mathcal{TP}$ by a corresponding extra-data independent transaction program tp', in which all accesses to an extra-data item are replaced by reading a constant value which corresponds to the initial value of the extra-data item, tp' would perform the same transformation on the database as would tp in a serial order. Thus, by Definition 12, it is easy to see that $\mathcal{TP}$ represents a set of extra-data independent transactions.

(Only If Part) Assume that at least one of the two conditions do not hold. If the first condition does not hold, then a transaction program in $\mathcal{TP}_{\dashv\ddagger\ddagger} - \mathcal{TP}$ *could* † perform some arbitrary transformation to the database which cannot be performed by any transaction in $\mathcal{TP}$. Thus, the

†Note that the existence of the possibility that a transaction could behave in a certain way is sufficient to prove that we cannot infer extra-data independence, without having additional information

set of transactions $\mathcal{TP}$ cannot be said to be extra-data independent. If the second condition does not hold, then a transaction could write an arbitrary value to an extra-data item which could be propagated back to the database through transactions. Again, the set of transactions $\mathcal{TP}$ cannot be said to be extra-data independent. □

The above result could be used to prove the extra-data independence of transaction programs in the Multi-database Serializability example (Section 5.1.5) and the Log example (Section 5.1.6).

For cases in which the condition outlined in the above theorem is too weak to prove extra-data independence, we would have to look into the semantics of transaction programs and data. One useful technique in this context is data flow analysis. If transaction programs are written in an imperative programming language, then using data flow analysis techniques, we can verify that extra-data information does not propagate into the database (this is similar to ensuring condition 2 in the above theorem). Then, the only effect extra-data may have on instances of transaction programs is manifested through flow of control. If we can verify that instances of transaction programs have the same effect on the database regardless of the flow of control, then we can infer that the transaction programs are extra-data independent with respect to serializability over the database. This technique is useful for proving extra-data independence in the car rental example (Section 5.1.2) and the page space map example (Section 5.1.3).

If both of the above techniques are not strong enough to prove the extra-data independence of transactions, then the actual application-dependent semantics of data and transactions would have to be used. This, for instance, is the case in the work queues examples (Section 5.1.1) and the proclamation locking example (Section 5.1.4).

7.2. Enforcing Correctness Criteria

In this section, we address some of the practical issues involved in enforcing the correctness criteria proposed in Section 5. We first discuss how $\mathcal{C}$ relationships can be tracked for concurrency control and then outline concurrency control mechanisms for each correctness criterion.

$\mathcal{C}$ relationships could be enforced in a manner similar to the Serialization Graph Testing (SGT) [1] concurrency control technique. In this technique, a serialization graph is maintained in order to perform concurrency control. An edge (t_i, t_j) exists in the serialization graph iff transaction t_i performs a conflicting operation on an object before transaction t_j. Only histories which give rise to acyclic serialization graphs are considered legal. This ensures serializability as defined in Definition 8. The Basic SGT algorithm checks to see if performing an operation would lead to a cycle in the Serialization Graph. If so, the operation is rejected, else it is performed. In the Conservative SGT, operations may be delayed before being scheduled in order to reduce the number of rejected operations. Algorithms for distributed SGT [1] have also been proposed.

The Serialization Graph Testing technique could be adapted in order to enforce $\mathcal{C}$ relationships induced by a set of axioms $\mathcal{K}$. Here an edge (t_i, t_j) exists in the serialization graph iff $t_i \mathcal{C} t_j$ is induced by the axioms in $\mathcal{K}$. It is easy to see that if only acyclic serialization graphs are considered legal, then the set of accepted histories would be serializable with respect to the set of axioms $\mathcal{K}$. The only potential problem in using the serialization graph technique to enforce serializability with respect to a set of axioms $\mathcal{K}$ would be if transactions could neither be committed nor aborted without forming a cycle in the serialization graph. In order to address this issue, we would need to prove that all cycles in the serialization graph could be avoided by committing and aborting transactions in some order. A proof of this propery would be similar to the proof presented in Section 5.2. We now show by means of an example that Conservative SGT mechanisms could be effectively used for enforcing $\mathcal{C}$ relationships.

Example 1 Consider the log example in Section 5.2. Suppose $(t_j\ \mathcal{C}\ t_i)$ already exists because, for example, t_j does a *write* of some object in the database and then t_i does a *read*. If t_j aborts (for some reason) then since we require failure atomicity, t_i must also abort (see Section 3). Suppose t_i performs the *last* operation and t_j is yet to terminate. That is, t_j will perform an *append* operation later, and hence, by axiom 4 will produce $(t_i\ \mathcal{C}\ t_j)$. Thus, if we wait until t_j's append to notice the cycle, the only way to break the cycle then is to abort t_i. But instead, given Axiom 4, if we had allowed t_i to perform *last* only after t_j terminates, then we could have perhaps committed both t_i and t_j. Thus, in general, ensuring that there are no $\mathcal{C}$ cycles can be achieved by delaying operations whenever possible. □

We now outline implementation techniques for the correctness criteria proposed in Section 5.

- *Serializability over Database:* This is the easiest correctness criterion to enforce because traditional concurrency control techniques can be directly applied [1]. This is because traditional concurrency control is performed on just the database and serializability is the commonly enforced correctness criterion.

- *Serializability over the Extended Database:* There are two main ways in which Serializability over the Extended Database can be enforced. In the first method, concurrency control could be performed over the extended database considering it as one large database. Here, each access to extra-data items is caught by the scheduler, which performs concurrency control for both the database and extra-data items. The scheduler could use traditional techniques for ensuring serializability. In the second method, the database and the extra-database can be considered as two separate databases and concurrency control performed for each of them. Multi-database concurrency control techniques could be used to address conflicts which span the separate databases.

- *Application Specific Correctness Criteria:* In this case too, there are two options depending on whether we treat the extended-database as one database or two databases. In the first case, we could use Serialization Graph Testing mechanisms in order to ensure that all accepted histories are serializable with respect to the set of axioms $\mathcal{K}$ specified as the correctness criteria. In the second case, we could use Distributed SGT techniques to enforce the desired correctness criterion.

- *Extra-Data Independent Transaction Programs:* Let the set of axioms to be ensured in order to achieve extra-data independence be $\mathcal{K}$. If $\mathcal{K}$ corresponds to Serializability over the Database or Serializability over the Extended Database, the corresponding techniques could be used to achieve Extra-data independence. Otherwise, an SGT based technique (like in Application Specific Correctness Criteria) could be used.

8. DISCUSSION

Recently, there have been many extensions to the classical work on concurrency control. One approach extends and elaborates the structure of data items, viewing them as abstract data type objects thus exploiting the semantics of the operations for better concurrency control [14]. Some of the other approaches include looking at serializability over a subset of data items [13] and relaxing the serializability correctness criterion by imposing instead specific constraints on acceptable schedules [8]. The work reported herein bears resemblance to these extensions in that, technically, we also view extra-data as objects with arbitrary operations defined on them and impose some restrictions on acceptable schedules. However, these are by-products of our main interest, that of enlarging the set of transaction accessible data to include structures that are traditionally either (a) hidden within, and are internal to, the database system itself or (b) local to a set of transactions. We have examined the consequences of such accesses and in doing so we are forced to use extensions to the traditional concurrency control setting. Specifically, our goal in this paper was twofold:

- To illustrate via detailed examples that allowing transactions to access extra-data not only improves the functionality of transactions but also has many performance benefits.

  We considered extra-data items that occur in typical database systems such as the log, the clock, and the concurrency control information. We also showed that other types of extra-data, such as proclaimed values, and queues used for coordinating pipelined and cooperating transactions prove very useful for structuring transactions in order to improve performance.

- To investigate, in detail, the correctness issues that arise when transactions are allowed access to extra-data.

  We saw that while traditional serializability can continue to be the mainstay of correctness, one needs to also consider extra-data independence of transactions. To precisely capture the interactions due to extra-data access, we axiomatized the operations on extra-data items to determine the serialization ordering requirements induced by extra-data access. These helped characterize the different types of correctness issues that must be considered. In this regard, we studied a variety of correctness notions.

The techniques presented in this paper are useful both for the transaction programmer as well as the database system implementor. The transaction programmer is concerned with correctness issues which arise when transactions access data items which are not part of the database (for example, the system clock or the log) and the concurrency control mechanisms required to achieve them. The database systems implementor, on the other hand, is concerned with the issues which arise when extra-data is used to improve the performance of transactions (for example, proclamation data items or work queues) or when transactions access extra-data items (for example, page space map tables) which are hidden at the transaction programmer level.

Allowing transactions to access extra-data improves transaction functionality. On the other hand, as we discussed at several points in the paper, performance consequences can be either positive or negative depending on the properties of the data. One implication is that extra-data access must be allowed only if the consequences are not detrimental to performance, and if they are and yet extra-data must be accessed, one must apply the least restrictive correctness criterion that fits the needs.

Acknowledgements — This research has been partially supported by the National Science Foundation under grant IRI-9314376.

REFERENCES

[1] P.A. Bernstein, V. Hadzilacos and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley (1987).

[2] P.K. Chrysanthis and K. Ramamritham. Synthesis of extended transaction models using ACTA. *ACM Transactions on Database Systems*, **19**(3):450–491 (1994).

[3] D. Georgakopoulos, M. Rusinkiewicz and A. Sheth. On serializability of multidatabase transactions through forced local conflicts. In *Proceedings of the IEEE Seventh International Conference on Data Engineering*, pp. 314–323, Kobe, Japan (1991).

[4] N. Gehani and O. Shmueli. *The LOG as Part of the Database*. Bell Laboratories Technical Memorandum (1992).

[5] N. Gehani, K. Ramamritham, J. Shanmugasundaram and O. Shmueli. *Accessing Extra Database Information: Concurrency Control and Correctness*. Technical Report 1996-016, University of Massachusetts, Amherst, Massachusetts (1996).

[6] J.N. Gray and A. Reuter. *Transaction Processing: Techniques and Concepts*. Morgan-Kaufman (1992).

[7] H.V. Jagadish and O. Shmueli. A proclamation-based model for cooperating transactions. In *Proceedings of the eighteenth International Conference on Very Large Databases*, pp. 265–276, Vancouver, Canada (1992).

[8] H.F. Korth and G. Speegle. Formal aspects of concurrency control in long-duration transaction systems using the NT/PV model. *ACM Transactions on Database Systems*, **19**(3):492-535 (1994).

[9] C. Mohan and D. Haderle. Algorithms for flexible space management in transaction systems supporting fine-granularity locking. In *Proc. 4th International Conference on Extending Database Technology*, pp. 113–144, Cambridge, United Kingdom (1994).

[10] K. Ramamritham and P.K. Chrysanthis. A taxonomy of correctness criteria in database applications. In *Very Large Data Bases Journal*, **5**(1):85–97 (1996).

[11] K. Ramamritham. Real-time databases. *International Journal of Distributed and Parallel Databases*, **1**(2):199–226 (1993).

[12] M.R. Stonebraker. Hypothetical data bases as views. In *Proceedings ACM-SIGMOD 1981 International Conference on Management of Data*, pp. 224–229 (1981).

[13] K. Vidyasankar. Generalized theory of serializability. *Acta Informatica*, **24**:105–119 (1987).

[14] W. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, **37**(12):1488–1505 (1988).