On the Serializability Theorem for Nested Transactions

R. F. Resende* A. El Abbadi†
Department of Computer Science
University of California
Santa Barbara, CA 93106

Abstract

The fundamental theorem of the classical serializability theory states the necessary and sufficient conditions for the conflict serializability of an execution. In this paper, we extend the correctness criteria of the classical theory by presenting a definition of conflict serializability for concurrency control of nested transactions. We define a serialization graph for nested transactions and we prove that its acyclicity is a sufficient and necessary condition for conflict serializability.

Keywords: Databases, Concurrency, Conflict Serializability, Nested Transactions.

1 Introduction

Concurrency has always been a major aspect of computing systems. In a database system the easier the task of the applications programmer the more reliable the product. It is vital with respect to the easy of use of databases that concurrency be transparent for the applications programmer. This has been done, and in most systems the programmer simply has to design a program without considering interferences from other programs accessing shared data. The simplicity of this concept was formally captured by the serializability theory [2] [6].

Most of the results of the serializability theory are related to a system model that has two levels, that of atomic operations, and above it, programs that use those operations. These systems are called *flat*, and the corresponding theory *classical*. In *nested transactions*, transactions may use subtransactions, which in turn may use subtransactions, and so on [5]. Nested transactions allow the benefits of atomicity to be used within a transaction, so that, for example, a transaction can include several simultaneous remote procedure calls, which can be coded without considering possible interference among them [4].

In this paper, we extend the correctness criteria of the classical theory by presenting a definition of conflict serializability for concurrency control of nested transactions. An execution of nested

*Work supported in part by CNPq - Conselho Nacional de Desenvolvimento Científico e Tecnológico
†Work supported in part by the NSF under grant number IRI-9117094.

transactions is correct, i.e., conflict serializable, if it is conflict equivalent to the execution of a serial nested transaction. Our contribution is a well balanced group of definitions based on [1] and [3] and a proof of the fundamental theorem of serializability theory. We define a serialization graph for nested transactions and we prove that its acyclicity is a sufficient and necessary condition for conflict serializability. In prior work on nested transactions, Hadzilacos and Hadzilacos [3] define a serialization graph and prove that its acyclicity is a sufficient condition for their notion of serializability. Fekete et al.[4] give a similar graph using a different system model, and again only prove sufficiency. This paper chooses a more restrictive notion of "conflict serializability", and then proves that this is equivalent to acyclicity of a serialization graph. Thus we show that one can obtain theorems for nested transactions systems that correspond to the standard theorems of the classical theory[2]. We conclude the paper by deriving a proof of the correctness of executions produced by a modular graph testing scheduler.

2 The Model

This section is divided in two subsections. In the first subsection we summarize part of the work presented in Beeri, Bernstein and Goodman [1]. In the second subsection we introduce our notion of equivalence and correctness.

2.1 Database System Model

The components of a computation are states, operations, transactions, return values, and a partial order. Each state represents the state of the entire database, states are denoted by $St = \{s_1, s_2, \ldots\}$. The set of transactions are denoted by $T = \{t_1, t_2, \ldots\}$. The set of operation occurrences in computations is denoted by OP.

A *DM* or *Data Manager* represents the storage subsystem. The DM is characterized by the set of computations it can execute. A computation is described by a tuple. A *DM tuple* is a tuple $(s_1, O, <, r, s_2)$, where s_1 and s_2 are states, O is a finite subset of OP, $<$ is a partial order on O, and r is a return value function with domain O. A *D-computation* is a DM tuple that obeys axioms (C1), (C2), and (C3) below. A *DM D* is the set of all D-computations of a given system. A DM tuple $(s_1, O, <, r, s_2)$ in D means that starting from state s_1, D can execute the operations in O in the execution order $<$, return the values specified by r, and end in state s_2. For $o_1, o_2 \in O$, we say that o_1 *precedes* o_2 in c if $o_1 < o_2$. To denote that o_1 does not precede o_2 we write $o_1 \not< o_2$. If $o_1 \not< o_2$ and $o_2 \not< o_1$ then o_1 and o_2 are *concurrent*. A D-computation $(s_1, O, <, r, s_2)$ is *serial* if $<$ is a total order on O. A D-computation $c = (s_1, O, <, r, s_2)$ is *order preserving atomic (OPA)* if there exists a total order $<_1$ such that $< \subseteq <_1$ and $\hat{c} = (s_1, O, <_1, r, s_2)$ is also a D-computation. That is, c has the same effect as a serial computation that preserves the order of c's operations. Let $O_1, O_2 \subseteq$ OP, such that $O_1 \cap O_2 = \emptyset$, let $<_1$ and $<_2$ be partial orders on O_1 and O_2 respectively, and let $<_1 \otimes <_2$ be defined as $<_1 \cup <_2 \cup O_1 \times O_2$. Square brackets are used to restrict domains, e.g., $r[O]$ denotes function r restricted to domain O. A DM D must satisfy:

(C1) (Composition axiom) If, for some s_1, s_2, s_3, the computations $c_1 = (s_1, O_1, <_1, r_1, s_2)$ and $c_2 = (s_2, O_2, <_2, r_2, s_3)$ are in D then so is their *composition*

$$c = c_1 \otimes c_2 = (s_1, O_1 \cup O_2, <_1 \otimes <_2, r_1 \cup r_2, s_3)$$

(C2) (Decomposition axiom) If, for some s_1 and s_3, there is a computation $c = (s_1, O_1 \cup O_2, <_1 \otimes <_2, r, s_3)$ in D, then there exists a state s_2 in St such that $c_1 = (s_1, O_1, <[O_1], r[O_1], s_2)$ and $c_2 = (s_2, O_2, <[O_2], r[O_2], s_3)$ are in D.

(C3) All D-computations are OPA.

Axioms (C1) and (C2) show how to compose and decompose tuples. Axiom (C3) entails the assumptions in the classical theory of serializability that operations are atomic and that noncommutative operations are related by $<$.

A transaction is an execution of a program. Formally, a *transaction* is a tuple $t = (O, <_t, r)$, where O is the set of operations executed in the computation, $<_t$ the *transaction order*, is a partial order on O, and r is a return value function with domain $O \cup \{t\}$. A *transaction tuple* for t, or a t-tuple, is any tuple $(O, <, r)$. A t*-computation* is a t-tuple that satisfies axiom (C4):

(C4) (Order extension axiom) If $t = (O, <_t, r)$ is a transaction, then $(O, <, r)$ is a t-computation if and only if $<_t \subseteq <$.

A subtransaction is viewed by its parent as an operation, though in reality it is an execution of a program. To achieve this dual view, we assume a function from the set T of transactions to a subset OP_T of OP, which gives the operation of which the (sub)transaction is an implementation. As an abuse of functional notation $tr(o)$ denotes any transaction associated with operation o. Axiom (C5) below states that $tr(o)$'s effect can be achieved by o, i.e., it relates an operation's specification to its implementation.

(C5) (Sequential correctness axiom) Let $tr(o)$ be a transaction associated with operation o. For all s_1, s_2, $<$, r, if $(O, <, r)$ is a $tr(o)$-computation, and if $(s_1, O, <, r[O], s_2)$ is a serial D-computation, then $(s_1, \{o\}, \emptyset, r[tr(o)], s_2)$ is also a D-computation.

A *hierarchical tuple* is a tuple $c = (s_1, F, <, r, s_2)$ where s_1 and s_2 are initial and final states respectively. F is a forest of subtransactions and operations, $<$ is an execution order, and r is a return value function. The special case in which F is a one-level forest, c is also a DM tuple. Sometimes it is convenient to treat a given forest as a tree. This can be done adding a root (in [4] this root is called "mythical" t_0). Formally, we assume that a hierarchical tuple $c = (s_1, F, <, r, s_2)$ can also be represented by a $\hat{c} = (s_1, A, <, \hat{r}, s_2)$, where A is the same as F together with a distinguished transaction, t_0, that invokes as subtransactions the roots of F, and $\hat{r}$ is the same as r with domain extended to t_0, $r(t_0)$ is irrelevant.

The set of *computations forests* CFs [1] is the set of hierarchical tuples that satisfies axioms (C6)-(C8) below. For a hierarchical tuple c, the set of leaves is $leaves(A)$ or $leaves(c)$; the roots (top-level transactions) are $roots(c)$. For a node x, the subtree rooted at x is A_x, and its leaves are $leaves(x)$. The set of descendents of x is $desc(x)$ and the set of children (i.e. immediate descendents) is $child(x)$, ($roots(c)$ is the same as $child(t_0)$ or $child(A)$). The lowest common ancestor of nodes x and y is $lca(x, y)$. Nodes x and y are *incomparable* if neither is a descendent of the other. The *DM projection* of c is $c[leaves(c)]$, also denoted as $c[D]$. For an internal node x, *c restricted to x* is the transaction tuple $(child(x), <[child(x)], r[child(x) \cup \{x\}])$. A CF $c = (s_1, A, <, r, s_2)$ must satisfy:

(C6) (Downward compatibility) Let $x, y \in A$, and let $p \in desc(x) \cup \{x\}, q \in desc(y) \cup \{y\}$, if $x < y$, then $p < q$.

(C7) (Transaction validity) For each internal node x, c restricted to x is an x-computation.

(C8) (DM validity) The DM projection of c, $c[D]$, is a D-computation.

Axiom (C6) says that if x completes before y begins, then each operation and subtransaction of x completes before any operation or subtransaction of y begins. (C7) requires the consistency of $<$ with the partial order of each constituent transaction, and (C8) says that a CF comprises a computation of a DM D.

Proposition 1. [1] Let $c = (s_1, A, <, r, s_2)$ be a CF. The partial order $<$ can be extended to a partial order $<'$ such that $c' = (s_1, A, <', r, s_2)$ is also a CF and for all nodes x and y, if, for all $p \in child(x)$ and all $q \in child(y)$, $p <' q$ then $x <' y$.

Proposition 1 establishes a "normalization" process for CFs. Such canonical partial orders will be assumed from now on, i.e., $<$ in the following CFs cannot be further extended as prescribed by Proposition 1. Furthermore, by DM-validity, (C8), for every CF c, $c[D]$ is a D-computation. By the OPA property of axiom (C3) we can extend the order to be serial on the leaves, and by Proposition 1, we can extend it further to be upward compatible. Hence we assume that all D-computations are serial.

2.2 Conflict consistency and conflict equivalence

In this section we define serializability for nested transactions based on the notion of conflicts. We then contrast this definition with other definitions.

Let o_1 and o_2 be operations and s be a state. We say that o_1 and o_2 *commute* with respect to s if for all r and $\bar{s}$, $(s, \{o_1, o_2\}, (o_1, o_2), r, \bar{s}) \in D$ iff $(s, \{o_1, o_2\}, (o_2, o_1), r, \bar{s}) \in D$. Two operations *(generally) commute* if they commute with respect to all states. Operation o_1 *conflicts* with o_2 if o_1 does not commute with o_2.

Let $<$ and $<'$ be total orders such that $<$ is a permutation of $<'$. We say that $<'$ is *conflict consistent* [3] with $<$ if whenever o conflicts with and precedes o' in $<$ then o precedes o' in $<'$.

[1] We will maintain the denomination "Computation Forest" or "CF", despite the introduction of t_0.

Two DM tuples $c = (s_1, O, <, r, s_2)$ and $c' = (s'_1, O', <', r', s'_2)$ are *conflict equivalent*, denoted $c \equiv c'$, iff $s_1 = s'_1$, $s_2 = s'_2$, $O = O'$, $<$ is conflict consistent with $<'$, and $r = r'$. Now we will prove that if two DM tuples are conflict equivalent and one of them is a D-computation then the other is also a D-computation. We will use the technique of reversing commutative operations presented in [1] and [6]. First we show that the DM-tuple obtained by inverting the order of two contiguous and commutative operations of a D-computation is also in D.

Lemma 1. Let c be a DM tuple with operations x and y contiguous and commutative with respect to the state in which the first one was executed. Let c' be a DM tuple obtained from c by reversing x and y. If $c \in D$ then $c' \in D$ and $c \equiv c'$.

Proof. Without loss of generality assume that $x < y$. Since $c \in D$, c obeys the decomposition axiom and can be represented as $c = c_1 \otimes (\hat{s_1}, \{x, y\}, (x, y), r[\{x, y\}], \hat{s_2}) \otimes c_2$. By definition of commutativity $(\hat{s_1}, \{x, y\}, (y, x), r[\{x, y\}], \hat{s_2}) \in D$ hence $c' = c_1 \otimes (\hat{s_1}, \{x, y\}, (y, x), r[\{x, y\}], \hat{s_2}) \otimes c_2 \in D$. That $c \equiv c'$ is obvious. □

Now we show the relation between conflict equivalence and membership in D.

Lemma 2. Let $c = (s_1, O, <, r, s_2)$ and $c' = (s'_1, O', <', r', s'_2)$ be DM tuples. If $c \in D$ and c is conflict equivalent to c' then $c' \in D$.

Proof. By induction on the number of pairs, k, of operations that are reversed in c with respect to c'. The basis (k=1) is trivial from Lemma 1. For the induction hypothesis lets assume that the lemma is true for $k-1$ pairs. Consider the pair of operations x and y that are reversed in c' and contiguous in c [2]. The operations of such a pair commute since the total order $<$ in c is conflict consistent with the order $<'$ in c'. Now we use Lemma 1 to obtain $\bar{c}$ which is in D, is conflict equivalent to c, and has only $k-1$ pairs of operations reversed with respect to c'. □

Two CFs $c = (s_1, A, <, r, s_2)$ and $c' = (s'_1, A', <', r', s'_2)$ are *conflict equivalent*, denoted $c_1 \equiv c_2$, iff $A = A'$, $r = r'$, and $c[D] \equiv c'[D]$. A CF $c = (s_1, A, <, r, s_2)$ is *serial* iff $<$ is a total order on the roots and on the children of each node. A CF is *conflict serializable* iff it is conflict equivalent to some serial CF.

Our notion of serializability differs from previously proposed notions. In [1] two CFs are equivalent if they have the same initial and final states and their roots have the same return values. In [3] the equivalence concept was strengthened by requiring not only the same roots but also the same forest (with the same return value for each node). We strengthen the latter definition by requiring not only the same final state but also that the ordering relation ($<$) of one be conflict consistent with the ordering of the other, which we showed guarantees the same final state. We consider states and return values only to take advantage of the state-dependent commutativity definition, thus remaining within the confines of the general theory presented in [1]. Note that in [4] serial

[2] Such a pair exists, by choosing x and y to be the pair of operations that are reversed in c' and closest together in c (without loss of generality, suppose x precedes y in c). Suppose (to obtain a contradiction) that any operation z came between x and y in c. We have that x must precede z in c', since they are closer together in c than x and y; similarly z must precede y in c'. The combination of these two facts shows x precedes y in c', contradicting the definition.

correctness for T_0 corresponds to serializability in [1]. To illustrate the difference between the equivalence definitions of [1] and [3], consider a transaction that increments an integer data item, using a read and a write operation. Consider two increment subtransactions belonging to two different top-level transactions. In [1] the order of the two increments is irrelevant as long as the underlying read and write operations are not interleaved. In contrast, in [3], the order is significant since all operations must return the same values, including the read and write that implement the increment operation. The difference between our notion of equivalence and that of [3] can be easily illustrated by any view serializable execution of flat transactions that is not conflict serializable [2]. Although more restrictive, our approach allows for the development of a serializability theory similar to the classical theory for flat transactions. In the next section we develop a serialization graph for nested transactions and show that the acyclicity of the graph is both necessary and sufficient for conflict serializability. Hence, for the subset of executions that strictly depends on conflict information from the leaves, our approach complements the theory of Beeri, Bernstein and Goodman [1]. In particular correctness in that theory is based on operational techniques such as commutativity-based reversals and substitution while conflict serializability can be proven using graph-theoretical techniques.

3 The Serializability Theorem

In this section we prove a theorem which establishes a necessary and sufficient condition for conflict serializability of nested transactions, similar to the fundamental theorem of serializability [2] for flat transactions. First we define the serialization graph for nested transactions then we present the theorem.

The *serialization graph* of a CF $c = (s_1, A, <, r, s_2)$, $SG(c)$, is a directed graph whose nodes correspond to the nodes A in c and which has an edge $x \rightarrow y$ iff x and y are siblings and either

(a) There exist leaves x' and y' descendents of x and y respectively, so that x' precedes and conflicts with y'; or

(b) $x <_t y$ where t is the parent of x and y.

In our model once two operations conflict, this conflict propagates up to the level of the children of their least common ancestor which are their ancestors. Type (a) edges capture the order of conflicting operations. Edges of type (b) correspond to the transaction order or algorithmic precedence relation between siblings. We first prove a technical lemma, then we show that the acyclicity of our graph is both necessary and sufficient for conflict serializability. Our sufficiency part of the proof follows the proof in [3].

Lemma 3. Given a graph $G = (V, E)$, a set of vertices $C \subseteq V, |C| \geq 2$, and P, a set of two or more disjoint nonempty partitions of vertices in C. Consider a graph $G' = (P, E')$ where $p \rightarrow p' \in E'$ iff there exists $v \rightarrow v' \in E$ and $v \in p, v' \in p'$. If C represents vertices which are involved in a (simple) cycle in G then there is at least one (simple) cycle in G'.

Proof. Let f map $v \in C$ to its partition $f(v) \in P$. Consider the cycle of C's vertices $v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_n \rightarrow v_1$. If f is applied to each element v_i of the cycle the result after removing possible self-cycles is a cycle between nodes of G'. Since there are two or more partitions, there is at least one simple cycle among them. □

Theorem 1. (The Serializability Theorem) Let $c = (s_1, A, <, r, s_2)$ be a Computation Forest, c is conflict serializable iff $SG(c)$ is acyclic.

Proof. (if) We will show how to construct c_s a CF that is serial and conflict equivalent to c.

[Definition of $<_s$] Let the level of a node in $SG(c)$ be the number of its proper ancestors in A (thus $root(A)$ is at level 0). $<_s$ is the relation at the end of the execution of the following Hierarchical Topological Sort (HTS) [3]:

> $<_s := \{(x, y) : \text{there is an edge } x \rightarrow y \text{ in } SG(c)\}$
> **for** $l := 1$ **to** "max. level of any node in $SG(c)$" **do**
>   "extend $<_s$ to totally order all level l nodes"
>   "further extend $<_s$ by setting $x <_s y$ for any x and y that
>   are descendents of level l nodes x' and y' respectively with $x' <_s y'$ "
> **end for**

[$<_s$ is acyclic] To see this we use induction on l. The basis follows from the assumption that $SG(c)$ is acyclic. Suppose, by way of contradiction, that before iteration l $<_s$ is acyclic but afterwards it has a cycle among the nodes in the set C'. Let us instantiate $<_s$ and C' as the graph G and the set of vertices C in Lemma 3. The hypothetical cycle created by HTS at iteraction l must include one of the new newly introduced edges and all new edges are between nodes whose least common ancestor is at level $l-1$ or closer to the root. Hence node $lca(C')$[3] must be at level $l-1$ or closer to the root. Now consider nodes p and p' that are children of the node $lca(C')$. The way HTS constructs $<_s$ guarantees that $p \rightarrow p'$ iff $v \rightarrow v'$ where v and v' are descendents of p and p' respectively. Hence the subgraph of $<_s$ whose vertices are the children of node $lca(C')$ corresponds to the graph G' in Lemma 3. By Lemma 3 a cycle among any set of nodes C' implies that there is a cycle among children of $lca(C')$, i.e., a cycle among nodes at level l or above, but HTS totally orders level l and it does not add edges above level l, a contradiction. Hence at iteration l no cycle can be created and $<_s$ produced by HTS is acyclic.

[Definition of c_s] We now construct a hierarchical tuple $c_s = (s_1, A, <_s^+, r, s_2)$ where $<_s^+$ is the irreflexive transitive closure of $<_s$.

[c_s is a CF] To show that c_s is a CF we have to prove that it satisfies (C6), (C7) and (C8). $<_s^+$ in c_s satisfies (C6) by construction: The iterated pseudo-command "further extend $<_s$ by setting $x <_s y$ for any x and y that are descendents of level l nodes x' and y' respectively with $x' <_s y'$ " of HTS guarantees this. Obviously extending $<_s$ to $<_s^+$ does not violate (C6). c_s satisfies (C7) trivially,

[3] The lowest common ancestor of a set of nodes C is the ancestor of every $x \in C$, but no proper descendent of x has this property. Assuming that the nodes of a CF are organized as a tree guarantees a uniform treatment since for any set of nodes there exist a lowest common ancestor.

since all transaction orders are captured by edges of type (b) of $SG(c)$ and embedded in $<_s$ by the first pseudo-command of HTS. By assumption c is a CF. Thus the DM projection of c, $c[D]$, is a D-computation. We now show that if x conflicts with and precedes y in $<$ then x precedes y in $<_s$. Recall that type (a) edges of $SG(c)$, which were originally installed between siblings that have conflicting descendents, are embedded in $<_s$, and HTS extended all these orderings to the descendents. Hence $<_s^+ [leaves(A)]$ is conflict consistent with $< [leaves(A)]$, and hence the DM tuple $c_s[D]$ is conflict equivalent to the D-computation $c[D]$, and by Lemma 2 $c_s[D]$ is a D-computation. Hence c_s satisfies (C8).

[c_s is serial] c_s is serial by construction, the iterated pseudo-command "extend $<_s$ to totally order all level l nodes" guarantees this.

[c is conflict serializable] By definition of c_s the two CFs have the same A, and as discussed above $c_s[D] \equiv c[D]$ hence c_s is conflict equivalent to c.

(only if) Suppose CF $c = (s_1, A, <, r, s_2)$ is conflict serializable, and hence there exists a serial CF c_s such that $c \equiv c_s$. For the sake of contradiction assume that $SG(c)$ has a cycle among siblings $x_1 \to x_2 \to \ldots \to x_k \to x_1$. Consider $x_i \to x_{i+1}$ in $SG(c)$. If $x_i \to x_{i+1}$ is a type (a) edge then since $c \equiv c_s$, c and c_s have conflict consistent order relations and therefore x_i appears before x_{i+1} in c_s. If $x_i \to x_{i+1}$ is a type (b) edge then since both executions have the same nodes and obey (C4), the transaction validity axiom, again x_i appears before x_{i+1} in c_s. Hence a cycle implies that each of $x_1, x_2, \ldots x_k$ appears before itself in the serial CF c_s, an absurdity. Thus no cycle can exist in $SG(c)$. That is, if CF c is conflict serializable then $SG(c)$ is acyclic. □

The graph $SG(c)$ that we have defined differs in two main ways from that defined by Hadzilacos and Hadzilacos[3] (which we refer to as $SG_{HH}(c)$). Firstly, SG_{HH} includes edges between incomparable nodes even if those nodes are not siblings. For example, condition (a) gives edges in SG_{HH} between any incomparable ancestors of x and y. Similarly condition (b) gives edges between any descendent of x and any descendent of y. Secondly, the graph SG_{HH} does not include vertices corresponding to the atomic operations of an execution; instead it only has vertices for subtransactions. The extra vertices for atomic operations seem essential for the definition to capture the intuitive notion of correct execution. In both respects, our definition is closer to the serialization graph defined in [4].

4 An application of serializability theorem

In this section we derive a simple proof of correctness of a serialization graph testing scheduler [7]. Hadzilacos and Hadzilacos [3] show proofs for a lock and for a timestamp based protocols.

The *Children's Serialization Graph* of transaction t, $CSG(t)$, is a directed graph where nodes are the children of t. $CSG(t)$ has an edge $x \to y$ iff (a) there exists descendents x' and y' of x and y respectively, so that x' precedes and conflicts with y', or (b) $x <_t y$. It follows from the definition of $SG(c)$ and $CSG(t)$ that $CSG(t)$ corresponds to $SG(c)$ restricted to the nodes $children(t)$. Conversely $SG(c)$ corresponds to the union of the $CSG(t)$'s.

The CSGT (Children's Serialization Graph Testing) rules are as follows. A graph, called the Stored Children's Serialization Graph of t, $SCSG(t)$, is created when a subtransaction t is invoked. Initially $SCSG(t)$ has no vertices. Each time t invokes a child y, t informs the scheduler about any other terminated child x such that $x <_t y$. This information gives the scheduler the ability to install type (b) edges. Each time a child y of t terminates, the scheduler adds the vertex y to $SCSG(t)$, adds the type (b) edges, and adds an edge $y' \rightarrow y$ ($y \rightarrow y'$) for each terminated sibling y' such that y' (y) or one of its descendents precedes and conflicts with y (y') or one of its descendents. Immediately after doing this, the scheduler tests if $SCSG(t)$ has cycles. If there are no cycles this means that up to now all children can be serialized. If a cycle is detected then CSGT signals y's abortion to t. It follows from the definition of $CSG(t)$ and $SCSG(t)$ that $CSG(t)$ is the same as $SCSG(t)$ when t terminates.

Theorem 2. Every history produced by CSGT is conflict serializable.

Proof. Let c be a CF representing an execution that follows the rules of CSGT. We will prove that $SG(c)$ is acyclic. For the sake of contradiction suppose that $SG(c)$ has a cycle, we will now prove that a cycle among any group C' of nodes implies a cycle in $SCSG(lca(C'))$. Let G, C, and P of Lemma 3 be instantiated as $SG(c)$, C' and the equivalence classes defined by the relation "descendents of the same child of $lca(C')$" respectively. Observe that graph G' of Lemma 3 is a subgraph of $CSG(lca(C'))$. By Lemma 3 a cycle among nodes in C' implies that there is at least one cycle in $CSG(lca(C'))$. As observed above, when $lca(C')$ terminates $CSG(lca(C'))$ is the same as $SCSG(lca(C'))$, but CSGT checks for cycles for every child of a transaction, a contradiction. Hence $SG(c)$ cannot have cycles and by the serializability theorem c must be serializable. $\square$

References

[1] C. Beeri, P. A. Bernstein, and N. Goodman, "A Model for Concurrency in Nested Transactions Systems", *Journal of the Association for Computing Machinery*, 36(2):230-269, April 1989.

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

[3] T. Hadzilacos and V. Hadzilacos, "Transaction Synchronization in Object Bases", *Journal of Computer and System Sciences*, V43, N1:2-24, August 1991.

[4] A. Fekete, N. Lynch, and W. Weihl, "A Serialization Graph Construction for Nested Transactions", *Proceedings of the 9th ACM Symposium on Principles of Database Systems*, April 2-4, 1990, Nashville, Tennessee, ACM, New York, 1990, pp. 94-108.

[5] J. E. B. Moss, "Nested Transactions: An Approach to Reliable Distributed Computing", Ph.D. Thesis, Technical Report MIT/LCS/TR-260, April 1981.

[6] C. H. Papadimitriou, *The Theory of Database Concurrency Control*, Computer Science Press, 1986.

[7] R. F. Resende and A. El Abbadi, "A Graph Testing Concurrency Control Protocol for Object Bases", *Proceedings of the 4th International Conference on Computing and Information*, May 28-30, 1992, Toronto, Canada, IEEE, Los Alamitos, 1992, pp. 316-317.