Making Consistency More Consistent:
A Unified Model for Coherence, Consistency and Isolation

Adriana Szekeres
University of Washington
aaasz@cs.washington.edu

Irene Zhang
Microsoft Research
irene.zhang@microsoft.com

ABSTRACT
Ordering guarantees are often defined using abstract execution models [2, 8–11, 19, 22]. Unfortunately, these models are complex and make different assumptions about system semantics. As a result, researchers find it impossible to compare the ordering guarantees of coherence, consistency and isolation. This paper presents a simple, unified model for defining ordering guarantees that is sufficiently general to model a wide range of systems, including processor memory, distributed storage, and databases. We define a new single constraint relationship, result visibility, which formalizes the “appears to execute before” relationship between operations. Using only result visibility, we define more than 20 ordering guarantees from different research areas, including PRAM [17], snapshot isolation [6] and eventual consistency session guarantees [21]. To our knowledge, these definitions form the broadest survey of ordering guarantees using a single constraint in the current literature.

ACM Reference Format:

1 OVERVIEW
Ordering guarantees, such as those of coherence, consistency and isolation, help programmers reason about the behavior of complex systems. Researchers commonly define ordering guarantees using abstract execution models [2, 8, 9, 11, 19, 22]. These models explain the system behavior, i.e., why a system produces a certain output for a given input. To meet an ordering guarantee, a system must allow only outputs producible by at least one allowable execution for all inputs.

Unfortunately, abstract execution models can be complex and often make limiting assumptions about a system’s semantics (e.g., the system only supports reads and writes). Thus, it becomes difficult for programmers to understand and impossible for them to translate these abstract models across different types of systems. For example, Adya’s model [2] uses dependencies and anti-dependencies between reads and writes; this requires programmers to reason about complex dependencies between operations and works only for systems limited to read and write operations. In contrast, Burckhardt’s model [9] uses visibility and arbitration relationships with more complex operations (e.g., increment), which are difficult to translate to Adya’s dependencies. As a result, it is hard to compare isolation guarantees defined in Adya’s model with consistency guarantees in Burckhardt’s model. These semantic differences complicate efforts by researchers from different communities to share or compare ordering guarantees. Worse still, each community has developed a different vocabulary to discuss different ordering guarantees.

This paper presents a simple, unified model for defining ordering guarantees. Our model uses a single operation relationship, result visibility, which formalizes the “appears to execute before” relationships used to define many guarantees. We demonstrate that result visibility is both general and expressive by defining more than 20 existing coherence, consistency and isolation ordering guarantees. This paper makes three contributions:

1. A new state-machine-based abstract execution model that considers execution ordering and state. It makes no assumptions about operation semantics, so it is sufficiently general to model a wide range of systems, including processor memory, distributed storage and relational databases.
(2) A new operation relationship, result visibility, which defines an invariant across all executions that produce a single result for a given program.

(3) A new set of definitions for a range of coherence, consistency and isolation guarantees, where each definition is simply a set of result visibility invariants. To our knowledge, these definitions form the broadest survey of ordering guarantees using a single constraint in the current literature.

2 ABSTRACT MODEL

Ordering guarantees commonly define limits on system behavior using abstract execution models. These models “explain” the behavior of complex systems, so programmers can reason about a system’s ordering guarantees without understanding its inner workings. Choosing an execution model requires meeting two competing demands: the model must be abstract enough to cover a wide range of systems, while also concrete and simple enough for programmers to understand. Existing execution models do not meet our requirements because they are either limited to a specific system type (e.g., processor memory) or they are too complex (e.g., they introduce a large number of complicated relationships between executed operations).

To avoid these limitations, we introduce a general-purpose, state machine-based model, where we represent systems as a set of one or state machines. We then assume that each operation executes on a subset of these state machines at any point in time. Operations have no relationship with each other except for executing on the same state machine, which provides a single, simple relationship for programmers to understand. We make no assumptions about how many state machines a given system contains or when operations execute on each state machine, allowing us to model a wide range of systems; for example, an 8-core multi-processor becomes 8 concurrently executing state machines, while a distributed storage system with 3 replicas as 3 state machines.

2.1 Abstract System Model

Like previous models, we assume the system holds a set of named data objects to which it applies operations. The system accepts sequences of atomic operations as input, executes those operations on the objects, and produces output sequences with a return value for each operation. However, we make no assumptions about the types of operations or data objects. We assume only that each operation is atomic, accesses or modifies (or both) one or more data objects, and returns a value as a result.

We refer to system input sequences as programs and a complete input as a program set:

DEFINITION 1 (PROGRAM). An ordered sequence of atomic operations on one or more data objects.

DEFINITION 2 (PROGRAM SET). An unordered set of programs submitted as input to the system.

Programs first impose a partial order on the operations in a program set. We call this relationship program order and annotate it as \( \prec_p \). The system then outputs one return value per operation in each program. These values are collected into a result, and the output for a full program set is its result set.

DEFINITION 3 (RESULT). An ordered set of return values produced by the execution of an input program \( P \), corresponding to the operations of \( P \) and their order in \( P \).

DEFINITION 4 (RESULT SET). An unordered set of results produced by the execution of a given program set.

In general, our abstract system model resembles those previously defined [1, 9, 11]. However, we do not assume any operation semantics (e.g., reads or writes) unless it is necessary in order to define an ordering guarantee.

2.2 State Machine Abstract Execution Model

Our execution model takes a state-machine-based approach, where state is represented as a sequence of executed operations. Like past models [8, 9, 11], we assume that the system executes operations in some total sequential order to form its execution history. Thus, we can define an executes-before relationship between operations:

DEFINITION 5 (EXECUTES-BEFORE). Given a total execution order \( e \) of a program set \( P \) and two operations \( \alpha, \beta \in P \), \( \alpha \) executes-before \( (\prec_e) \beta \) iff \( \alpha \) precedes \( \beta \) in the sequential order imposed by \( e \).

Unlike previous models, we assume the system consists of one or more state machines executing in parallel, where each state machine has a complete copy of the system’s data objects but does not execute all operations in the execution history. Instead, at any point in time, we assume that each state machine has executed a subsequence of the system’s execution history. Thus, we define the execution state of an operation as follows:

DEFINITION 6 (EXECUTION STATE). Given a total execution order \( e \) of a program set \( P \), the execution state of operation \( \alpha \), \( \sigma_\alpha \), contains some sequence of operations \( \beta \) in execution order, where \( \beta \prec_e \alpha \).

We annotate execution states as subscripts on the executing operation. For example, consider a program with two writes and a read: \( w_{1x}, 0w_{2x}, 1rx \). We denote the
execution of the read operation as \( rx, 1_{w_{1},0}w_{2},x,1 \), which indicates that \( rx \) returned 1 and had an execution state consisting of the two preceding writes. Together, the execution ordering, return values, and state for all program set operations form our version of an execution history:

**Definition 7 (Execution).** A total sequential ordering of operations from the program set input. Each executing operation has a corresponding execution state, which determines the return value of the operation.

For an execution to be valid, each operation state must include, at a minimum, the preceding operations whose semantics dictated the return value of the corresponding operation. However, we put no constraints on the operation semantics, so they are completely defined by the system. For example, in our previous program, \( rx, 1_{w_{1},0}w_{2},x,1 \) and \( rx, 1_{w_{1},0}w_{2},x,1 \) are valid executions while \( rx, 1_{w_{1},0}w_{2},x,1 \) is not because the semantics of read and write operations imply that the read should reflect the latest write in the operation’s execution state.

Using execution state lets us easily model operations that execute in parallel or on different replicas. Those are simply operations in the execution history that do not appear in each other’s execution state. For example, take the following set of possible executions for our previous example program:

\[
e_1 : w_{1},x,\text{0} \prec_e w_{2},x,1, 1_{w_{1},0}w_{2},x,1 \prec_e \text{w}_{1},x,\text{0}w_{2},x,1
\]

\[
e_2 : w_{1},x,\text{0} \prec_e w_{2},x,1 \prec_e \text{w}_{1},x,\text{0}w_{1},x,0
\]

\[
e_3 : w_{1},x,\text{0} \prec_e w_{2},x,1 \prec_e \text{w}_{1},x,0w_{2},x,1
\]

Based on execution state, all operations in execution \( e_1 \) executed sequentially on the same state; however, in execution \( e_2 \), the read operation either executed in parallel with or on a different replica from the second write operation. In both cases, the execution clearly explains the returned results.

Our new execution model also effectively captures potential asynchronous and non-deterministic behavior. For example, the two writes in execution \( e_3 \) executed in parallel or on different replicas. However, we know there was some coordination or synchronization before the read because it executed on state that contains both write operations.

**2.3 New Properties and Equivalence**

Finally, we define relationships between executions that we later use to define result visibility. In our new model, the executes-before relationship, defined above as \( \prec_e \), is not meaningful without execution state. For example, in execution \( e_4 \) below, \( w_{1},x,\text{0} \prec_e w_{2},x,1 \) while in execution \( e_2 \), \( w_{2},x,1 \prec_e \text{w}_{1},x,\text{0} \); however, \( e_4 \) returns the same result as \( e_2 \).

\[
e_4 : w_{1},x,\text{0} \prec_e \text{w}_{1},x,\text{0}w_{1},x,0 \prec_e w_{2},x,1
\]

Therefore, we define a new strictly-executes-before relationship between two operations that captures ordering and state:

**Definition 8 (Strictly-executes-before).** Given an execution \( e \) of a program \( P \) and two operations \( \alpha, \beta \in P \), \( \alpha \) strictly-executes-before \( (\prec_\sigma) \beta \) iff \( \alpha \prec \beta \) and \( \alpha \in \sigma_\beta \).

The strictly-executes-before relationship guarantees that \( \alpha \) executes sequentially before \( \beta \) and that \( \alpha \) and \( \beta \) execute on shared state (i.e., they execute in sequence on the same state machine).

Note that different executions can lead to the same result set. We use the notation \( e \sim R \) to mean that execution \( e \) produced result set \( R \). For example, executions \( e_1 \) and \( e_3 \) from the previous example return the same result set. Without knowledge of system internals, programmers cannot distinguish between these executions, so we consider them to be equivalent:

**Definition 9 (Equivalent Execution).** Given an input program set \( P \), and two executions, \( e_1 \) and \( e_2 \) of \( P \), \( e_1 \) and \( e_2 \) are equivalent iff both executions return the same result set \( R \) (i.e., both \( e_1, e_2 \sim R \)).

We then define the set of all equivalent executions for a given program set that produce a particular result set as the equivalent execution set:

**Definition 10 (Equivalent Execution Set).** Given an input program set \( P \), the set \( E_P \) of all valid executions of \( P \), and a result set \( R \), the equivalent execution set, \( E_{P\sim R} \), is \( \{ e \in E_P \mid e \sim R \} \).

The equivalent execution set is needed to define result visibility: given a \( P \) and \( R \), our visibility constraint provides an invariant over all executions in \( E_{P\sim R} \).

**3 RESULT VISIBILITY**

Result visibility defines a formal equivalent to “operation A appears to run before B.” That is, if a system guarantees result visibility, then the programmer knows that it will produce only executions that appear to order the operations one at a time on a single copy of system state. Thus, we define result visibility as a single invariant on all executions that produce a single result set:

**Definition 11 (Result Visibility).** Given a program \( P \), operation \( \alpha \) is result visible to \( (\prec) \) operation \( \beta \) for result set \( R \) iff \( \forall e \in E_{P\sim R} \alpha \prec \beta : \exists \sigma e' \in E_{P\sim R} \alpha \prec_\sigma \beta \) and for all other pairs \( \delta, \gamma \) of result visible operations for \( R \) (i.e., \( \delta \prec \gamma \) for \( R \) \( \delta \sim_\sigma \gamma \in e \implies \delta \sim_\sigma \gamma \in e' \)).

Intuitively, result visibility guarantees that the system produces the return value to \( \beta \) by executing it after \( \alpha \) on a simple system. For executions in \( E_{P\sim R} \) where \( \alpha \prec_\sigma \beta \),
we need not perform any checks because they already executed α before β on the same state and met our invariant. For executions where α \not≺_σ β, there must be an equivalent execution where α \sim_σ β that does not break any of the other result visibility invariants.

With the result visibility invariant, programmers know that they can reason about α and β as operations executing in order on a simple state machine. For example, if our simple program uses a system that guarantees wx, 1 ⊈ rx, then we know that the system will always return 1. Result visibility is an equivalent, but easier to understand, relationship to allowable executions. In particular, for all result visibility constraints, results visibility implies that there exists at least one execution that leads to the result set where, if α \preceq β, then α \sim_σ β. We state the lemma formally and prove it in Appendix A:

**Lemma 1.** Given a program set P and a set of result visibility constraints V = {α, β ∈ P|α \preceq β for R}, a result set R satisfies V \iff ∃e ∈ E_P→_R such that for every pair α \preceq β ∈ V: α \sim_σ β in e.

Since result visibility is equivalent to ∃e|α \sim_σ β, and α \sim_σ β represents “executes sequentially on a single copy of system state,” we can consider result visibility a formalization of “α appears to execute before β on a simple state machine.” The result visibility invariant works as expected. In our previous program – w_1x, 0w_2x, 1rx – we could require w_1 \preceq w_2. However, this single constraint is not sufficient to impose a useful ordering on w_1 and w_2 since, without an intervening read, the two writes could execute in any order. Thus, to enforce program order, we need to add relationships between the writes and the read: w_1 \preceq r and w_2 \preceq r. These two constraints are sufficient to ensure that rx always returns 1. However, these three constraints do not unduly limit the execution; for example, the following executions are all allowable:

\[
\begin{align*}
e_1: & \quad w_1x, 0 \sim_e w_2x, 1w_1z, 0 \sim_e rx, 1w_1x, 0w_2x, 1 \\
e_2: & \quad w_1x, 0 \sim_e w_2x, 1 \sim_e rx, 1w_1x, 0w_2x, 1 \\
e_3: & \quad w_2x, 1 \sim_e w_1x, 0 \sim_e rx, 1w_2x, 1
\end{align*}
\]

Even with these three constraints, note that result visibility still allows executions where \(w_1 \not\sim_e w_2\) and \(w_1 \not\sim_σ w_2\): from the programmer’s perspective, these executions still “appear” (i.e., are equivalent) to an execution where \(w_1\) executed before \(w_2\) on a simple state machine.

## 4 Unified Ordering Guarantees

Ordering guarantees dictate the result sets that a system returns for a given program. *Coherence* typically refers to ordering guarantees on single data objects, while *consistency* covers multi-object guarantees. In contrast, *isolation* assumes operations are grouped into transactions and restricts how operations from concurrently executing transactions can interleave. Table 1 uses result visibility to define a range of coherence, consistency and isolation ordering guarantees. We formally define the stricter ordering guarantees in each category below.

### 4.1 Sequential Consistency

Sequential consistency simplifies a complex system by having it appear to execute operations one at a time on a single copy of system state.

**Definition 12 (Sequential Consistency).** A system provides sequential consistency if, for any given program set, P, it produces only result sets R where: (SC1) for every two operations where \(α \sim_p β, α \preceq β\), and (SC2) for every two operations \(α, β\) in different programs, either \(α \preceq β\) or \(β \preceq α\).

Appendix B.1 proves that sequential consistency limits systems to result sets that lead from sequentially consistent executions, making it consistent with existing definitions [16].

### 4.2 Serializability

To use our execution model for isolation, we consider each program in the program set to be a separate transaction. For isolation guarantees, we refer to programs and program sets as transactions and transaction sets, respectively. Because isolation guarantees apply to all operations in a transaction, we define a new set visibility property for simplicity:

**Definition 13 (Set Visibility).** Given two sets \(S_1\) and \(S_2\), \(S_1\) is set visible to \(S_2\) if \(∀α ∈ S_1, ∀β ∈ S_2|α \sim β\).

We also allow set visibility between a set and a single operation and vice versa. Serializability [7] resembles a sequential consistency guarantee. However, instead of a sequential ordering of operations, serializability requires a sequential ordering of transactions. That is, transaction appear as if they ran one at a time on a single copy of system state. Program ordering is always maintained within operations in a transaction, but there are no additional ordering guarantees across transactions (e.g., all read-only transactions could be ordered first and not return any writes). For this reason, some researchers consider serializability to be weaker than sequential consistency.

**Definition 14 (Serializability).** A database system provides serializability if for any given transaction set \(T\), it produces only result sets \(R\) where, for every two committed transactions \(t_1, t_2 \in T\), either \(t_1 \preceq t_2\) or \(t_2 \preceq t_1\).
Table 1: Ordering guarantees. We use result visibility to define a range of coherence and consistency models. Checkmarks indicate constraints that are enforced in the model. We assume that programs, sessions and transactions are equivalent. Note that, because result visibility is an invariant, the constraints are additive and there is no need to mention executions. As a result, we can compactly represent a wide range of ordering guarantees using only result visibility to construct the columns of this table.

<table>
<thead>
<tr>
<th>Coherence (same object)</th>
<th>Consistency (different objects)</th>
<th>Session Guarantees</th>
<th>Isolation (transactional)</th>
<th>Real-time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program ordering</td>
<td>Write Atomicity</td>
<td>Program ordering</td>
<td>Write Atomicity</td>
<td>Monotonic Views</td>
</tr>
<tr>
<td>α≺β</td>
<td>w₁ ≤ w₂ ≤ w₃ ≤ γ</td>
<td>α≺β</td>
<td>w₁ ≤ w₂ ≤ w₃ ≤ γ</td>
<td>w₁ ≤ r₁ &lt; r₂</td>
</tr>
<tr>
<td>w₁ &lt; w₂ &lt; w₃ &lt; w₄ &lt; w₅</td>
<td>w₁ ≤ w₂ ≤ w₃ &lt; w₄ ≤ w₅</td>
<td>w₁ &lt; w₂ &lt; w₃ &lt; w₄ &lt; w₅</td>
<td>w₁ &lt; w₂ &lt; w₃ &lt; w₄ &lt; w₅</td>
<td>w₁ &lt; w₂ &lt; w₃ &lt; w₄ &lt; w₅</td>
</tr>
</tbody>
</table>

Strict Serializability
Serializability [7]
Snapshot Isolation [6]
Parallel SI [20]
Non-monotonic SI [4]
Monotonic Atomic Views [5]
Repeatability
Read Committed [14]
Read Uncommitted [14]
Linearizability [15]
Sequential Consistency [16]
Total Store Order [1]
Partial Store Order [1]
Weak Ordering [12]
Processor Consistency [13]
PRAM [17]
Cache Consistency [13]

Read-Your-Writes [21]
Monotonic Reads [21]
Monotonic Writes [21]
Writes-Follow-Reads [21]

1Only if WS₁ ∩ WS₂ ≠ ∅.
We ensure that transactions appear to execute in a serial order by requiring that either all of the operations from one transaction are result visible to the other or vice versa. In Adya’s model [2], this requirement would translate to no cycles in the dependency graph. In Crooks’s definitions [11], the requirement is equivalent to each transaction having a complete parent. Appendix B.2 proves equivalence between our and Crooks’s definitions.

4.3 Real-time Guarantees

Some ordering guarantees have real-time requirements. The most commonly used ones are linearizability [15] and strict serializability. To define these guarantees, we add real-time properties to our abstract execution model.

**Definition 15 (Strictly-Returns-Before).** Given a program set \( P \) and two operations \( \alpha, \beta \in P \), we define \( \alpha \) strictly-returns-before \( (\prec_r) \) \( \beta \) iff the system produces the return value to \( \alpha \) before the programmer submits \( \beta \).

The strictly-returns-before relationship is a partial ordering; if \( \alpha \not\prec_r \beta \) and \( \beta \not\prec_r \alpha \), then the two operations are considered to be externally concurrent (i.e., the programmer submits both operations at the same time). External concurrency does not reflect internal concurrency (i.e., the system executes both operations in parallel), although it is impossible to have internal without external concurrency. Given this new relationship, we define linearizability as an additional constraint on sequential consistency:

**Definition 16 (Linearizability).** A system provides linearizability if, for any given program set, \( P \), it produces only result sets \( R \) where: (L1) for every two operations where \( \alpha \prec_r \beta \), \( \alpha \prec \beta \), and (L2) for every two operations \( \alpha \not\prec_r \beta \) and \( \beta \not\prec_r \alpha \), either \( \alpha \not\prec \beta \) or \( \beta \not\prec \alpha \).

Strict serializability is a similar addition to serializability:

**Definition 17 (Strict Serializability).** A system provides strict serializability if, for any program set \( P \), it produces only result sets \( R \) where: (SS1) for every two transactions \( t_1, t_2 \in P \), if all operations \( \alpha \in t_1 \) and \( \beta \in t_2 \), \( \alpha \prec_r \beta \), then \( \alpha \prec \beta \); otherwise: (SS2) for every two transactions \( t_1, t_2 \in P \), either \( \forall \alpha \in t_1, \forall \beta \in t_2 | \alpha \prec \beta \) or \( \forall \alpha \in t_1, \forall \beta \in t_2 | \beta \prec \alpha \).

Although we included an additional real-time relationship between operations, both linearizability and strict serializability are easily expressed in our model. Further, the real-time relationship is simply a pre-condition, similar to program ordering or transactions; result visibility remains sufficient to constrain allowable executions. A similar technique could be used to add other pre-condition relationships needed to define different classes of ordering guarantees. For example, adding a causality relationship between operations would let us define causal ordering guarantees [3, 18].

5 SUMMARY

In this paper, we specified a new unified framework for defining coherence, consistency and isolation guarantees. Our framework includes a new abstract execution model, an invariant on executions, and the ability to define ordering guarantees for processor memory to database systems. We demonstrate the power and expressiveness of our framework by capturing definitions for more than 20 ordering guarantees, from PRAM [17] to snapshot isolation [6]. To our best knowledge, the definitions collected in Table 1 represent the broadest survey of ordering guarantees using a single constraint in the current literature.

**REFERENCES**


for all pairs of operations with visibility relationships in $E$. Now, it is easy to see that we can repeat the same process since there is some other pair of operations $\gamma \prec_\beta \delta$ in $\sigma$. If any executions $e_{\gamma} \in E_{P \to \sigma}\setminus\alpha \prec_\sigma \beta$ automatically has a matching $e_{\delta}^\ast \in e$ in $E_{P \to \sigma}$, then there exists at least one execution in $E_{P \to \sigma}$ to meet the visibility condition for all pairs $\alpha \prec_\beta e$. □

### B PROOFS OF EQUIVALENCE FOR ORDERING GUARANTEE DEFINITIONS

#### B.1 Sequential Consistency Proof

We prove that systems which provide sequential consistency produce the same result sets as sequentially consistent executions.

**THEOREM 18 (SEQUENTIALLY CONSISTENT EXECUTION).** Given a program set $P$, we define a sequentially-consistent execution $e$ as one where: (S1) for every two operations $\alpha \prec_\beta p$, $\alpha \prec_\beta \beta$ in $e$; and (S2) for every two operations $\alpha \prec_\beta e$, $\alpha \prec_\beta e$ in $e$. Execution $e$ imposes a total order of the $\prec_\sigma$ relation that preserves program order, and thus it is a sequentially-consistent execution.

**PROOF.** (SC1) and (SC2) form a set of visibility constraints. By Lemma 1 this means that there exists an execution $e$ such that: (1) for every two operations $\alpha \prec_\beta p$, $\alpha \prec_\beta \beta$ and (2) for every two operations $\alpha \prec_\beta e$, $\alpha \prec_\beta e$ in $e$. Execution $e$ imposes a total order of the $\prec_\sigma$ relation that preserves program order, and thus it is a sequentially-consistent execution.

#### B.2 Isolation Proofs

We prove that our definition for isolation are equivalent to the definitions in Crooks [11].

**THEOREM 3 (SERIALIZABILITY).** **DEPENDENCY-CYCLE-FREE(\text{WS} \prec \text{WS}, \text{RS} \prec \text{WS}, \text{WS} \prec \text{RS}) \equiv \exists e : \forall t \in T_\text{C} : \text{COMPLETE}_e \not\ni s_p.

**PROOF.** The visibility constraints for serializability require a sequential visibility order on committed transactions. By Lemma 1 this means that there must exist a valid execution $e$ such that for every committed transaction $t$ all its operations executed on a state machine which executed, in the given execution order, all previous operations from the previous transactions which precede $t$ in the sequential order. This execution is equivalent to an execution in which for every committed transaction its parent state is complete, which also imposes a sequential
order of all transactions’ operations, i.e. every transac-
tions reads from a state which was obtained by executing
all preceeding transactions.

THEOREM 4 (SNAPSHOT ISOLATION). \( \text{DEPENDENCY-}
\text{CYCLE-FREE}(WS \prec WS, RS \prec WS, WS \prec RS^*) \land \text{CONSISTEN-
T-READS} \equiv \exists e : \forall t \in T_C : \exists s \in S_e, \text{COMPLETE}_{e,s} \land \text{NO-CONF}_t \).

PROOF. The visibility constraints for snapshot isolation require a sequential order of all committed transac-
tions such that every transaction’s reads and writes appear
to have executed before the writes of all the subsequent
transactions. Additionally, if two transactions have write-
write conflicts, they are completely ordered with respect
to eachother, i.e., the writes of the earlier transaction in
the sequence must appear to have executed before the
reads of the later transaction. By Lemma 1 this means
that there must exist an execution \( e \) where transactions
actually executed as described above, i.e., with the corre-
sponding \( \prec_\sigma \) relations. This execution is so far identical
to an execution where \( \forall t \in T_C : \text{PREREAD}_e.t \) and where
write-write conflicting transactions are totally ordered.
The additional set of constraints, \( \text{CONSISTENT – READS} \),
limit the transactions to reading from a committed snap-
shot, i.e., if transaction \( t_1 \) reads from another transaction
\( t_2 \), thus forming a visibility constraint, all of the writes
of \( t_1 \) must be visible to all the reads of \( t_1 \). This limits our
execution \( e \) to an execution where for every transaction \( t \)
there is a state such that \( \text{COMPLETE}_{e,s} \), i.e. each trans-
action reads from a snapshot of committed transactions,
and no write-write conflicting transaction committed be-
fore \( t \) since that state, i.e. \( \text{NO-CONF}_t \).

THEOREM 5 (READ COMMITTED). \( \text{DEPENDENCY-}
\text{CYCLE-FREE}(WS \prec WS, RS \prec WS) \land \text{NO-INTERMEDIATE-}
\text{READS} \equiv \exists e : \forall t \in T_C : \text{PREREAD}_e.t \).

PROOF. The visibility constraints for read committed
require a sequential order of all committed transac-
tions such that every transaction’s reads and writes appear
to have executed before the writes of all the sub-
sequent transactions. Additionally, if a visibility con-
straint exists from a write in a transaction \( t_1 \) to a read
in a more recent transaction, then all the writes in \( t_1 \)
to that object must be visible to the read as well, i.e.
\( \text{NO – INTERMEDIATE – READS} \). By Lemma 1 this
means that there must exist an execution \( e \) where transac-
tions actually executed as described above, i.e., with the corre-
sponding \( \prec_\sigma \) relations. This execution is identical
to an execution where \( \forall t \in T_C : \text{PREREAD}_e.t \), i.e., where
for every operation in every transaction there must exist
a read state that precedes the transaction’s commit state –
by definition this read state does not contain intermediate
reads.