We consider a sequential specification defined by the language $S = (a())^*(b())^*$ where all the invocations of $a()$ occur before invocations of $b()$.

1. Describe a reduction of checking linearizability w.r.t. the specification $S$ to a reachability problem. More precisely, describe a labeled transition system (monitor) that accepts exactly all the histories of a given implementation (sequences of call and return actions) that are not linearizable w.r.t. $S$. The synchronized product between a transition system representing an implementation and this monitor (where the synchronization actions are call and returns) reaches an accepting state of the monitor iff the implementation is not linearizable.
Exercices (2)

- What is the complexity of checking linearizability of a differentiated history of a concurrent queue?
Exercises (2)

- What is the complexity of checking linearizability of a differentiated history of a concurrent queue?

“Value v dequeued without being enqueued”

```
deq: v
```

“Value v dequeued before being enqueued”

```
deq: v  enq: v
```

“Value v dequeued twice”

```
deq: v  deq: v
```

“Values dequeued in the wrong order”

```
enq: v_1  enq: v_2  deq: v_2  deq: v_1
```

“Dequeue wrongfully returns empty”

```
deq: empty
```

```
enq: v_1  deq: v_1
```

```
enq: v_2  deq: v_2
```

```
enq: v_n  deq: v_{n-1}
```

```
enq: v_n  deq: v_n
```

```
```
Weak Consistency

Constantin Enea
IRIF, University Paris Diderot
Replicated objects

Distributed systems

Conflicting concurrent updates: how are they observed on different replicas?

Adversarial environments: crashes, network partitions
Pessimistic Replication

Using consensus algorithms to agree on an order between conflicting concurrent updates

- **Strong consistency**
- **Availability**

CAP theorem [Gilbert et al.’02]: strong cons. + availability + partition tolerance is impossible
Optimistic Replication

Each update is applied on the local replica and propagated **asynchronously** to other replicas.

- **Strong consistency** is **false**.
- **Availability** is **true**.

Replicas may store different versions of data: **weak consistency**.
Optimistic data replication

Optimistic replication: replicas are allowed to diverge
– operations are applied immediately at the submission site
Optimistic data replication

Optimistic replication: replicas are allowed to diverge
– operations are applied immediately at the submission site
– in the background, sites exchange and apply remote operations
Concurrent operations

{2, 3}

add 3

{2, 3}

rem 3

{2, 3}

{2, 3}

{2, 3}
Concurrent operations

Solving conflicts between concurrent operations
- speculate and roll-back, e.g., Google App Engine Datastore
Concurrent operations

Solving conflicts between concurrent operations

- speculate and roll-back, e.g., Google App Engine Datastore
Concurrent operations

Solving conflicts between concurrent operations

- speculate and roll-back, e.g., Google App Engine Datastore
Concurrent operations

Solving conflicts between concurrent operations

– speculate and eventually, roll-back, e.g., Google App Engine Datastore
– convergent conflict resolution, e.g., CRDTs [Shapiro et al.'11]

- Correct operations? Allowed level of consistency between replicas?
  - by CAP theorem [Gilbert et al.'02], achieving strong consistency (linearizability) is impossible
  - various correctness criteria: eventual consistency, causal consistency, etc
Example: Key-value map

```plaintext
struct Timestamp(number: nat; rid: nat);
function lessthan(Timestamp(n1, rid1), Timestamp(n2, rid2)) : boolean {
  return (n1 < n2) \lor (n1 == n2 \land rid1 < rid2);
}

message Update(key: Key, val: Val, ts: Timestamp) : reliable

type Replica(rid: nat) {
  var localclock: nat;
  var store: pmap(Key, pair(Val, Timestamp));

  operation read(key: Key) {
    match store[key] with
    ⊥ → { return undef; }
    (val, ts) → { return val; }
  }

  operation write(key: Key, val: Val) {
    localclock++; // advance logical clock
    store[key] := (val, ts);
    send Update(key, val, Timestamp(localclock, rid));
    return ok;
  }

  receive Update(key, val, ts) {
    if (store[key] = ⊥ \lor store[key].second.lessthan(ts))
      store[key] := (val, ts);
    if (ts.number > localclock) // keep up with time
      clock := ts.number;
  }
```

Figure 6.7: A broadcast-based eventually consistent key-value store with last-writer-wins resolution

Fkvs.
operation write(key: Key, val: Val) {
    localclock++; // advance logical clock
    store[key] := (val, ts);
    send Update(key, val, Timestamp(localclock, rid));
    return ok;
}

receive Update(key, val, ts) {
    if (store[key] = ⊥ ∨ store[key].second.lessthan(ts))
        store[key] := (val, ts);
    if (ts.number > localclock) // keep up with time
        clock := ts.number;
}

struct Timestamp(number: nat; rid: nat);
function lessthan(Timestamp(n1, rid1), Timestamp(n2, rid2)) : boolean {
    return (n1 < n2) ∨ (n1 == n2 ∧ rid1 < rid2);
}
operation write(key: Key, val: Val) {
    localclock++; // advance logical clock
    store[key] := (val, ts);
    send Update(key, val, Timestamp(localclock, rid));
    return ok;
}

receive Update(key, val, ts) {
    if (store[key] = ⊥ ∨ store[key].second.lessthan(ts))
        store[key] := (val, ts);
    if (ts.number > localclock) // keep up with time
        clock := ts.number;
}

struct Timestamp(number: nat; rid: nat);
function lessthan(Timestamp(n1, rid1), Timestamp(n2, rid2)) : boolean {
    return (n1 < n2) ∨ (n1 == n2 ∧ rid1 < rid2);
}
operation write(key: Key, val: Val) {
    localclock++; // advance logical clock
    store[key] := (val, ts);
    send Update(key, val, Timestamp(localclock, rid)) ;
    return ok;
}

receive Update(key, val, ts) {
    if (store[key] = ⊥ ∨ store[key].second.lessthan(ts))
        store[key] := (val, ts);
    if (ts.number > localclock) // keep up with time
        clock := ts.number;

};
Example: OR-Set

payload set $E$, set $T$

initial $\emptyset$, $\emptyset$
query contains (element $e$) : boolean $b$
let $b = (\exists n : (e, n) \in E)$
query elements () : set $S$
let $S = \{e|\exists n : (e, n) \in E\}$
update add (element $e$)
  prepare (e)
  let $n = unique()$
  effect $(e, n)$
  $E := E \cup \{(e, n)\} \setminus T$
update remove (element $e$)
  prepare (e)
  let $R = \{(e, n)|\exists n : (e, n) \in E\}$
  effect ($R$)
  $E := E \setminus R$
  $T := T \cup R$
Figure 2 shows our specification for an add-wins replicated set CRDT. Its concurrent state from another replica of the same object. The monotonic evolution of replica states is described by a compare operation, supplied with each CRDT specification.

An update applies its side effect, which is called the source. To this effect, an update is modeled as an operation; the source executes the prepare and effect first to the source replica, then (eventually) at all replicas.

update \textit{add} (element \( e \))
\begin{align*}
\text{prepare} (e) \\
\text{let } n = \text{unique}(e) \\
\text{effect} (e, n) \\
E := E \cup \{(e, n)\} \setminus T
\end{align*}

update \textit{remove} (element \( e \))
\begin{align*}
\text{prepare} (e) \\
\text{let } R = \{(e, n)\mid \exists n : (e, n) \in E\} \\
\text{effect} (R) \\
E := E \setminus R \\
T := T \cup R
\end{align*}
Figure 2 shows our specification for an add-wins replicated set CRDT. Its concurrent specification allows an update to be applied to a source replica, and then propagated downstream to other replicas.

**4.1 Observed Remove Set**

The monotonic evolution of replica states is crucial for maintaining consistency. A replica state can evolve by applying an update operation or by merging with another replica state. The notation is easy to infer from the standard CRDT model [9, 8, 10].

An optimized conflict-free replicated set model is defined as follows:

- **update** `add (element e)`
  ```
  prepare (e)
  let n = unique()
  effect (e, n)
  E := E ∪ {(e, n)} \ T
  ```

- **update** `remove (element e)`
  ```
  prepare (e)
  let R = {(e, n)|∃n : (e, n) ∈ E}
  effect (R)
  E := E \ R
  T := T ∪ R
  ```
**update** *add* (element *e*)

prepare (*e*)

let *n* = *unique*()

effect (*e*, *n*)

\[ E := E \cup \{(e, n)\} \setminus T \]

**update** *remove* (element *e*)

prepare (*e*)

let *R* = \{(e, *n*)|\*n* : (e, *n*) \in E\}

effect (*R*)

\[ E := E \setminus R \]
\[ T := T \cup R \]
update *add* (element e)
  prepare (e)
    let n = *unique*()
  effect (e, n)
    $E := E \cup \{(e, n)\} \setminus T$

update *remove* (element e)
  prepare (e)
    let $R = \{(e, n) | \exists n : (e, n) \in E\}$
  effect (R)
    $E := E \setminus R$
    $T := T \cup R$
Formal Definitions

- Histories
- Abstract Executions
- Operation Context
- Replicated Data Types
- Return-value Consistency

(Using the formal framework in “Principles of Eventual Consistency” by S. Burckhardt)
Histories

Definition 3.1 (History). A history is an event graph \((E, \text{op}, \text{rval}, \text{rb}, \text{ss})\) where

(h1) \(\text{op} : E \rightarrow \text{Operations}\) describes the operation of an event.

(h2) \(\text{rval} : E \rightarrow \text{Values} \cup \{\nabla\}\) describes the value returned by the operation, or the special symbol \(\nabla\) \((\nabla \notin \text{Values})\) to indicate that the operation never returns.

(h3) \(\text{rb}\) is a natural partial order on \(E\), the returns-before order.

(h4) \(\text{ss}\) is an equivalence relation on \(E\), the same-session relation.

Definition 3.2 (Well-formed History). A history \((E, \text{op}, \text{rval}, \text{rb}, \text{ss})\) is well-formed if

(h5) \(x \xrightarrow{\text{rb}} y\) implies \(\text{rval}(x) \neq \nabla\) for all \(x, y \in E\).

(h6) for all \(a, b, c, d \in E\): \((a \xrightarrow{\text{rb}} b \land c \xrightarrow{\text{rb}} d) \Rightarrow (a \xrightarrow{\text{rb}} d \lor c \xrightarrow{\text{rb}} b)\).

(h7) For each session \([e] \in E/\approx_{ss}\), the restriction \(\text{rb}|_{[e]}\) is an enumeration.
Abstract Executions

Definition 3.3 (Abstract Executions). An abstract execution is an event graph \((E, \text{op}, \text{rval}, \text{rb}, \text{ss}, \text{vis}, \text{ar})\) such that

(a1) \((E, \text{op}, \text{rval}, \text{rb}, \text{ss})\) is a history.

(a2) \text{vis} is an acyclic and natural relation.

(a3) \text{ar} is a total order.

Definition 4.4 (Operation Context). An operation context is a finite event graph \(C = (E, \text{op}, \text{vis}, \text{ar})\) where \(\text{op} : E \to \text{Operations}\) describes the operation of each event, \(\text{vis}\) is an acyclic relation representing visibility among the elements of \(E\), and \(\text{ar}\) is a total order representing arbitration of the elements in \(E\). We let \(C\) be the set of all operation contexts.
Replicated Data Types

**Definition 4.5 (Replicated Data Type).** A replicated data type \( F \) is a function \( \text{Operations} \times C \rightarrow \text{Values} \) that, given an operation \( o \) and an operation context \( C \), specifies the expected return value \( F(o, C) \) to be used when performing \( o \) in context \( C \), and which does not depend on the events, *i.e.* is the same for isomorphic (as in Definition 2.2) contexts: \( C \approx C' \Rightarrow F(o, C) = F(o, C') \) for all \( o, C, C' \).

**Replicated Counter:** a read returns the number of increment operations in the context

\[
F_{\text{ctr}}(\text{rd}, (E, \text{op}, \text{vis}, \text{ar})) = |\{e' \in E | \text{op}(e') = \text{inc}\}|
\]

**Last-Writer-Wins Register:** a read returns the value of the last write in the context (w.r.t. arbitration order), or “\text{undef}”, if there is no write

\[
F_{\text{reg}}(\text{rd}, (E, \text{op}, \text{vis}, \text{ar})) = \begin{cases} 
\text{undef} & \text{if } \text{writes}(E) = \emptyset \\
v & \text{if } \text{op}(\max_{\text{ar}} \text{writes}(E)) = \text{wr}(v)
\end{cases}
\]

**Multi-Value Register:** a read returns a set of values, one for each write in the context that has not been overwitten by some other write

\[
F_{\text{mvr}}(\text{rd}, (E, \text{op}, \text{vis}, \text{ar})) = \\
\{v | \exists e \in E : \text{op}(e) = \text{wr}(v) \text{ and } \forall e' \in \text{writes}(E) : e \not\rightarrow^{\text{vis}} e'\}
\]
payload set $E$, set $T$

initial $\emptyset, \emptyset$

query contains (element $e$) : boolean $b$
  let $b = (\exists n : (e, n) \in E)$

query elements () : set $S$
  let $S = \{e|\exists n : (e, n) \in E\}$

update add (element $e$)
  prepare $(e)$
    let $n = unique()$

  effect $(e, n)$
    $E := E \cup \{(e, n)\} \setminus T$

update remove (element $e$)
  prepare $(e)$
    let $R = \{(e, n)|\exists n : (e, n) \in E\}$

  effect $(R)$
    $E := E \setminus R$
    $T := T \cup R$
Return-Value Consistency

**Definition 4.8.** For a replicated data type $\mathcal{F}$, we define the return value consistency guarantee as

$$RVal(\mathcal{F}) \overset{\text{def}}{=} \forall e \in E : \text{ rval}(e) = \mathcal{F}(\text{op}(e), \text{context}(A, e))$$

where context is defined as follows:

**Definition 4.9.** Let $A = (E, \text{op}, \text{rval}, \text{rb}, \text{ss}, \text{vis}, \text{ar})$ be an abstract execution containing an event $e \in E$. Then

$$\text{context}(A, e) \overset{\text{def}}{=} A|_{\text{vis}^{-1}(e), \text{op}, \text{vis}, \text{ar}}$$
Formal Definitions

\[
\begin{align*}
\text{ReadMyWrites} & \quad \text{def} \quad (so \subseteq \text{vis}) \\
\text{MonotonicReads} & \quad \text{def} \quad (\text{vis};so) \subseteq \text{vis} \\
\text{ConsistentPrefix} & \quad \text{def} \quad (ar;(\text{vis} \cap \neg\text{ss})) \subseteq \text{vis} \\
\text{NoCircularCausality} & \quad \text{def} \quad \text{acyclic}(hb) \\
\text{CausalVisibility} & \quad \text{def} \quad (hb \subseteq \text{vis}) \\
\text{CausalArbitration} & \quad \text{def} \quad (hb \subseteq \text{ar}) \\
\text{Causality} & \quad \text{def} \quad \text{CausalVisibility} \land \text{CausalArbitration} \\
\text{SingleOrder} & \quad \text{def} \quad \exists E' \subseteq \text{rval}^{-1}(\nabla) : \text{vis} = \text{ar} \setminus (E' \times E) \\
\text{Realtime} & \quad \text{def} \quad rb \subseteq \text{ar}
\end{align*}
\]

\[
\begin{align*}
\text{Linearizability}(\mathcal{F}) & \quad \text{def} \quad \text{SingleOrder} \land \text{Realtime} \land \text{RVal}(\mathcal{F}) \\
\text{SequentialConsistency}(\mathcal{F}) & \quad \text{def} \quad \\
& \quad \text{SingleOrder} \land \text{ReadMyWrites} \land \text{RVal}(\mathcal{F}) \\
\text{CausalConsistency}(\mathcal{F}) & \quad \text{def} \quad \\
& \quad \text{EventualVisibility} \land \text{Causality} \land \text{RVal}(\mathcal{F}) \\
\text{BasicEventualConsistency}(\mathcal{F}) & \quad \text{def} \quad \\
& \quad \text{EventualVisibility} \land \text{NoCircularCausality} \land \text{RVal}(\mathcal{F})
\end{align*}
\]

EventualVisibility: the nb. of operations not “seeing” an operation is finite
Causal Consistency (CC) [Lamport’78]

If an update is visible, then all the updates is dependent on should be also visible

- write(x,1) and write(y,1) are causally dependent:

```plaintext
write(x,1)
write(y,1)
```

```plaintext
read(y): 1
read(x): 1 0
```
Causal Consistency (CC) [Lamport’78]

If an update is visible, then all the updates it is dependent on should be also visible

- write(x,1) and write(y,1) are causally dependent:
Causal Consistency (CC)

If an update is visible, then all the updates is dependent on should be also visible

- write(x,1) and write(y,1) are concurrent:
Formalization: Visibility

A is visible to operation B at replica R if the effect of A had been included in R at the time when B was executed.

∃ \text{vis}. \text{vis} \supseteq \text{po} \land \text{vis} \text{ transitive} \\
\land \forall \text{read. \text{return value} is consistent with the set of visible ops}
Formalization: Visibility

A is visible to operation B at replica R if the effect of A had been included in R at the time when B was executed.

∃ vis. vis ⊇ po ∧ vis transitive ∧ ∀ read. "return value is consistent with the set of visible ops"
Arbitration: conflict resolution between concurrent writes using timestamps

Formalization: Arbitration

[∃ vis, arb. arb ⊇ vis ⊇ po ∧ vis transitive ∧ arb total order ∧ ∀ read. “return value = value of the last visible write in arbitration order”]

Arbitration: conflict resolution between concurrent writes using timestamps
Formalization: Arbitration

[Burckhardt et al.’14]

Arbitration: conflict resolution between concurrent writes using timestamps

∃ vis, arb. arb ⊑ vis ⊑ po ∧ vis transitive ∧ arb total order ∧ ∀ read. “return value = value of the last visible write in arbitration order”
Checking CC for a single execution is NP-complete (proof on whiteboard)

Checking CC for a finite-state implementation (given as a regular language) and a regular specification is undecidable
Characterizing CC

\[ \exists \text{vis}, \text{arb}. \text{arb} \supseteq \text{vis} \supseteq \text{po} \wedge \text{vis} \text{ transitive} \wedge \text{arb} \text{ total order} \wedge \forall \text{read. } \text{``return value} = \text{value of the last visible write in arbitration order''} \]

is equivalent to

\[ \exists \text{rf}. \text{po} \cup \text{rf} \text{ is acyclic and } (\text{po} \cup \text{rf}); \text{rb} \text{ is acyclic and } \text{po} \cup \text{rf} \cup \text{cf} \text{ is acyclic} \]

- **rf (read-from):** relating each read to a write with the same variable/value
- **rb (read-before):** relating each read to a write that overwrites the read value
- **cf (conflict):** a necessary subset of \text{arb} derived from \text{po} and \text{rf}
Characterizing CC

CC = ∃ rf. … and
  po ∪ rf ∪ cf is acyclic

Example:

write(x,1) → write(x): 2

write(x,_) → write(x,_)
Characterizing CC

\[ CC = \exists \text{rf. ... and } \text{po} \cup \text{rf} \cup \text{cf} \text{ is acyclic} \]

Example:

\[ \text{write}(x,1) \xleftarrow{\text{cf}} \text{write}(x): 2 \]
\[ \text{read}(x): 2 \]

\[ \text{write}(x): 2 \]
\[ \text{read}(x): 1 \]

\[[POPL'17]\]
Checking CC

CC = ∃ rf. po ∪ rf ∪ rb is acyclic and
    po ∪ rf ∪ cf is acyclic

Each value is written once (data independence) => fixed read-from

Testing: acyclicity can be checked in polynomial time

Verification: using finite-state automata to represent “cyclic” executions