Verification of Concurrent Systems under Weak Consistency Models

Ahmed Bouajjani
University of Paris

MPRI, Paris, December 2020
Lecture 1
Concurrency

Concurrency is natural

- Parallelization of tasks
- Shared resources
- Distribution: remote resources, partners, etc.

Concurrency is ubiquitous

- Multi-core architectures
- Shared-memory concurrent data structures
- Communication protocols
- Distributed geo-replicated data structures
- Internet applications
Consistency: What is Assumed/Guaranteed

Users
- cancel
- book
- cancel
- book1
- book2

Booking application
- cancel
- book
- write
- read

Storage System
- send
- receive
- receive
- send

Network
- Assume
- Guarantee
- Assume
- Guarantee
- Assume
- Guarantee
Consistency Issues

- What is consistency?
- Why there are different levels of consistency?
- How to verify that a system (an application) is correct for a given consistency level of its environment?
- How to verify that a system (an infrastructure) implements a given consistency level?
Concurrent interactions

How to guarantee that

Every room and every flight seat is assigned to at most one user
Reasoning about concurrent interactions

Application programmers need abstraction

=> Strong Assumptions on:

- **Atomicity** of sequence of actions
- **Synchrony** of interactions
Abstraction based on system layers

Sets of observable behaviors at interfaces?
Abstraction based on system layers

Users

Booking application

Storage System

Network

Sets of observable behaviors at interfaces?

Let us focus on this interface
Concurrent interactions with storage systems

What are the expected observable behaviors?
Concurrent interactions with storage systems

What are the expected observable behaviors?

Strong consistency:
- updates are visible to all participants without delay
- updates are visible in the same order to everybody
Concurrent interactions with storage systems

Strong consistency:
- updates are visible to all participants without delay
- updates are visible in the same order to everybody
Concurrent interactions with storage systems

**Strong consistency:**
- updates are visible to all participants without delay
- updates are visible in the same order to everybody
Concurrent interactions with storage systems

Strong consistency:
- updates are visible to all participants without delay
- updates are visible in the same order to everybody
Concurrent interactions with storage systems

Strong consistency:
- updates are visible to all participants without delay
- updates are visible in the same order to everybody
Reasoning about concurrent interactions

Application programmers need abstraction

=> Strong Assumptions on:
   - Atomicity of sequence of actions
   - Synchrony of interactions
Reasoning about concurrent interactions

Application programmers need abstraction

=> Strong Assumptions on:
    - Atomicity of sequence of actions
    - Synchrony of interactions

… but they also need performant and available systems

=> Less synchronization in the system implementation

=> May lead to relaxing some of the system guarantees
Reasoning about concurrent interactions

Application programmers need abstraction

=> Strong Assumptions on:
  - Atomicity of sequence of actions
  - Synchrony of interactions

... but they also need performant and available systems

=> Less synchronization in the system implementation

=> May lead to relaxing some of the system guarantees
Concurrent interactions with storage systems

Replicated memory

read(x,?)  
read(x,?)

x = 0

x = 0

x = 0

read(x,?)  
read(x,?)

x:=2

x:=5

Geo-replicated databases, Modern hardware architectures
Concurrent interactions with storage systems

- Replicated memory
- Geo-replicated databases, Modern hardware architectures

Weak consistency:
- participants may see different sets of updates
- updates may be visible in different orders to participants
Concurrent interactions with storage systems

Weak consistency:
- participants may see different sets of updates
- updates may be visible in different orders to participants
Concurrent interactions with storage systems

Weak consistency:
- participants may see different sets of updates
- updates may be visible in different orders to participants
Concurrent interactions with storage systems

Weak consistency:
- participants may see different sets of updates
- updates may be visible in different orders to participants
Concurrent interactions with storage systems

Weak consistency:
- participants may **see different sets of updates**
- updates may be **visible in different orders** to participants
Concurrent interactions with storage systems

Weak consistency:
- participants may **see different sets of updates**
- updates may be **visible in different orders** to participants
Concurrent interactions with storage systems

Weak consistency:
- participants may see different sets of updates
- updates may be visible in different orders to participants
Concurrent interactions with storage systems

Weak consistency:
- participants may see different sets of updates
- updates may be visible in different orders to participants
Concurrent interactions with storage systems

Weak consistency:
- participants may see different sets of updates
- updates may be visible in different orders to participants
Concurrent interactions with storage systems

Weak consistency:
- participants may see different sets of updates
- updates may be visible in different orders to participants
Concurrent interactions with storage systems

Weak consistency:
- participants may see different sets of updates
- updates may be visible in different orders to participants
Reasoning about concurrent interactions

Application programmers **need abstraction**

=> **Strong Assumptions on:**
   - **Atomicity** of sequence of actions
   - **Synchrony** of interactions

… but they also **need performant and available systems**

=> **Less synchronization** in the system implementation

=> May lead to **relaxing** some of the **system guarantees**

**Applications should be immune against these relaxations**
or to be aware about them
Consistency Models

Define the expected interactions with the system

==> Returned values by read actions?
Consistency Models

Define the expected interactions with the system

$$\Rightarrow \text{Returned values by read actions?}$$

Returned values by read actions depend on:
- the current visibility set, and
- the order of execution of the visible actions
Sequential Consistency

- updates are totally ordered => visible in the same order to all proc.
- program order is respected => e.g., reads cannot overtake writes
Sequential Consistency

- updates are totally ordered => visible in the same order to all proc.
- program order is respected => e.g., reads cannot overtake writes
y:=1
read(x,?)

Sequential Consistency

- updates are totally ordered => visible in the same order to all proc.
- program order is respected => e.g., reads cannot overtake writes

x:=1
y=1

x:=1
read(y,?)
Sequential Consistency

- updates are totally ordered => visible in the same order to all proc.
- program order is respected => e.g., reads cannot overtake writes
Sequential Consistency

Possible read values:
(0, 1), (1, 0), (1, 1)

- updates are totally ordered => visible in the same order to all proc.
- program order is respected => e.g., reads cannot overtake writes
Sequential Consistency

- updates are totally ordered => visible in the same order to all proc.
- program order is respected => e.g., reads cannot overtake writes
Relaxing order constraints

\[ x = y = 0 \]

Sequential Consistency
Relaxing order constraints

Relax the Program Order Constraints

\[ x = y = 0 \]

\[ \text{write}(x, 1) \quad \text{read}(x, 0) \quad \text{read}(y, 0) \]

Sequential Consistency

Swap operations

\[ \text{read}(x, 0) \quad \text{write}(x, 1) \quad \text{read}(y, 0) \]

\[ \text{read}(x, 0) \quad \text{read}(y, 0) \quad \text{write}(x, 1) \]
Relaxing order constraints

Relax the Program Order Constraints

x = y = 0

write(x, 1) \rightarrow read(y, 0)

read(y, 0) \rightarrow write(x, 1)

Sequential Consistency

Swap operations

Execute in parallel

Fast execution of reads!
Relaxing order constraints

Relax the Program Order Constraints

\[ x = y = 0 \]

\[ \text{read}(x, 0) \quad \text{write}(x, 1) \quad \text{read}(y, 0) \quad \text{read}(x, 0) \]

\[ \text{read}(x, 0) \quad \text{write}(x, 1) \quad \text{read}(x, 0) \quad \text{read}(y, 0) \]

\[ \text{read}(x, 0) \quad \text{write}(x, 1) \quad \text{read}(y, 0) \]

Swap operations

Sequential Consistency

Execute in parallel

Fast execution of reads!

Reach the same state!
TSO (Total Store Ordering)

- Writes can be delayed, but they are visible to the issuer
- They become visible to all processes simultaneously
- Visible writes are visible in the same order to all processes
- Writes by a same process become visible in their issue order
**TSO (Total Store Ordering)**

- Used in several architectures: Intel / AMD x-86, Sparcv8, etc.
- The kernel of all (weaker) memory models
- Also considered in distributed systems
TSO : Operational Model

- **writes** are sent to **store buffers** (one per process)
- **writes** are committed to **memory** at any time
- **reads** are from
  - **own store buffer** if a value exists (last write to the variable)
  - otherwise from the **memory**
- **atomic read-writes** executed when **own buffer is empty**
- **fence** = flush the buffer (simulated with atomic read-write)
Total Store Ordering (TSO)

- updates are totally ordered => visible in the same order to all proc.,
- updates can be delayed => reads may overtake writes
- updates are totally ordered => visible in the same order to all proc.,
- updates can be delayed => reads may overtake writes
Total Store Ordering (TSO)

It is also possible to read \((0, 0)\).

- updates are totally ordered \(\Rightarrow\) visible in the same order to all proc.,
- updates can be delayed \(\Rightarrow\) reads may overtake writes.
Total Store Ordering (TSO)

Possible read values:
(0, 1), (1, 0), (1, 1), (0, 0)

- updates are totally ordered => visible in the same order to all proc.,
- updates can be delayed => reads may overtake writes
TSO: Non SC Behaviors

x = y = 0

- Impossible under SC: Cyclic happen-before relation
**TSO: Non SC Behaviors**

\[ x = y = 0 \]

**CS1** and **CS2**?

- **Impossible under SC**: Cyclic happen-before relation

- **Possible under TSO**!
  - writes are **delayed**: pending in store buffers
  - reads get old values in the memory (0's)
Avoiding Reordering: Fences

\[ x=y=0 \]

- A fence forces **flushing** the store buffer
- \( \Rightarrow \) reaching CS1 and CS2 becomes **impossible**
Avoiding Reordering: Fences

\[ x = y = 0 \]

- A fence forces flushing the store buffer
- \( \Rightarrow \) reaching CS1 and CS2 becomes impossible

SC can be enforced: insert a fence after each write
TSO: Enforcing SC Behaviors

$P1$

\[ w(x,1) \]

\[ \text{fence} \]

\[ r(y,0) \]

$P2$

\[ w(y,1) \]

\[ \text{fence} \]

\[ r(x,0) \]
TSO: Enforcing SC Behaviors

P1

\[ w(x,1) \]
\[ \text{fence} \]
\[ r(y,0) \]

P2

\[ w(y,1) \]
\[ \text{fence} \]
\[ r(x,0) \]
TSO: Enforcing SC Behaviors

P1

w(x,1)
fence
r(y,0)

P2

w(y,1)
fence
r(x,0)
TSO: Enforcing SC Behaviors

P1

w(x,1)
fence
r(y,0)

P2

w(y,1)
fence
r(x,0)
TSO: Enforcing SC Behaviors

P1

<table>
<thead>
<tr>
<th>w(x,1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>fence</td>
</tr>
<tr>
<td>r(y,0)</td>
</tr>
</tbody>
</table>

P2

<table>
<thead>
<tr>
<th>w(y,1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>fence</td>
</tr>
<tr>
<td>r(x,0)</td>
</tr>
</tbody>
</table>

x=1

y=1
TSO: Enforcing SC Behaviors

P1

w(x,1)
fence
r(y,0)

> 

P2

w(y,1)
fence
r(x,0)

x=1
y=1
Relaxed Consistency

At different levels:

- **Multicores:** Weak Memory Models *(TSO, Power, ARM, …)*
- **Concurrent programming languages:** WMM *(Java, C, …)*
- **Distributed systems:** Weak Consistency Models
  *(Eventual Consistency, Causal Consistency, etc.)*
Automated Verification

Checks if all observable behaviors of a program are conform to a specification

- decidability and complexity
  dealing with unbounded partial order constraints

- algorithmic approaches (precise/approximate)
  efficient bug detection procedures, abstract analyses

- deductive approaches
  sound and complete proof rules
Automated Verification

Checks if all observable behaviors of a program are conform to a specification

- decidability and complexity
dealing with unbounded partial order constraints

- algorithmic approaches (precise/approximate)
efficient bug detection procedures, abstract analyses

- deductive approaches
sound and complete proof rules

Compositional reasoning across system layers
Abstraction based on system layers

Let us focus on this interface

Sets of observable behaviors at interfaces?
Verifying Correctness: Application Layer

Application Layer

Guarantee: Functional Specification ($\Sigma$)

Implementation (A)

Assume: Consistency Model (M)
Verifying Correctness: System Layer

Guarantee: Functional Specification ($\Sigma$)

Assume: Consistency Model (M)

Guarantee: Consistency Model (M)

Implementation (A)

Assume: Infrastructure Model (N)

Implementation (S)
Verifying Correctness: System Layer

Assume: Infrastructure Model (N)

Assume: Consistency Model (M)

Guarantee: Consistency Model Model (M)

Implementation (S)

Consistency Model = Specification

Guarantee: Functional Specification ($\Sigma$)

Assume: Consistency Model (M)

Guarantee: Consistency Model Model (M)

Implementation (S)
Library implementing a Concurrent Data Structure e.g., queue, stack

Library implementing a Shared memory
Client

Service

Infrastructure

Invariant verification

Refinement

Imp2

CLM

Imp1/Spec

ICM

refines?

satisfies?

Assume

Assume

Guarantee

Guarantee

Assume

Guarantee
In the context of client service infrastructure, the CLM (Classical Invariant Verification) satisfies the guarantee assumption. 

The notation 'satisfies?' indicates a verification process. 

The diagram illustrates the relationship between client service and infrastructure, with guarantees and assumptions marked with arrows. 

ICM = SC

The textual content suggests a focus on invariant verification within a classical framework.
Client

Service

Infrastructure

Verification under weak CM

Imp2

CLM

Imp1/Spec

ICM

ICM’

weaker

refines?

satisfies?

Assume

Guarantee

Assume

Guarantee

Assume

Guarantee

Assume

Guarantee
Robustness checking

Client

Service

Infrastructure

Imp2 ≈ Imp2

Robust?

ICM'

weaker

ICM

CLM

Assume

Guarantee

Imp1/Spec

refines?

satisfies?

≈

Robust?
Verifying a program over SC

Safety verification problem

<--> State reachability problem

- Let \( P \) be a shared-memory concurrent program
- A \textit{state} is a vector of control locations + memory valuation
- Consider the \textit{configuration graph} of \( P \) under the SC semantics

Given a state \( s \),
determine if \( s \) is \textit{reachable} from an initial state
in the state graph of \( P \)
Verifying a program over SC: Issues

Safety verification problem

\[ \iff \]

State reachability problem

- **Infinite data domains:**
  
  \[ \Rightarrow \] We consider a finite data domain (abstraction/fix a bound)

- **Complexity:**
  
  - State space grows exponentially w.r.t. the nb of processes
  - State reachability is PSPACE-complete

- **Dynamic creation of processes/parametric nb of proc.**:
  
  - Still decidable by reduction to coverability in Petri Nets
  - State reachability is EXPSPACE-complete
Verifying a program over TSO

Safety verification problem

$\iff$

State reachability problem

- Let $P$ be a shared-memory concurrent program
- A *state* is a vector of control locations + memory valuation
- A *configuration* of $P$ in the TSO semantics = state + store buffers
- Consider the *configuration graph* of $P$ under the TSO semantics

Given a state $s$, determine if $s$ is *reachable* from an initial configuration in the configuration graph of $P$ under TSO semantics
Verifying a program over TSO: Issues

Safety verification problem

\[ \iff \]

State reachability problem

In addition to issues for SC

Unbounded store buffers = queues!

- Systems with queues are Turing powerful in general
- But, the use of queues in TSO is special
- Decidability? Complexity?
Verifying a program over TSO: Decidability

[Atig, Bouajjani, Burckhardt, Musuvathi, 2010]

The state reachability under TSO is decidable

- Reduction to reachability in Lossy Channel Systems (LCS)
- LCS’s are well-structured systems (WSS)
- State reachability is decidable for WSS’s

[Abdulla, Jonsson, 1993], [Abdulla, Cerans, Jonsson, Tsay, 1996], [Finkel, Schnoebelen, 2001]
Verifying a program over TSO: Decidability

[Atig, Bouajjani, Burckhardt, Musuvathi, 2010]

The state reachability under TSO is decidable

- Reduction to reachability in Lossy Channel Systems (LCS)
- LCS’s are well-structured systems (WSS)
- State reachability is decidable for WSS’s

[Abdulla, Jonsson, 1993], [Abdulla, Cerans, Jonsson, Tsay, 1996], [Finkel, Schnoebelen, 2001]

The state reachability under TSO is non-primitive recursive

- Reduction of the reachability problem in LCS’s
- Reachability in LCS’s is non-primitive recursive

[Schnoebelen, 2001]
FIFO Channel Systems

Finite-state transition systems + unbounded FIFO channels
Lossy FIFO Channel Systems

Finite-state transition systems + unbounded Lossy FIFO channels

Messages can be lost at any time
Well-Structured Systems

• Let \( U \) be a universe.

• **Well-quasi ordering** \( \preceq \) over \( U \): \( \forall c_0, c_1, c_2, \ldots, \exists i < j, \ c_i \preceq c_j \)

• \( \Rightarrow \) Each (infinite) set has a finite minor set.

• Let \( S \subseteq U \). Upward-closure \( \overline{S} = \) minimal subset of \( U \) s.t.
  - \( S \subseteq \overline{S} \),
  - \( \forall x, y. (x \in S \text{ and } x \preceq y) \Rightarrow y \in \overline{S} \).

• A set is upward closed if \( \overline{S} = S \)

• Upward closed sets are definable by their minor sets
  - Assume there is a function \( \text{Min} \) which associates a minor to each set.
  - Assume \( \text{pre}(\text{Min}(S)) \) is computable for each set \( S \).

• **Monotonicity:** \( \preceq \) is a simulation relation

\[ \forall c_1, c_1', c_2. ((c_1 \rightarrow c_1' \text{ and } c_1 \preceq c_2) \Rightarrow \exists c_2'. c_2 \rightarrow c_2' \text{ and } c_1' \preceq c_2') \]
Well-Structured Systems

Lemma:

The \( \text{pre} \) and \( \text{pre}^* \) images of upward closed set are upward closed

1. Let \( S \) be an upward closed set.
2. Assume \( \text{pre}(S) \) is not upward closed.
3. Let \( c_1 \in \text{pre}(S) \), and let \( c_2 \in U \) such that \( c_1 \leq c_2 \) and \( c_2 \notin \text{pre}(S) \)
4. Let \( c_1' \in S \) such that \( c_1 \rightarrow c_1' \)
5. Monotonicity \( \Rightarrow \) there is a \( c_2' \) such that \( c_2 \rightarrow c_2' \) and \( c_1' \leq c_2' \)
6. \( S \) is upward closed \( \Rightarrow \ c_2' \in S \)
7. \( \Rightarrow c_2 \in \text{pre}(S) \), contradiction.

8. For \( \text{pre}^* \): the union of upward closed sets is upward closed.
Well-Structured Systems

Backward Reachability Analysis

Consider the increasing sequence \( X_0 \subseteq X_1 \subseteq X_2 \ldots \) defined by:

- \( X_0 = \text{Min}(S) \)
- \( X_{i+1} = \text{Min}(X_i \cup \text{Min}(\text{pre}(\overline{X_i}))) \)

Termination:

There is a index \( i \geq 0 \) such that \( X_{i+1} = X_i \)

- \( \text{pre}^*(S) \) is upward closed \( \Rightarrow \) has a finite minor
- Wait until a minor is eventually collected.
Well-Structured Systems

Many examples

- Petri nets / Vector Addition Systems
- Lossy channel systems
- Timed Petri nets
- Broadcast protocols
- …
Well-Structured Systems

The case of Lossy Channel Systems

- Subword relation over a finite alphabet is a WQO (Higman’s lemma)
- Upward-closed sets = finite unions of
  \[ \Sigma^* a_1 \Sigma^* a_2 \cdots a_m \Sigma^* \]
- Computation of the Pre:
  - Send: Left concatenation + Upward closure
  - Receive: Right derivation
- Lossyness $\Rightarrow$ Monotonicity
From TSO programs to LCS?

Store buffers are not lossy!
An example of TSO program

P1
\[ > \quad w(x,1) \]
\[ > \quad w(y,1) \]
\[ > \quad w(x,2) \]

P2
\[ > \quad r(x,2) \]
\[ > \quad r(y,0) \]

x=y=0

\[ \rightarrow \]

TSO store buffer of P1

x=0
y=0
An example of TSO program

P1
- w(x,1)
- w(y,1)
- w(x,2)

x = y = 0

P2
- r(x,2)
- r(y,0)

TSO store buffer of P1

x = 0
y = 0
An example of TSO program

P1
w(x,1)
w(y,1)
w(x,2)

P2
> r(x,2)
r(y,0)

x=y=0

> x=1
y=0

TSO store buffer of P1
An example of TSO program

P1
w(x,1)
w(y,1)
w(x,2)
> x=y=0
P2
> r(x,2)
r(y,0)

TSO store buffer of P1

P1
x=1
y=1
An example of TSO program

P1
- w(x,1)
- w(y,1)
- w(x,2)

P2
- > r(x,2)
- r(y,0)

x=0, y=0

TSO store buffer of P1

P1→ w(x,2) w(y,1) w(x,1) → x=2 y=1
An example of TSO program

P1
- w(x, 1)
- w(y, 1)
- w(x, 2)

TSO store buffer of P1

x = y = 0

P2
- r(x, 2)
- r(y, 0)

x = 2
y = 1
An example of TSO program

Deadlock under the TSO semantics
TSO Store Buffers -> Lossy Channels?

P1
w(x,1)
w(y,1)
w(x,2)

x=y=0

P2
r(x,2)
r(y,0)

Lossy Fifo Channel

x=0
y=0
TSO Store Buffers $\rightarrow$ Lossy Channels?

\[
\begin{align*}
&\text{P1} \\
&w(x,1) \\
&w(y,1) \\
&w(x,2) \\
&\text{P2} \\
&\text{r}(x,2) \\
&\text{r}(y,0) \\
\end{align*}
\]

Lossy Fifo Channel
TSO Store Buffers $\rightarrow$ Lossy Channels?
TSO Store Buffers $\rightarrow$ Lossy Channels?

P1
\[ w(x,1) \]
\[ w(y,1) \]
\[ w(x,2) \]

\[ \text{Lossy Fifo Channel} \]

P2
\[ x=y=0 \]
\[ r(x,2) \]
\[ r(y,0) \]

\[ x=2 \]
\[ y=0 \]
TSO Store Buffers $\rightarrow$ Lossy Channels?

Diagram:

- **P1**
  - $w(x,1)$
  - $w(y,1)$
  - $w(x,2)$

- **P2**
  - $r(x,2)$
  - $r(y,0)$

**Lossy Fifo Channel**

- $x=y=0$
- $x=2$
- $y=0$
TSO Store Buffers $\rightarrow$ Lossy Channels?

Unsound simulation of TSO!
Store Memory Snapshots

Future Snapshots of the Memory

\( P_1 \)
\[ \xrightarrow{w(x,1)} \]
\[ \xrightarrow{w(y,1)} \]
\[ \xrightarrow{w(x,2)} \]
\[ x=y=0 \]

\( P_2 \)
\[ \xrightarrow{r(x,2)} \]
\[ \xrightarrow{r(y,0)} \]
\[ x=0 \]
\[ y=0 \]
Store Memory Snapshots

P1

w(x, 1)

w(y, 1)

w(x, 2)

x = y = 0

P2

r(x, 2)

r(y, 0)

Future Snapshots of the Memory

P1

x = 1

y = 0

P1

x = 0

y = 0
Store Memory Snapshots

Future Snapshots of the Memory

P1
w(x,1)
w(y,1)
\[ w(x,2) \]

\[ x=y=0 \]

P2
r(x,2)
r(y,0)

x=0
y=0

P1

Future Snapshots of the Memory

x=1
y=1

x=1
y=0
Store Memory Snapshots

Future Snapshots of the Memory

P1
w(x,1)
w(y,1)
w(x,2)

x=y=0

P2
> r(x,2)
> r(y,0)

P1

Future Snapshots of the Memory

x=0
y=0

x=2
y=1
x=1
y=1
x=1
y=0
Store Memory Snapshots

P1
w(x,1)
w(y,1)
w(x,2)

P2
r(x,2)
r(y,0)

x=y=0

Future Snapshots of the Memory

x=2
y=1
x=1
y=1
x=1
y=0

x=1
y=0
Store Memory Snapshots with Losses

Future Snapshots of the Memory + Lossyness
Store Memory Snapshots with Losses

Future Snapshots of the Memory + Lossyness
Store Memory Snapshots with Losses

Future Snapshots of the Memory
+ Lossyness
Store Memory Snapshots with Losses

Future Snapshots of the Memory + Lossyness

Valid Simulation of TSO
From TSO to Lossy Channel Systems

- 1-channel machine per process + composition
From TSO to Lossy Channel Systems

- 1-channel machine per process + composition

- Each process:
  - **write**: puts a new memory state at the tail of the channel
  - **read**: checks the channel, then the memory
  - **memory update**: moves the head of the channel to the memory
From TSO to Lossy Channel Systems

- 1-channel machine per process + composition

- Each process:
  - write: puts a new memory state at the tail of the channel
  - read: checks the channel, then the memory
  - memory update: moves the head of the channel to the memory

Problem: Interferences between processes ?
Processes must agree on the same order of memory updates
From TSO to Lossy Channel Systems

- 1-channel machine per process + composition

- Each process:
  - write: puts a new memory state at the tail of the channel
  - read: checks the channel, then the memory
  - memory update: moves the head of the channel to the memory

**Problem:** *Interferences between processes?*
Proceses must agree on the same order of memory updates

  - guesses writes by other processes; put them in the channel

- Validation of the guesses by composition:
  - transitions are labelled by *write operations + process id*
  - machines are *synchronized* on these actions
From Lossy Channel Systems to TSO programs

Property of LCS:

Every LCS can be simulated by an LCS with one process and one channel
From Lossy Channel Systems to TSO programs

- **P1** simulates a LCS with one channel using x and y:
  - send(m) $\rightarrow$ write(x, m)
  - receive(m) $\rightarrow$ read(y,m)

- **P2** forwards values from x to y
From Lossy Channel Systems to TSO programs

- **P1** simulates a LCS with one channel using **x** and **y**:
  - `send(m) → write(x, m)`
  - `receive(m) → read(y, m)`

- **P2** forwards values from **x** to **y**
  - **P2** can miss some values
Other Weak(er) Memory Models

- Power, ARM, …
- C, Java, …

Various types of relaxations

- atomicity of operations
- reordering of operations
- visibility of operations by different processes
PSO (Partial Store Ordering)

- Writes can be delayed, but they are visible to the issuer
- They become visible to all processes simultaneously
- Visible writes are visible in the same order to all processes
- Writes by a same process become visible in their issue order on a same variable
PSO (Partial Store Ordering)

- Writes can be delayed, but they are visible to the issuer
- They become visible to all processes simultaneously
- Visible writes are visible in the same order to all processes
- Writes by a same process become visible in their issue order on a same variable

- Weaker model than TSO: writes on ≠ variables can be reordered
PSO (Partial Store Ordering)

- Writes can be delayed, but they are visible to the issuer
- They become visible to all processes simultaneously
- Visible writes are visible in the same order to all processes
- Writes by a same process become visible in their issue order
  on a same variable

- Weaker model than TSO: writes on ≠ variables can be reordered

  - Operational Model ?
  - Reachability Problem ?
Articles


What’s Decidable about Weak Memory Models, M-F. Atig, A. Bouajjani, S. Burckhardt, M. Musuvathi, ESOP 2012.