

A Causality-Based Runtime Check for (Rollback) Atomicity

Serdar Tasiran
Koc University
Istanbul, Turkey

Tayfun Elmas
Koc University
Istanbul, Turkey

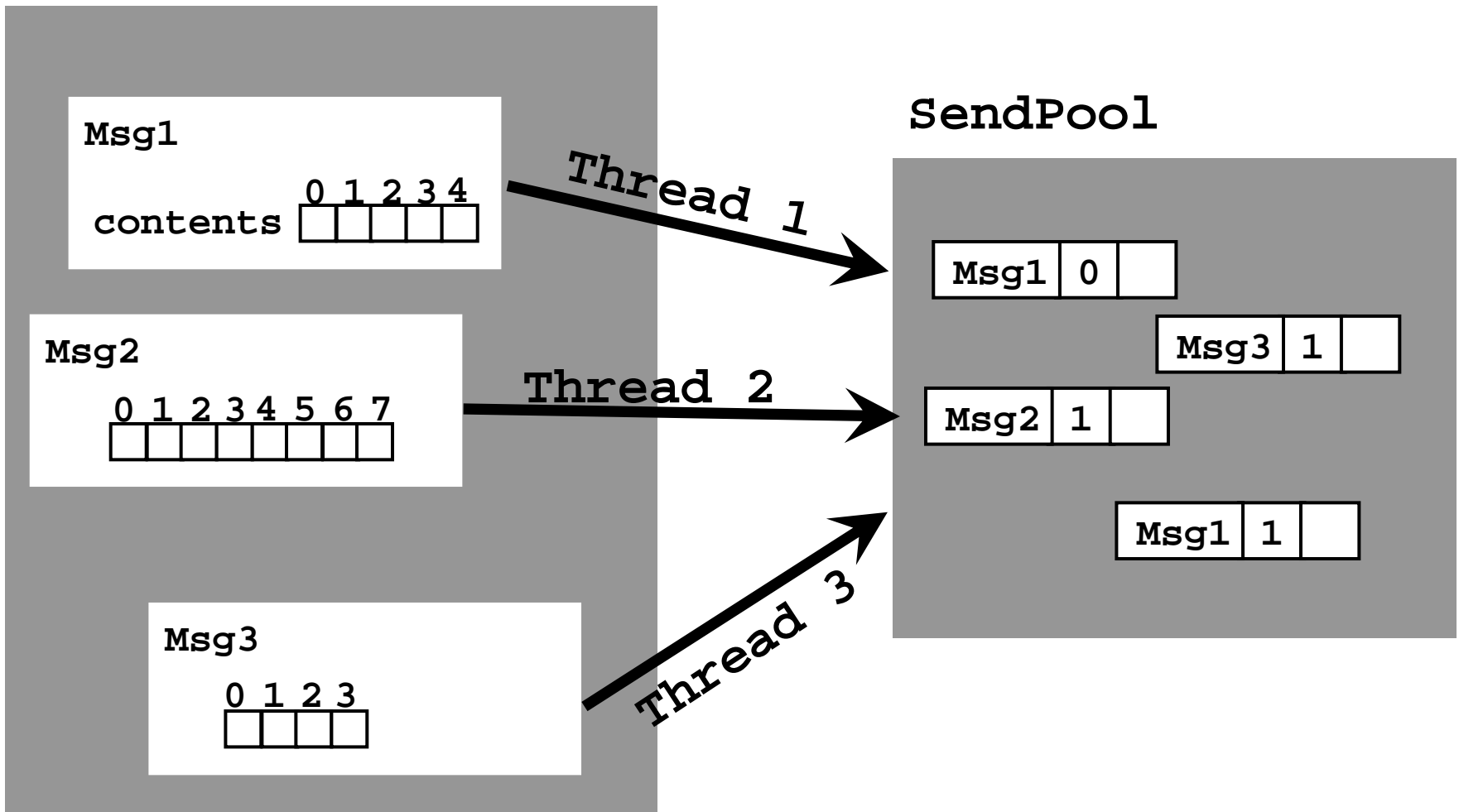
RV 2007
March 13, 2007

Outline

- This paper:
 - Define rollback atomicity of an execution
 - A special case of refinement
 - An algorithm and tool for checking rollback atomicity of an execution
- Why?
 - How is rollback atomicity different from existing definitions of atomicity?
 - Less restrictive
 - Better observability
 - Motivating example
- Formal definition
- Checking rollback atomicity

Motivating Example

ToSendQueue



Example Code

```
0: class Msg {
1:     long msgId;
2:     static long KBSentThisSec = 0;
3:     boolean sent = false;
4:     byte[] contents;

5:     static synchronized long getKBSentThisSec() {
6:         return KBSentThisSec;
7:     }
8:     static synchronized long getKBSentThisSecIncr() {
9:         return ++KBSentThisSec;
10:    }
```

 Reset every second

```
11:   synchronized atomic void send() {
12:
13:       if ( sent  || !toSendQueue.isIn(this))
14:           abort;
15:
16:       if (Msg.getKBSentThisSec() > MAXRATE)
17:           abort; // Caller must retry
18:
19:       int i = 0;
20:       while (i < contents.length) {
21:
22:           sendPool.insert(msgId, i, contents[i]);
23:           if ( (++i % 1000) == 0 )
24:               if (Msg.getKBSentThisSecIncr() > MaxRate)
25:                   abort; // Caller must retry
26:       }
27:
28:       sent = true;
29:       toSendQueue.remove(this);
30:   }
31: }
```

An interleaving

- Note: `KBsentThisSec++` is a Read-Modify-Write

Thread 1

...

send 1000 bytes of Msg1

`KBsentThisSec++`

send 1000 bytes of Msg1

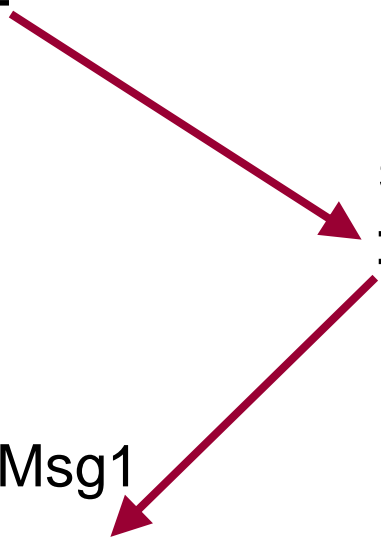
`KBsentThisSec++`

Thread 2

...

send 1000 bytes of Msg2

`KBsentThisSec++`



- These two atomic blocks are not conflict or view serializable!

Focus vs. Peripheral Variables

- But `Msg.KBSentThisSec` is there only for rate control!
- We want `Msg.send()` to update atomically only :
 - `ToSendQueue`, and
 - the `sent` and `contents` fields of `Msg` objects
- But arbitrary values of `Msg.KBSentThisSec` will not work
 - Atomic method aborted if rate is exceeded.
 - Not purely a performance counter
 - Cannot abstract away `Msg.KBSentThisSec` from implementation

```
0: class Msg {
1:     long msgId;                /* @Focus */
2:     static long KBSentThisSec = 0; /* @Peripheral */
3:     boolean sent = false;      /* @Focus */
4:     byte[] contents;          /* @Focus */
```

Defining Rollback Atomicity

- A special case of view refinement [Elmas et. al., PLDI '05]
- A concurrent execution σ^{conc} is **rollback atomic** if there exists an **equivalent** serial execution σ^{ser} .
- **Equivalent:**
 - For each thread t , $\sigma^{\text{conc}}|_t$ and $\sigma^{\text{ser}}|_t$ must consist of the same sequence of atomic blocks
 - Not necessarily the same actions
 - Abstracted states of σ^{conc} and σ^{ser} must match “after” each atomic block completes.
- **Abstraction map:** Values of focus variables
 - Project out peripheral variables
 - Roll back effects of uncommitted atomic blocks
 - History variables remember previous values of uncommitted writes

Rollback Atomicity vs. Others

- State match required at each atomic block
 - not only at the end of the execution
 - not only at quiescent points (no atomic block in progress)
 - but only for focus variables
- Incomparable to view serializability
 - No requirement about what reads see
- Incomparable to commit atomicity
 - Requires a state match at more points along the execution
- If focus variables = all shared variables
 - Implies commit atomicity
- Weaker (more permissive) than
 - reduction
 - conflict serializability

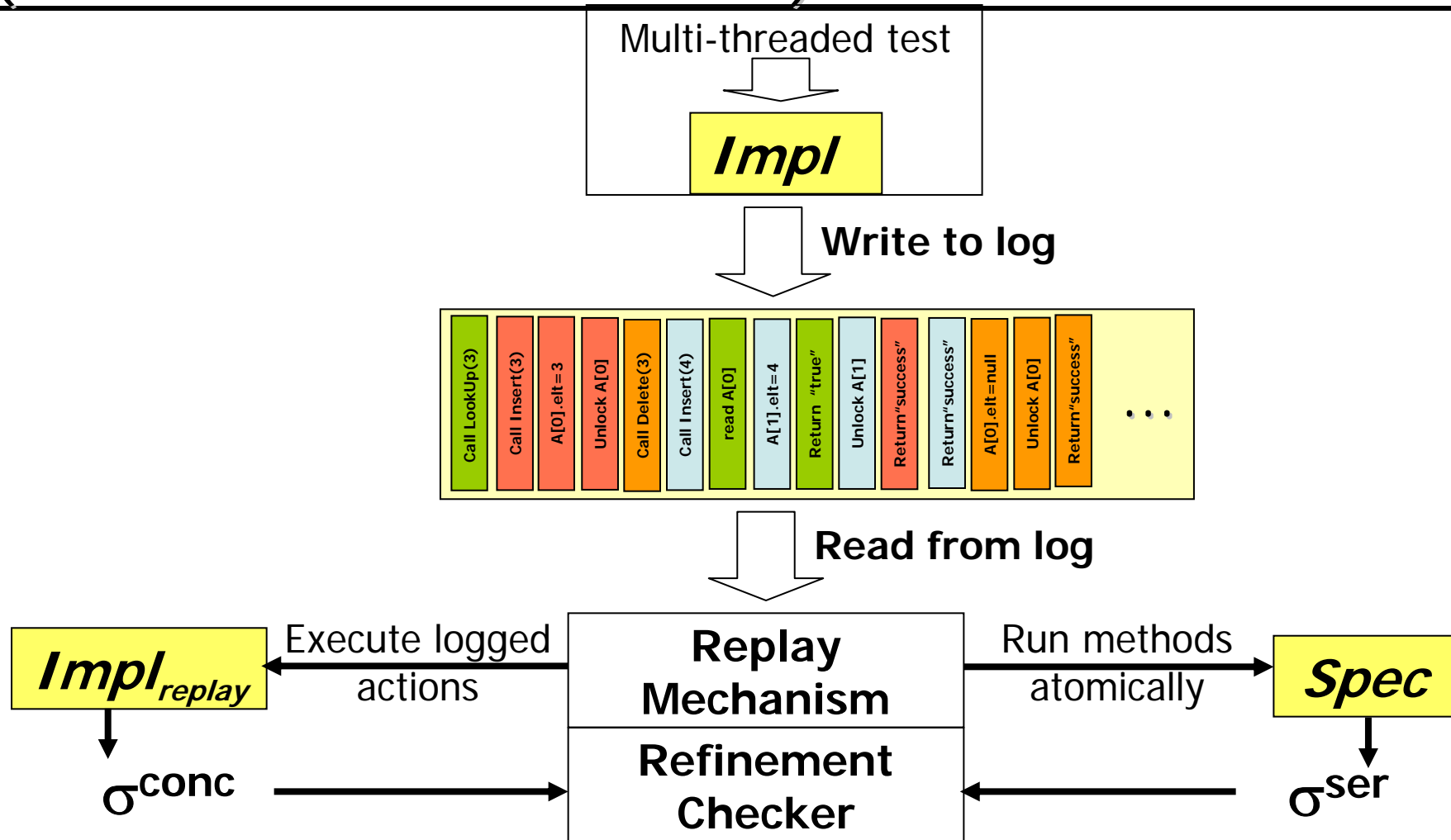
Inferring the commit order: Causality graphs

- **Commit order:** Order of atomic blocks in σ^{ser}
- **Causality graph:** Extracted from logged execution
- **Vertices:**
 - Shared variable accesses
 - Atomic blocks
- **Edges: (u,v)**
 - u comes earlier in σ^{conc}
 - u and v are/contain
 - accesses to the same variable
 - at least one is a write
 - Program order edges:
 - u precedes v in program order

Inferring the commit order: Causality graphs

- Two versions
 - CG_{under} : Focus variables causality graph
 - Only accesses to focus variables tracked
 - CG_{full} : Full causality graph
 - All shared variables tracked
- Cycle detection algorithm infers linear order consistent with causality edges
 - Use CG_{full} if it is acyclic
 - Use CG_{under} otherwise
 - If CG_{full} has cycle also, use last focus variable access as commit point
 - Note: Serialization conflicts with causality in this case.

Checking Rollback Atomicity Using the VYRD Tool (The Java PathFinder Version)



- At certain points for each atomic block, take "focus state snapshots"
- Check that abstraction functions match

Rollback Atomicity Violations

- If tool does not declare warning, execution rollback atomic.
 - Invariants expressed using focus variables hold
 - At each commit point
 - At end of execution
 - If CG_{under} is acyclic
 - within atomic blocks, sequential reasoning about focus variables is valid.
- If tool declares warning, two possibilities
 - Rollback atomicity violation
 - Could not infer correct commit order
 - But in this case there are conflict-edge cycles between atomic blocks

Conclusion

- New concept of atomicity
 - More permissive for peripheral variables
 - More observability for focus variables
- Refinement-based definition and checking
 - Commit order inferred from causality graph