A Programming Language Perspective on Transactional Memory Consistency

Hagit Attiya, Technion

Joint work with Sandeep Hans,
Alexey Gotsman, Noam Rinetzky

Research funded by the **European Union**'s 7th Framework Programme (FP7/2007-2013) under grant agreement n° 238639 – MC-**TRANFORM**.

Transactional Memory (TM)

[Herlihy & Moss 93]

- **Atomic blocks** (accessing transactional variables)
  - Appear to execute atomically
  - May abort
- **Local variables** (also inside blocks)
- **Global variables** (only outside blocks)

```
node := new(StackNode);
node.val := val;
result := abort;
while(result == abort){
  result := atomic{
    node.next =
       Top.read();
    node.val++ ;
    Top = node;
   }
}
g := val;
```

TM Consistency Condition

- How should the TM implementation behave?

- No single answer…

TM Consistency Conditions

Opacity

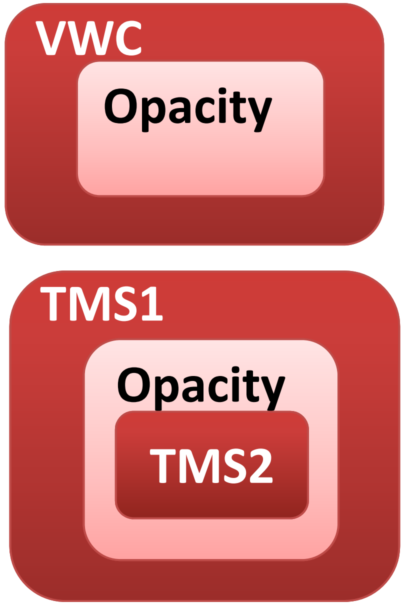
[Guerraoui & Kapalka 08]

Virtual World Consistency

[Imbs & Raynal 09]

TMS1 / TMS2

[Doherty, Groves, Luchangco, Moir 09]



What is the **"right"** condition?

But we are not social scientists

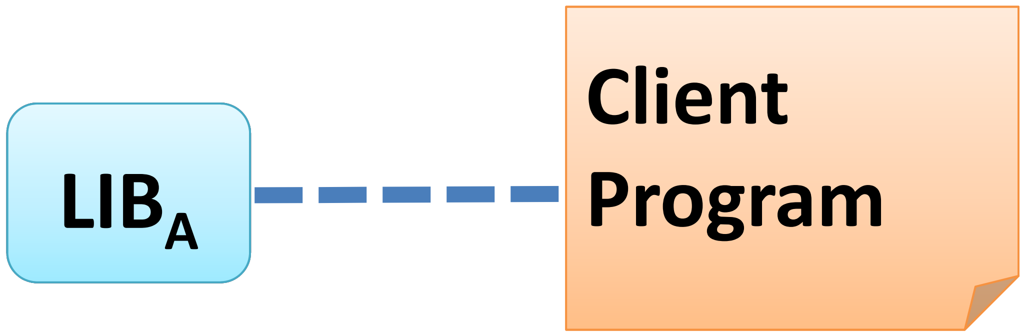
al

..

Observational Refinement

[He, Hoare, Sanders 86]

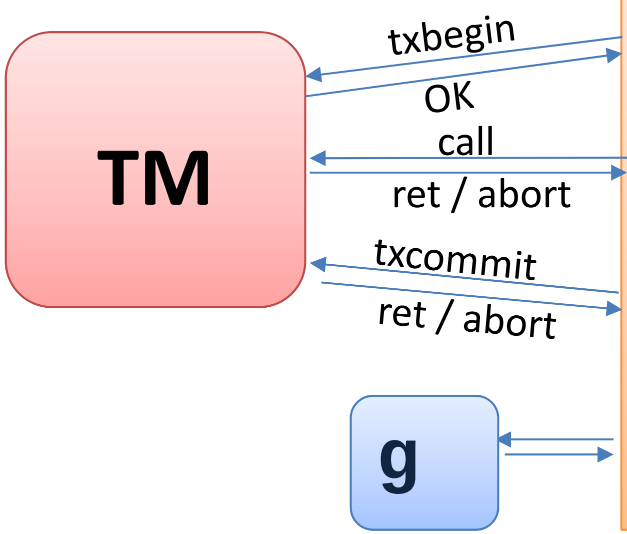
Preserve the observations of a client program, when an **abstract** library implementation is substituted with a **concrete** one



Our work uses observational refinement as a yardstick to evaluate TM consistency conditions

Interactions of a Program using TM

- **Local actions:** access only the local variables
- **Global actions:** interact with other client programs
- **Interface actions:** interact with TM

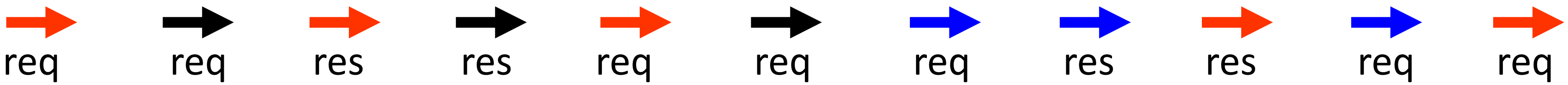



```
node := new(StackNode);
node.val := val;
result := abort;
while(result == abort){
  result := atomic{
    node.next =
       Top.read();
    node.val++ ;
    Top = node;
   }
}
g := val;
```

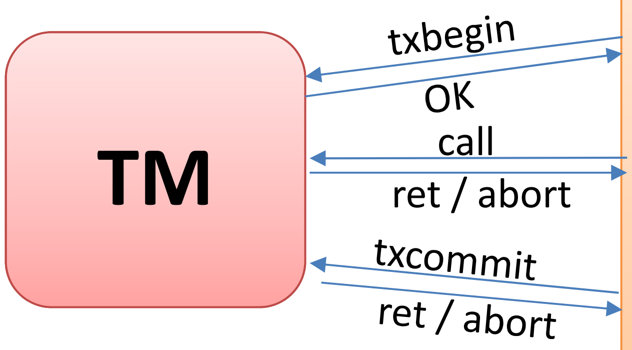
Client Program

Interactions of a Program using TM

History: Finite sequence of interface actions



Transactional System (**TM**): set of histories

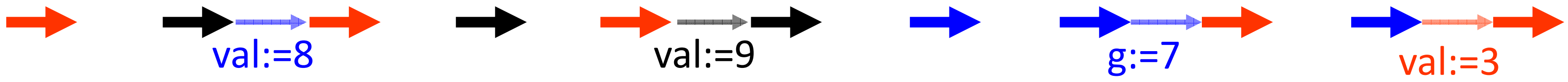


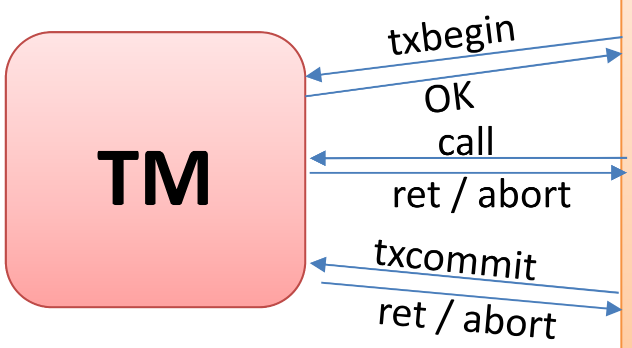

```
node := new(StackNode);
node.val := val;
result := abort;
while(result == abort){
  result := atomic{
    node.next =
       Top.read();
    node.val++ ;
    Top = node;
   }
}
g := val;
```

Client Program

More than Just TM

Trace: includes also local and global actions



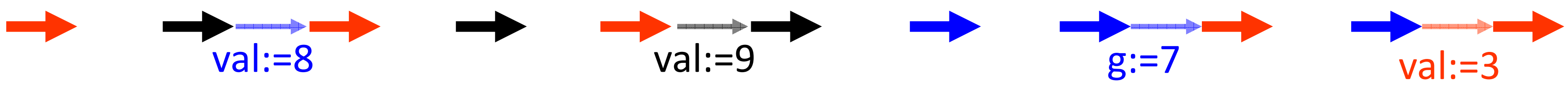



```
node := new(StackNode);
node.val := val;
result := abort;
while(result == abort){
  result := atomic{
    node.next =
       Top.read();
    node.val++ ;
    Top = node;
   }
}
g := val;
```

Client Program

Trace Equivalence

Trace: includes also local and global actions

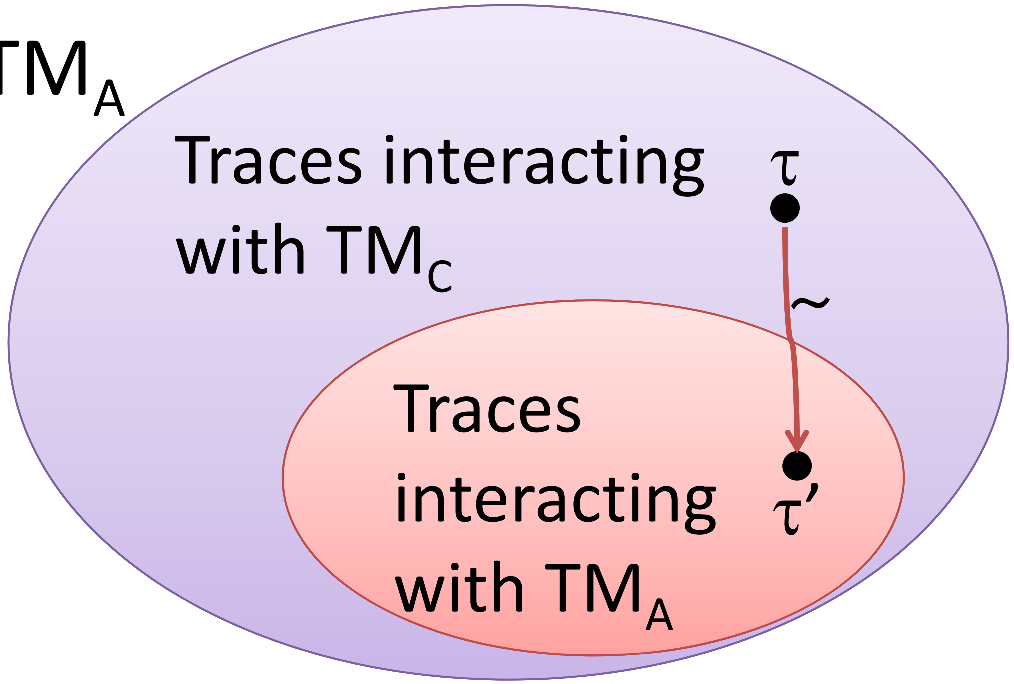


Two traces are **observationally equivalent** $\tau \sim \tau'$
if threads see the same sequence of local values

TM_C **observationally refines** TM_A if
every trace τ with history in TM_C
has a trace $\tau' \sim \tau$ with history in TM_A

Why Observational Refinement?

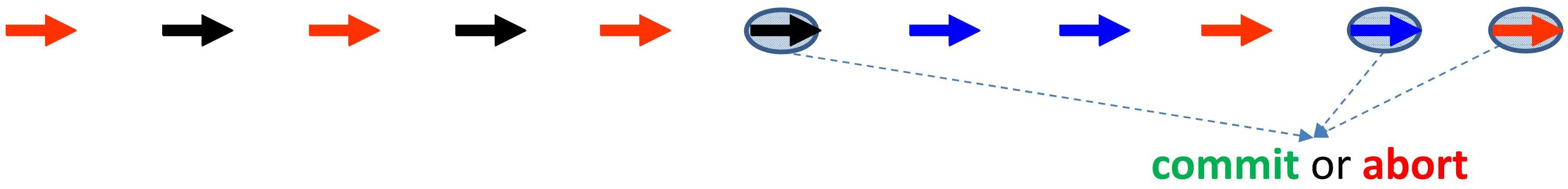
Prove properties of TM_A and deduce same properties for TM_C



TM_C **observationally refines** TM_A if every trace τ with history in TM_C has a trace **$\tau' \sim \tau$** with history in TM_A

Abstract System for Opacity

Complete history: all transactions commit / abort



Sequential history: no interleaving of transactions

Legal history: read from committed transactions

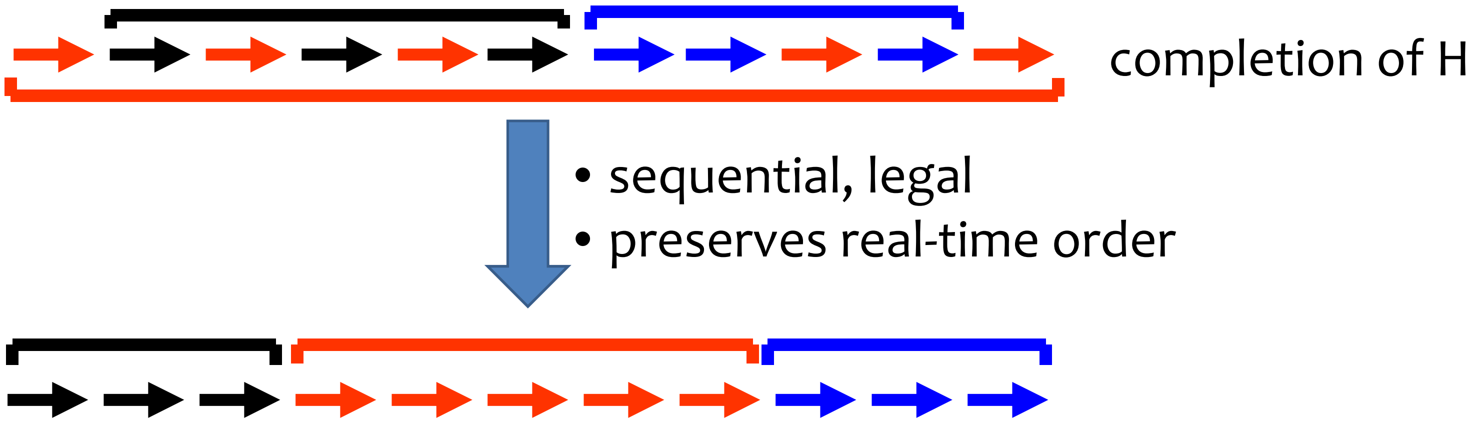
TM_{ATOMIC}: all sequential and legal histories

Opacity

History H is **opaque** if we can [Guerraoui & Kapalka 08]

– Complete H

– Find a permutation S of H that is **sequential**, **legal** and preserves the **real-time order** of H



TM is **opaque** if every history in TM is opaque

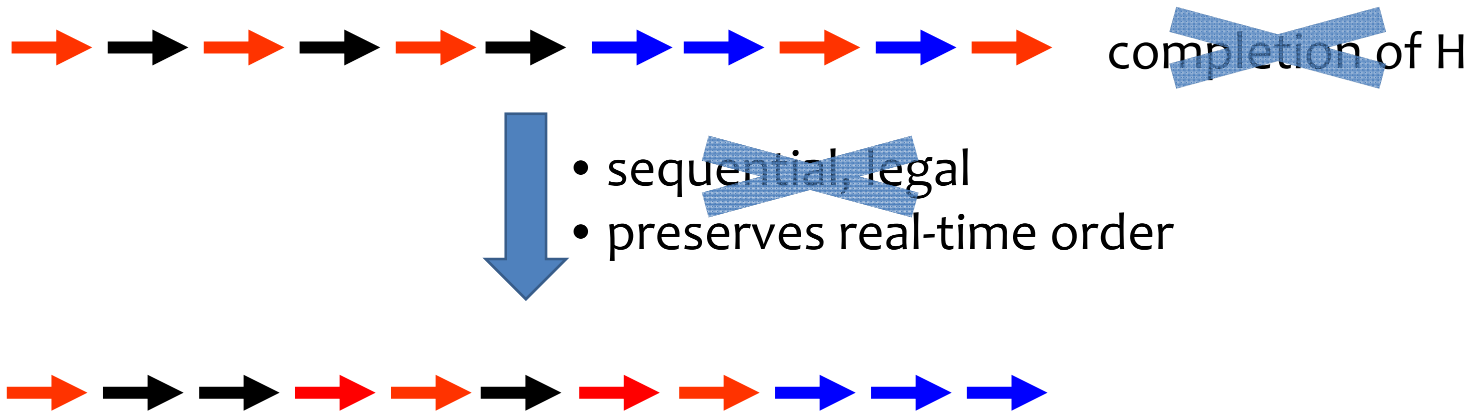
Opacity Relation

$H \sqsubseteq S$

S preserves the **per-thread** and **real-time** order of H

$TM_C \sqsubseteq TM_A$

for every $H \in TM_C$, $H \sqsubseteq S$, for some $S \in TM_A$



TM_C is opaque $\Leftrightarrow$ $TM_C \sqsubseteq TM_{ATOMIC}$

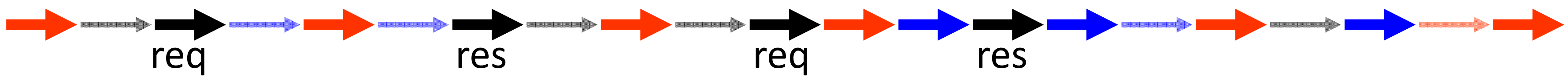
Opacity Relation vs. Linearizability

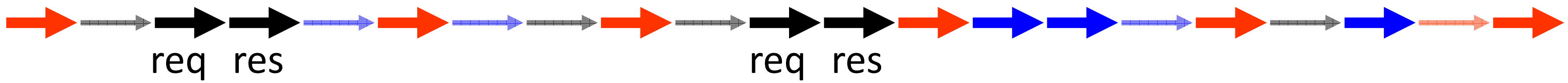
Linearizability: consistency condition for library calls

– client is suspended when waiting for a response

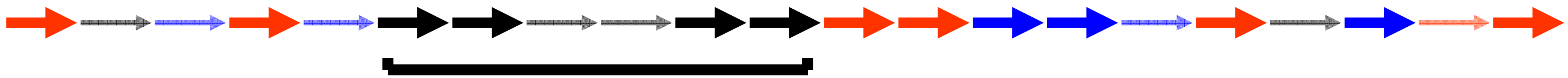
Observational refinement for linearizability

[Filipovic, O'Hearn, Rinetzky, Yang 09]





☝ For opacity relation, we need to do more…



Main Result

$TM_C \sqsubseteq TM_A \Leftrightarrow TM_C$ observationally refines TM_A

- global variables can be accessed, but only outside atomic blocks

(returns 1)

```
result := atomic{
  l := g ;
  write(1,tx) ;
}
```

```
result := atomic{
  g := 1 ;
  read(tx) ;
}
```

(returns 1)

Main Result

$TM_C \sqsubseteq TM_A \Leftrightarrow TM_C$ observationally refines TM_A

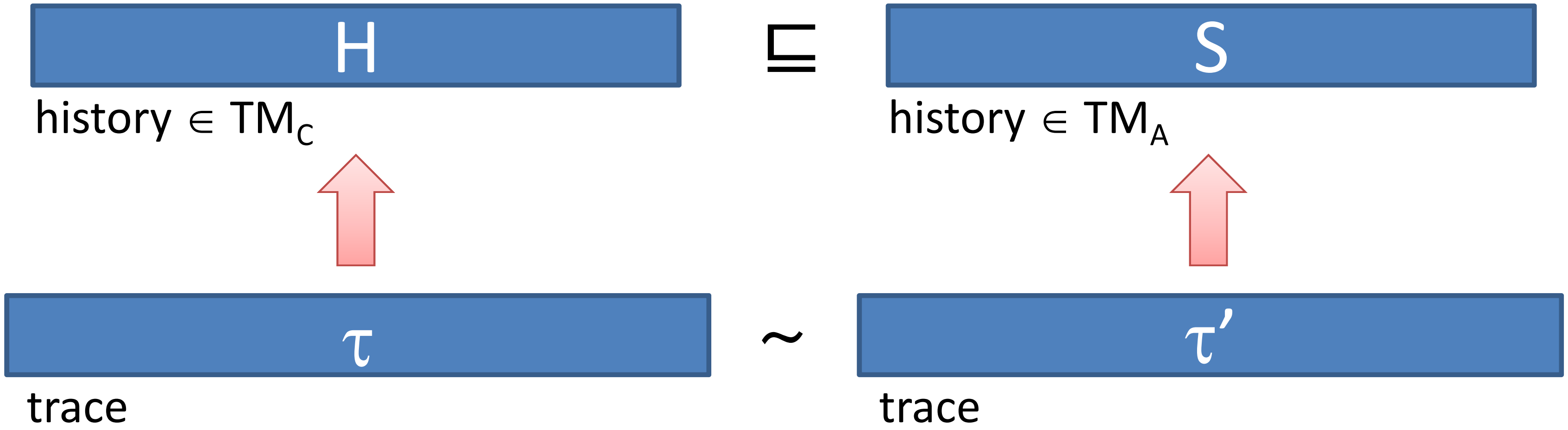
- global variables can be accessed,
  but only outside atomic blocks

- finite histories

- no nesting of atomic blocks

Soundness: $\sqsubseteq$ is Sufficient

Assume $TM_C \sqsubseteq TM_A$ and prove that a trace τ observed with TM_C has an equivalent trace τ' observed with TM_A

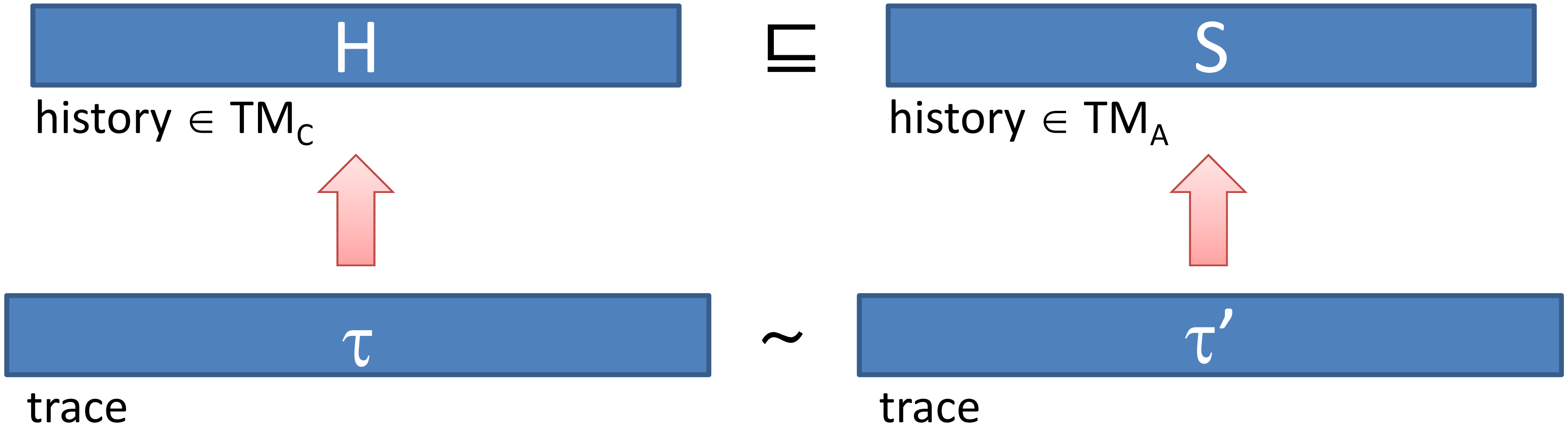


- Consider a trace τ whose history H is in TM_C
- $TM_C \sqsubseteq TM_A \Rightarrow H \sqsubseteq S$ for some history S in TM_A

From τ and S, get a trace $\tau' \sim \tau$ of TM_A whose history is S

Soundness: $\sqsubseteq$ is Sufficient

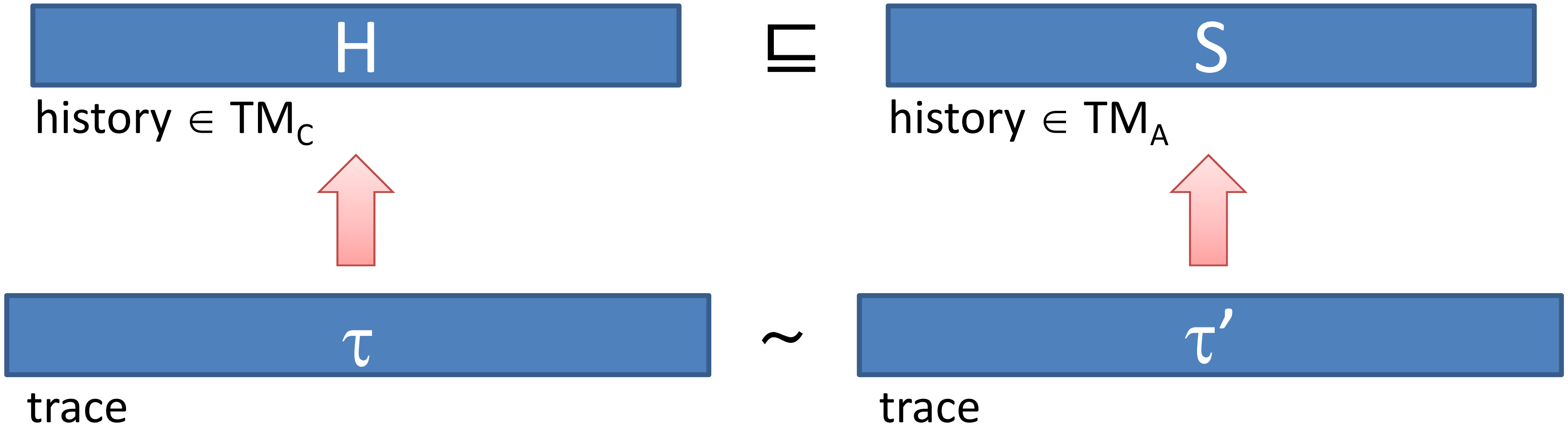
Assume $TM_C \sqsubseteq TM_A$ and prove that a trace τ observed with TM_C has an equivalent trace τ' observed with TM_A



☝ Two traces are **equivalent** if they have the **same order** for **same-thread** actions and for **global** actions

Soundness: $\sqsubseteq$ is Sufficient

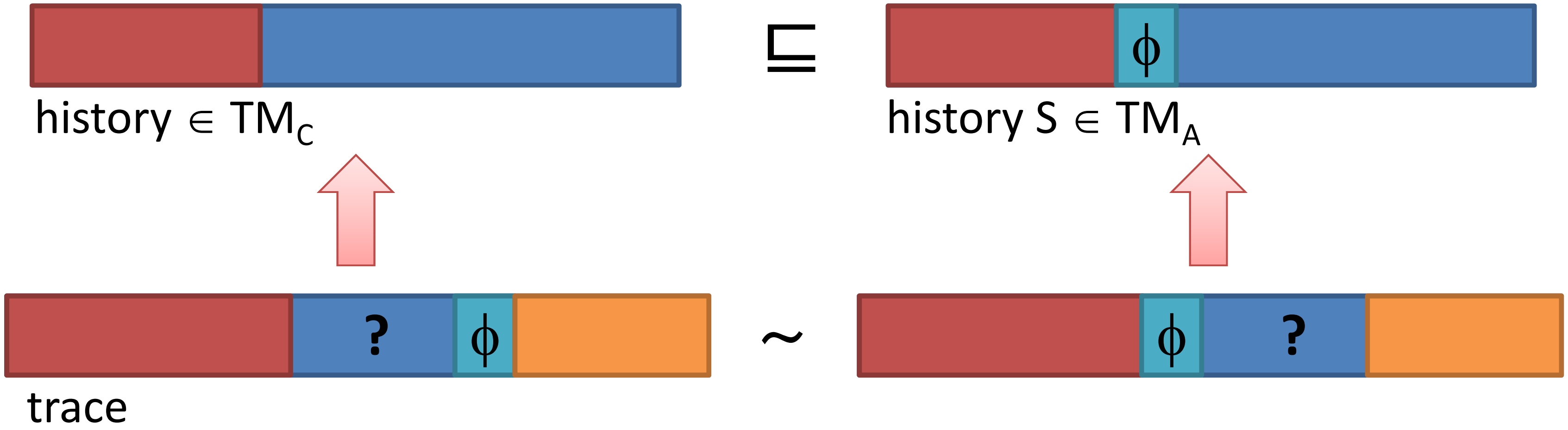
Assume $TM_C \sqsubseteq TM_A$ and prove that a trace τ observed with TM_C has an equivalent trace τ' observed with TM_A



☝ Inductively **permute** τ to get τ', while **preserving the order of same-thread and global actions**

Soundness: Inductive Step

Assume we have permuted a prefix of τ so its history is a prefix of a history in TM_A

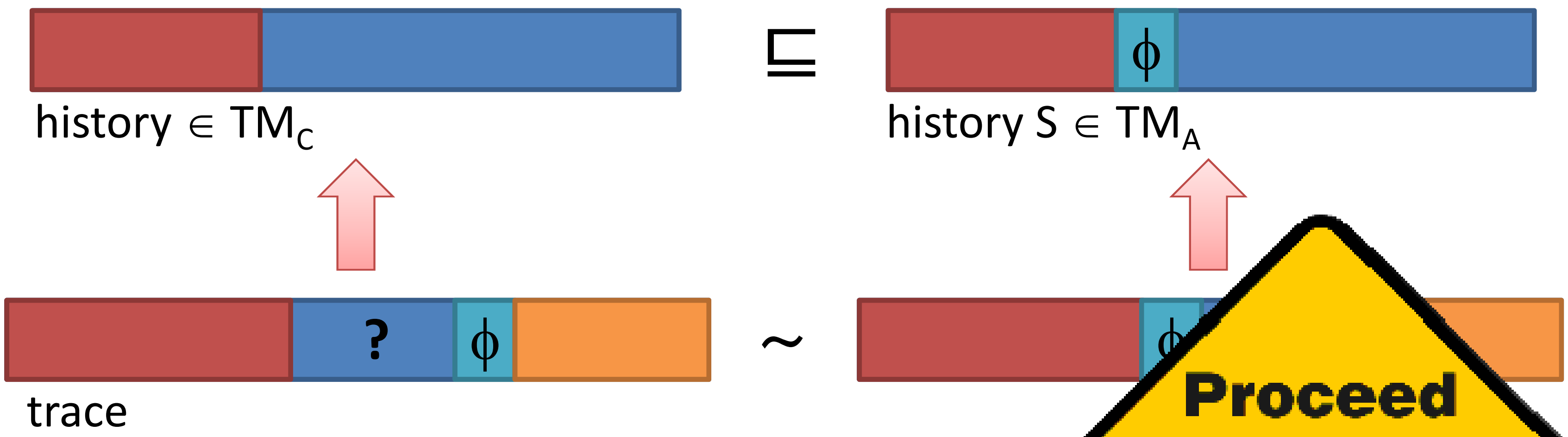


☞ Locate ϕ, the next interface action in S, and move it

☝ Reordering ϕ relative to earlier actions may violate equivalence

Soundness: Inductive Step

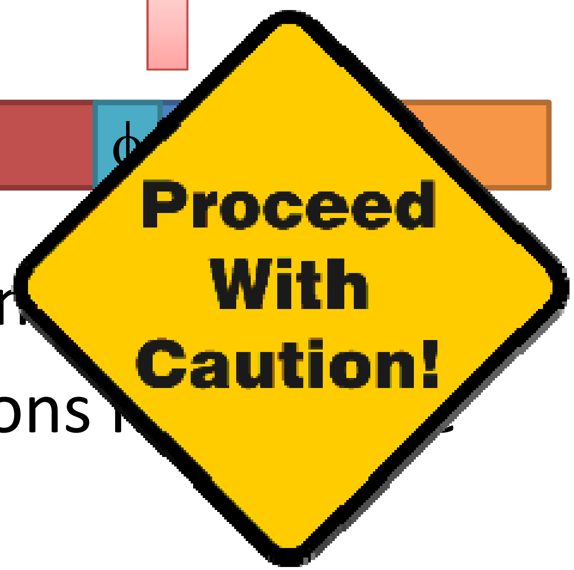
Assume we have permuted a prefix of τ so its history is a prefix of a history in TM_A



☞ Locate ϕ, the next interface action in

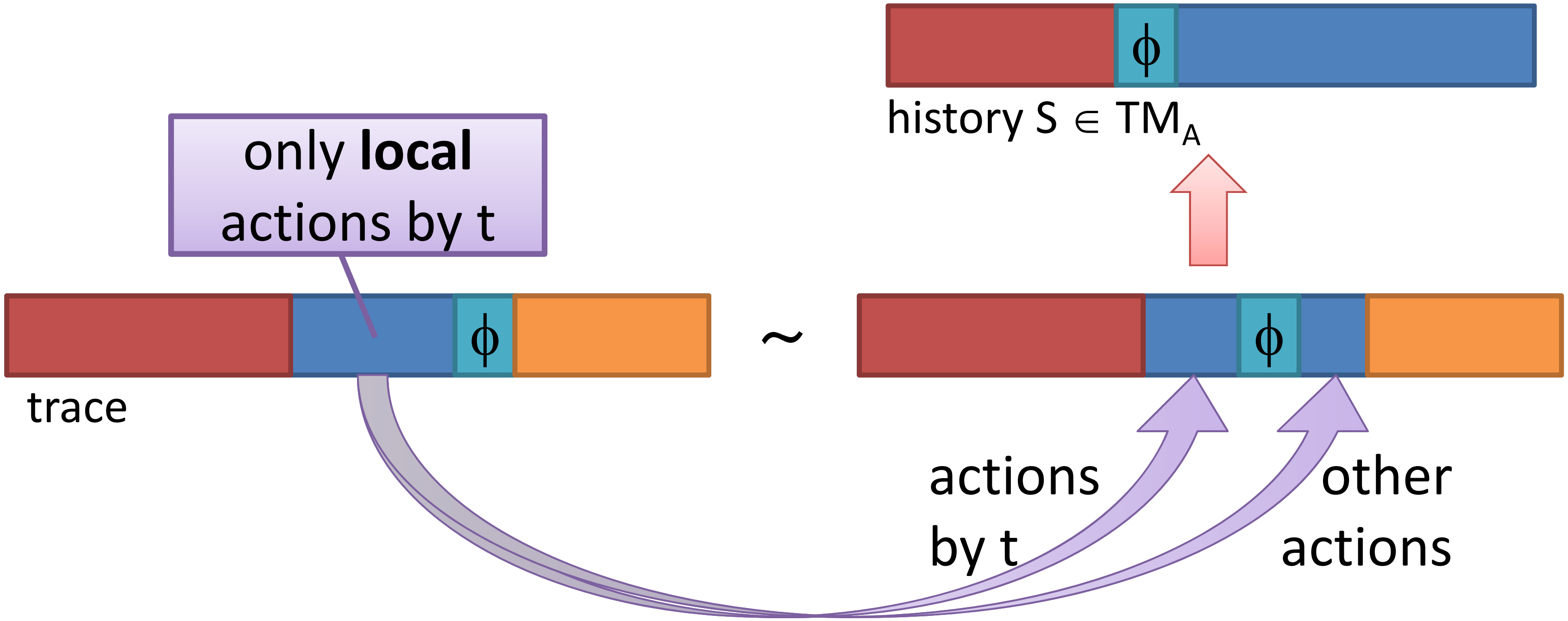
☝ Reordering ϕ relative to earlier actions

equivalence



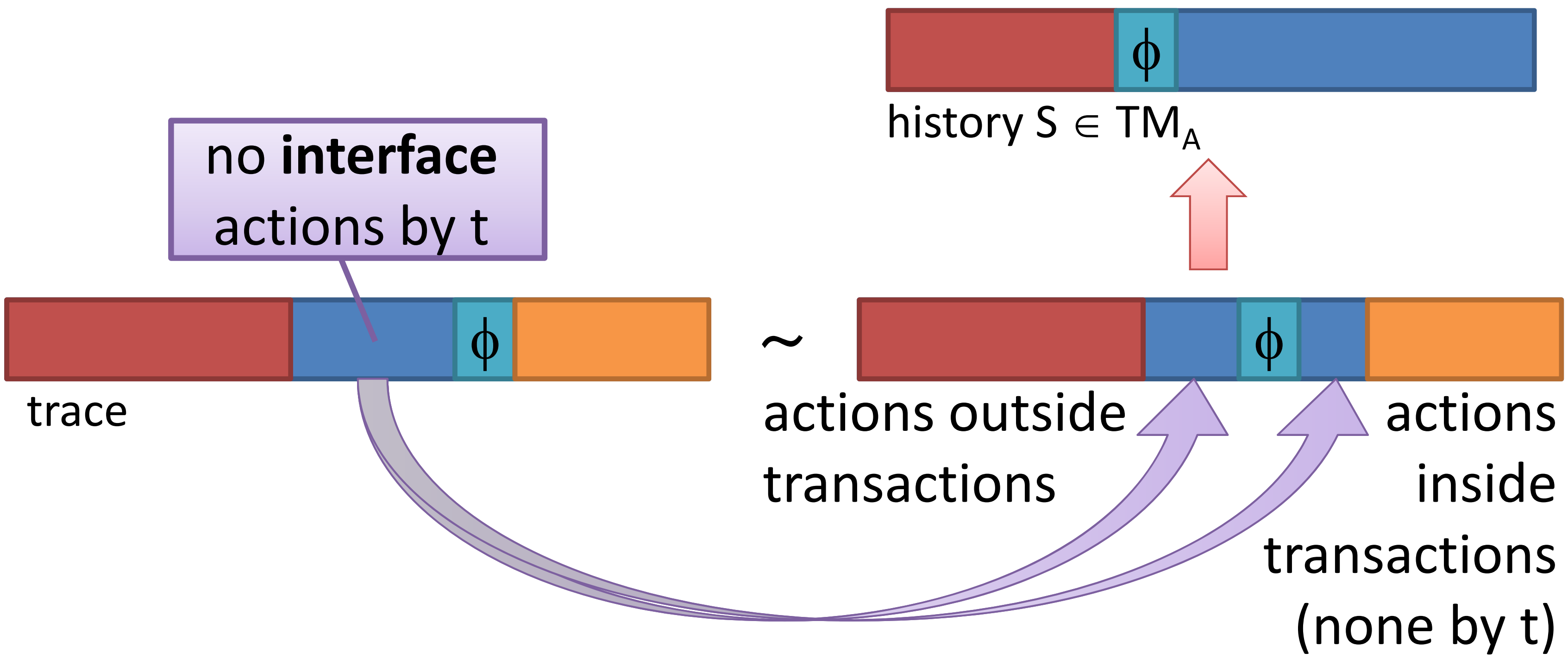
Inductive Step: Case 1

$\phi \neq$ txbegin by thread t



Inductive Step: Case 2

ϕ = txbegin by thread t



Example: $\sqsubseteq$ is Necessary

```
while (g <> 1 ) ;

result := atomic{
     node.next =
         Top.read();
     node.val++ ;
     Top = node;
    }
}
```

```
result := atomic{
     node.next =
         Top.read();
     node.val++ ;
     Top = node;
    }
}
g := 1 ;
```

Completeness: $\sqsubseteq$ is Necessary

$TM_C \sqsubseteq TM_A$ ⇦ TM_C observationally refines TM_A

- For every history H, construct a program P_H ensuring the opacity relation

- I.e., the real-time order between transactions in every trace of P_H must agree with the real-time order of the transactions in H

☝ Use global variables & leaking of local variables

Leaking Information from Aborted Transactions

Completeness result assumes we can read local state of aborted transactions

From ScalaTM Quick Start

Be careful about rollback

ScalaSTM might need to try an atomic block more than once before optimistic concurrency can succeed. Any call into the STM might potentially discover the failure and trigger the rollback and retry. Local non-`Ref` variables that have a lifetime longer than the atomic block won't be rolled back, and so they should be avoided. Local variables used only inside or only outside the atomic block are fine, though.

Below, `badToString` is incorrect because it uses a mutable `StringBuilder` both outside and inside its atomic block. The return value will definitely mention all of the elements

Weaker Observations, Weaker Consistency Conditions

- When local variables are rolled back after a transaction aborts, TMS1 may suffice
  - I/O automata based definition
  - In TMS, the validity of each response is checked against a "coherent" subset of the transactions
  - May include commit-pending transactions

3.2. Why TMS1 enables transactional programming

The purpose of TMS1 is to specify what guarantees the TM runtime must make in order to ensure that programmers who think about their programs as if only serial executions (i.e., executions in which the events of each transaction appear consecutively) are possible do not receive any unpleasant surprises as a result of the concurrent execution of transactions. We explain below how TMS1's validation conditions ensure that all responses given by the TM runtime are consistent with some serial execution of the program. In particular, for each response, we describe how to transform the actual program execution into a serial execution (i.e., one in which transactions are not interleaved with each other) such that the program cannot distinguish between the actual execution and the constructed serial execution.

First consider a commitOk or abort response that occurs when there are no other commit-pending transac-

What We Know about VWC

Sequence-based definition

- Each aborted transaction is checked for consistency (separately)

If atomic blocks return abort / commit
(typically assume
not preserve eve

VWC suffices if tl
codes or just one
thread and no gl


```
tmp0 := commit;
tmp0 := atomic{
    read tx ;
    write ty ;
}
if (tmp0 == abort )
    gv = 1 ;
    tmp1 = atomic{
        tz = 1 ;
    }
```

```
tmp3 = gv ;
result  = atomic{
    tmp4 = read(tz)
    if ((tmp3 == 1)
       or (tmp4 == 1))
        !!!!
}
```

Future Work

- infinite histories
- nesting
- access global variables inside atomic blocks (?)
- mixing transactional and non-transactional accesses
-
-
-

Possibly by considering other consistency conditions (**TMS2**, **DU-Opacity**)

Thank You