Verifying Transactional Memory

Simon Doherty [1] Brijesh Dongol [2] John Derrick [1]
Gerhard Schellhorn [3] Heike Wehrheim [4]

[1]University of Sheffield

[2]Brunel University

[3]University of Augsburg

[4]University of Paderborn

Outline

Section 1

Transactional Memory and Opacity

Transactional Memory

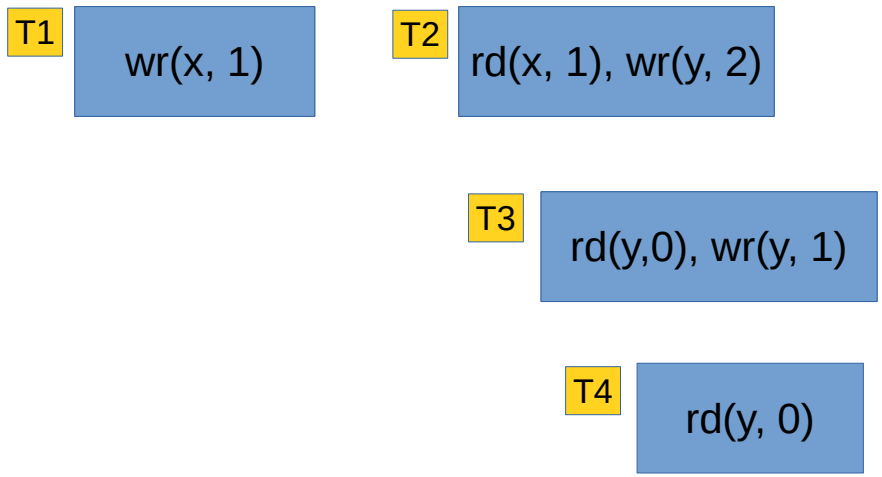
- Transactional memory allows the programmer to specify *transactions* that appear to execute atomically
- Transactions appear to be serialized into some (valid) total order
- Transactions might abort, and aborting transactions are invisible to other transactions
- These conditions are formalized in a correctness condition called *opacity* (Guerraoui and Kalka, 2008)

Opacity

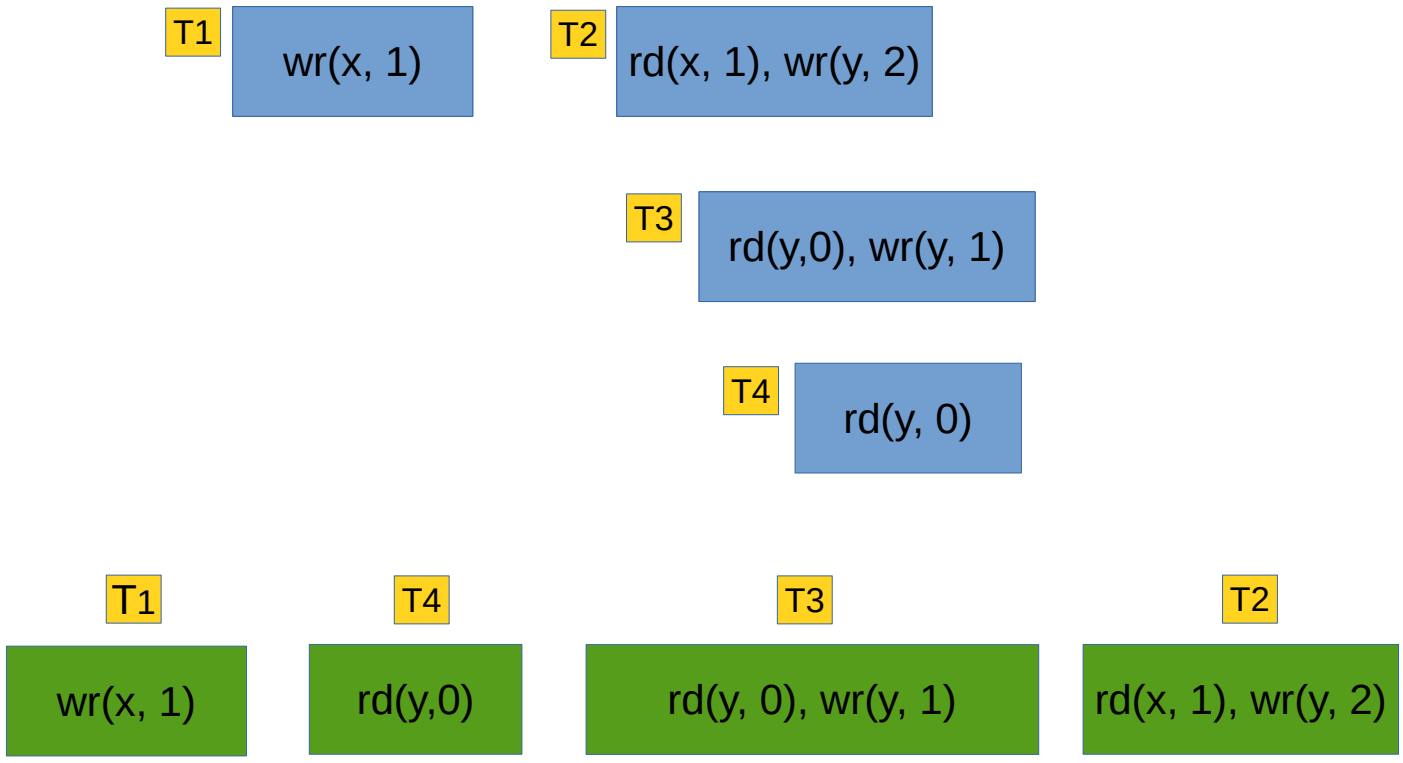
Opacity is a correctness condition akin to linearizability:

- Opacity defines a set of *legal histories* (sequences of invocations and responses)
- Transactions have *begin*, *commit*, *read* and *write* operations
- The serialization order must be consistent with semantics of read and write operations
- The serialization order must be consistent with the transactions' *real-time partial-order*

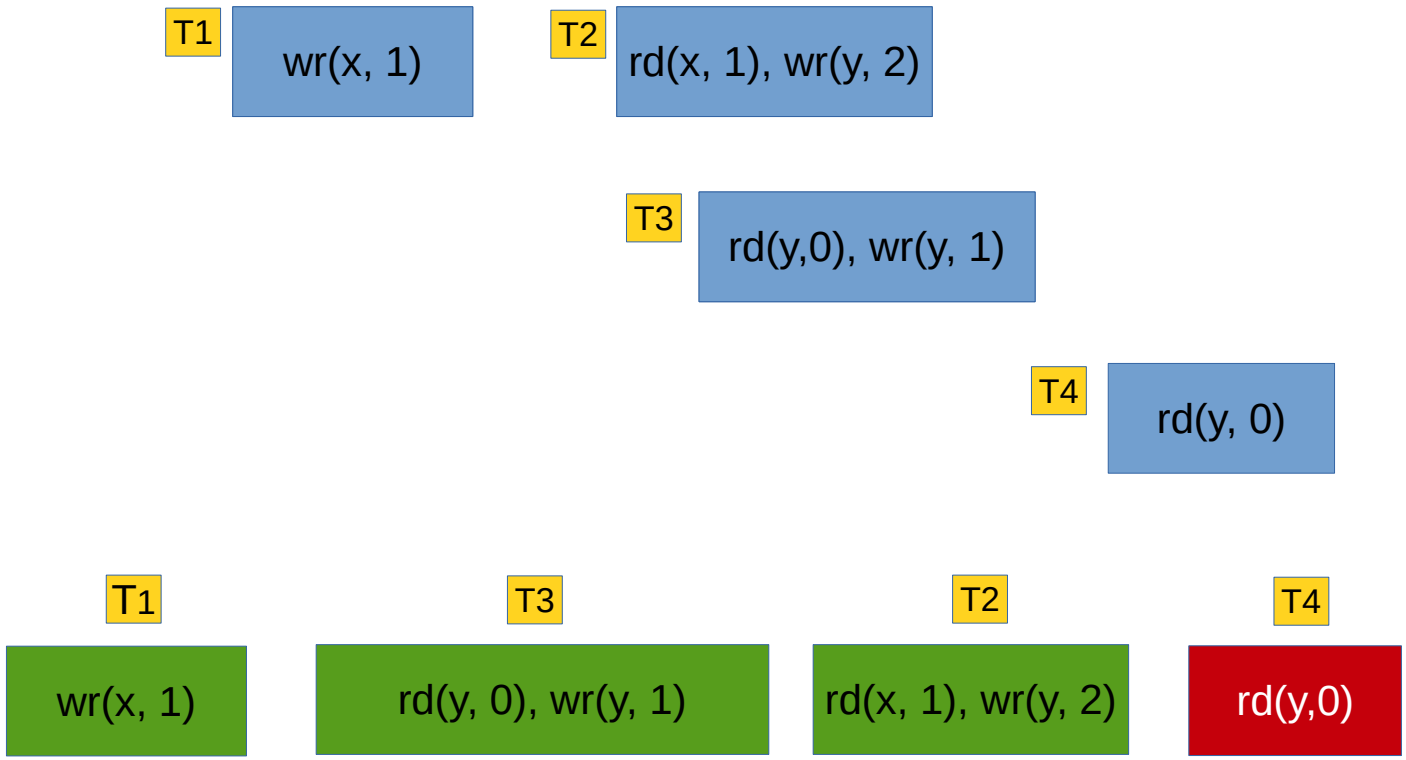
Opacity



Opacity


T1
wr(x, 1)
T2
rd(x, 1), wr(y, 2)
T3
rd(y,0), wr(y, 1)
T4
rd(y, 0)
T1
wr(x, 1)
T4
rd(y,0)
T3
rd(y, 0), wr(y, 1)
T2
rd(x, 1), wr(y, 2)

Opacity


T1
wr(x, 1)
T2
rd(x, 1), wr(y, 2)
T3
rd(y,0), wr(y, 1)
T4
rd(y, 0)
T1
wr(x, 1)
T3
rd(y, 0), wr(y, 1)
T2
rd(x, 1), wr(y, 2)
T4
rd(y,0)

Proving Opacity

We prove opacity with the help of an intermediate specification

$$\mathit{TM\ Implementation} \leq \mathit{TMS2} \leq \mathit{Opacity}$$

- TMS2 (Doherty *et al.*, 2013) is an I/O automaton (Lynch and Tuttle, 1987)
- An I/O automaton is a *labelled transition system* with a notion of *trace*
- Traces are sequences of externally visible behaviour, and all traces of TMS2 are opaque sequences of transactional operations: including reads, writes and commits

Refinement

TMS2 is defined as an IO-automaton, which comes equipped with a simple notion of refinement called *trace inclusion*

$$traces(A) \subseteq traces(B)$$

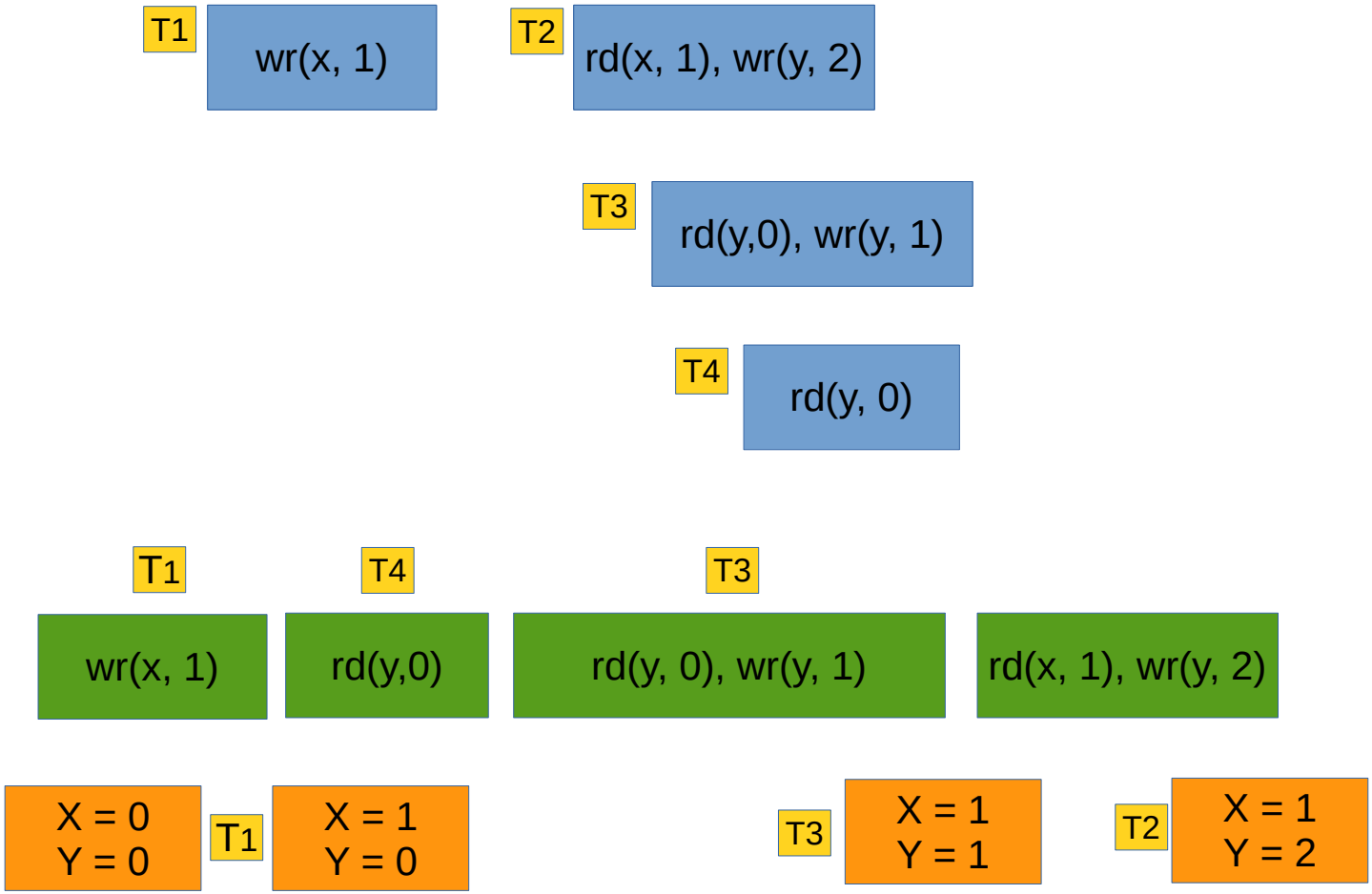
This can be verified using the standard technique of *simulation relations*.

TMS2

The states of TMS2 record a *sequence of stores*:

- During its *commit* operation, each writing transaction adds a new store onto the end of this sequence
- During its *begin* operation, each transaction records its *begin-index*, which is the current length of this sequence
- TMS2 ensures that each transaction is *valid* w.r.t some store appearing no earlier than the *begin-index*.

TMS2


T1
wr(x, 1)
T2
rd(x, 1), wr(y, 2)
T3
rd(y,0), wr(y, 1)
T4
rd(y, 0)
T1
T4
T3
wr(x, 1)
rd(y,0)
rd(y, 0), wr(y, 1)
rd(x, 1), wr(y, 2)
X = 0
Y = 0
T1
X = 1
Y = 0
T3
X = 1
Y = 1
T2
X = 1
Y = 2

Proving Opacity via TMS2

The proof framework described so far has been developed in prior work

- Doherty *et al.* 2009, 2013 for the development of TMS2 and related models
- Lesani *et al.* 2012 show that TMS2 is opaque
- Lesani *et al.* CONCUR 2012 develop a TM verification framework in PVS, and verify the NoRecs algorithm.

In this work we contribute

- A mechanization of this verification framework in Isabelle
- An adaptaton of rely/guarantee reasoning into this framework
- A mechanized verification of a subtle *pessimistic transactional-memory* algorithm

Section 2

Pessimistic Transactional Memory

Aborting Transactions

Typically, aborted transactions get retried. This pattern of aborting and retrying has drawbacks

- performance?
- liveness?
- interacts poorly with *irrevocable* operations: writing to a log, printing to a console, launching a rocket.

Pessimistic Transactional Memory

One solution: simply prohibit aborts.

- Matveev-Shavit, TRANSACT 2012 (MSPessTM)
- Transactions partitioned into *reader-only transactions* and (potential) *writers*
- Concurrency between readers
- Readers and writer can be concurrent (in contrast to R/W locks)
- One *active* writer can execute concurrently with one *committing writer*

Section 3

Rely/Guarantee Reasoning

Invariants

For this verification, the simulation relation is relatively simple:

- It is obvious how transactions should be ordered
- The relationship between the contents of the shared memory and the abstract memory sequence is straightforward

The challenging aspects of the MSPessTM algorithm are to do with the synchronization betwen different writers, and between readers and writers.

Structuring an Invariant

Our invariant is composed of a shared invariant S, and a transaction local invariant I_t

$$I = S \wedge \bigwedge_t I_t$$

- S constrains the shared state of MSPessTM
- I_t relates t's local state to the shared state

Structuring an Invariant

The local invariant is further indexed by actions a, so $I_{t,a}$ is t's local invariant when action a is enabled for t.

For each action a

$$S \wedge I_{t,a} \implies S' \qquad \text{[shared preservation]}$$

$$S \wedge I_{t,a} \implies I'_{t,a'} \qquad \text{[local preservation]}$$

We also need to prove some kind of noninterference condition.

Noninterference

For all distinct processes t, t', and for each action b, show

$$S \wedge I_{t,a} \wedge I_{t',b} \implies I'_{t,a}$$

This rule has disadvantages

- We need to prove it for every pair of actions a, b
- The rule can obscure the reason why $I_{t,a}$ is preserved

Using a Rely condition

We introduce a *rely* condition R_t, a relation over states of MSPessTM that preserves each I_t.

For each action a

$$S \wedge I_{t,a} \wedge R_t \implies I'_{t,a} \qquad \text{[stability]}$$

and for each t' and action b

$$S \wedge I_{t',b} \implies R_t \qquad \text{[guarantee]}$$

These rules plus shared and local preservation are sufficient to prove preservation of $I = S \wedge \bigwedge_t I_t$.

Where is the Guarantee?

Conventional rely/guarantee approaches employ a *guarantee relation* G satisfying something like

$$S \wedge I_{t,a} \implies G$$

and

$$G \implies R$$

This is unnecessary for us, because transactions are uniform in their behaviour.

Section 4

Completing the Proof

Active and Committing Writers

MSPessTM requires that there be at most one *active writer* and at most one *committing writer* at a time.
We equip the state with *activeWriter* and *committingWriter* auxilliary variables.
A writer must acquire permission before becoming *active*.

- MSPessTM has a lock, which can be acquired when no other writer is present
- If the lock is not available, writers wait in a *writer array*
- When a writer begins it's commit operation (and thereby stops being the active writer), it checks the writer array, and unblocks a writer if it finds one.

The Active Writer's Rely

Our rely condition requires for each t that whenever $activeWriter = t$

- $activeWriter' = t$
- $lock' = lock$
- $committingWriter = \bot \implies committingWriter' = \bot$
- $version'(l) = version(l)$

The Committing Writer's Rely

Our rely condition requires for each t that whenever $committingWriter = t$

- $committingWriter' = t$
- $glb' = glb$
- $store'(l) = store(l)$
- $version'(l) = version(l)$

Section 5

Conclusions

Summary

- Proved opacity of MSPessTM
- Applying the TMS2 specification
- Developed a rely/guarantee scheme for this setting
- Mechanized in Isabelle

Ongoing Work

Transactional memory and weak-memory

- Implementations running on weak memory
- Hybrid (hardware/software) implementations
- Formalizing and mechanizing semantics of interaction between TM and non-TM operations
- Development of framework for verifying TM in Isabelle