


ORACLE®



**ORACLE<sup>®</sup>**

## **Formal Machine-Checked Verification of a Real Transactional Memory Algorithm**

Victor Luchangco (joint work with Mark Moir)  
Oracle Labs



Copyright © 2011 Oracle and/or its affiliates (“Oracle”). All rights are reserved by Oracle except as expressly stated as follows. Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted, provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers, or to redistribute to lists, requires prior specific written permission of Oracle.



# Transactional Memory

- Facilitate concurrent programming
- Transactions encapsulate abstract operations
  - operations appear to take effect atomically

As a foundation used by other concurrent programs, we want to be especially sure that TM implementations are correct!



# Vision

- Precise formal specifications for transactional memory
  - one size will not fit all uses
- Precise formal models of TM algorithms
- Rigorous proofs that algorithms meet specifications



# Our approach

- Model algorithms as automata
  - relatively straightforward
  - well-developed theory: invariants, etc.
- Model specifications as automata
  - well-developed theory: simulation relations (refinements), etc.
  - hierarchical proofs: intermediate specifications
- Machine-checked proofs
  - greater confidence; repeatable
  - reusable: for different algorithms or specifications



# Hierarchical, reusable proofs

- High-level specification captures abstract requirements
- Intermediate specification for implementation approach
- Model algorithms at multiple levels



**ORACLE<sup>®</sup>**

## **Towards Formal Specification and Verification of Transactional Memory: Machine-Checked Proofs**

Victor Luchangco (joint work with Simon Doherty and Mark Moir)  
Sun Labs  
Scalable Synchronization Research Group





# What's new?

- Formalized NOrec algorithm
  - NOrecAtomicCommitValidate, NOrec, NOrecPseudocode
- Proved that NOrec implements TMS2
  - verified using PVS specification and verification system
  - (previously proved TMS2 implements TMS1)
- Rewrote fundamental definitions
  - Automata, Simulations, Sequences
- Proved basic theorems
  - e.g., simulation implies (finite) trace inclusion



# This talk

- I/O automata and simulations
- PVS
- TM specifications
- NOrec algorithm and automata
- Proof
- Challenges



# Background: I/O automata (simplified)

- states
- start states (nonempty)
- actions: external (input/output), internal
- transition relation: (state, action, state)
  
- execution fragment: state, action, state, action, state, ...
  - each consecutive (state, action, state) triplet is a step
- execution: starts with start state
- trace: projection onto external actions

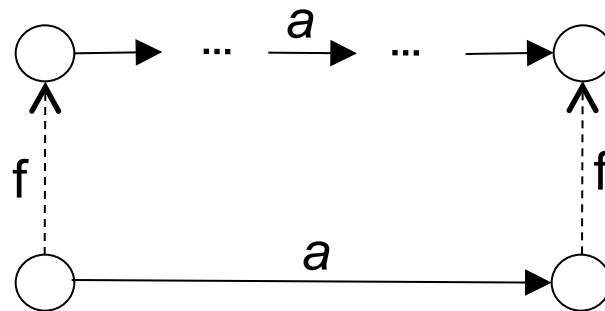


# Background: I/O automata (simplified)

- Automaton “generates” executions/traces
  - traces define visible behavior of automaton
    - as implementation: possible behavior
    - as specification: allowed behavior
  - any safety property can be encoded as an automaton
- Implementation = trace inclusion
  - not bisimulation

# Background: Simulation proofs

- Forward simulation  $f$  from implementation automaton **C** to specification automaton **A**
  - relation on  $\text{states}(C) \times \text{states}(A)$
  - for every start state of  $C$ , there is a corresponding start state of  $A$
  - for every step  $(s,a,s')$  of  $C$  and every state  $u$  of  $A$  corresponding to  $s$ , there is a state  $u'$  of  $A$  corresponding to  $s'$  such that there is a (possibly empty) sequence of steps from  $u$  to  $u'$  that appears identical to the step of  $C$ .





## Background: Simulation proofs

- Forward simulation  $f$  from implementation automaton **C** to specification automaton **A**
  - relation on  $\text{states}(C) \times \text{states}(A)$
  - for every start state of  $C$ , there is a corresponding start state of  $A$
  - for every step  $(s, a, s')$  of  $C$  and every state  $u$  of  $A$  corresponding to  $s$ , there is a state  $u'$  of  $A$  corresponding to  $s'$  such that there is a (possibly empty) sequence of steps from  $u$  to  $u'$  that appears identical to the step of  $C$ .
- Similar for backward simulation
- Forward and backward simulations are complete.
  
- We've used this successfully to verify concurrent data structures correct.



# Some alternative approaches

- Trace properties
  - e.g., linearizability, sequential consistency, serializability
  - graph-based proofs
- Operational semantics



# PVS

- Typed higher-order logic
- Rewriting-based theorem prover



```

Automata[State, Action: TYPE+,
    start: nonempty_pred[State],
    trans: pred[[State,Action,State]]]: THEORY BEGIN

FiniteStepSeq: TYPE = [# actions: finseq[Action],
    states: { ss: nonempty_finseq[State] |
        length(ss) = length(actions) + 1 } #]

stepseq: VAR FiniteStepSeq

length(stepseq): nat = stepseq`actions`length

steps(stepseq): finseq[Step] =
    (# length := length(stepseq`actions),
    seq := LAMBDA (n: below[length(stepseq`actions)]):
        (stepseq`states(n), stepseq`actions(n), stepseq`states(n+1)) #)

finiteExecFrag(stepseq): bool =
    FORALL (n: below[length(stepseq)]): trans(steps(stepseq)(n))

finiteExecution(stepseq): bool =
    finiteExecFrag(stepseq) AND start(first(stepseq))

```

```

reachable(s): INDUCTIVE bool =
  start(s) OR (EXISTS s0,a: reachable(s0) AND trans(s0,a,s))

reachableAlt: LEMMA
  reachable(s) IFF
    EXISTS (fexec, (n: below[length(fexec`states)])):
      s = fexec`states(n)

invariant(p: pred[State]): bool =
  FORALL s: reachable(s) IMPLIES p(s)

invariantInduction: LEMMA
  FORALL (p: pred[State]):
    (FORALL s: start(s) IMPLIES p(s))
    AND
    (FORALL s0, a, s1:
      reachable(s0) AND reachable(s1) AND p(s0) AND trans(s0,a,s1)
      IMPLIES p(s1))
    IMPLIES invariant(p)

END Automata

```




# TM specifications (previous work)

- Serializability
  - does not constrain behavior of aborted transactions
- Opacity [Guerraoui, Kapalka]
  - single serialization “explains” aborted and committed txns
- Virtual world consistency (VWC) [Imbs, et al.]
  - different system model (also, read/write memory only)
  - no external consistency
- TMS1: high-level abstract specification
  - “as permissive as possible”
- TMS2: captures intuition of many TM implementations
  - implies TMS1 (opacity?)



## TMS2: “Write-latest”

- $\text{beginIdx}_t$ : “timestamp” of state at beginning of txn  $t$
- $\text{mem}$ : sequence of memory states
- $\text{wrSet}_t$ : write set of  $t$
- $\text{rdSet}_t$ : read set of  $t$
- $\text{pc}_t$ : bookkeeping



```
TMS2[Txn, Loc, Val: TYPE+, validInit: nonempty_pred[[Loc -> Val]]]:  
THEORY BEGIN
```

```
ActionType: DATATYPE ...
```

```
Action: TYPE+ = [# txn: Txn, acttype: ActionType #]
```

```
State: TYPE =  
  [# pc: [Txn -> PCValue],  
    beginIdx: [Txn -> nat],  
    mem: nonempty_finseq[RWState],  
    wrSet: [Txn -> PartialFunction[Loc,Val]],  
    rdSet: [Txn -> PartialFunction[Loc,Val]] #]
```


```
start(s): bool =  
  s`mem`length = 1  
  AND validInit(last(s`mem))  
  AND (FORALL t:    s`pc(t) = notStarted  
           AND s`rdSet(t) = emptyMap  
           AND s`wrSet(t) = emptyMap)
```

```
precondition(a)(s): bool = ...
```

```
effect(a,s): State = ...
```

```
trans(s0,a,s1): bool = precondition(a)(s0) AND s1 = effect(a,s0)
```

```
IMPORTING Automata[State, Action, start, trans]
```



```
ActionType: DATATYPE WITH SUBTYPES external, internal
BEGIN
  beginTxn: beginTxn?                : external
  beginOk: beginOk?                  : external
  inv(i: Invocation): inv?           : external
  resp(r: Response): resp?          : external
  commit: commit?                    : external
  commitOk: commitOk?                : external
  cancel: cancel?                     : external
  abort: abort?                       : external
  doReadWritten(l: Loc): doReadWritten? : internal
  doReadUnwritten(l: Loc, n: nat): doReadUnwritten? : internal
  doWrite(l:Loc, v: Val): doWrite?    : internal
  doCommitReadOnly: doCommitReadOnly? : internal
  doCommitWriter: doCommitWriter?    : internal
END ActionType
```

```

precondition(a)(s): bool = LET t = a`txn IN
CASES a`acttype OF
  beginTxn: s`pc(t) = notStarted,
  beginOk: s`pc(t) = beginPending,
  inv(i): s`pc(t) = ready,
  resp(r): (readResp?(s`pc(t)) AND r = readOk(v(s`pc(t))))
            OR (writeRespOk?(s`pc(t)) AND r = writeOk),
  commit: s`pc(t) = ready,
  commitOk: s`pc(t) = commitRespOk,
  cancel: s`pc(t) = ready,
  abort:   s`pc(t) = beginPending
            OR reading?(s`pc(t))
            OR writing?(s`pc(t))
            OR s`pc(t) = doCommit
            OR s`pc(t) = cancelPending,
  doReadWritten(l): s`pc(t) = reading(l) AND dom(s`wrSet(t))(l),
  doReadUnwritten(l,n): s`pc(t) = reading(l)
                        AND NOT dom(s`wrSet(t))(l)
                        AND validIndex(s,t,n),
  doWrite(l,v): s`pc(t) = writing(l,v),
  doCommitReadOnly: s`pc(t) = doCommit
                    AND dom(s`wrSet(t)) = emptyset,
  doCommitWriter:  s`pc(t) = doCommit
                    AND dom(s`wrSet(t)) /= emptyset
                    AND readCons(last(s`mem),s`rdSet(t))
ENDCASES

```

```

effect(a,s): State =
  IF precondition(a)(s) THEN LET t = a`txn IN
    CASES a`acttype OF
      beginTxn: s WITH [`pc(t) := beginPending,
                        `beginIdx(t) := s`mem`length-1],
      beginOk: s WITH [`pc(t) := ready],
      inv(i): s WITH [`pc(t) :=
        IF read?(i) THEN reading(l(i)) ELSE writing(l(i),v(i)) ENDIF],
      resp(r): s WITH [`pc(t) := ready],
      commit: s WITH [`pc(t) := doCommit],
      commitOk: s WITH [`pc(t) := committed],
      cancel: s WITH [`pc(t) := cancelPending],
      abort: s WITH [`pc(t) := aborted],
      doReadWritten(l): s WITH [`pc(t) := readResp(down(s`wrSet(t)(l)))] ,
      doReadUnwritten(l,n): (s WITH [`pc(t) := readResp(v),
                                     `rdSet(t)(l) := up(v)]
                            WHERE v = s`mem(n)(l)),
      doWrite(l,v): s WITH [`pc(t) := writeRespOk,
                            `wrSet(t)(l) := up(v)],
      doCommitReadOnly: s WITH [`pc(t) := commitRespOk],
      doCommitWriter: s WITH [`pc(t) := commitRespOk,
                              `mem := s`mem o oride(last(s`mem),
                                                    s`wrSet(t))]

    ENDCASES
  ELSE
    arbitraryState
  ENDIF

```





## **NOrec algorithm [Dalessandro, et al.]**

- Simple deferred-update alg: “no ownership records”
  - write shared memory on commit
  - maintain private read and write sets
  - reads are invisible
- Sequence lock to protect writeback
  - serializes commit of writing transactions
  - readers check that lock is not held
- Value (re)validation when sequence lock changes
- Low overhead
  - good when conflicts are rare



# **NOrec automata**

- NOrecAtomicCommitValidate
- NOrecDerived
- NOrec
- NOrecPaperPseudocode

```

NorecAtomicCommitValidate[Txn, Loc, Val: TYPE+,
                           validInit: nonempty_pred[[Loc -> Val]]]: THEORY
BEGIN

ActionType: DATATYPE ...

Action: TYPE+ = [# txn: Txn, acttype: ActionType #]

State: TYPE =
  [# pc: [Txn -> PCValue],
   currMem: RWState,
   globalSN: nat,
   snapshotSN: [Txn -> nat],
   wrSet: [Txn -> PartialFunction[Loc,Val]],
   rdSet: [Txn -> PartialFunction[Loc,Val]]
  #]

start(s): bool = ...


precondition(a)(s): bool = ...

effect(a,s): State = ...


trans(s0,a,s1): bool = precondition(a)(s0) AND s1 = effect(a,s0)

IMPORTING Automata[State, Action, start, trans]

```



```
ActionType: DATATYPE WITH SUBTYPES external, internal
BEGIN
  beginTxn: beginTxn?                : external
  beginOk: beginOk?                  : external
  inv(i: Invocation): inv?           : external
  resp(r: Response): resp?          : external
  commit: commit?                    : external
  commitOk: commitOk?                : external
  cancel: cancel?                    : external
  abort: abort?                      : external
  doBegin: doBegin?                  : internal
  readValidate: readValidate?        : internal
  doReadWritten(l: Loc): doReadWritten? : internal
  doReadUnwritten(l: Loc): doReadUnwritten? : internal
  doWrite(l:Loc, v: Val): doWrite?    : internal
  doCommitReadOnly: doCommitReadOnly? : internal
  doCommitWriter: doCommitWriter?    : internal
END ActionType
```




```
precondition(a)(s): bool = LET t = a`txn IN
CASES a`acttype OF
  beginTxn: s`pc(t) = notStarted,
  beginOk: s`pc(t) = begun,
  ...
  doBegin: s`pc(t) = beginPending,
  readValidate: readCons(s`currMem, s`rdSet(t)),
  doReadWritten(l): s`pc(t) = reading(l) AND dom(s`wrSet(t))(l),
  doReadUnwritten(l): s`pc(t) = reading(l)
    AND NOT dom(s`wrSet(t))(l)
    AND s`snapshotSN(t) = s`globalSN,
  doWrite(l,v): s`pc(t) = writing(l,v),
  doCommitReadOnly: s`pc(t) = doCommit
    AND dom(s`wrSet(t)) = emptyset,
  doCommitWriter: s`pc(t) = doCommit
    AND dom(s`wrSet(t)) /= emptyset
    AND s`snapshotSN(t) = s`globalSN
ENDCASES
```

```

effect(a,s): State =
  IF precondition(a)(s) THEN LET t = a`txn IN
    CASES a`acttype OF
      beginTxn: s WITH [ `pc(t) := beginPending],
      ...
      doBegin: s WITH [ `pc(t) := begun,
                       `snapshotSN(t) := s`globalSN ],
      readValidate: s WITH [ `snapshotSN(t) := s`globalSN ],
      doReadWritten(l): s WITH [ `pc(t) := readResp(down(s`wrSet(t)(l)) )],
      doReadUnwritten(l,n): (s WITH [ `pc(t) := readResp(v),
                                     `rdSet(t)(l) := up(v) ]
                              WHERE v = s`mem(n)(l) ),
      doWrite(l,v): s WITH [ `pc(t) := writeRespOk,
                            `wrSet(t)(l) := up(v) ],
      doCommitReadOnly: s WITH [ `pc(t) := commitRespOk],
      doCommitWriter: s WITH [ `pc(t) := commitRespOk,
                              `currMem := oride(s`currMem, s`wrSet(t)),
                              `globalSN := s`globalSN + 1 ]

    ENDCASES
  ELSE
    arbitraryState
  ENDIF

```




```
NOrec[Txn, Loc, Val: TYPE+,
      validInit: nonempty_pred[[Loc -> Val]]]: THEORY
BEGIN

ActionType: DATATYPE ...

Action: TYPE+ = [# txn: Txn, acttype: ActionType #]

State: TYPE =
  [# pc: [Txn -> PCValue],
   currMem: RWState,
   globalSN: nat,
   commitLock: Lock,
   l: [Txn -> Loc],
   v: [Txn -> Val],
   snapshotSN: [Txn -> nat],
   wrSet: [Txn -> PartialFunction[Loc,Val]],
   rdSet: [Txn -> PartialFunction[Loc,Val]],
   validationSN: [Txn -> nat],
   validated: [Txn -> setof[Loc]],
   writtenBack: [Txn -> setof[Loc]]
  #]
```



```
ActionType: DATATYPE WITH SUBTYPES external, internal
BEGIN
  beginTxn: beginTxn?                : external
  beginOk: beginOk?                  : external
  inv(i: Invocation): inv?           : external
  resp(r: Response): resp?          : external
  commit: commit?                    : external
  commitOk: commitOk?                : external
  cancel: cancel?                    : external
  abort: abort?                      : external
  doBegin: doBegin?                  : internal
  doReadWritten(l: Loc): doReadWritten? : internal
  doReadUnwritten(l: Loc): doReadUnwritten? : internal
  validateNewRead: validateNewRead?  : internal
  doWrite(l:Loc, v: Val): doWrite?    : internal
  startValidation: startValidation?   : internal
  validateLoc(l: Loc): validateLoc?   : internal
  confirmValidation: confirmValidation? : internal
  doCommitReadOnly: doCommitReadOnly? : internal
  doCommitWriter: doCommitWriter?    : internal
END ActionType
```





# Machine-checked proofs using PVS

- Formalize automata in PVS language
  - need to formalize basic tools: automata, sequences
- State lemmas (also in PVS language)
  - invariants, “unchanged lemmas”, simulation theorem
  - also lots of basic lemmas about automata, sequences, etc.
- Prove lemma using PVS prover
  - theorem prover (not model checker)
    - equational rewriting-based, with decision procedures



## Our experience

- Learning how to use prover takes time
- Lots of “obvious” lemmas needed
- Typed logic is mixed blessing
- Better “development environment” would help



## Previous TM verification work

- Guerraoui, Henzinger, Jobstmann, Singh [PLDI'08]
  - verify “abort consistency” (like opacity) using model checkers
    - impl must satisfy certain structural properties (not verified)
- Cohen, O’Leary, Pnueli, Tuttle, Zuck [FMCAD’07]
  - automata-based models formalized in PVS
  - does not constrain aborted transactions, not real protocol
- O’Leary, Saha, Tuttle [ICDCS’09]
  - Use Spin to model check small instances of McRT STM
    - only serialisability, no constraints on aborted transactions
- Moore, Grossman [POPL’07]
  - small step operational semantics for language level
  - no real implementation



# Summary

- Verified real TM algorithm (NOrec)
- Developed libraries for modeling and verifying algorithms using PVS



# Future directions

- Verify other TM algorithms
- Specify opacity as automaton
  - prove that it implements TMS1 and is implemented by TMS2
- Extend to infinite traces
- Extend interface/specification
  - nontransactional operations
  - weakly consistent memory models
  - nested transactions
  - ...

**ORACLE®**