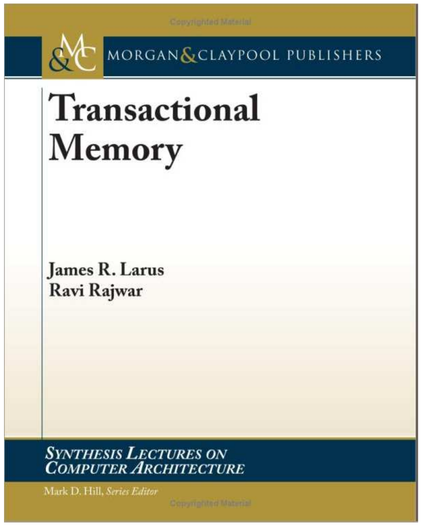
An Abort-Aware Model of Transactional Programming

Kousha Etessami
U. of Edinburgh

Patrice Godefroid
Microsoft Research

Background: Multi-Core Revolution

- New multi-core machines
- The masses will have to learn concurrent programming
- But using locks and shared memory is hard and messy
- Can we use something better?
- One proposal: transactional programming
  - Processes communicate using atomic transactions
  - Gives the illusion of sequential programming
  - Requires an underlying "Transactional Memory"
  - Lots of recent research on efficient software and hardware "Transactional Memory" implementations



Motivation of This Work

- A program analysis/verification point of view of Transactional Programming
  - If Transactional Programs are really easier to write, then they should be easier to verify ! Is that true?
  - But wait, what is a transactional program?
- Part 1: high-level semantics for transactional programs
  - A critique of single-lock semantics
  - An abort-aware semantics for transactions
- Part 2: TSMs = Transactional State Machines
  - A finite-state abstract model of transactional programs
  - Some first verification results

Transactional Program

- A transactional program runs on top of a STM or HTM implementation
  - Processes communicate using atomic transactions accessing shared memory

- What is the API? What is a transaction?

- Syntax: atomic { . . . }

- Semantics? "single-lock semantics"
  - easy
  - But not satisfactory because overly simplified ("as if")
    - No parallelism allowed, blocking (known)
    - Ignores non-terminating transactions (known)
  - Ignores STM/HTM aborted transactions: for responsiveness, "abort" cannot be equal to "retry" (new)

Single-Lock Semantics is too Simplistic

- Example: flight reservation program

```
Transaction book(Agent, Flight_Nbr, Customer_Id ) {
  forall possible Seat_Number:
    if (Agent.Flights[Flight_Nbr,Seat_Number] == available)
      then { Agent.Flights[Flight_Nbr,Seat_Number] = Customer_Id;
             return; // attempt to commit }
  return full; // explicit abort for “no seats available”
}
Main() { ...
  status = book(Expedia, AA175, JohnDoe);
  if (status == full) . . . // try another flight
  if (status == commit) . . . // great, move on
  if (status == abort) . . . // notify user & retry with Orbitz
}
```

the transactional program wants to be notified of any automatic abort (for responsiveness)

dead code with single-lock semantics

- Transactions may abort; aborted transactions must have side-effects to be able to test for success (commit) or failure (abort)
- Transactional programs must deal with aborted transactions

Transactional State Machines (TSMs)

- A foundation for the analysis of Transactional Programs
- TSM = “a FSM model for transactional programs”
- TSM = concurrent Recursive State Machines (RSMs) + shared variables + transactions
  - RSMs = (finite-state) procedures which can call each other
  - Each thread/process executes one RSM
  - Shared variables (with finite domain) for communication
  - Transactions: some procedures are transactional
  - Nesting: recursion is allowed
- Each terminating transaction ends in a commit or abort

An Abort-Aware Semantics for Transactions

- There is a universal copy of all shared variables v
- After a transaction has started, for every shared variable v,
  - every first read of v is recorded in a fixed copy
  - every write and subsequent read of v is performed on a mutable copy
- If/when the transaction terminates,
  - If fixed copy == universal copy, universal copy = mutable copy (commit)

    Assumption: this is done with a single atomic compare-and-swap
  - Else there is a memory conflict and the transaction is aborted (abort)
- Notes : (this is just one possible semantics, variants are ok too)
  - Transactions are non-blocking and concurrent
  - Memory conflicts are based on values, not accesses
  - Nested transactions are "closed" (because we think open nested transactions with inner commits and outer aborts do not make sense !)

Examples and Remarks

```
    Initially, x = 0
Process 1 | Process 2
----------+----------
atomic {  | r1 = x;
  x = 1;  |
  x = 2;  |
}         |
  Can r1 == 1? No.
```

```
     Initially, x = y = 0
Process 1        | Process 2
-----------------+-----------
atomic {         | r1 = y;
  y = 1;         | atomic {
  if (x == 0)    |   x = 1;
      abort;     | }
}                |
     Can r1 == 1? No.
```

```
    Initially, x = y = 0
Process 1 | Process 2
----------+----------
atomic {  | r1 = x;
  x = 1;  | r2 = y;
  y = 1;  |
}         |
Can r1 == 1, r2 == 0? No.
```

- Remarks : (choices that simplify the TSM semantics)
  - We assume strong isolation/atomicity (ex1)
    - To be able to define an interleaving semantics
  - Explicit aborts supported (exception-raising mechanism) (ex2)
  - Deferred update (not direct update) (ex2)
  - Possible compiler re-orderings are not part of semantics (ex3)

Stutter-Serializability

- Properties of abort-aware semantics:
  - If a transaction commits, it is as if it can be entirely scheduled at the time of its successful compare-and-swap operation
  - Therefore, any sequence of changes to the universal copy can be witnessed by a serial execution of committed transactions

    (Like with "single-lock semantics", yet we accommodate aborts !)
- Formally,
  - a TSM is stutter-serializable if, for every run r of the TSM, there is a run r' such that r[U] is stutter-equivalent to r'[U] and all committed transactions during r' are serial
- Theorem: All TSMs are stutter-serializable

Model Checking (MC)

- Theorem: in general, MC of TSMs for stutter-invariant linear (LTL) properties of shared memory is undecidable
  - Proof idea: with possibly unbounded recursion, TSMs can simulate concurrent PDAs, i.e., a Turing machine
- Theorem: if recursion only occurs inside transactions, MC of TSMs for stutter-invariant linear (LTL) properties of shared memory is decidable
  - Proof idea: (for finitely-many finite-domain variables) state transformations performed by recursive transactions can be "summarized" in finitely-many possible ways, both for shared variables (commits) and local variables
- Note: same results with other TSM semantics like abort=retry, nondeterministic aborts, single-lock, etc.

Conclusions

- Plan:
  - Transactional programming is embraced by the masses
  - Transactional programs are automatically abstracted into TSMs
  - These TSMs are analyzed with tools (model checking, etc.)
- TSM = a model for Transactional Programs
- Abort-aware semantics to allow reactive programming
  - Responsiveness is important yet mostly ignored so-far in TM world !
  - With abort-aware semantics, all TSMs are stutter-serializable
  - Clean high-level semantic property akin “single-lock semantics”, yet formal and includes aborts
- Some model-checking results