Software Transactional Memory & Automatic Mutual Exclusion

Martin Steffen

University of Oslo, Norway

Oslo

Introduction

Transactional Java
  Operational semantics without transactions
  Transactional semantics
  Versioning semantics
  Two-phase locking

Conclusion
Motivation

- concurrency $\Rightarrow$ concurrency control
- nowaday's languages: lock-based (good ol' mutex)
- disadvantages:
  - low-level of abstraction
  - difficult to reason about
  - "conservative" protection $\Rightarrow$ performance penalty / deadlocks
  - pessimistic approach to concurrency control
- here: "optimistic" approach
  - reduce crit-secs, more concurrency $\Rightarrow$ non-blocking
Transactions

• coming from the data-base community
• control abstraction
• important correctness/failure properties: ACID transaction semantics = “illusion” of mutex
  1. atomicity
  2. isolation
  3. consistency
  4. durability
Introduction

Transactional Java
- Operational semantics without transactions
- Transactional semantics
- Versioning semantics
- Two-phase locking

Conclusion
TFJ

- taken from [Jagannathan et al., 2005]
- extending Featherweight Java with transactions
  - state
  - multi-threading (of course)
  - transactions
- featuring: nested and multi-threaded transactions
- operational semantics, 2 concretizations
  - versioning
  - 2-phase locking
- correctness proof: serializability
Why are transactions more high-level?

class Transactor {
    u: Updater;
    r: Runner;
    init (r: Runner, u: Updater) {
        this.u := u;
        this.r := r;
        this
    }

    run () {
        this.u.update(); // write
        this.r.run(); // spawn intervening activity
        thus.u.n.val; // read
    }
}
}
Why are transactions more high-level?

class Transactor {
  u: Updater;
  r: Runner;
  init (r: Runner, u: Updater) {
    this.u := u;
    this.r := r;
    this
  }

  run () {
    onacid
    this.u.update(); // write
    this.r.run(); // spawn intervening activity
    thus.u.n.val; // read
    commit
  }
}
Syntax

\[ P ::= 0 \mid P \parallel P \mid t\langle e \rangle \]  
process

\[ L ::= \text{class}C\{\vec{f}; \vec{M}\} \]  
class definition

\[ M ::= m(\vec{x})\{e\} \]  
method

\[ e ::= x \mid e.f \mid e, m(\vec{e}) \mid e.f := e \]  
expression

\[ \mid \text{news}C \mid \text{spawn} e \mid \text{onacid} \mid \text{commit} \mid \text{null} \]  
values/basic expressions

\[ v ::= r \mid v, f \mid v.m(\vec{v}) \mid b.f := v \]  

- basically 2 additions:
  - onacid: start a transaction
  - commit: end a transaction
Semantics

- given operationally (SOS, as usual ...)
  - labelled transition system
  - evaluation-contexts
- 2 “stages”:
  1. first “general” semantics
  2. afterwards: 2 concretizations
- 2-level semantics
  1. local = per thread
  2. global = many threads
Underlying semantics: no transactions

- for illustration here, only
- no separation in local ↔ global steps
- no transaction handling (but concurrency)
- heap-manipulations (read, write, extend) left “unspecified”
- configuration (local/global): $\Gamma \vdash e$
Operational semantics: no transactions

\[
\begin{align*}
\text{read}(r, \Gamma) &= C(\vec{u}) & \textit{fields} (C) &= \vec{f} & \text{R-FIELD} \\
\Gamma \vdash r. f_i & \xrightarrow{rd_r} & \Gamma \vdash u_i \\
\text{read}(r, \Gamma) &= C(\vec{r}) & \text{write}(r \mapsto C(\vec{r}) \downarrow_{i'}, \Gamma) &= \Gamma' & \text{R-ASSIGN} \\
\Gamma \vdash r. f_i := r' & \xrightarrow{wr rr'} & \Gamma' \vdash r' \\
\text{read}(r, \Gamma) &= C(\vec{r}) & \text{mbody}(m, C) = (\vec{x}, e) & \text{R-INVOKE} \\
\Gamma \vdash r. m(\vec{r}) & \xrightarrow{rd_r} & \Gamma \vdash e[\vec{r}/\vec{x}][r/\text{this}] \\
r \fresh & \quad \text{extend}(r \mapsto C(\vec{\null}), \Gamma) = \Gamma' & \text{R-NEW} \\
\Gamma \vdash \text{new} C() & \xrightarrow{xt_r} & \Gamma' \vdash r \\
\end{align*}
\]

\[
\begin{align*}
P &= P'' \parallel n\langle E[\text{spawn } e] \rangle & P' &= P'' n\langle E[\text{null}] \rangle \parallel n'\langle e' \rangle \\
n' \fresh & \quad \text{spawn}(n, \mathcal{E}', \Gamma) = \Gamma' & \text{R-SPAWN}
\end{align*}
\]
Introducing transactions

- as said: syntax: onacid + commit
- steps: split into 2 levels
  1. local: per thread
  2. global: “inter”-thread
- more complicated “memory model”
  - each thread has a local copy
  - how that exactly works ⇒ depending on the kind of transaction implementation (see later)
- general idea: optimistic approach
  - each thread works on its local copy (no locks, no regard of others)
  - local copy ⇒ isolation
  - when committing: check for conflicts ⇒
    - no: ⇒ make the effect visible
    - yes: ⇒ abort
Transactions and threads

- both are **dynamic**
  - thread creation by **spawn**
  - transaction “creation” by **onacid**

- transaction structure: **nested**¹
  - a transaction can contain inner transactions
  - child transactions must commit **before** outer transaction
  - child transaction
    - commits \(\Rightarrow\) effects become visible to **outer** transaction
    - aborts \(\Rightarrow\) outer transaction does not abort

- relationship:
  - each thread **inside** an enclosing transaction²
  - “**multi**” threads in one transaction

¹Thread structure: flat. One could make a hierarchical “father-child” structure, but it’s irrelevant here.
²or toplevel
Local steps

- steps concerning one thread
- basic “single-threaded”, “non-transactional” steps
- local state/configuration:
  - “simple” expression \( e + \text{ local environment } \mathcal{E} \)
  \[ \mathcal{E} |- e \]
  - \( \mathcal{E} : \)
    - per transaction (labelled with \( l \)): local (partial) “state” = assoc of references to values
    - manipulated by read/write/extend
    - details determine the transactional model
    - Note: read-access may change \( \mathcal{E} \)

---

\(^3\)The paper itself is undecided whether to call it transaction environment or a sequence of transaction environments.
Local steps: rules

\[
\begin{align*}
\text{read}(r, \mathcal{E}) &= \mathcal{E}', \ C(\vec{u}) \\
\text{fields}(C) &= \vec{f} \\
\end{align*}
\]

R-FIELD

\[
\mathcal{E} \vdash r.f_i \quad \xrightarrow{\text{rd} \ r} \quad \mathcal{E}' \vdash u_i
\]

\[
\begin{align*}
\text{read}(r, \mathcal{E}) &= \mathcal{E}', \ C(\vec{r}) \\
\text{write}(r \mapsto C(\vec{r}) \downarrow_{i}', \mathcal{E}') &= \mathcal{E}''
\end{align*}
\]

R-ASSIGN

\[
\mathcal{E} \vdash r.f_i := r' \quad \xrightarrow{\text{wr} \ rr'} \quad \mathcal{E}'' \vdash r'
\]

\[
\begin{align*}
\text{read}(r, \mathcal{E}) &= \mathcal{E}', \ C(\vec{r}) \\
\text{mbody}(m, C) &= (\vec{x}, e)
\end{align*}
\]

R-INVoke

\[
\mathcal{E} \vdash r.m(\vec{r}) \quad \xrightarrow{\text{rd} \ r} \quad \mathcal{E}' \vdash e[\vec{r}/\vec{x}][r/\text{this}]
\]

\[
\begin{align*}
r \ \text{fresh} \\
\text{extend}(r \mapsto C(\text{null}), \mathcal{E}) &= \mathcal{E}'
\end{align*}
\]

R-NEW

\[
\mathcal{E} \vdash \text{new} \ C() \quad \xrightarrow{\text{xt} \ r} \quad \mathcal{E}' \vdash r
\]
Global steps

- behavior of **multiple** interacting threads
  \[ n_1\langle e_1 \rangle \parallel \ldots \parallel n_k\langle e_k \rangle = P \]

- **global** state/configuration
  \[ \Gamma \vdash P \]
  = program \( P \) + **global environment** \( \Gamma \) = local environment per thread:
  \[ n_1:\mathcal{E}_1, \ldots n_k:\mathcal{E}_k \vdash n_1\langle e_1 \rangle \ldots n_k\langle e_k \rangle \]

- transitions
  \[ \Gamma \vdash P \xrightarrow{\alpha} n\Gamma' \vdash P' \]
Global steps: rules (1)

\[
P = P'' \parallel n\langle e \rangle \quad \mathcal{E} \vdash e \xrightarrow{\alpha} \mathcal{E}' \vdash e' \quad P' = P'' \parallel n\langle e' \rangle
\]

\[
\text{reflect}(n, \mathcal{E}', \Gamma) = \Gamma'
\]

\[
\Gamma \vdash P \xrightarrow{\alpha} n\Gamma' \vdash P'
\]

\[
P = P'' \parallel n\langle E[\text{spawn } e] \rangle \quad P' = P'' \parallel n\langle E[\text{null}] \rangle \parallel n'\langle e' \rangle
\]

\[
n' \text{ fresh} \quad \text{spawn}(n, \mathcal{E}', \Gamma) = \Gamma'
\]

\[
\Gamma \vdash P \xrightarrow{\text{sp } n'} n\Gamma' \vdash P'
\]

\[
P = P' \parallel n\langle r \rangle \quad \Gamma = n:\mathcal{E}, \Gamma'
\]

\[
\Gamma \vdash P \xrightarrow{k_i} n\Gamma' \vdash P''
\]

**G-PLAIN**

**G-SPAN**

**G-THKILL**
Global steps: transaction handling

- **start** a transaction:
  - basically straightforward
  - create a new transaction label

- finish a transaction (commit)
  - “publish” the result
  - slightly more complex, because of multi-threaded transactions
  - join all threads that are about to commit the transaction in question

- transaction in question: the “innermost” meant by the commit-action
Global steps: transaction rules (2)

\[
P = P'' \parallel n\langle E[onacid]\rangle \quad P' = P'' \parallel n\langle E[null]\rangle
\]

\[
\text{fresh} \quad \text{start}(l, n, \Gamma) = \Gamma'
\]

\[
\Gamma \vdash P \xrightarrow{ac} n\Gamma' \vdash P'
\]

\[
P = P'' \parallel n\langle E[commit]\rangle \quad P' = P'' \parallel n\langle E[null]\rangle
\]

\[
\Gamma = \Gamma'', n: E = E', l: \emptyset \quad \text{intranse}(l, \Gamma) = \tilde{n} = n_1 \ldots n_k
\]

\[
\text{commit}(\tilde{n}, \tilde{E}, \Gamma) = \Gamma' \quad n_1:E_1, n_2:E_2, \ldots , n_k:E_k \in \Gamma \quad \tilde{E} = E_1, E_2, \ldots , E_k
\]

\[
\Gamma \vdash P \xrightarrow{co} n\Gamma' \vdash P'
\]
Versioning semantics

- so far: the **core** has been **left abstract**
- one **concretization** of the general semantics
- concretization of the **memory manipulations**
- local environment \( \mathcal{E} \)

\[
l_1: \varrho_1, \ldots, l_k: \varrho_k
\]

- \( l \): transaction label
- \( \varrho \):
  - \( \log \) (of that transaction/of the given thread)
  - (part of the) dynamic context of the transaction \( l \)
- \( \mathcal{E} \) is **ordered**, ,
- current enclosing one: on the **right**
- reflects the **nesting** of transactions
Environment manipulations (local)

remember the **local steps**, for one thread
\[ E \vdash r \rightarrow E' \vdash r' \]

**read:** given a reference \( r \), find the assoc. value
- **look-up** the value for \( r \), not necessary in the innermost (= rightmost) transaction
- **log** the found value for the innermost transaction, i.e., **copy/record** it into that transactions log

**write:** similarly, the old value is logged locally, too

**extend:** similarly, **no old** value is logged (fresh reference)
Environment manipulation (local)

\[
\begin{align*}
\mathcal{E} &= \mathcal{E}', \quad l: \varnothing \quad \text{findlast}(r, \mathcal{E}) = C(\vec{r}) \quad \mathcal{E}'' &= \mathcal{E}', \quad l: (\varnothing, r \mapsto C(\vec{r})) \\
\text{read}(r, \mathcal{E}) &= \mathcal{E}'', \ C(\vec{r}) \\
\mathcal{E} &= \mathcal{E}', \quad l: \varnothing \quad \text{findlast}(r, \mathcal{E}) = D(\vec{r}') \quad \mathcal{E}'' &= \mathcal{E}', \quad l: (\varnothing, r \mapsto D(\vec{r}'), \ r \mapsto C(\vec{r})) \\
\text{write}(r \mapsto C(\vec{r}), \mathcal{E}) &= \mathcal{E}'' \\
\mathcal{E} &= \mathcal{E}', \quad l: \varnothing \\
\mathcal{E}'' &= \mathcal{E}', \quad l: (\varnothing, r \mapsto C(\vec{r})) \\
\text{extend}(r \mapsto C(\vec{r}), \mathcal{E}) &= \mathcal{E}'' \\
\Gamma &= n: \mathcal{E}, \quad \Gamma' \\
\Gamma'' &= n': \mathcal{E}', \quad \Gamma \\
\text{spawn}(n, n', \Gamma) &= \Gamma''
\end{align*}
\]
Environment manipulation: for transactions

- 2 operations: **start** and **commit**

  **start:**
  - easy (“optimistic”)
  - create a new [label] for the transaction
  - start with an [empty log] for the new transaction

  **commit:**
  - more tricky.
  - [propagate] (“reflect”) bindings from the transaction to the parent
  - commit only, if no [conflict] is detected
  - [conflict]: values used (r/w) in / must coincide with values as in parent transaction
Environment manipulation: transactions

\[
\begin{align*}
\Gamma &= n:E, \Gamma' \quad \Gamma'' = (l: (E, l: \langle \rangle)), \Gamma \\
\text{E-START} & \quad \text{start}(l, n, \Gamma) = \Gamma''
\end{align*}
\]

\[
\begin{align*}
\text{E-COMMIT}_1 & \quad \text{commit}(\langle \rangle, \langle \rangle, \Gamma) = \Gamma \\
\text{E-COMMIT}_2 & \quad \text{commit}(n, \vec{n}, \vec{E}, \Gamma) = \Gamma''
\end{align*}
\]

\[
\begin{align*}
\mathcal{E} &= \mathcal{E}', l:\varrho & \quad \text{readset}(\varrho, \langle \rangle) = \varrho' & \quad \text{writeset}(\varrho, \langle \rangle) = \varrho'' \\
\text{check}(\varrho', \mathcal{E}') & \quad \mathcal{E}' = \mathcal{E}'', l':\varrho''' & \quad \text{reflect}(n, (\mathcal{E}'', l':\varrho''', \varrho'''), \Gamma) = \Gamma' \\
& \quad \text{commit}(\vec{n}, \vec{\mathcal{E}}, \Gamma') = \Gamma'' \\
\text{commit}(n \vec{n}, \mathcal{E}, \vec{\mathcal{E}}, \Gamma) &= \Gamma''
\end{align*}
\]
Checking an environment
Modsets
Modsets

\[
\text{readset}(\langle \rangle, \_ ) = \langle \rangle
\]

\[
\varrho = u \mapsto C(\vec{u}) \quad u \notin \vec{r} \quad \text{readset}(\varrho'', \vec{r}u) = \varrho'
\]

\[
\text{readset}(\varrho, \vec{r}) = u \mapsto C(\vec{u}), \varrho'
\]

\[
\varrho = u \mapsto C(\vec{u}), \varrho'' \quad u \in \vec{r} \quad \text{readset}(\varrho'', \vec{r}) = \varrho'
\]

\[
\text{readset}(\varrho, \vec{r}) = \varrho'
\]

\[
\text{writeset}(\langle \rangle, \_ ) = \langle \rangle
\]

\[
\varrho? r \mapsto C(\vec{r}), \varrho'' \quad \text{writeset}(\varrho'', \varrho') = \varrho'''' \quad r \mapsto C(\vec{r}) \neq \text{first}(r, \varrho')
\]

\[
\text{writeset}(\varrho, \varrho') = u \mapsto D(\vec{u}), \varrho''''
\]
Two-phase locking

- different instantiation of the general semantics, slight alteration
- based on **locks**
- **strict**
- **pessimistic**
- two **phases**:
  1. first get hold of all the locks needed for a transaction
  2. then release them again
- **strict**: all acquiring is done **before** all releasing.
Two-phase locking transactional semantics

- “slight” alteration of the previous one
- transaction & locks
  - objects have locks for protection
  - locks are held by transactions.\(^4\)
  - enter a transaction: all locks held by transaction or prefix
  - creating an object.
- to support locking
  - unique transaction label \(l_L +\)
  - lock environment \(\varrho_L\).
- \(\varrho\) stores lock ownership (per reference): which transactions hold the lock = sequence to reflected nesting
- given \(l_1, l_2, \ldots, l_k\)
- change of lock-ownership:
  - acquire by grabbing
  - commit by child, and propagate the lock upwards

\(^4\)Note the difference to multi-threaded Java
Environment manipulation with locks (local)

\[ \mathcal{E} = \mathcal{E}', l: \varnothing \quad \text{findlast}(r, \mathcal{E}) = C(\vec{r}) \]
\[ \mathcal{E}'' = \mathcal{E}', l: (\varnothing, r \mapsto C(\vec{r})) \quad \text{checklock}(r, \mathcal{E}) = \top \]
\[ \text{read}(r, \mathcal{E}) = \mathcal{E}''', C(\vec{r}) \]  
\hline
\[ \text{E-READ} \]

\[ \text{findlast}(r, \mathcal{E}) = D(\vec{r}') \quad \mathcal{E}' = \text{acquirelock}(r, \mathcal{E}) \]
\[ \mathcal{E}' = \mathcal{E}'', l: \varnothing \quad \mathcal{E}''' = \mathcal{E}'', l: (\varnothing, r \mapsto D(\vec{r}'), r \mapsto C(\vec{r})) \]
\[ \text{write}(r \mapsto C(\vec{r}), \mathcal{E}) = \mathcal{E}''' \]  
\hline
\[ \text{E-WRITE} \]

\[ \text{acquirelock}(r, \mathcal{E}) = \mathcal{E}', l: \varnothing \quad \mathcal{E}'' = \mathcal{E}', l: (\varnothing, r \mapsto C(\vec{r})) \]
\[ \text{extend}(r \mapsto C(\vec{r}), \mathcal{E}) = \mathcal{E}'' \]  
\hline
\[ \text{E-EXTEND} \]
Environment manipulation: transactions

\[ \Gamma = n: \mathcal{E}, \Gamma' \quad \Gamma'' = (l:(\mathcal{E}, l:\langle\rangle)), \Gamma \]

\[ \text{E-START} \]

\[ \text{start}(l, n, \Gamma) = \Gamma'' \]

\[ \text{commit}(\langle\rangle, \langle\rangle, \Gamma) = \Gamma \]

\[ \text{E-COMMIT}_1 \]

\[ \mathcal{E} = l_L: q_L, \mathcal{E}' \quad q'_L = \text{release}(l(\mathcal{E}), q_L) \quad \mathcal{E}'' = l_L: q'_L, \mathcal{E}' \]

\[ \text{reflect}(n, (\mathcal{E}'', l': q''', q'''), \Gamma) = \Gamma' \quad \text{commit}(\vec{n}, \vec{\mathcal{E}}, \Gamma') = \Gamma'' \]

\[ \text{E-COMMIT}_2 \]

\[ \text{commit}(n \vec{n}, \mathcal{E} \vec{\mathcal{E}}, \Gamma) = \Gamma'' \]
Further development in the paper

• After the formalization: prove some “soundness results”
  • ultimately: “ACID”, serialization
  • techniques: “permutation lemmas”
Introduction

Transactional Java
  Operational semantics without transactions
  Transactional semantics
  Versioning semantics
  Two-phase locking

Conclusion
Further reading

- **wait-free** data structures
- old, related theoretical results: [Lipton, 1975]: theory of left/right movers
- [Herlihy and Wing, 1990]: linearizability for concurrent objects
- **futures** [Welc et al., 2005]
- transactions for Java [Garthwaite and Nettles, 1996]
- **software transactional memory** [Shavit and Toitu, 1995]
- **automatic mutual exclusion** [Abadi et al., 2008] and originally [Isard and Birell, 2007]
- and another POPL’08 paper?
References I

Semantics of transactional memory and automatic mutual exclusion.
In Proceedings of POPL ’08. ACM.

Transactions for Java.

Linearizability: A Correctness Condition for Concurrent Objects.
ACM Transactions on Programming Languages and Systems, 12(3):463–492.

Automatic mutual exclusion.
In Proceedings of the 11th Workshop on Hot Topics in Operating Systems.

A transactional object calculus.

In Second Annual Symposium on Principles of Programming Languages (POPL) (Palo Alto, CA), pages 78–86. ACM.

Software transactional memory.

Safe futures in Java.
In Twentieth Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) ’05, pages 439 – 453. ACM.
In SIGPLAN Notices.