Anticipation of Method Execution in Mixed Consistency Systems

Marco Giunti, Hervé Paulino, António Ravara

NOVA School of Science and Technology, Portugal

ACM/SIGAPP Symposium On Applied Computing
March 30 2023

Weak Consistency

- Citing Burckhardt (Foundations and Trends in Programming Languages 2014):
- In globally distributed systems, shared state is never perfect.
- When communication is neither fast nor reliable, we cannot achieve at the same time
  1. strong consistency
  2. low latency
  3. availability
- *Weak consistency* to overcome these limitations in *replicated systems*

Commutative concurrent operations

- We are interested in analysing concurrent operations in replicated systems
- Standard consistency prerequisite: operations must *commute*
- Focus on *language level*: Object-Oriented Programming (*OOP*)
- Rephrasing, our goal is:

identify commutative method calls and gather information on calls that in distributed runtime featuring replicas can be anticipated

Example: executing calls concurrently in two Sites

- Locally permissible ops immediately executed
- Strongly consistent ops require coordination among replicated sites

- Site 1

```
acc1.deposit(20);
acc2.withdraw(5);
acc2.deposit(10);
acc3.deposit(20);
acc2.getBalance();
```

- Site 2

```
acc3.getBalance();
acc1.deposit(20);
acc2.withdraw(5);
acc2.deposit(10);
acc3.deposit(20);
acc2.getBalance();
```

Example: executing calls concurrently in two Sites

- Locally permissibile ops immediately executed
- Strongly consistent ops require coordination among replicated sites

- Site 1

```
acc2.withdraw(5);
acc2.deposit(10);
acc3.deposit(20);
acc2.getBalance();
```

- Site 2

```
acc1.deposit(20);
acc2.withdraw(5);
acc2.deposit(10);
acc3.deposit(20);
acc2.getBalance();
```

Example: executing calls in Site 1

- While `acc2.withdraw(5)` is put under coordination, Site 1 processes the next call
- Can we locally execute `acc2.deposit(10)`?
- Let's try

- Site 1

```
acc2.deposit(10);
acc3.deposit(20);
acc2.getBalance();
```

- Under coord.

```
acc2.withdraw(5);
```

- Site 2

```
acc2.withdraw(5);
acc2.deposit(10);
acc3.deposit(20);
acc2.getBalance();
```

Example: executing calls in Site 2

- After a step the system might fatally diverge
- E.g. if balance before deposit was < 5 then Site 1 allows withdrawal while Site 2 disallows it
- Eventual consistency is broken

- Site 1

```
acc3.deposit(20);
acc2.getBalance();
```

- Under coord.

```
acc2.withdraw(5);
```

- Site 2

```
acc2.withdraw(5);
acc2.deposit(10);
acc3.deposit(20);
acc2.getBalance();
```

Example: executing calls in Site 1

- Therefore in previous step deposit is put on hold as well
- Differently, the next call can be executed since it operates on a different account
- That is, the call can be "anticipated" w.r.t. operations received before

- Site 1

```
acc3.deposit(20)
acc2.getBalance();
```

- Under coord.

```
acc2.withdraw(5);
acc2.deposit(10);
```

- Site 2

```
acc2.withdraw(5);
acc2.deposit(10);
acc3.deposit(20);
acc2.getBalance();
```

Related work

- Logical-based specifications (e.g. Bansal et al., JAR 2020)

```
name: set
preamble: (declare-sort E 0)
state: -name: S -type: (Set E) -name: size -type: Int
methods:
  name: add
  args: -name: v -type: E
  return: - name: result -type: Bool
  requires: true
  ensures:
    (ite (member v S)
         (and (= S_new S)
              (= size_new size)
              (not result))
         (and (= S_new (union S (singleton v)))
              (= size_new (+ size 1)) result))
```

Related work

- Abstract languages (e.g. Houshmand & Lesani, POPL 2019)

```
Class Courseware
let Student = Set <sid:SId> in
let Course = Set <cid:CId> in
let Enrolment = Set <esid : SId, ecid : CId> In
Σ = Student × Course × Enrolment
I = λ <ss,cs,es>.
      refIntegrity(es, esid, ss, sid) ∧
      refIntegrity(es, ecid, cs, cid)
addCourse(c) = λ <ss, cs, es>.
                 <⊤, <ss, cs ∪ c, es>, ⊥>
deleteCourse(c) = λ <ss, cs, es>.
                 <⊤, <ss, cs \ c, es>, ⊥>
...
```

Challenge

- Can we process source code at *compile-time* to gather information on calls to anticipate?
- Idea: code analysis decides that deposit cannot anticipate withdraw on the *same instance*

```
class Account {
balance : int
...

def withdraw(amount : int) : Unit {
 this.balance -= amount
}

def deposit(amount : int) : Unit {
 this.balance += amount
}

...
}
```

Purpose

- How we will use the information on anticipation generated at compile-time?
- Idea: populate a *table* that can be accessed fast by the runtime system
- Scheduler checks the table in order to anticipate `acc2.deposit(10)` w.r.t. `acc2.withdraw(5)`
- Anticipation is forbidden
- Instead, `acc3.deposit(20)` can anticipate the previous calls `acc2.withdraw(5)` and `acc2.deposit(10)`

```
/* WITHOUT TABLE */
acc2.withdraw(5);
acc2.deposit(10);
acc3.deposit(20;)
acc2.getBalance();
```

```
/* VIRTUAL VIEW WITH TABLE */
acc3.deposit(20);
  acc2.withdraw(5);  /* ON HOLD */
  acc2.deposit(10);  /* ON HOLD */
acc2.getBalance();
```

Methodology: language-level consistency requirements

- We need to provide the consistency semantics
- Idea: *single keyword* to allow weak consistency of *fields*
- Remaining fields have strong consistency
- Design choice: field decoration requires less effort and fragmentation than assigning consistency to operations

```
class Account {
balance   : int weak
...

def withdraw(amount : int) : Unit  {
 this.balance -= amount
}

def deposit(amount : int) : Unit {
 this.balance += amount
}
```

State invariant

- We need to provide state invariant to infer the permissibility of ops
- Idea: field invariants and method preconditions

```
class Account {
balance  : int weak [this.balance≥0]
...

def withdraw(amount : int) : Unit [?] {
 this.balance -= amount
}

def deposit(amount : int) : Unit [amount>0]{
 this.balance += amount
}

...
}
```

Method's preconditions

- We cannot rely on the value of the weak field balance
- Withdraw has no preconditions

```
class Account {
balance : int weak [this.balance≥0]
...

def withdraw(amount : int) : Unit {
 this.balance −= amount
}

def deposit(amount : int) : Unit [amount>0]{
 this.balance += amount
}

...
}
```

OOP core language

- We stand on the tradition of languages with formal semantics
- We consider a variant of OOlong (Castegren&Wrigstad, ACM SAC 2018)

$$
\begin{array}{llll}
d & ::= & x \mid x.f \mid v & \textit{(Invariant values)} \\
c & ::= & d_1 \ \mathit{Rel} \ d_2 & \textit{(Invariants)} \\
\mathit{Md} & ::= & m(x : t_1) : t_2 \ [\tilde{c}] \ \{e\} & \textit{(Methods)} \\
\mathit{Fd} & ::= & f : t \sim\!\mathsf{weak} \ [\tilde{c}] & \textit{(Fields)} \\
e & ::= & sv \mid x.f \mid x.f = e \mid x.m(e) \mid & \\
 & & \mathsf{let}\ x = e_1 \ \mathsf{in} \ e_2 \mid \mathsf{new}\ C \mid (t)\ e & \textit{(Expressions)} \\
sv & ::= & v \mid x \mid sv_1 \ \mathit{Op} \ sv_2 & \textit{(Symbolic Values)}
\end{array}
$$

Program reductions

- Standard Heap (H)/Stack (V) single-thread semantics

$$\frac{V(x) = \iota \quad H(\iota) = (Account, balance \mapsto n) \quad \text{this}', y' \text{ fresh} \qquad e = \text{let } z\text{=this}'.balance \text{ in this}'.balance\text{=}z + y'}{\langle H, V, \_, \_, x.deposit(10)\rangle \hookrightarrow \langle H, V[\text{this}' \mapsto \iota, y' \mapsto 10], \_, \_, e\rangle}$$

- What about methods' preconditions?

$$\frac{}{\langle H, V, \_, \epsilon, x.deposit(10)\rangle \hookrightarrow \langle H, V', \_, 10 > 0, e\rangle}$$

- A state Σ is composed by the four entries above

$$\langle \Sigma, x.deposit(10)\rangle \hookrightarrow \langle \Sigma', e\rangle$$

Formal semantics of permissible operations

- We leverage the Hamsaz model of conflicts (Houshmand & Lesani, POPL 2019)
- State invariant holds iff $f : t \sim$weak $[\tilde{c}] \in \Sigma$ and $c \in \tilde{c}$ implies instantiation of c evals to true
- A call is *guarded* if post-state satisfies constraint, e.g. $10 > 0$
- A call is *permissible* in pre-state if pre-state invariant implies post-state invariant
- A method is *locally permissible (LP)* if all guarded calls are permissible

Method calls under coordination

- Calls of non-LP methods require coordination
- E.g. state invariant of *balance* requires non-negativity

```
class Account {
balance   : int weak [this.balance≥0]
...
}
```

- The call `x.withdraw(5)` can break state invariant
- E.g. if $V(x) = \iota$ and $H(\iota) = (Account, balance \mapsto 3)$
- Therefore, *withdraw* is non-LP

Formal semantics of commutative ops

- Commutative calls defined as expected
- E.g. $x_1.deposit(v_1)$ and $x_2.withdraw(v_2)$ commute in Σ iff

1. Sequence $x_1.deposit(v_1); x_2.withdraw(v_2)$ gives rise to Σ_1
2. Sequence $x_2.withdraw(v_2); x_1.deposit(v_1)$ gives rise to Σ_2
3. $\Sigma_1 = \Sigma_2$

Call anticipation algorithm

- Novel notion relying on *weak fields integer generalization*
- Quantification over all possible integer values

```
let anticipation (Σ : state) (mc1 mc2 : call) : bool =
if commute Σ mc1 mc2 then
  let m1 = methodOf mc1 in let m2 = methodOf mc2 in
  match LP Σ m2 ∨ ∀ n1,..., nm. Permissible (WIFG Σ (n1,...,nm)) mc2 with
  | true -> (* mc1 LP in post−state for all possible weak ints *)
            (* or mc1 preserves permissibility *) ···
  | false -> false
else false
```

- Limitation: algorithm is non-effective for runtime use
- Overhead of invoking constraint solver on all post-states unbearable

Aim of static analysis: parametric call anticipation

- Establishing call anticipation as yes or no is too restrictive
- We need to generate parameters for anticipations

```
def interest(interest : int) : Unit {
    this.balance += this.balance * interest / 100
}
...
x1.method(n);
x2.interest(i);
```

	getBalance	deposit	withdraw	interest
interest	✓	$x_1 \neq x_2 \wedge$ $i \geq -100$	$x_1 \neq x_2 \wedge$ $i \geq -100$	$x_1 \neq x_2 \wedge$ $i \geq -100$

Symbolic memory

- The static analysis is built on top of *symbolic values*

  $sv \quad ::= \quad v \mid x \mid sv_1 \; Op \; sv_2$
- Symbolic semantics : transitions with open terms to establish commutativity
- We consider transitions of methods, rather than method calls
- Example: *deposit* and *withdraw* commute?
- Symbolic heap: $balance \mapsto x$, where x is fresh
- Technique: execute sequences *deposit*; *withdraw* and *withdraw*; *deposit* and produce *heap equations*

Method commutativity, statically

- Execution *deposit*; *withdraw*, same “instance”
  1. Symbolic execution of *deposit*, d is formal parameter, leads to heap: $balance \mapsto x + d$
  2. Symb. execution of *withdraw*, w is formal parameter, leads to $balance \mapsto (x + d) - w$
- Execution *withdraw*; *deposit*; same “instance”
  1. Symb. execution of *withdraw* leads to $balance \mapsto x - w$
  2. Symbolic execution of *deposit* leads to heap: $balance \mapsto (x - w) + d$
- Algorithm produces equation $((x + d) - w, (x - w) + d)$
- Equation is SAT for all values $x \geq 0$, $d > 0$ (inferred from state invariant)
- In general, symbolic values in fields and methods parametrize commutativity

Method anticipation, statically

- Our objective is populate table with parameters that allow call anticipation
- Use: fast access by runtime system to take decision
- Methodology: generate a *list of logical conjunctions*
- We distinguish equality case: e.g. $\text{this}_1 = \text{this}_2$ and $\text{other}_1 \neq \text{other}_2$
- Init state populated with init objects with symbolic integers

```
let ant (h :eqCase) (c1 c2 : classDef) (md1 md2 : methodDef) : prop list =
 let Σ = gen h md1 md2 in
 let eqs, cc = scommute h md1 md2 in
 let eff2 = hasEffect c2 md2 in
 let <Σ'',_,_> = update_s Σ eff2 null in
 eqs :: cc :: sLP Σ md2 ::
   (sLP Σ'' md1 ∨
    ∀x in (weak_inv Σ).
     ((sP Σ md1 => sP Σ'' md1) ∧
     (sNP Σ md1 => sNP Σ'' md1))) :: [SFNI h md1 md2]
```

Example: anticipating LP method

- Formula:

```
eqs :: cc :: sLP Σ md2 ::
  (_ ∨
   ∀x in (weak_inv Σ).
    ((sP Σ md1 => sP Σ'' md1) ∧
    (sNP Σ md1 => sNP Σ'' md1))) :: [SFNI h md1 md2]
```

- Consider positive example that relies on right disjunction

```
persons  int  weak  [this.persons  mod 2 = 0]
tables   int  weak  [this.tables ≥ 0]

/* commutative methods */
def addTable(){ this.tables += 1 } /* LP */
def addPerson () { this.persons += 1} /* non−LP */
```

Anticipation of `addTable` w.r.t. `addPerson` ($\text{this}_1 = x = \text{this}_2$)

$$\Sigma = x \mapsto (\_, pers \mapsto p, tbl \mapsto t) \qquad \Sigma' = x \mapsto (\_, pers \mapsto p+1, tbl \mapsto t)$$
$$\Sigma'' = x \mapsto (\_, pers \mapsto p, tbl \mapsto t+1) \qquad \mathrm{inv}(\Sigma) = p \bmod 2 = 0 \wedge t \geq 0$$
$$\mathrm{inv}(\Sigma') = p + 1 \bmod 2 = 0 \wedge t \geq 0 \qquad \mathrm{inv}(\Sigma'') = p \bmod 2 = 0 \wedge t + 1 \geq 0$$

- Simplified formula:

  ```
  eqs :: cc :: sLP Σ addT ::
  ∀p,t.((sP Σ addP => sP Σ'' addP) ∧ (sNP Σ addP => sNP Σ'' addP))
  ```

- We have the following subformulas

$$eqs = (p+1, p+1), (t+1, t+1) \qquad cc = \text{true}, \text{true}$$
$$sLP\ \Sigma\ addT = \forall p, t.\, \mathrm{inv}(\Sigma) \Rightarrow \mathrm{inv}(\Sigma'') \qquad sP\ \Sigma\ addP = \mathrm{inv}(\Sigma) \Rightarrow \mathrm{inv}(\Sigma'$$
$$sP\ \Sigma''\ addP = \mathrm{inv}(\Sigma'') \Rightarrow p + 1 \bmod 2 = 0 \wedge t + 1 \geq 0$$
$$snP\ \Sigma\ addP = \mathrm{inv}(\Sigma) \Rightarrow \neg\mathrm{inv}(\Sigma')$$
$$snP\ \Sigma''\ addP = \mathrm{inv}(\Sigma'') \Rightarrow \neg(p + 1 \bmod 2 = 0 \wedge t + 1 \geq 0)$$

Anticipation of `addTable` w.r.t. `addPerson` ($\text{this}_1 = x = \text{this}_2$)

$$\Sigma = x \mapsto (\_, pers \mapsto p, tbl \mapsto t) \qquad \Sigma' = x \mapsto (\_, pers \mapsto p+1, tbl \mapsto t)$$

$$\Sigma'' = x \mapsto (\_, pers \mapsto p, tbl \mapsto t+1) \qquad \text{inv}(\Sigma) = p \bmod 2 = 0 \wedge t \geq 0$$

$$\text{inv}(\Sigma') = p+1 \bmod 2 = 0 \wedge t \geq 0 \qquad \text{inv}(\Sigma'') = p \bmod 2 = 0 \wedge t+1 \geq 0$$

- Simplified formula:

```
eqs :: cc :: sLP Σ addT ::
∀p, t.((false => sP Σ'' addP) ∧ (true => sNP Σ'' addP))
```

- We have the following subformulas

$$eqs = (p+1, p+1), (t+1, t+1) \qquad cc = \text{true}, \text{true}$$

$$sLP\ \Sigma\ addT = \forall p, t.\, \text{inv}(\Sigma) \Rightarrow \text{inv}(\Sigma'') \qquad sP\ \Sigma\ addP = \text{false}$$

$$sP\ \Sigma''\ addP = \_$$

$$snP\ \Sigma\ addP = \text{true}$$

$$snP\ \Sigma''\ addP = \text{inv}(\Sigma'') \Rightarrow \neg(p+1 \bmod 2 = 0 \wedge t+1 \geq 0)$$

Formula is SAT! `addTable` is anticipated

Use cases

Results obtained with Java prototype (Thank Rúben Vaz)

Use-case	# weak/ # strong	# invs	# mtds	# non-LP	# pairs	# conflicts
Account	1/1	2	6	3	21	5
Auction	1/1	2	4	0	9	3
Counter	1/0	0	3	0	6	0
Register	3/0	0	2	0	3	0

Use cases

$m_2 \xrightarrow{c} m_1$ indicates that, given sequence $m_1; m_2$, method m_2 can anticipate m_1 when c holds

Use-case	# anticipations
Account	$\mathtt{a} \xrightarrow{i \geq -100} \mathtt{g}$, $\mathtt{a} \xrightarrow{\mathsf{this}_1 \dot{\neq} \mathsf{this}_2 \wedge i_2 \geq -100} \{\mathtt{d}, \mathtt{t}, \mathtt{w}, \mathtt{a}\}$, $\mathtt{g} \to *$, $\mathtt{d} \to \{\mathtt{d}, \mathtt{g}\}$, $\mathtt{d} \xrightarrow{\mathsf{this}_1 \dot{\neq} \mathsf{this}_2} \{\mathtt{t}, \mathtt{w}, \mathtt{a}\}$
Auction	$\mathtt{b} \to \mathtt{cb}$, $\mathtt{cb} \to *$, $\mathtt{c} \to \mathtt{c}$, $\mathtt{b} \to \mathtt{b}$
Counter	$* \to *$
Register	$\mathtt{g} \to *$, $* \to \mathtt{g}$, $\mathtt{s} \to \mathtt{s}$,

Thanks!

- Evaluation with Java prototype
  1. querying the table < 0.001 *ms*
  2. several orders of magnitude lower than performing global synchronization
- Main limitation: consistency only supported for primitive types
- Directions for future work
  - Extend the language, e.g. conditionals, loops...
  - Mechanization of proof of soundness, e.g. in the Coq proof assistant