Static Analysis of Concurrent Programs

Mooly Sagiv
Why Concurrent Programs

- Partial execution order
- More efficient implementations
  - Multicore machines
- Distributed environments
Challenges in Concurrent Programs

• Hard to define the semantics
• Hard to debug
  – Assertion checking /partial does not suffice
  – Many potential executions
  – Testing is very hard
  – Reproducing bugs is hard
• Hard to verify
Java Concurrency

• JVM supports **threads** of execution
  – Many threads can be executed “simultaneously”
  – Communicate via shared memory
  – **Synchronize** activities using **locks**
  – Both threads and locks are objects

• **Standard methods** for efficient transfer of control
  – `wait`
  – `notify`
  – `...`

• Special rules about allowed orders of executions
Java Memory Model

Thread engine

assign

use

working memory

unlock

store

write

unlock

Main memory

Thread engine

assign

use

working memory

unlock

store

load

read

unlock
Java code fragment

```
[p.x→0, p.y→0]

synchronized(p) {
    p.y = 2;
}
a = p.x;
b = p.y;
c = p.y;
```

```
synchronized(p) {
    p.y = 3;
    p.y = 100;
}
p.x = 1;
```
Correctness Criteria

- Partial correctness (assertion checking)
  - No runtime errors
  - No memory leaks
- Absence of deadlocks
- Termination
- Absence of data races
- Linearizability
Deadlocks

• Happens when two or more competing threads are waiting for the other to finish, and thus neither ever does

  lock(l1);
  lock(l2);
  ...
  unlock(l2);
  unlock(l1);
  ...
  lock(l1);
  lock(l2);
  ...
  unlock(l1);
  unlock(l2);
Dataraces

- Occurs when two threads access the same resource
  - At least one of the accesses are write
  - No mechanism to prevent order conflicts
A Simple Example

\[ x := y; \quad y := y + 1; \]
Java code fragment

[p.x→0, p.y→0]

synchronized(p) {
  p.y = 2;
}

a = p.x;
b = p.y;
c = p.y;

synchronized(p) {
  p.y = 3;
  p.y = 100;
}

p.x = 1;
No Datarace

lock(l);

v := v + 1;

unlock(l);

lock(l);

v := v + 1;

unlock(l);
Concurrency and Alias Analysis

Different locks

sync(e_1) { }
e2.f = }

sync(e_3) { }
... = e4.f }

Same Object
Programs and Properties

- Concurrent programs
- Unbounded number of threads
  - parametric systems
- Unbounded number of objects
- Pointers and destructive updates

- Memory safety
  - Absence of null dereferences
  - Absence of memory leaks
- Preservation of data structure invariants
- Linearizability
- User-specified invariants
A Generic Framework for Verifying Concurrent Programs

Eran Yahav

POPL’01
Main Ideas

• An interleaving semantics for Java
  – A program configuration encodes
    • global store
    • program-location of every thread
    • status of locks and threads

• First-order logical structures used to represent program configurations

• Use TVLA to obtain a static analyzer

• Works well for tiny programs
Configurations

• Predicates model properties of interest
  – $\text{eq}(v_1, v_2)$
  – $\text{is}_T(v)$
  – $\{ \text{at}[\text{lab}](t) : \text{lab} \in \text{Labels} \}$
  – $\{ \text{rv}[\text{fld}](o_1, o_2) : \text{fld} \in \text{Fields} \}$
  – $\text{heldBy}(l, t), \text{blocked}(t, l), \text{waiting}(t, l)$

• Can use the framework with different predicates
Structural Operational Semantics

• An action consists of:
  – precondition formula
  – update formulae

• Precondition formula may use a free variable $t_s$ for “currently scheduled” thread

• Semantics is non-deterministic
### Structural Operational Semantics - actions

| lock(v) |  
|---|---|
| **precondition** | $\neg \exists t \neq t_s: \text{rval}[v](t_s,l) \land \text{held_by}(l,t)$ |
| **predicate update** | $\text{held_by}'(l_1,t_1) = \text{held_by}(l_1,t_1) \lor (l_1 = l \land t_1 = t_s)$  
| | $\text{blocked}'(t_1,l_1) = \text{blocked}(t_1,l_1) \land ((l_1 \neq l) \lor (t_1 \neq t_s))$ |
Safety Properties

• Configuration-local property as logical formula

• Example: mutual exclusion

\[ \forall t_1, t_2: (t_1 \neq t_2) \rightarrow \neg(at[l_{crit}](t_1) \land at[l_{crit}](t_2)) \]

• Example: no total deadlock

\[ \exists t, \forall l_b: is\_thread(t) \land \neg blocked(t, l_b) \]
l_0: while (true) {
l_1:       synchronized(myLock) {
l_C:           // critical actions
l_2:               } 
l_3: }
Concrete Configuration

at[l_1] held_by rval[myLock]

at[l_C]

at[l_1]

rval[myLock]

at[l_0]

at[l_0]

rval[myLock]

blocked
Abstract Configuration

- `at[l_1]`
- `rval[myLock]` to `blocked`
- `held_by`
- `at[l_C]` to `rval[myLock]`
- `at[l_0]`
Abstract Interpretation

1_0: while (true) {
1_1:       synchronized(myLock) {
1_C:            // critical actions
1_2:       }
1_3:   }

rval[myLock]
at[l_0]
Abstract Interpretation

l_0: while (true) {
  l_1:     synchronized(myLock) {
    l_C:         // critical actions
    l_2:          }
  l_3:        }

l_0: while (true) {
l_1:    synchronized(myLock) {
l_C:        // critical actions
l_2:    }
l_3: }

Abstract Interpretation
Abstract Interpretation

l_0: while (true) {
l_1:     synchronized(myLock) {
l_C:         // critical actions
l_2:     }
l_3: }

at[l_0]
rval[myLock]
Abstract Interpretation

l_0: while (true) {
l_1:    synchronized(myLock) {
l_C:      // critical actions
l_2:    }
l_3: }

\[\text{Abstract Interpretation}\]

\[l_0: \text{while (true) \{}
l_1: \quad \text{synchronized(myLock) \{}
l_C: \quad \text{\quad \# critical actions}
l_2: \quad \text{\}}
l_3: \text{\}}\]
Abstract Interpretation

```java
l_0: while (true) {
  l_1:   synchronized(myLock) {
    l_C:       // critical actions
  }
  l_2: }
  l_3: }
```
<table>
<thead>
<tr>
<th>Program</th>
<th>Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>twoLock Q</td>
<td>No interference</td>
</tr>
<tr>
<td></td>
<td>No memory leaks</td>
</tr>
<tr>
<td></td>
<td>Partial correctness</td>
</tr>
<tr>
<td>Producer/consumer</td>
<td>No interference</td>
</tr>
<tr>
<td></td>
<td>No memory leaks</td>
</tr>
<tr>
<td>Apprentice Challenge</td>
<td>Counter increasing</td>
</tr>
<tr>
<td>Dining philosophers with resource ordering</td>
<td>Absence of deadlock</td>
</tr>
<tr>
<td>Mutex</td>
<td>Mutual exclusion</td>
</tr>
<tr>
<td>Web Server</td>
<td>No interference</td>
</tr>
</tbody>
</table>
Scalability

• The number of abstract configurations is doubly exponential in the program size
• Interleaving semantics quickly leads to state explosion
Techniques for Scalability

- Course abstractions
  - Refinement
- Program restrictions
- Limited semantics
- Specialization
- Unsoundness
Flow Insensitive Analysis

• Ignore control flow statements
• Obtain a sound solution w.r.t. all interleavings
• But can it be precise enough?
  – Initializations
  – Strong updates
  – Object sensitivity
  – Context sensitivity
Critical Section Example

l_0: while (true) {
l_1:     synchronized(myLock) {
l_C:         // critical actions
l_2:     }
l_3: }

Thread Modular Shape Analysis

Alexey Gotsman

PLDI’07
Is this a well-formed cyclic doubly-linked list?

```c
{  PIRQP irp;
  LIST_ENTRY listHead, *entry;
  KIRQL irql;

  InitializeListHead(&listHead);

  KeAcquireSpinLock(&DeviceExtension->SpinLock, &irql);
  
  do {
      irp = KeyboardClassDequeueReadIRP();
      if (irp) {
          irp->IoStatus.Status = STATUS_SUCCESS;
          irp->IoStatus.Information = 0;
          InsertTailList(&listHead, &irp->Tail.Overlay.ListEntry);
      }
  } while (irp != NULL);

  KeReleaseSpinLock(&DeviceExtension->SpinLock, irql);

  //
  // Complete these irps outside of the spin lock
  //
  while (!IsListEmpty(&listHead)) {
      entry = RemoveHeadList(&listHead);
      irp = CONTAINING_RECORD(entry, IRP, Tail.Overlay.ListEntry);
  }
```
Main Ideas

- Compile-time names for threads and locks
- Static association of the locks and parts of the heap
- Analyze the behavior of every thread separately with a resource invariant describing the state of others
  - Break the heap into two parts
- Overapproximate local and resource invariants
- Prove the absence of dataraces
A motivating example (unrealistic)

class List {
    public List n;
    public int d;
}
List g;

void producer() {
    List p;
    while (1) {
        ...
        create a list in p
        synchronized (g) {
            g = p;
            p = null;
        }
    }
}

void consumer() {
    List c;
    while (1) {
        synchronized (g) {
            c = g;
            g = null;
        }
        ...
        consume c;
    }
}
A motivating example (producer)

\[ g == \text{null} \]

- \textbf{while} \( p == \text{null} \)
- \textbf{Create A list} \( p \)
- \textbf{lock}(g);
- \( g == p; \)
- \textbf{unlock}(g);
- \( p == \text{null} \)
- \( g == \text{null} \land p == \text{list} \)
- \( p == \text{list} \)
- \( g == p \land p == \text{list} \)
- \( p == \text{null} \land g == \text{list} \)
A motivating example (producer)

\[ g = \text{null} \lor g = \text{list} \]

while \( p = \text{null} \)

Create A list \( p \)

lock(g);

\( g = p; \) \( g = \text{null} \land p = \text{list} \) \( g = p \land p = \text{list} \)

unlock(g);

\( p = \text{null} \) \( p = \text{null} \land g = \text{list} \) \( p = \text{null} \land g = \text{list} \) \( p = \text{null} \)
A motivating example (producer)

$$g \equiv \text{null} \lor g \equiv \text{list}$$

```
while (g == null || g == list) && p == list
{
    p = null
    g = p
    lock(g);
    p = null && g == list
    unlock(g);
}
```

Potential memory leak
A motivating example (consumer)

g == null
\lor

g == list

while

lock(g);

c == null
\land

(g == null \lor g == list)

c == g
\land

(g == null \lor g == list)

g = null
\land

(g == null \lor c == list)

unlock(g);

c == null \lor c == list

consume(c)
A Separation Domain

• \( \langle D, \sqsubseteq, \sqcup, \bot, \top, e, * \rangle \)
• \( \langle D, \sqsubseteq, \sqcup, \bot, \top \rangle \) is the usual lattice
• \( * : D \times D \rightarrow D \)
• \( \langle D, \sqsubseteq, e, * \rangle \) is a partially ordered commutative monoid
  – * is associative and commutative
  – * has the unit element \( e \)
  – * is monotone
Locality

- A separated domain \( \langle D, \sqsubseteq, \sqcup, \bot, \top, e, * \rangle \)
- A function \( f: D \rightarrow D \) is **local** if for all \( u, v \):
  \[
  f(u * v) \sqsubseteq f(u) * v
  \]
Relating Concrete and Abstract Separated Domains

- A separated concrete domain
  \[<C, \subseteq_C, \sqcup_C, \bot_C, \tau_C, e_C, \ast_C>\]
- A separated abstract domain
  \[<A, \subseteq_A, \sqcup_A, \bot_A, \tau_A, e_A, \ast_A>\]
- A concretization \(\gamma: A \rightarrow C\) such that
  - \(\gamma\) is monotone
  - Abstract transformers over-approximate concrete ones
  - \(\gamma\) is a homomorphism between abstract and concrete monoids
    - \(\forall a, b: \gamma(a \ast_A b) = \gamma(a) \ast_C \gamma(b)\)
Decomposition Based Shape Analysis

Roman Manevich
Thread-Modular Analysis

- Abstract away the correlations between local states of different threads
  - No correlations between program counters
  - Cartesian Abstraction

- Information maintained
  - Correlations between the local state and global state of each thread

- “The quadratic cost of computing transformers can be greatly reduced…”  
  [Flanagan & Qadeer SPIN, 2003]

- Naturally handles unbounded number of threads
A Singleton Buffer

Boolean empty = true;
Object b = null;

produce() {
1: Object p = new();
2: await (empty) then {
   b = p;
   empty = false;
}
3:
}

consume() {
Object c;
4: await (!empty) then {
   c = b;  b=null;
   empty = true;
}
5: use(c);
6: dispose(c);
7:
}
Thread-Modular Abstraction
Thread-Modular Abstraction

not all combinations are feasible
Partial Abstract Interpretation

4: C1: await !empty then {
   c_1 = b; empty = true;
}

4: C2: await !empty then {
   c_2 = b; empty = true;
}

5: C1: use(c_1); dispose(c_1)

5: C2: use(c_2); dispose(c_2)

Potential Double Free!!!
**A Singleton Buffer**

Boolean empty = true;
Object b = null;

produce() {
1: Object p = new();
2: await (empty) then {
    b = p;
    empty = false;
}
3: }

consume() {
Object c;
4: await (!empty) then {
    c = b;   b=null;
    empty = true;
}
5: use(c);
6: dispose(c);
7: }

Safe Dereference
No Double free
Thread-Modular Analysis

- Abstract away the correlations between local states of different threads
  - No correlations between program counters
  - Cartesian Abstraction

- Information maintained
  - Correlations between the local state of each thread and the globals

- Scales with the number of threads
- Handles unbounded number of threads
- But limited precision
Increasing Precision

- Enforce program restrictions
  - Limited aliasing
  - Ownership relations
  - Limited concurrency

- Enhanced analysis
  - Global instrumentation
  - Separation Domains [Gotsman et. al. PLDI’07]
  - Semi-Thread Modular Analysis [Segalov et. al., TR, Berdine et. al. CAV’08]
Thread-Modular Analysis

Non-disjoint resource invariants
[Roman’s work]
Fine-grained concurrency

Separated resource invariants
[Gotsman et al., PLDI 07]
Coarse-grained concurrency

**Single** global resource invariant
[Flanagan & Qadeer, SPIN 03]
Thread Quantification for Concurrent Shape Analysis

J. Berdine, T. Lev-Ami, R. Manevich, G. Ramalingam, M. Sagiv

CAV’08

Semi-Thread-Modular Analysis

M. Segalov, T. Lev-Ami, R. Manevich, G. Ramalingam, M. Sagiv
Main Results

- A refinement of thread-modular analysis
  - Not fully modular
- Precise enough to prove properties of fine-grained concurrent programs
  - Were not automatically proved before
- Two effective methods for efficiently computing transformers
  - Summarizing Effects
  - Summarizing Abstraction
  - On a concurrent set imp. speedup is x34!
Semi-Thread-Modular Analysis

- Abstract away correlations between local states of more than two threads
- Information maintained
  - Correlations between the local state of each thread and the global state
  - May-correlations between local states of every pair of threads
    - Not necessarily symmetric
Semi-Thread-Modular Abstraction

\[
\begin{align*}
\text{C1,C2} & \quad \text{C1,C3} & \quad \text{C1,C4} & \quad \text{C2,C1} & \quad \text{C2,C3} & \quad \text{C2,C4} \\
\text{C1,C2} & \quad \text{C1,C3} & \quad \text{C1,C4} & \quad \text{C2,C1} & \quad \text{C2,C3} & \quad \text{C2,C4}
\end{align*}
\]
Worst-Case Complexity

- Full state analysis
  - Shared state – $G$, Local state – $L_{tid}$
  - State space = $\mathcal{O}(G \times L_1 \times \ldots \times L_n)$
  - #states: $O(|G| \cdot |L|^n)$

- Thread-modular analysis
  - State space = $\mathcal{O}(G \times L_1) \times \ldots \times \mathcal{O}(G \times L_n)$
  - #states: $O(n \cdot |G| \cdot |L|)$

- Semi-thread-modular analysis
  - State space = $\mathcal{O}(G \times L_1 \times L_2) \times \ldots \times \mathcal{O}(G \times L_{n-1} \times L_n)$
  - #states: $O(n \cdot |G| \cdot |L|^2)$
Point-wise Transformer

\[ \text{dispose}(c_1) \]

6: C1: \text{dispose}(c_1)

6: C1: \text{dispose}(c_1)
Point-wise Transformer

dispose($c_1$)

6: C1: dispose($c_1$)

is this command safe in this configuration?
missing information on $c_1$
Abstract element \( \gamma' \) maps to concrete element \( \alpha' \) through operational semantics. Concrete element \( \text{statement } st \) maps to abstract element \( \text{statement } st \) through abstract semantics \([CC'79]\). The most-precise abstract semantics \([CC'79]\) maps abstract element \( \gamma' \) to concrete element \( \alpha' \).
Sound Transformer

refined element

“simple” abstract semantics

statement st

partial concretization

abstract element

refined element

γ'

abstract element

most-precise abstract semantics [CC’79]

statement st

abstract element
Partial Concretization Based Transformer

![Diagram showing the relationship between 3-thread substate, exec, statement st, abstract semantics, factoids, and proj.]

- 3-thread substate
- exec
- statement st
- abstract semantics
- factoids
- proj

The diagram illustrates the flow between these components, indicating how they interact within the context of partial concretization.
Transformer for Concurrent Systems

\[
TR(F) = \{<l', g', o>: <l, g, o> \in F, <l, g> \tau <l', g'>\} \cup \\
\left\{<l_2, g', o>, <l_2, g', \alpha(l_1')>: f_1, f_2, f_3, f_4 \in F: \\
<l_1, g, l_2, o> \in \text{substates}(f_1, f_2, f_3, f_4), \\
<l_1, g> \tau <l_1', g'>\right\}
\]

factoids

\[
\text{3-thread substate} \xrightarrow{\text{Chose other component}} \text{3-thread substate}
\]

\[
\text{substates} \xrightarrow{\text{Chose 1st component as executing thread}} \text{factoids}
\]

\[
\text{Chose other component statement st} \xrightarrow{\text{proj}} \text{factoids}
\]

\[
\text{executing thread statement st}
\]
Partial Concretization

C1, C2

C1, C3

C1, C4

C2, C1

C2, C3

C2, C4

C1: Executing
C2: Tracked
C3: Other
Partial Concretization (Substates)

C1, C2

C1, C3

C2, C1

C2, C3
6: C1: dispose(c) (exec)
6: C1: dispose(c) (project)
Reducing Quadratic Factors

\[
\text{TR (F)} = \{ <l', g', o> : <l, g, o> \in F, <l, g> \tau <l', g'> \} \cup
\begin{cases}
<l_2, g', o>, <l_2, g', \alpha(l_1')> : f_1, f_2, f_3, f_4 \in F: \\
<l_1, g, l_2, o> \in \text{substates}(f_1, f_2, f_3, f_4), \\
<l_1, g> \tau <l_1', g'>
\end{cases}
\]

- Exploit redundancies in the action
  - Cannot affect locals of other threads
- Use asymmetry between the two abstraction
- Can prove no loss of information
- Summarizing Effects
- Apply aggressive abstraction to the executing threads
  - Summarizing Abstraction
Other Methods for Static Analysis of Concurrency

- Start by sequential analysis and then coarsen abstractions
- Rely/Guarantee reasoning
- ...

...
Summary

• Static analysis of concurrency is interesting
• Thread modular analysis yields impressive results for well behaved coarse-grained concurrency
• Decompositions can be applied to scale abstractions even for fine-grained concurrency