The SCOOP Concurrency Model in
Java-like Languages

Faraz TORSHIZI\textsuperscript{a}, Jonathan S. OSTROFF\textsuperscript{b}, Richard F. PAIGE\textsuperscript{c} and Marsha CHECHIK\textsuperscript{a}

\textsuperscript{a} University of Toronto, Canada
\textsuperscript{b} York University, Canada
\textsuperscript{c} University of York, UK

http://www.cs.toronto.edu/~faraz
Why SCOOP?

- Techniques for writing concurrent code are still low-level
  - semaphores, locks, sync blocks, monitors etc.
  - hard to test and maintain

- There is a large gap between the above mechanisms and the popular object-oriented concepts

- The SCOOP model [Meyer97] is an attempt to bridge this gap in OO context
  - Originally developed for Eiffel language
What is SCOOP?
Simple Concurrent Object Oriented Programming

- Basic concept of OO computation: routine call $x.f(a)$
What is SCOOP?

Simple Concurrent Object Oriented Programming

- Basic concept of OO computation: routine call $x.f(a)$

- SCOOP adds the notion of a **processor** (handler)

- Processor is an **abstract concept** used to define behavior
The “separate” keyword

- $x.f(a)$ – execute routine $f$ on the object attached to $x$.

- In a sequential context $f$ is synchronous

- In a concurrent context, if $x$ denotes an object handled by another processor, $f$ is asynchronous

- This semantic difference (synchronous vs. asynchronous) has a syntactic marker: separate
Separate call

\[ x: \text{separate} \ X \]

\[ \ldots \]

\[ x.f(a) \]
SCOOP requirements

- **Handling:** All calls on an object are executed by its associated processor (no object sharing)

- **Mutual exclusion:** At most one method may execute on an object at a time

- **Separateness:**
  - Calls on *non-separate* objects are *synchronous*
  - Calls on *separate* objects are *asynchronous*

SCOOP programs are free of data races and atomicity violations by construction
class PHILOSOPHER

create
  make

feature
  left, right: separate FORK

  make (l, r: separate FORK)
    do
      left := l; right := r
    end

act
  do
    from until False loop
      eat (left, right)
    end
  end

  eat (l, r: separate FORK)
    require not (l.inuse or r.inuse)
    do
      l.pickup; r.pickup
      -- local activity --
      l.putdown; r.putdown
    end
Using sequential library in concurrent context

class PHILOSOPHER

create
  make

feature
  left, right: separate FORK

make (l, r: separate FORK)
  do
    left := l; right := r
  end

act
  do
    from until False loop
      eat (left, right)
    end
  end

eat (l, r: separate FORK)
  require not (l.inuse or r.inuse)
  do
    l.p pickup; r.p pickup
    -- local activity --
    l.putdown; r.putdown
  end

class FORK

feature
  inuse: BOOLEAN

  pickup is
    do
      inuse := True
    end

  putdown is
    do
      inuse := False
    end
end
Using sequential library in concurrent context

Automatic locking of arguments
Using sequential library in concurrent context

Automatic locking of arguments

Wait condition

```java
class PHILOSOPHER
create
  make
feature
  left, right: separate FORK
  make (l, r: separate FORK)
    do
      left := l; right := r
    end
  act
    do
      from until False loop
        eat (left, right)
      end
    end
  eat (l, r: separate FORK)
    require not (l.inuse or r.inuse)
    do
      l.pickup; r.pickup
      -- local activity --
      l.putdown; r.putdown
    end

class FORK
feature
  inuse: BOOLEAN
  pickup is
    do
      inuse := True
    end
  putdown is
    do
      inuse := False
    end
end
```
Using sequential library in concurrent context

Automatic locking of arguments

Wait condition

Async calls
Our Goal

- The SCOOP model is developed as an extension to Eiffel language

**Eiffel**
- clean, DbC, full OO
- not popular as it should

**SCOOP**
- High-level abstraction for concurrency
- Automatic synchronization
- Data race freedom
- Atomicity violation freedom
- Fair scheduling
- Using sequential libraries in concurrent context

- A pattern for SCOOP that makes it feasible to apply the SCOOP concurrency model to other OO languages

**Java/C#**
- used by many people
- no support for DbC
Input SCOOP program

Consistency checking

Core library

Translation rules

Multi-threaded output
To write the SCOOP program we use the **meta-data facility** of the supporting language.

Annotations in Java and attributes in C#

One keyword in Eiffel (**separate**) vs. two annotations in other languages (**separate** and **await**).
public class Philosopher {
    private @separate Fork rightFork;
    private @separate Fork leftFork;

    public Philosopher (@separate Fork l, @separate Fork r) {
        leftFork = l; rightFork = r;
    }

    public void act() {
        while (true) {
            eat(leftFork, rightFork); // non-separate call
        }
    }

    @await("!l.isInUse() && !r.isInUse()")
    public void eat (@separate Fork l, @separate Fork r) {
        l.pickUp(); r.pickUp(); // separate calls
        if (l.isInUse() && r.isInUse()) {
            l.putDown(); r.putDown();
        }
    }
}
Annotated C#: CSCOOP

```csharp
public class Philosopher {
    [separate] private Fork rightFork;
    [separate] private Fork leftFork;

    public Philosopher ([separate] Fork l, [separate] Fork r) {
        leftFork = l; rightFork = r;
    }

    public void act() {
        while (true) {
            eat(leftFork, rightFork); // non-separate call
        }
    }

    [await("!l.isInUse() && !r.isInUse()")]
    public void eat([separate] Fork l, [separate] Fork r) {
        l.pickUp(); r.pickUp();
        if (l.isInUse() && r.isInUse()) {
            l.putDown(); r.putDown();
        }
    }
}
```
Architecture

Input SCOOP program

Consistency checking

Core library

Translation rules

Multi-threaded output
@separate X x1;
A a1;
...
public void r (@separate X x)
{
    a1 = x.a2;
}
...
r (x1);
a1....

public class X {
    A a2;
    ...
}

Consistency issues

Client

Supplier
@separate X x1;
A a1;
...
public void r (@separate X x)
{
    a1 = x.a2;
}
...
r (x1);
a1....

public class X {
    A a2;
    ...
}

Client

Supplier

client

supplier

x1

a1

a2
@separate X x1;
A a1;
...
public void r (@separate X x) {
    a1 = x.a2;
}
...
r (x1);
a1....

Data race on a1

public class X {
    A a2;
    ...
}

Supplier

Client
@separate X x1;
A a1;
...
public void r (@separate X x)
{
    a1 = x.a2;
}
...
r (x1);
a1....

public class X {
    A a2;
    ...
}

Not allowed -- Compile-time error

Datarace on a1

Client

Supplier

client

supplier

x1

a1

a2
Architectured

- Input SCOOP program
- Consistency checking
- Translation rules
- Multi-threaded output
- Core library
Core library

- The core library provides the essentials for modeling SCOOP:
  - Processors
  - Separate and non-separate calls
  - Atomic locking of multiple resources
  - Wait semantics
  - Wait-by-necessity
  - Fair scheduling
Core library

**Core Library**

- **ScoopThread**
  - setProcessor()
  - getProcessor()
  - checkPreconditions()
  - setCall()
  - getCall()

- **Scheduler**
  - lock_requests
  - all_processors
  - locked_processors
  - sleep_semaphore: Semaphore
  - run()
  - createProcessor(…)
  - removeProcessor(…)
  - releaseLocks(…)
  - addRequest(…)

- **Processor**
  - locked_by: Processor
  - local_call_stack
  - remote_call_queue
  - sleep_semaphore: Semaphore
  - lock_semaphore: Semaphore
  - scheduler
  - run()
  - lockProcessor(…)
  - unlockProcessor(…)
  - addLocalCall(…)
  - addRemoteCall(…)
  - invokeCall(…)

- **Call**
  - method_name
  - arg_types
  - return_value
  - scoop_processor
  - scoop_object
  - call_semaphore: Semaphore
  - lock_request
  - getMethodname()
  - getObject()
  - getObjectProcessor()
  - getCallSemaphore()
  - getLockRequest()

- **LockRequest**
  - semaphore: Semaphore
  - requester
  - locks
  - getLocks()
  - getSemaphore()
  - getRequester()
Processors are instances of the Processor class.

Every processor has a
- Local call stack for local calls
- Remote call queue for remote calls

Processor repeatedly performs the following:
1. Pops an item off the stack and executes it
2. If the stack is empty, dequeues an item from the remote call queue and pushes it onto the local call stack
3. If both the stack and the queue are empty, waits for new requests to be enqueued
public class A
{
    @separate B b;
    @separate C c;
    ...
    this.m(b, c);
    ...
    @await("condition()")
    public void m(@separate B bb, @separate C cc){
        bb.f();
        cc.g();
        ...
    }
    ...
}
Dynamic behavior
Dynamic behavior

Send request
Dynamic behavior

(1) Acquire locks
(2) Check wait condition

Send request
Dynamic behavior

(1) Acquire locks
(2) Check wait condition
Send the “go ahead” signal
Dynamic behavior

1. Acquire locks
2. Check wait condition
   - Send the “go ahead” signal

Send request

- Enqueue call on proc-B and release the lock
- Same for call on proc-C
Dynamic behavior

1. Acquire locks
2. Check wait condition

Send the "go ahead" signal

Enqueue call on proc-B and release the lock

Same for call on proc-C

Continue with local activity
Architecture

Input SCOOP program

Consistency checking

Core library

Translation rules

Multi-threaded output
Translation rule for remote call

Input: annotated code

<table>
<thead>
<tr>
<th>Class %ClassName</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td>//method body containing a separate call</td>
</tr>
<tr>
<td>%ReturnType %Method( %Type0 %SepArg,...);</td>
</tr>
<tr>
<td>{ ...</td>
</tr>
<tr>
<td>%SepArg0.%SepMethod([ %SepArg0,...,%SepArgN,...]);</td>
</tr>
<tr>
<td>... }</td>
</tr>
<tr>
<td>Class %Type</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>//supplier side</td>
</tr>
<tr>
<td>void %SepMethod( %Type00 %Arg0,...,%TypeNN %ArgN)</td>
</tr>
<tr>
<td>{ ... }</td>
</tr>
</tbody>
</table>

Output: multi-threaded using core library

Class SCOOP_%ClassName |
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td>//translated method</td>
</tr>
<tr>
<td>%ReturnType %Method(SCOOP_ %Type0 %SepArg,...);</td>
</tr>
<tr>
<td>{ ...</td>
</tr>
<tr>
<td>Call call;</td>
</tr>
<tr>
<td>List&lt;Processor&gt; locks;</td>
</tr>
<tr>
<td>//loop</td>
</tr>
<tr>
<td>locks.add( %SepArg0.getProcessor());</td>
</tr>
<tr>
<td>...//do this for all separate arguments of %SepMethod</td>
</tr>
<tr>
<td>locks.add( %SepArgN.getProcessor());</td>
</tr>
<tr>
<td>//end loop</td>
</tr>
<tr>
<td>lock_request = new LockRequest( %SepArg, locks,</td>
</tr>
<tr>
<td>%SepArg.getProcessor().getLockSemaphore());</td>
</tr>
<tr>
<td>List&lt;Object&gt; args_types, args;</td>
</tr>
<tr>
<td>//loop</td>
</tr>
<tr>
<td>args_types.add( %Type00); //use the SCOOP_ for separate types</td>
</tr>
<tr>
<td>args.add( %Arg0);</td>
</tr>
<tr>
<td>...//do this for all arguments of %SepMethod</td>
</tr>
<tr>
<td>args_types.add( %TypeNN);</td>
</tr>
<tr>
<td>args.add( %ArgN); ...</td>
</tr>
<tr>
<td>//end loop</td>
</tr>
<tr>
<td>call = new Call(&quot;%SepMethod&quot;, args_types, args,</td>
</tr>
<tr>
<td>void, lock_request, %SepArg0, void);</td>
</tr>
<tr>
<td>%SepArg0.getProcessor().addRemoteCall(call);</td>
</tr>
<tr>
<td>...//move on to the next operation without waiting</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
Tool support: JSCOOP
Tool support: JSCOOP

```java
public class Client{
    public static X x1;
    public static X x2;
    public static A a;

    public void sc1(){
        @separate X x1 = new X();
        X x2 = new X();
        x2 = x1; // invalid: traitor
        x1 = x2; // valid
        r1 (x1); // invalid
    }

    public void r1 (X x){}

    public void sc2(){
        @separate X x1 = new X();
        r2 (x1);
    }

    public void r2 (@separate X x){
        x.f(a); // invalid
        x.g(a); // valid
    }

    public void sc3(){
        X x1 = new X();
        s (x1);
    }
}
```

Violation of consistency rules
Tool support: JSCOOP
Limitations / Future work

- Not shown the correctness of translation
  - develop more examples in JSCOOP
  - check the bi-similar behavior to programs written in Java

- Need for empirical studies
  - access the efficiency and effectiveness of the tool

- Add full support for inheritance and genericity

- **SCOOP is still prone to deadlocks**
  - Apply model-checking techniques to detect deadlocks at compile-time
Summary

- Design pattern for SCOOP that makes it feasible to apply the SCOOP concurrency model to other OO languages
  - Annotation processing and consistency checking
  - Core library
  - Translation rules

- A prototype implementation for Java based on an Eclipse plug-in called JSCOOP

- http://code.google.com/p/jscoop
Thank you!
References