The SCOOP Concurrency Model in Java-like Languages

Faraz TORSHIZI[a], Jonathan S. OSTROFF[b], Richard F. PAIGE[c] and Marsha CHECHIK[a]
[a] ***University of Toronto, Canada***
[b] *York University, Canada*
[c] *University of York, UK*

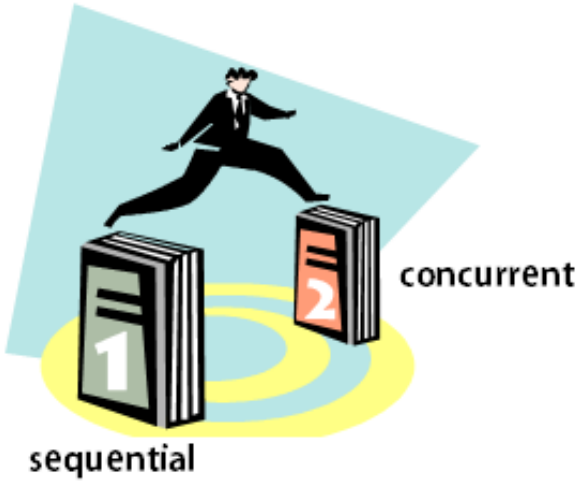
`http://www.cs.toronto.edu/~faraz`

Communicating Process Architectures (CPA'09)

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

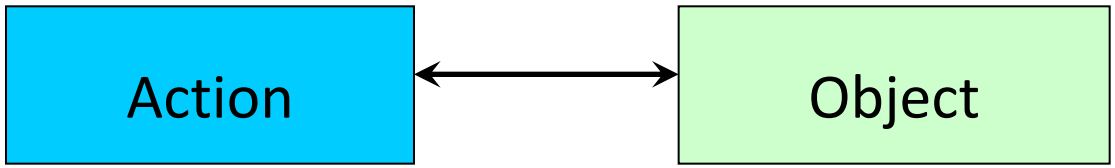


- The SCOOP model [Meyer97] is an attempt to bridge this gap in OO context
  - Originally developed for Eiffel language

What is SCOOP?

Simple Concurrent Object Oriented Programming

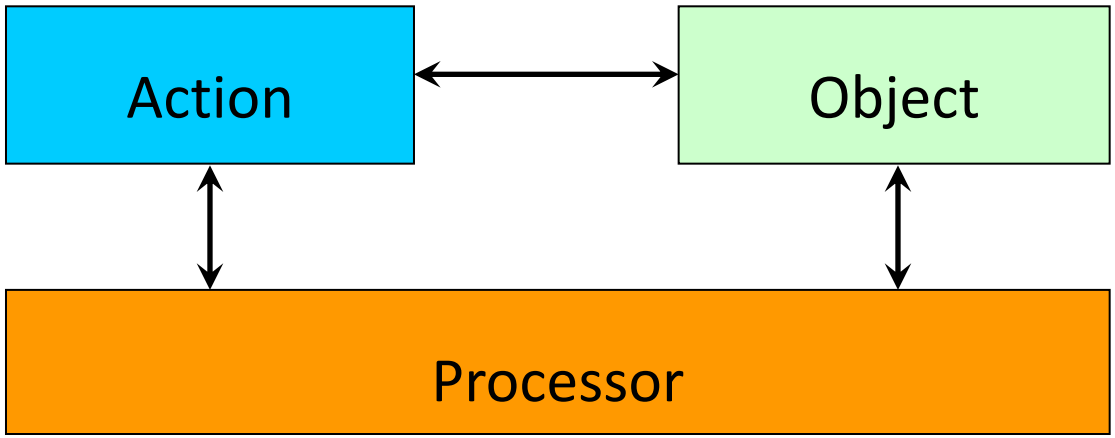
- Basic concept of OO computation: routine call **x.f(a)**



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

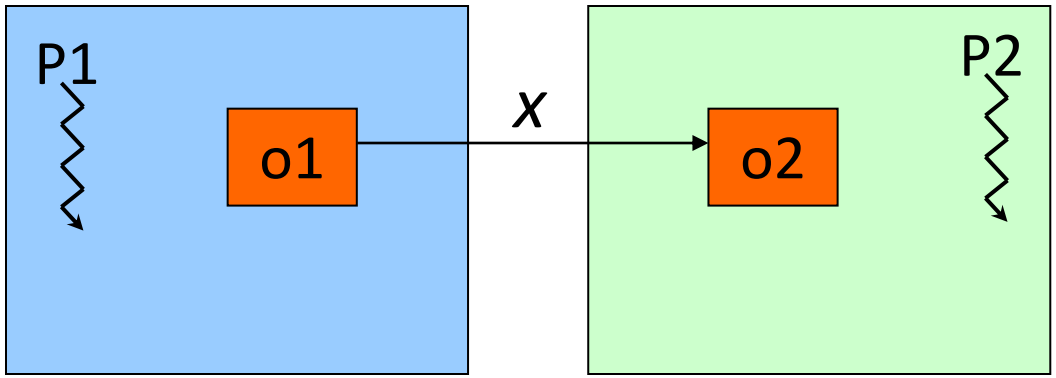
- **x.f (a)** – execute routine ***f*** on the object attached to ***x***.
- In a sequential context ***f*** is **synchronous**
- In a concurrent context, if ***x*** denotes an object handled by another processor, ***f*** is **asynchronous**
- This semantic difference (synchronous vs. asynchronous) has a syntactic marker: **separate**

Separate call

x: **separate** *X*

...

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
```

```
class FORK

feature
   inuse: BOOLEAN

   pickup is
      do
         inuse := True
      end

   putdown is
      do
         inuse := False
      end
end
```

```
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 FORK

feature
    inuse: BOOLEAN

    pickup is
        do
            inuse := True
        end

    putdown is
        do
            inuse := False
        end
end
```

```
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 FORK

feature
    inuse: BOOLEAN

    pickup is
        do
            inuse := True
        end

    putdown is
        do
            inuse := False
        end
end
```

Automatic locking of arguments

```
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

```
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

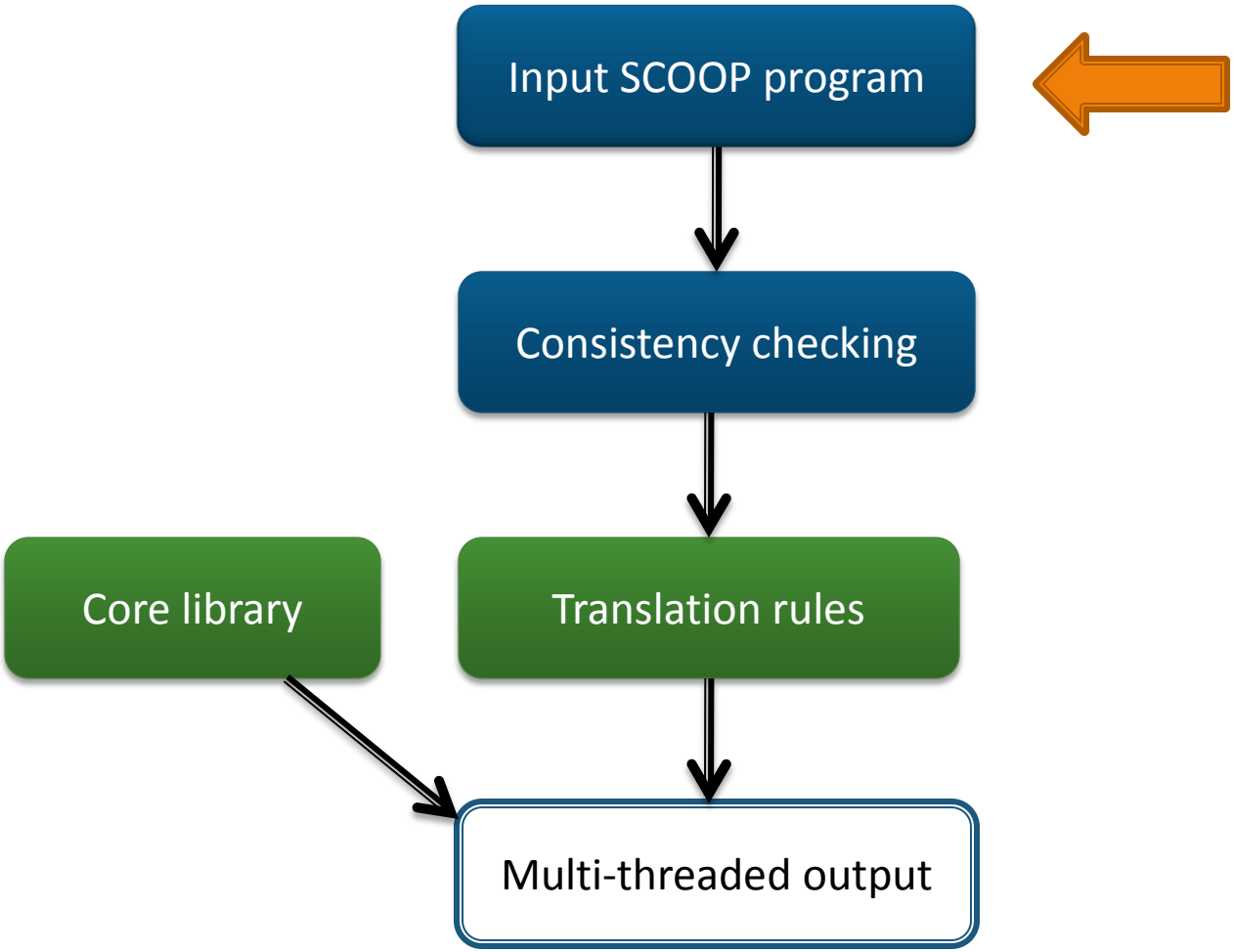
Eiffel
- clean, DbC, full OO
- not popular as it should

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

Architecture



Input SCOOP program

- 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**)

Annotated Java: JSCOOP

```
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

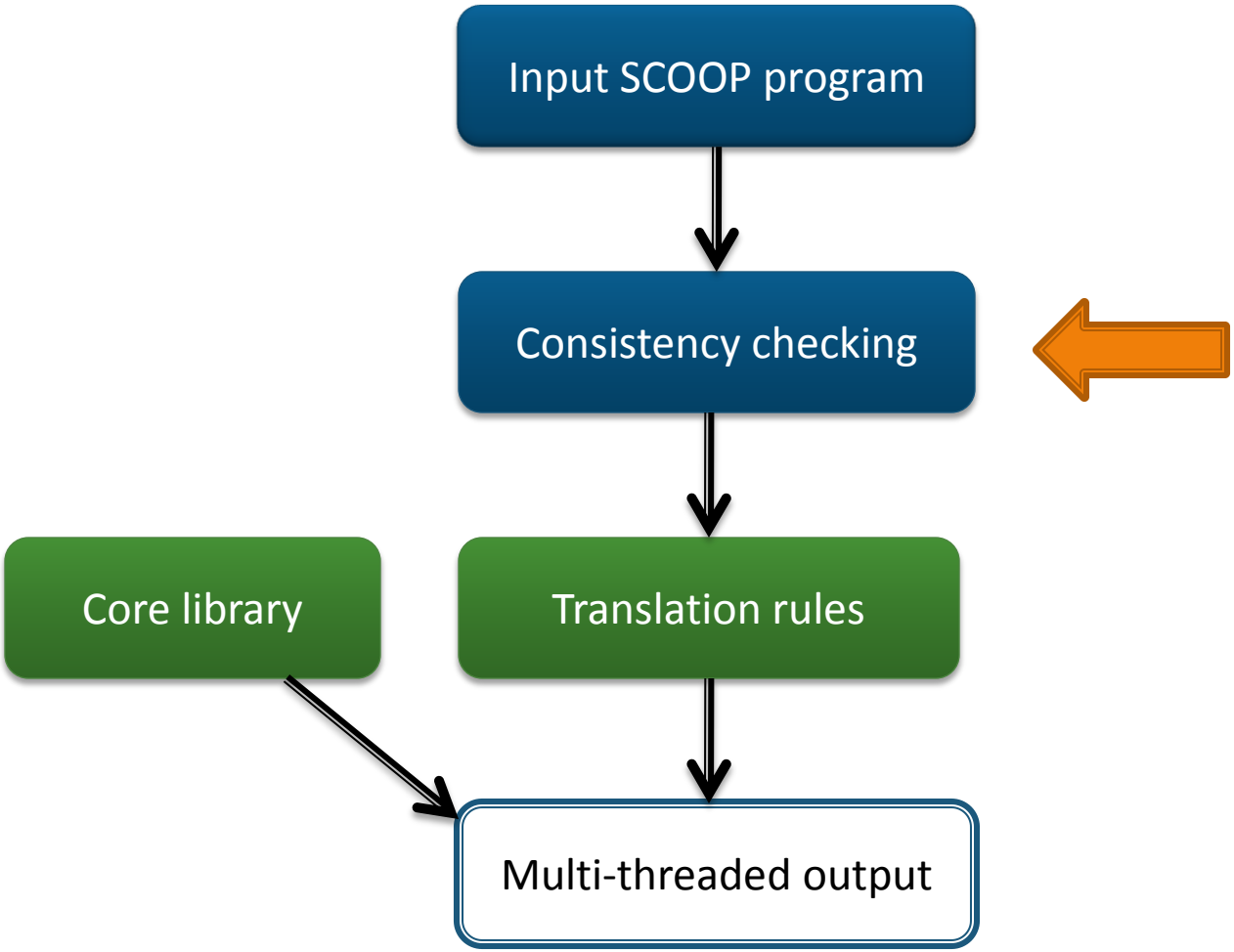
```
public class Philosopher {
   [separate] private Fork rightFork;
   [separate] private Fork leftFork;

   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



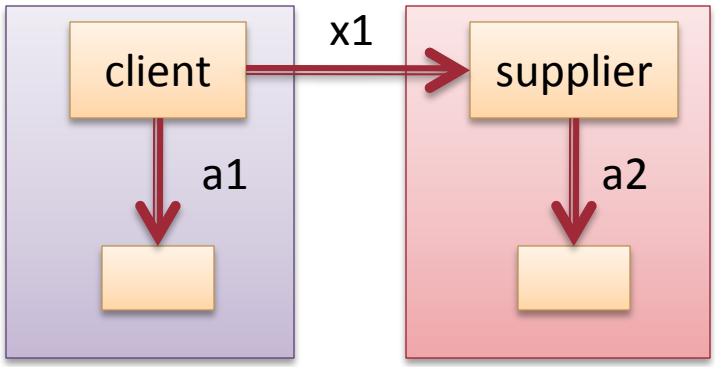
Consistency issues

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

Client

```
public class X {
   A a2;
   …
}
```

Supplier



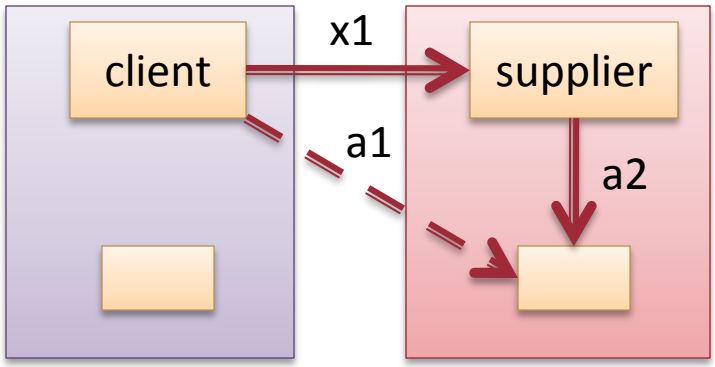
Consistency issues

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

Client

```
public class X {
   A a2;
   …
}
```

Supplier



Consistency issues

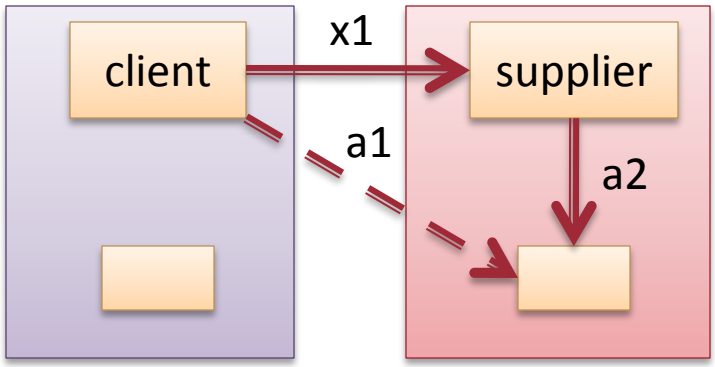
```
@separate X x1;
A a1;
…
public void r (@separate X x)
{
   a1 = x.a2;
}
…
r (x1);
a1....
```

Datarace on a1

Client

```
public class X {
   A a2;
   …
}
```

Supplier



Consistency issues

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

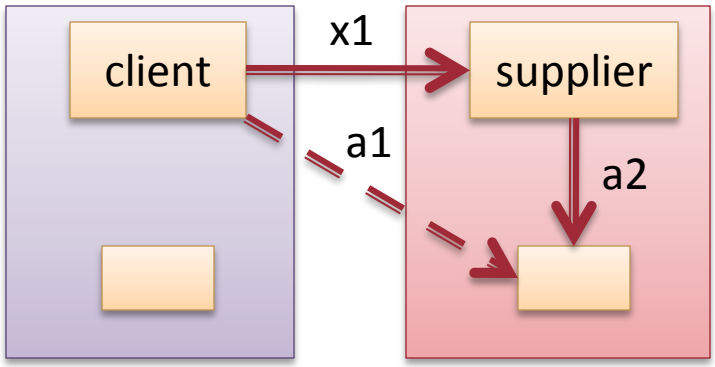
Not allowed -- Compile-time error

Datarace on a1

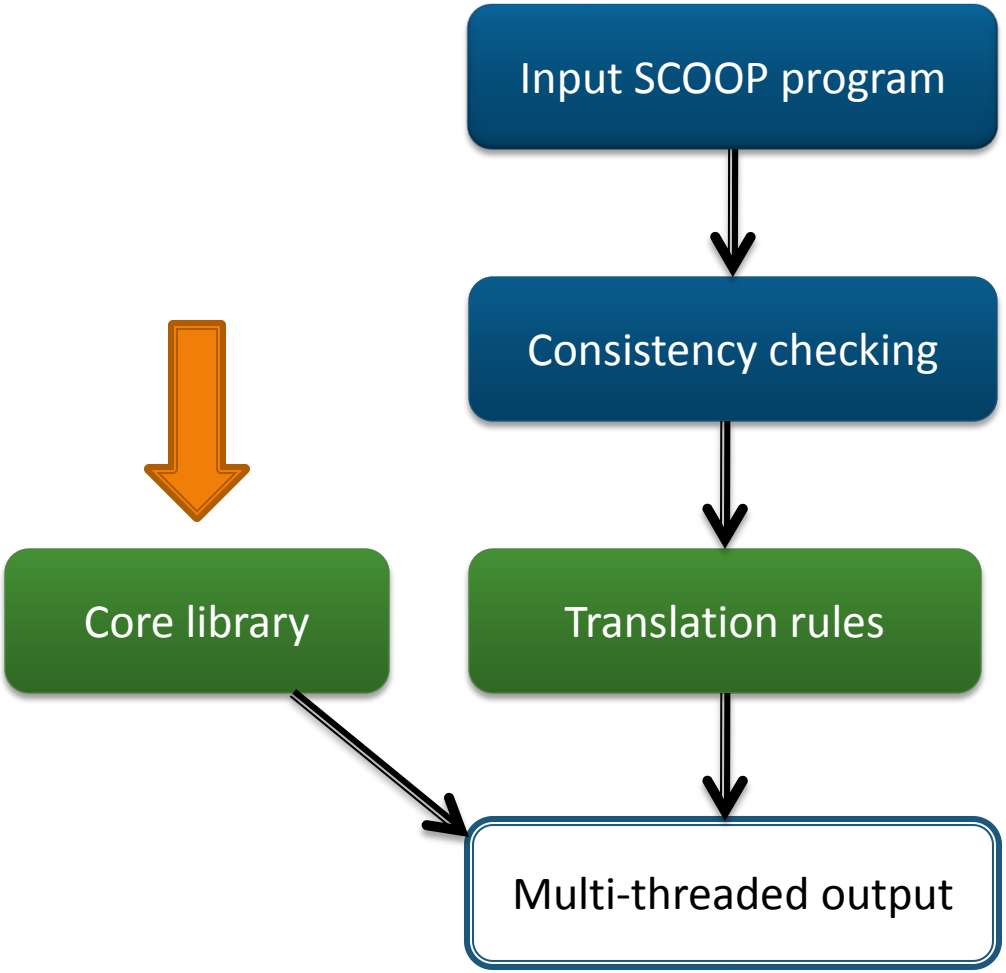
Client

```
public class X {
   A a2;
   …
}
```

Supplier



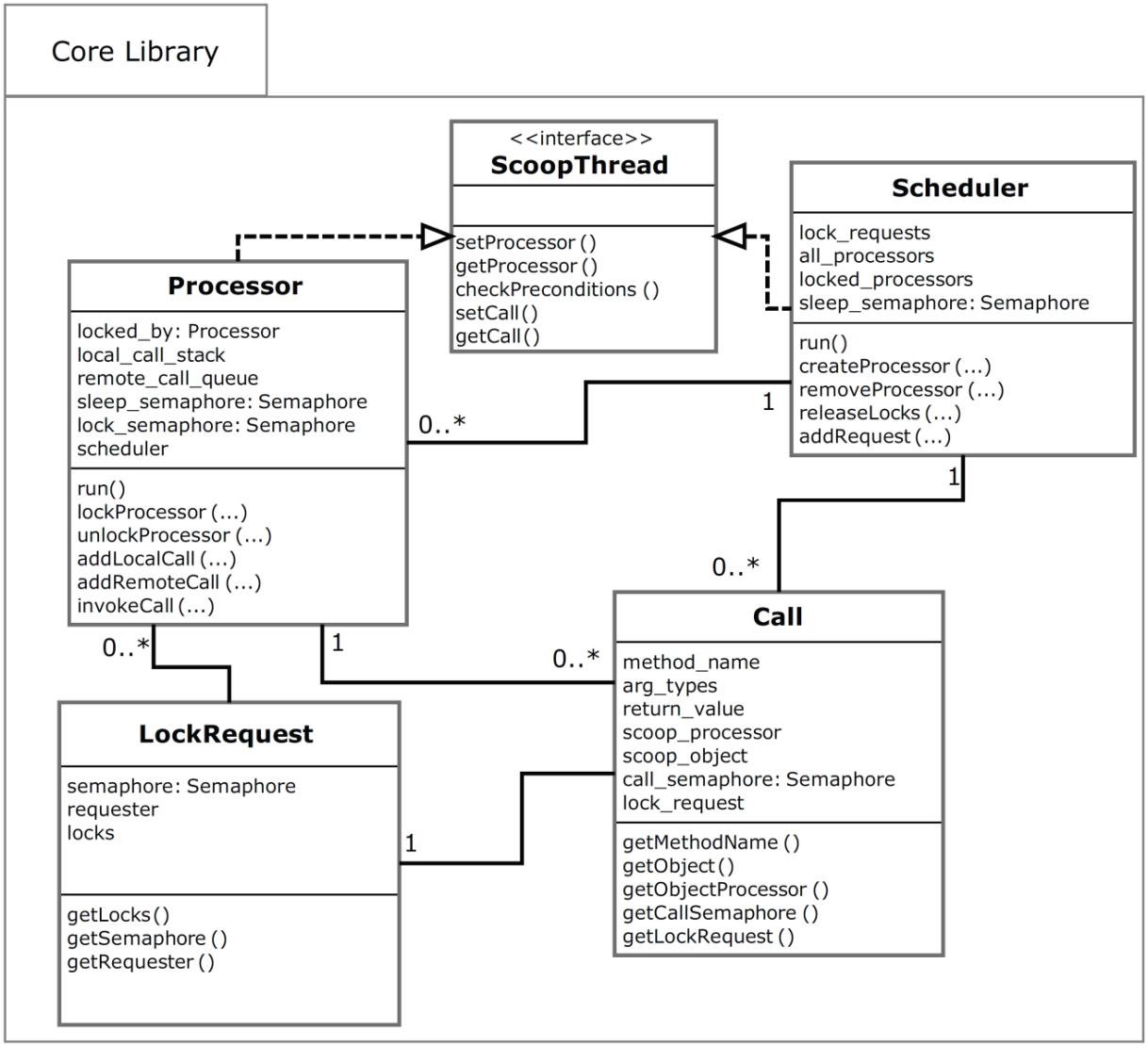
Architecture



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



Processor

- Processors are instances of the **Processor** class.

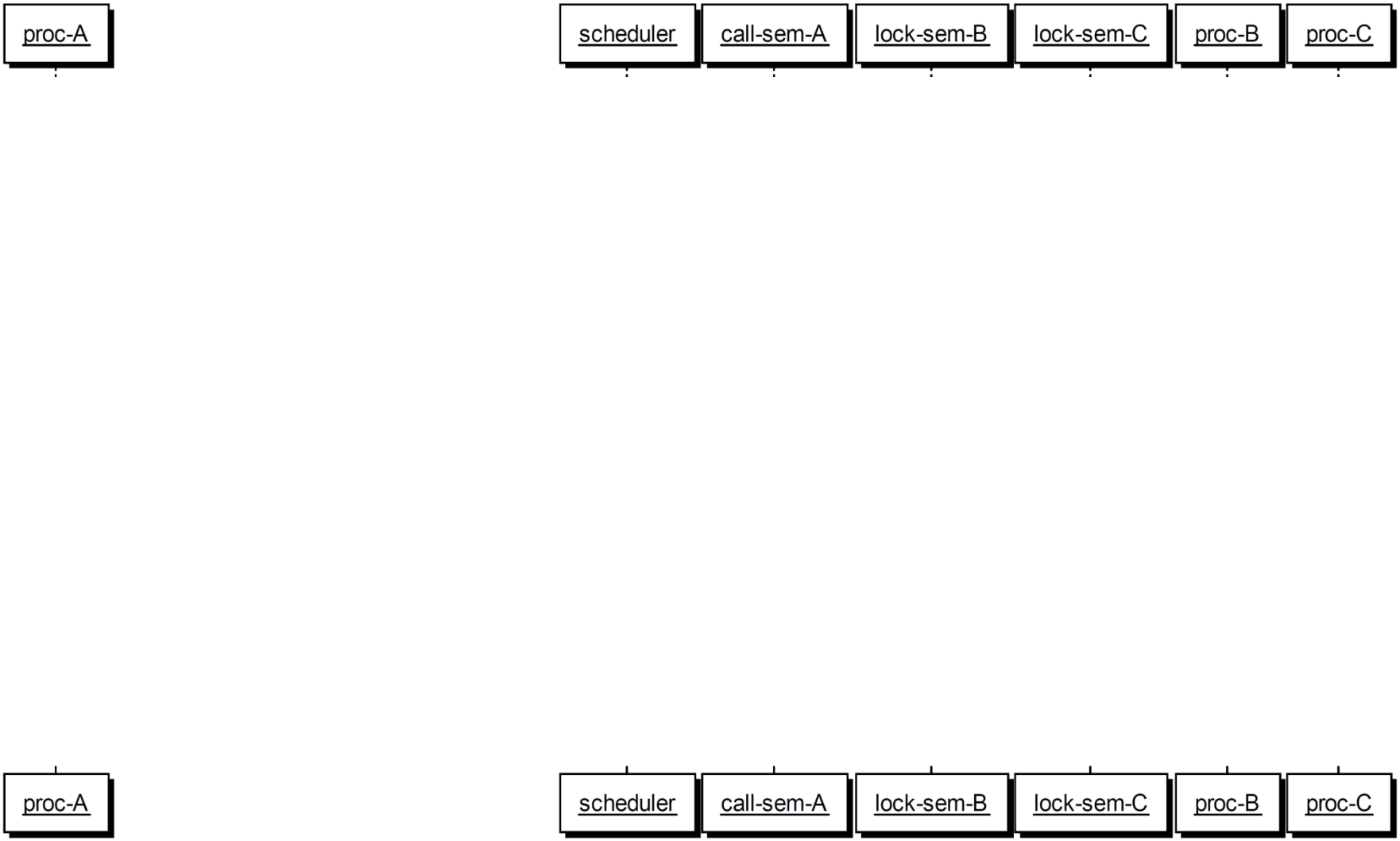
- Every processor has a
  - **Local call stack** for local calls
  - **Remote call queue** for remote calls

- 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

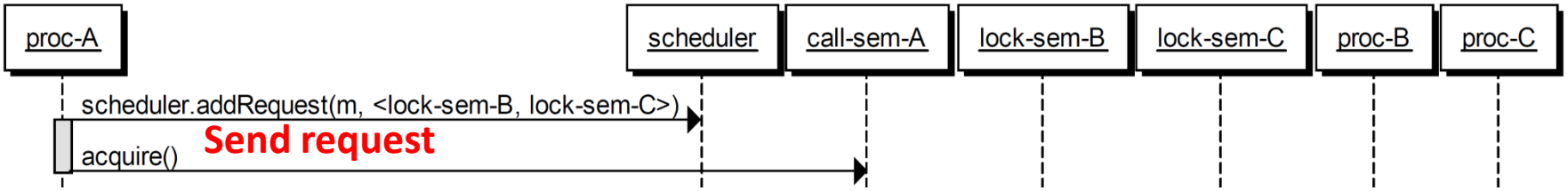
SCOOP dynamic behavior

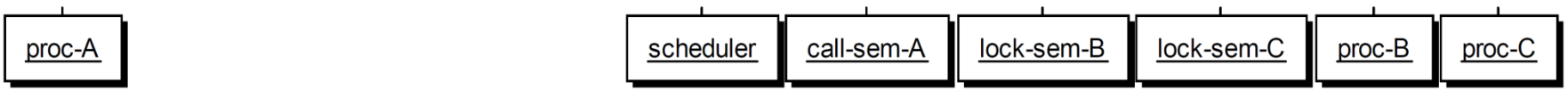
```
public class A
{
    @separate B b;
    @separate C c;
      ...
    this.m(b, c);
      ...
    @await("condition()")
    public void m(@separate B bb, @separate C cc){
       bb.f();
       cc.g();
       ...
    }
      ...
}
```

Dynamic behavior

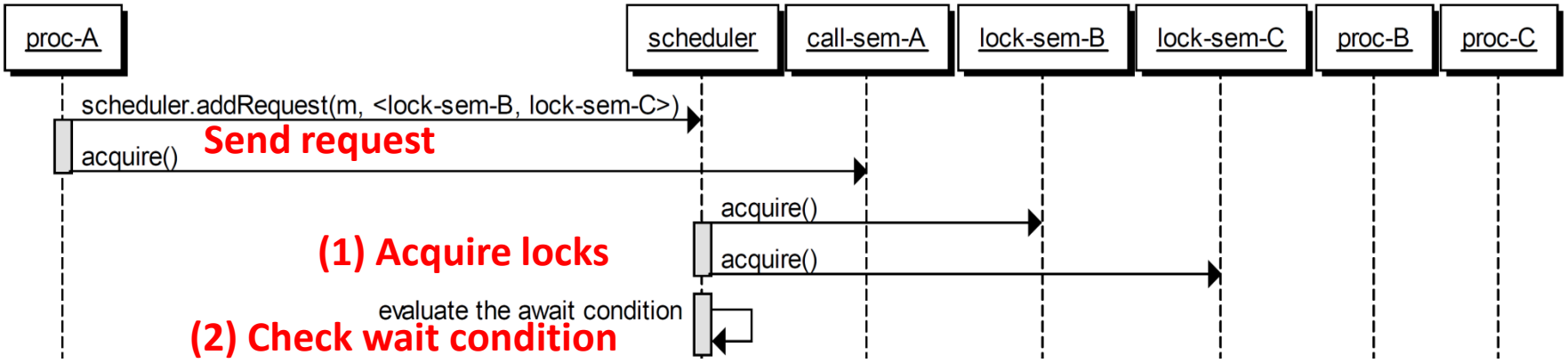


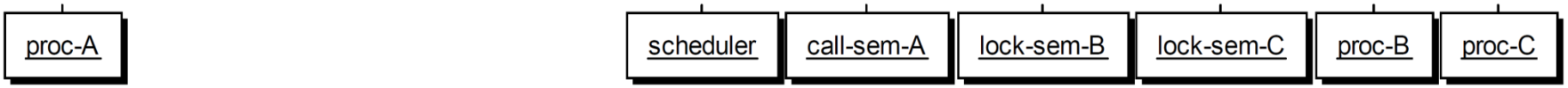
Dynamic behavior



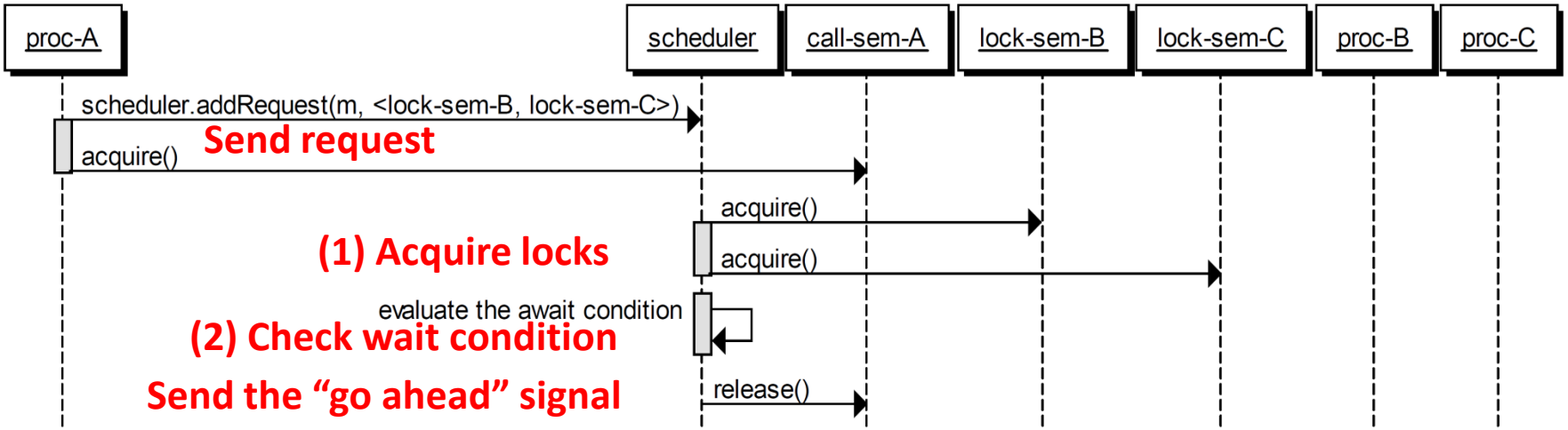


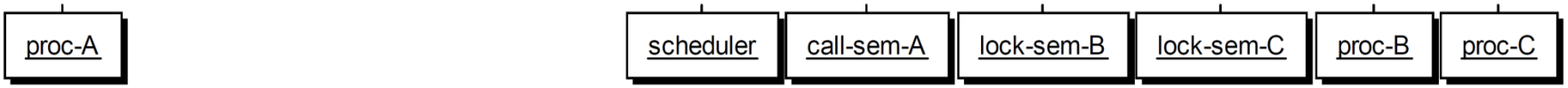
Dynamic behavior



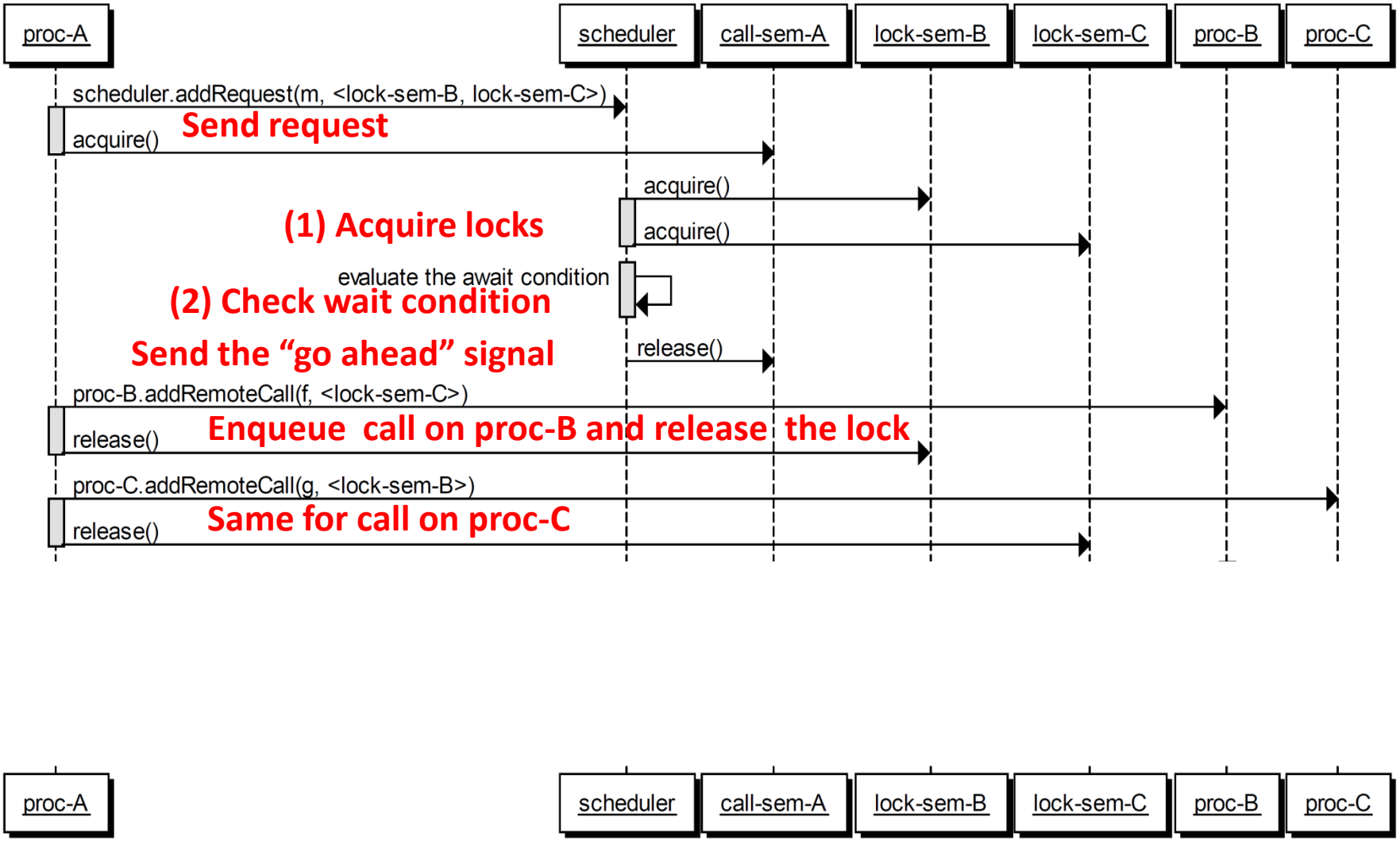


Dynamic behavior



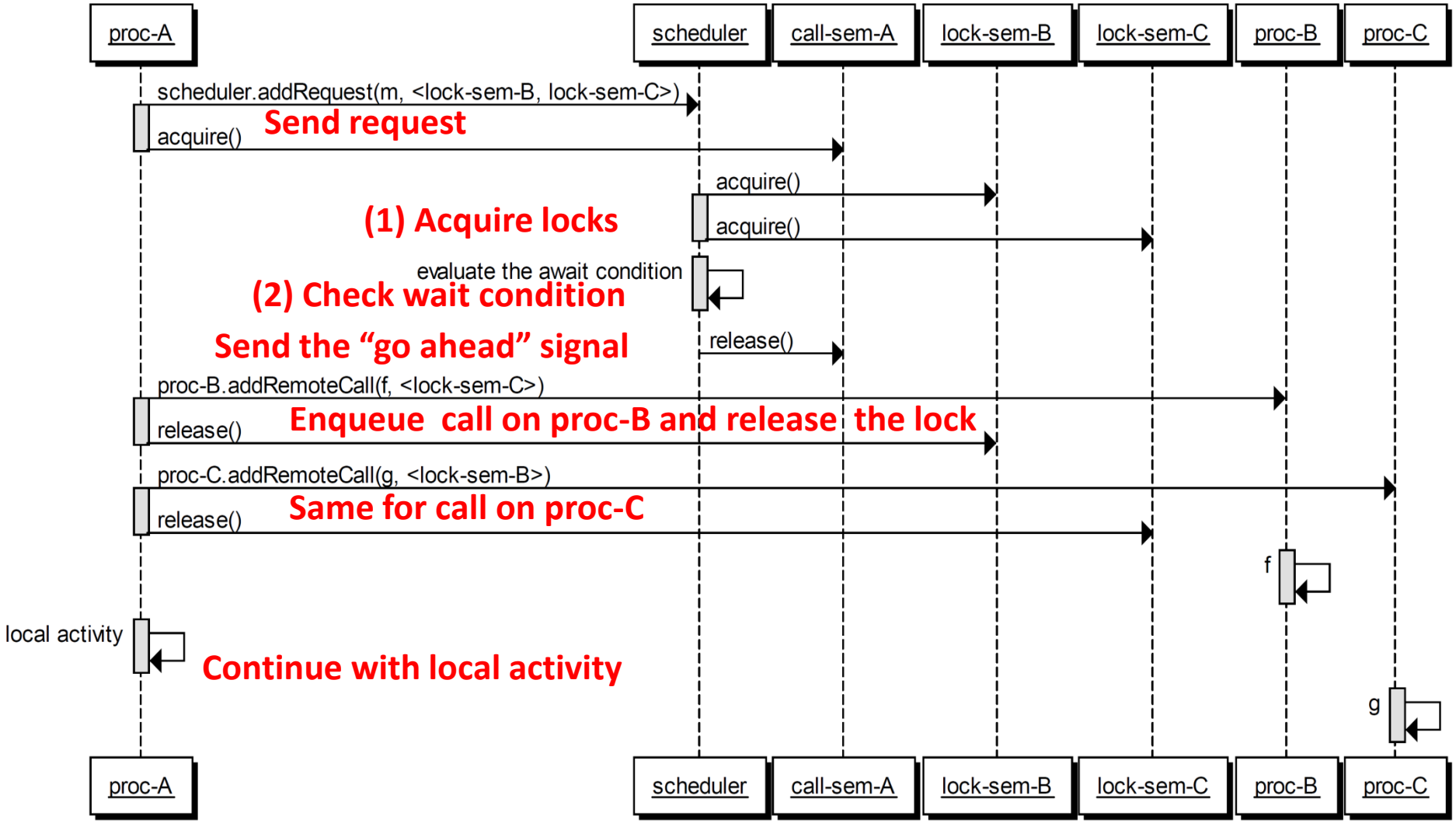


Dynamic behavior

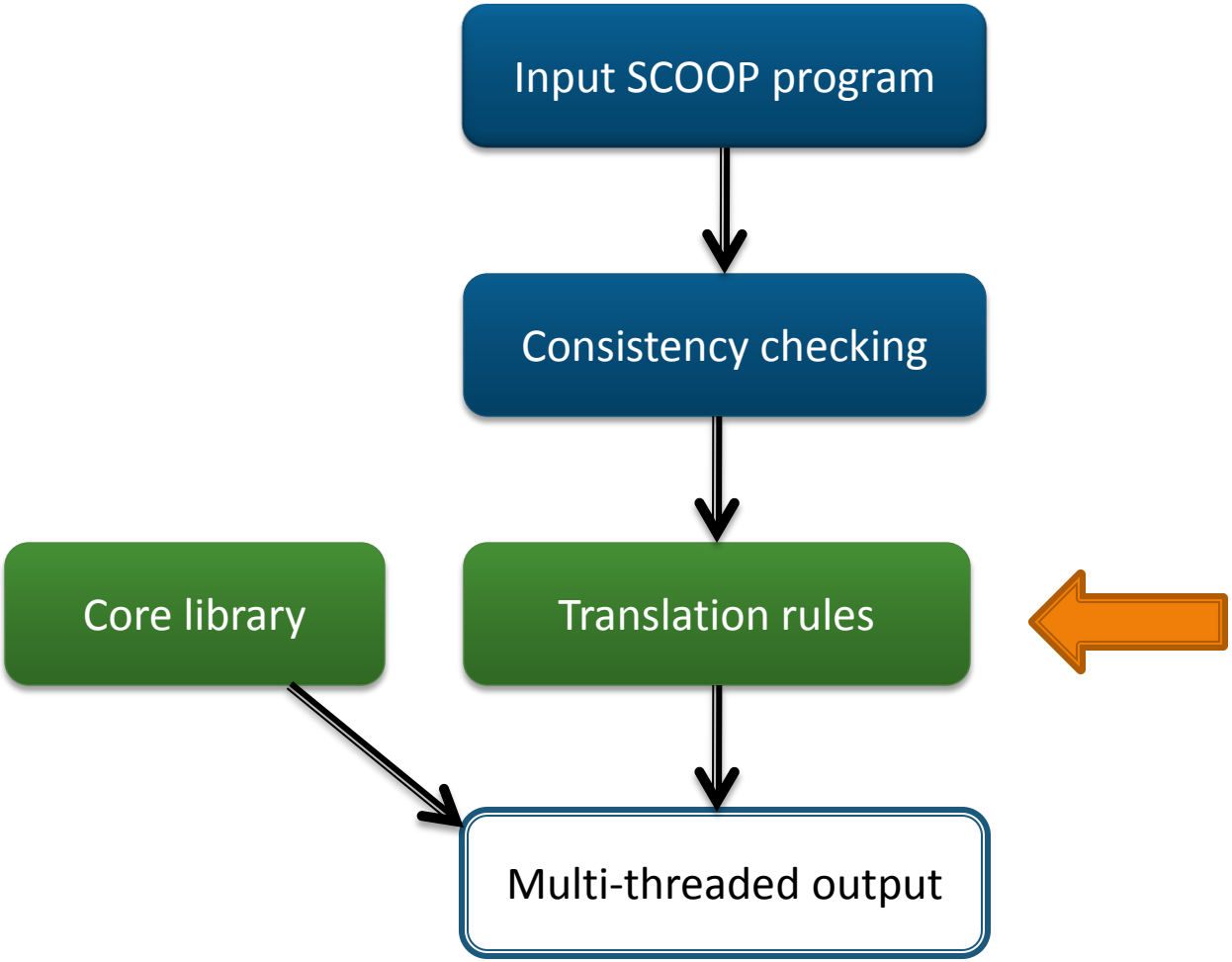


proc-A
scheduler
call-sem-A
lock-sem-B
lock-sem-C
proc-B
proc-C

Dynamic behavior



Architecture



Translation rule for remote call

Annotated code

```
Class %ClassName {
  ...
  //method body containing a separate call
  %ReturnType %Method(%Type0 %SepArg,...);
  { ...
   %SepArg0.%SepMethod([%SepArg0,...,%SepArgN,...]);
    ...
  }
  ...
}

Class %Type {
  ...
  //supplier side
  void %SepMethod(%Type00 %Arg0,...%TypeNN %ArgN)
  { ... }
  ...
}
```

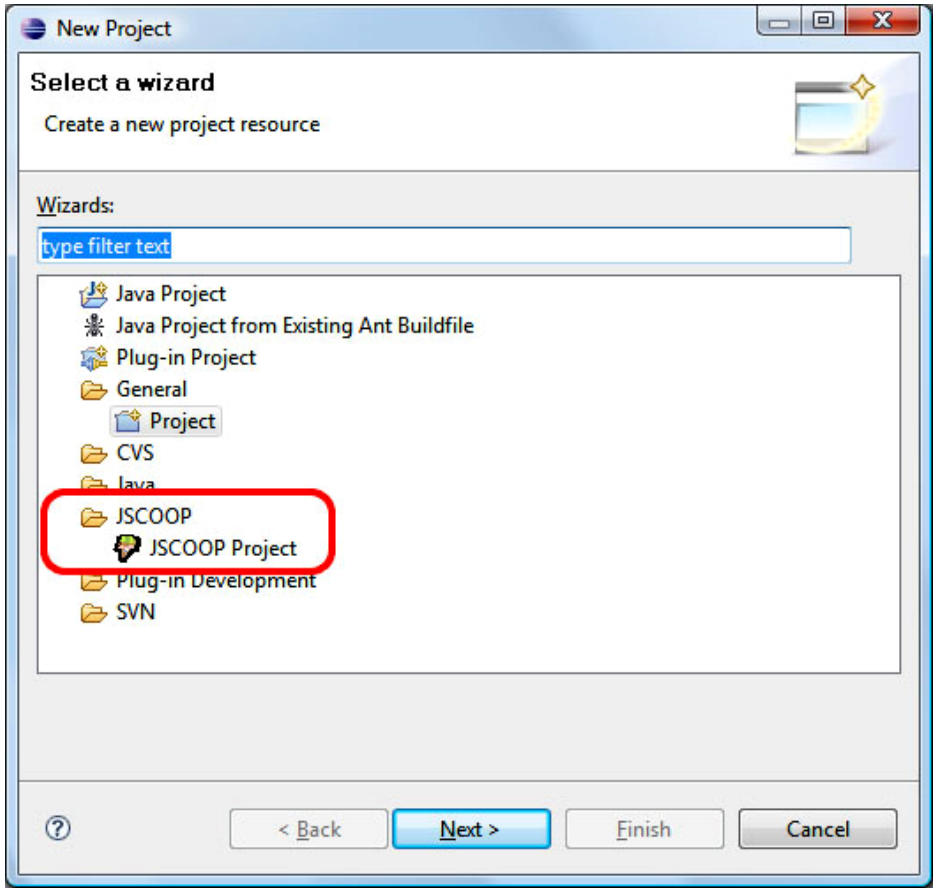
Input: annotated code

Translation

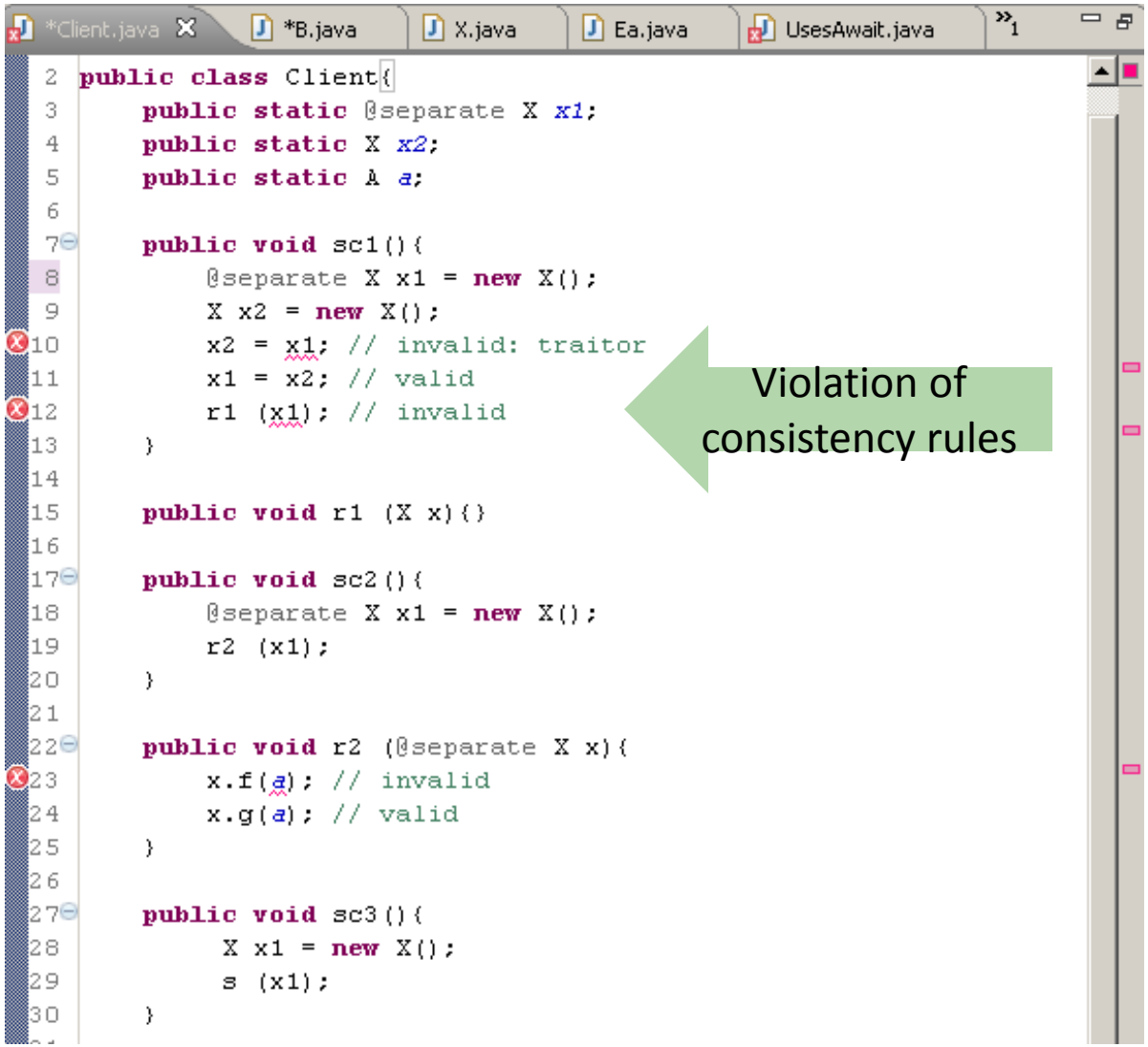
```
Class SCOOP_%ClassName {
  ...
  //translated method
  %ReturnType %Method(SCOOP_%Type0 %SepArg,...);
  { ...
    Call call;
    List<Processor> locks;
    //loop
    locks.add(%SepArg0.getProcessor());
    ...//do this for all separate arguments of %SepMethod
    locks.add(%SepArgN.getProcessor());
    //end loop
    lock_request = new LockRequest(%SepArg, locks,
       %SepArg.getProcessor().getLockSemaphore());
    List<Object> args_types, args;
    //loop
    args_types.add(%Type00); //use the SCOOP_ for separate types
    args.add(%Arg0);
    ...//do this for all arguments of %SepMethod
    args_types.add(%TypeNN);
    args.add(%ArgN);...
    //end loop
    call = new Call("%SepMethod", arg_types, args,
                    void, lock_request, %SepArg0, void);
    %SepArg0.getProcessor().addRemoteCall(call);
    ...//move on to the next operation without waiting
  }
  ...
}
```

Output: multi-threaded using core library

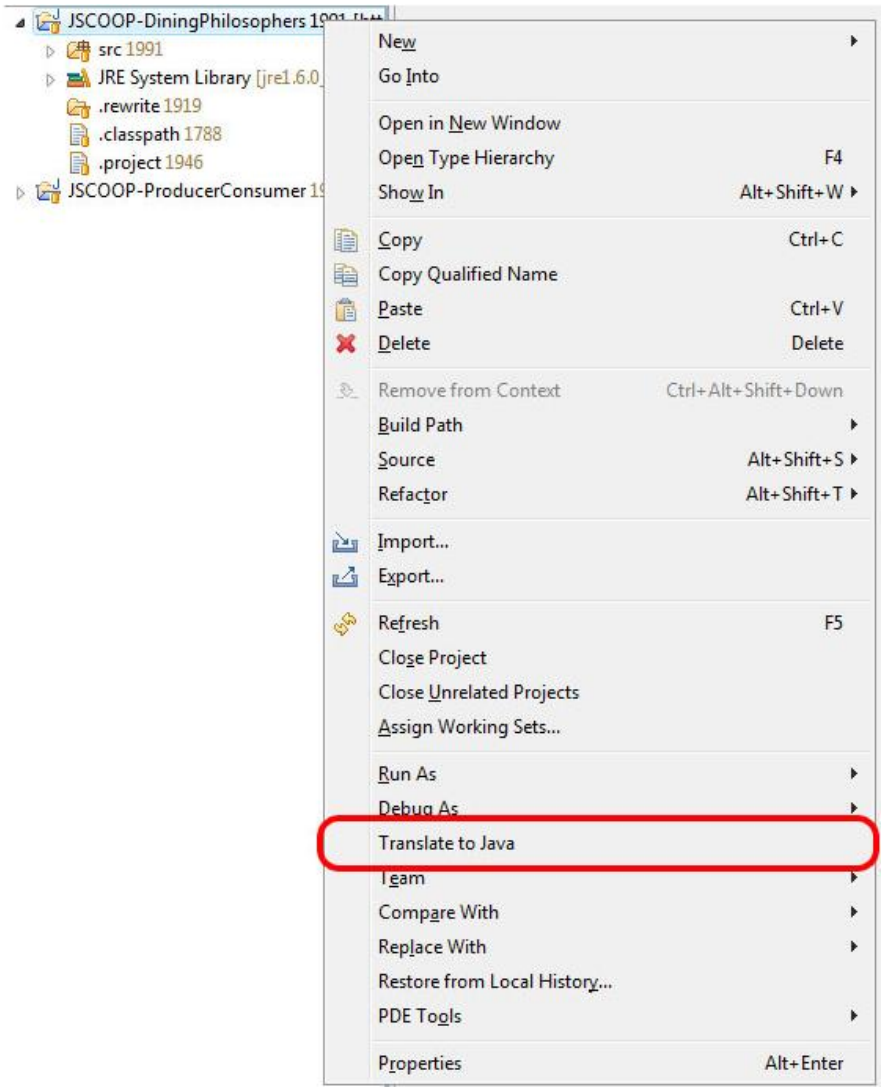
Tool support: JSCOOP



Tool support: JSCOOP



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 genercity
- **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

1. Bertrand Meyer. *Object-Oriented Software Construction. Prentice Hall, 1997.*
2. Piotr Nienaltowski. *Practical framework for contract-based concurrent object-oriented programming, PhD thesis 17031. PhD thesis, Department of Computer Science, ETH Zurich, 2007.*
3. Jonathan S. Ostroff, Faraz Torshizi, Hai Feng Huang, and Bernd Schoeller. Beyond contracts for concurrency. *Formal Aspects of Computing, 21(4):319–346, 2009.*
4. Piotr Nienaltowski. Flexible access control policy for SCOOP. *Formal Aspects of Computing, 21(4):347–*362, 2009.
5. Phillip J. Brooke, Richard F. Paige, and Jeremy L. Jacob. A CSP model of Eiffel's SCOOP. *Formal Aspects of Computing, 19(4):487–512, 2007.*