Communicating Process Architectures 2009
Peter Welch, Herman Roebbers and Tobe Announced (Eds.)
IOS Press, 2009

The SCOOP Concurrency Model in Java-like Languages

Faraz TORSHIZI [a,1], Jonathan S. OSTROFF [b], Richard F. PAIGE [c] and Marsha CHECHIK [a]

[a] *Department of Computer Science, University of Toronto, Canada*
[b] *Department of Computer Science and Engineering, York University, Canada*
[c] *Department of Computer Science, University of York, UK*

Abstract.
SCOOP is a minimal extension to the sequential object-oriented programming model for concurrency. The extension consists of one keyword (**separate**) that avoids explicit thread declarations, synchronized blocks, explicit waits, and eliminates data races and atomicity violations by construction, through a set of compiler rules. SCOOP was originally described for the Eiffel programming language. This paper makes two contributions. Firstly, it presents a design pattern for SCOOP, which makes it feasible to transfer SCOOP's concepts to different object-oriented programming languages. Secondly, it demonstrates the generality of the SCOOP model by presenting an implementation of the SCOOP design pattern for Java. Additionally, we describe tools that support the SCOOP design pattern, and give a concrete example of its use in Java.

Keywords. Object Oriented, Concurrency, SCOOP, Java

Introduction

Concurrent programming is challenging, particularly for less experienced programmers who must reconcile their understanding of programming logic with the low-level mechanisms (like threads, monitors and locks) needed to ensure safe, secure and correct code. Brian Goetz (a primary member of the Java Community Process for concurrency) writes:

> [...] multicore processors are just now becoming inexpensive enough for midrange desktop systems. Not coincidentally, many development teams are noticing more and more threading-related bug reports in their projects. In a recent post on the NetBeans developer site, one of the core maintainers observed that a single class had been patched over 14 times to fix threading related problems. Dion Almaer, former editor of TheServerSide, recently blogged (after a painful debugging session that ultimately revealed a threading bug) that most Java programs are so rife with concurrency bugs that they work only "by accident". ...
>
> One of the challenges of developing concurrent programs in Java is the mismatch between the concurrency features offered by the platform and how developers need to think about concurrency in their programs. The language provides low-level mechanisms such as synchronization and condition waits, but these mechanisms must be used consistently to implement application-level protocols or policies. [1, p.xvii]

Java 6.0 adds richer support for concurrency but the gap between low-level language mechanisms and programmer cognition persists.

The SCOOP (Simple Concurrent Object Oriented Programming) model of concurrency [2,3] contributes towards bridging this gap in the object-oriented context. Instead of

[1] Corresponding Author: *Faraz Torshizi, 10 King's College Road, Toronto, ON, Canada, M5S 3G4*. E-mail: `faraz@cs.toronto.edu`.

providing typical concurrency constructs (e.g., threads, semaphores and monitors), SCOOP operates at a different level of abstraction, providing a single new keyword **separate** that is used to denote an object running on a different processor. Processor is an abstract notion used to define behavior. Processors can be mapped to virtual machine threads, OS threads, or even physical CPUs (in case of multi-core systems). Calls within a single processor are synchronous (as in sequential programming), whereas calls to objects on other processors are dispatched asynchronously to those processors for execution while the current execution continues. This is all managed by the runtime, invisible to the developer. The SCOOP compiler eliminates race conditions and atomicity violations by construction thus automatically eliminating a large number of concurrency errors via a static compiler check. However, it is still prone to user-introduced deadlocks. SCOOP also extends the notion of Design by Contract (DbC) to the concurrent setting. Thus, one can specify and reason about concurrent programs with much of the simplicity of reasoning about sequential programs while preserving concurrent execution [4].

Contributions

The SCOOP model is currently implemented in the Eiffel programming language, making use of Eiffel's built-in DbC support. In this paper, we make two contributions:

1. We provide a design pattern for SCOOP that makes it feasible to apply the SCOOP concurrency model to other object-oriented programming languages such as Java and C# that do not have Eiffel constructs for DbC. The design has a front-end and a back-end. The front-end is aimed at software developers and allows them to write SCOOP code in their favorite language. The back-end (invisible to developers) translates their SCOOP programs into multi-threaded code. The design pattern adds **separate** and **await** keywords using the language's meta-data facility (e.g., Java annotations or C# attributes). The advantage of this approach is that developers can use their favorite tools and compilers for type checking. The translation rules enforced by the design pattern happen in the back-end.
2. We illustrate the design pattern with a prototype for Java, based on an Eclipse plug-in called JSCOOP. Besides applying the SCOOP design pattern to Java, and allowing SCOOP programs to be written in Java, the JSCOOP plug-in allows programmers to create projects and provides syntax highlighting and compile-time consistency checking for "traitors" that break the SCOOP concurrency model (Section 2.1). The plug-in uses a core library and translation rules for the generic design pattern to translate JSCOOP programs to multi-threaded Java. Compile time errors are reported at the JSCOOP level using the Eclipse IDE.

Currently the Eclipse plug-in supports preconditions (**await**) but not postconditions. This is sufficient to capture the major effects of SCOOP, but future work will need to address the implementation of full DbC.

In Section 1 we describe the SCOOP model in the Eiffel context. In Section 2 we describe the front-end design decisions for transferring the model to Java and C#. In Section 3, we describe the back-end design pattern via a core library (see the UML diagram in Fig. 1 on page 8). This is followed by Section 4 which goes over a number of generic translation rules used by the pre-processor to generate multi-threaded code using the core library facilities. In Section 5 we describe the Eclipse plug-in which provides front-end and back-end tool support for JSCOOP (the Java implementation of SCOOP). This shows that the design pattern is workable and allows us to write SCOOP code in Java. In Section 6, we compare our work to other work in the literature.

1. The SCOOP Model of Concurrency

The SCOOP model was described in detail by Meyer [2], and refined (with a prototype implementation in Eiffel) by Nienaltowski [3]. The SCOOP model is based on the basic concept of OO computation which is a routine call `t.r(a)`, where `r` is a routine called on a target `t` with argument `a`. SCOOP adds the notion of a *processor* (handler). A processor is an abstract notion used to define behavior — routine `r` is executed on object `t` by a processor `p`. The notion of processor applies equally to sequential or concurrent programs. In a sequential setting, there is only one processor in the system and behaviour is synchronous. In a concurrent setting there is more than one processor. As a result, a routine call may continue without waiting for previous calls to finish (i.e., behaviour is asynchronous).

By default, new objects are created on the same processor assigned to handle the current object. To allow the developer to denote that an object is handled by a different processor than the current one, SCOOP introduces the **separate** keyword. If the **separate** keyword is used in the declaration of an object `t` (e.g., an attribute), a new processor `p` will be created as soon as an instance of `t` is created. From that point on, all actions on `t` will be handled by processor `p`. Assignment of objects to processors does not change over time.

```
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) -- eating
            -- thinking
         end
      end

   eat (l, r: separate FORK)
      require not (l.inuse or r.inuse)
      do
         l.pickup; r.pickup
         if l.inuse and r.inuse then
           l.putdown; r.putdown
         end
      end
end
```

Listing 1: `PHILOSOPHER` class in Eiffel

The familiar example of the dining philosophers provides a simple illustration of some of the benefits of SCOOP. A SCOOP version of this example is shown in listings 1 and 2. Attributes `left` and `right` of type `FORK` are declared **separate**, meaning that each fork object is handled by its own processor (thread of control) separate from the current processor (the thread handling the philosopher object). Method calls from a philosopher to a fork object (e.g., `pickup`) are handled by the fork's dedicated processor in the order that the calls are received.

The system deadlocks when forks are picked up one at a time by competing philosophers. Application programmers may avoid deadlock by recognizing that two forks must

```
class FORK feature
   inuse: BOOLEAN

   pickup is
      do inuse := True end

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

Listing 2: `FORK` class in Eiffel

be obtained at the same time for a philosopher to safely eat to completion. We encode this information in an `eat` method that takes a left and right fork as **separate** arguments.

The `eat` routine invoked at line 13 waits until both fork processors are allocated to the philosopher and the precondition holds. The precondition (**require** clause) is treated as a guard that waits for the condition to become true rather than a correctness condition that generates an exception if the condition fails to hold. In our case, the precondition asserts that both forks must not be in use by other philosophers. A scheduling algorithm in the back-end ensures that resources are *fairly* allocated. Thus, philosophers wait for resources to become available, but are guaranteed that *eventually* they will become available provided that all reservation methods like `eat` terminate. Such methods terminate provided they have no infinite loops and have reserved all relevant resources.

One of the difficulties of developing multi-threaded applications is the limited ability to re-use sequential libraries. A naive re-use of library classes that have not been designed for concurrency often leads to data races and atomicity violations. The mutual exclusion guarantees offered by SCOOP make it possible to assume a correct synchronization of client calls and focus on solving the problem without worrying about the exact context in which a class will be used. The **separate** keyword does not occur in class `FORK`. This shows how classes (such as `FORK`) written in the sequential context can be re-used without change for concurrent access. Fork methods such as `pickup` and `putdown` are invoked atomically from the `eat` routine (which holds the locks on these forks).

Once we enter the body of the `eat` routine (with a guarantee of locks on the left and right forks and the precondition **`True`**), no other processors can send messages to these forks. They are under the control of this routine. When a philosopher (the client) calls fork procedures (such as `l.pickup`) these procedures execute asynchronously (e.g., the `pickup` routine call is dispatched to the processor handling the fork, and the philosopher continues executing the next instruction). In routine queries such as `l.inuse and r.inuse` at line 22, however, the client must wait for a result and thus must also wait for all previous separate calls on the fork to terminate. This is called *wait-by-necessity*.

2. The Front-end for Java-like Languages

Fair versions of the dining philosophers in pure Java (e.g., see [5, p137]) are quite complex. The low level constructs for synchronization add to the complexity of the code. In addition, the software developer must explicitly implement a fair scheduler to manage multiple resources in the system. The scheduling algorithm for a fair Java implementation in [5] requires an extra 50 lines of dense code in comparison to the SCOOP code of listings 1 and 2.

Is there a way to obtain the benefits of SCOOP programs for languages such as Java and C#? Our design goal in this endeavor is to allow developers to write SCOOP code using their favorite editors and compilers for type checking.

```
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(pre="!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();
      }
   }
}
```

Listing 3: Example of a SCOOP program written in Java

```
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();
      }
   }
```

Listing 4: Example of a SCOOP program written in C#

At the front-end, we can satisfy our design goal by using the meta-data facility of modern languages that support concurrency as shown in listing 3 and in listing 4. The meta-data facility (e.g., annotations in Java and attributes for C#) provide data about the program that is not part of the program itself. They have no direct effect on the operation of the code they annotate. The listings show SCOOP for Java (which we call JSCOOP) and SCOOP for C# (which we call CSCOOP) via the annotation facility of these languages.

In Eiffel, the keyword **require** is standard and in SCOOP is re-used for the guard. Thus only one new keyword (**separate**) is needed. Java and C# languages do not support contracts.

So we need to add two keywords via annotations: **separate** and **await** (for the guard). The front-end uses the Java or C# compiler to do type checking. In addition, the front-end must do number of consistency checks to assure the correct usage of the **separate** construct (see below).

2.1. Syntax Checking

In SCOOP, method calls are divided into two categories:

- A *non-separate call* where the target object of the call is not declared as separate. This type of call will be handled by the current processor. Waiting happens only if the parameters of this method contain references to separate objects (i.e., objects handled by different processors) or there exists an unsatisfied await condition. Before executing a non-separate call, two conditions have to be true: (a) locks should be acquired automatically on all processors handling separate parameters and (b) await condition should be true. Locks are passed down further (if needed) and are released at the end of the call. Therefore, atomicity is guaranteed at the level of methods.
- A *separate call*. In the body of a non-separate method one can safely invoke methods on the separate objects (i.e., objects declared as separate in the parameters). This type of call, where the target of the call is handled by a different processor, is referred to as a separate call. Execution of a separate call is the responsibility of the processor that handles the target object (not the processor invoking the call). Therefore, the processor invoking the call can send a message (call request) to the processor handling the target object and move on to the next instruction without having to wait for the separate call to return. The invoking processor waits for the results only at the point that it is necessary (e.g., the results are needed in an assignment statement) this is referred to as wait-by-necessity.

Due to the differences between the semantics of separate and non-separate calls, it is necessary to check that a separate object is never assigned to a variable that is declared as non-separate. Entities declared as non-separate but pointing to separate objects are called *traitors*. It is important to detect traitors at compile-time so that we have a guarantee that remote objects cannot be accessed except through the official locking mechanism that guarantees freedom from atomicity and race violations.

The checks for traitors must be performed by the front-end. We use the Eclipse plug-in (see Section 5) for JSCOOP to illustrate the front-end which must check the following separateness consistency (SC) rules [3]:

- **SC1**: If the source of an attachment (assignment instruction or parameter passing) is separate, its target entity must be separate too. This rule makes sure that the information regarding the processor of a source entity is preserved in the assignment. As an example, line 9 in Listing 5 is an invalid assignment because its source `x1` is separate but its target is not. Similarly, the call in line 11 is invalid because the actual argument is separate while the corresponding formal argument is not. There is no rule prohibiting attachments in the opposite direction — from non-separate to separate entities (e.g., line 10 is valid).
- **SC2**: If an actual argument of a separate call is of a reference type, the corresponding formal argument must be declared as separate. This rule ensures that a non-separate reference passed as actual argument of a separate call be seen as separate outside the processor boundary. Let's assume that method `f` of class `X` (line 22 in Listing 5) takes a non-separate argument and method `g` takes a separate argument (line 23). The client is not allowed to use its non-separate attribute `a` as an actual argument of `x.f` because

from the point of view of `x`, `a` is separate. On the other hand, the call `x.g(a)` is valid.

- **SC3**: If the source of an attachment is the result of a separate call to a function returning a reference type, the target must be declared as separate. If query `q` of class `X` returns a reference, that result should be considered as separate with respect to the client object. Therefore, the assignment in line 34 is valid while the assignment in line 35 is invalid.

```
 1  public class Client{
 2     public static @separate X x1;
 3     public static X x2;
 4     public static A a;
 5
 6     public void sc1(){
 7           @separate X x1 = new X();
 8           X x2 = new X();
 9           x2 = x1; // invalid: traitor
10           x1 = x2; // valid
11           r1 (x1); // invalid
12     }
13
14     public void r1 (X x){}
15
16     public void sc2(){
17           @separate X x1 = new X();
18           r2 (x1);
19     }
20
21     public void r2 (@separate X x){
22           x.f(a); // invalid
23           x.g(a); // valid
24     }
25
26     public void sc3() {
27            @separate X x1 = new X();
28            s (x1);
29     }
30
31     public void s (@separate X x){
32           @separate A res1;
33           A res2;
34           res1 = x.q; // valid
35           res2 = x.q; // invalid
36     }
37  }
```

Listing 5: Syntax checking

A snapshot of the tool illustrating the above compile errors is shown in Fig. 6 on page 16.

3. The Core Library for the Back-end

In this section we focus on the design and implementation of the core library classes shown in Fig. 1; these form the foundation of the SCOOP implementation. The core library provides support for essential mechanisms such as processors, separate and non-separate calls, atomic locking of multiple resources, wait semantics, wait-by-necessity, and fair scheduling.

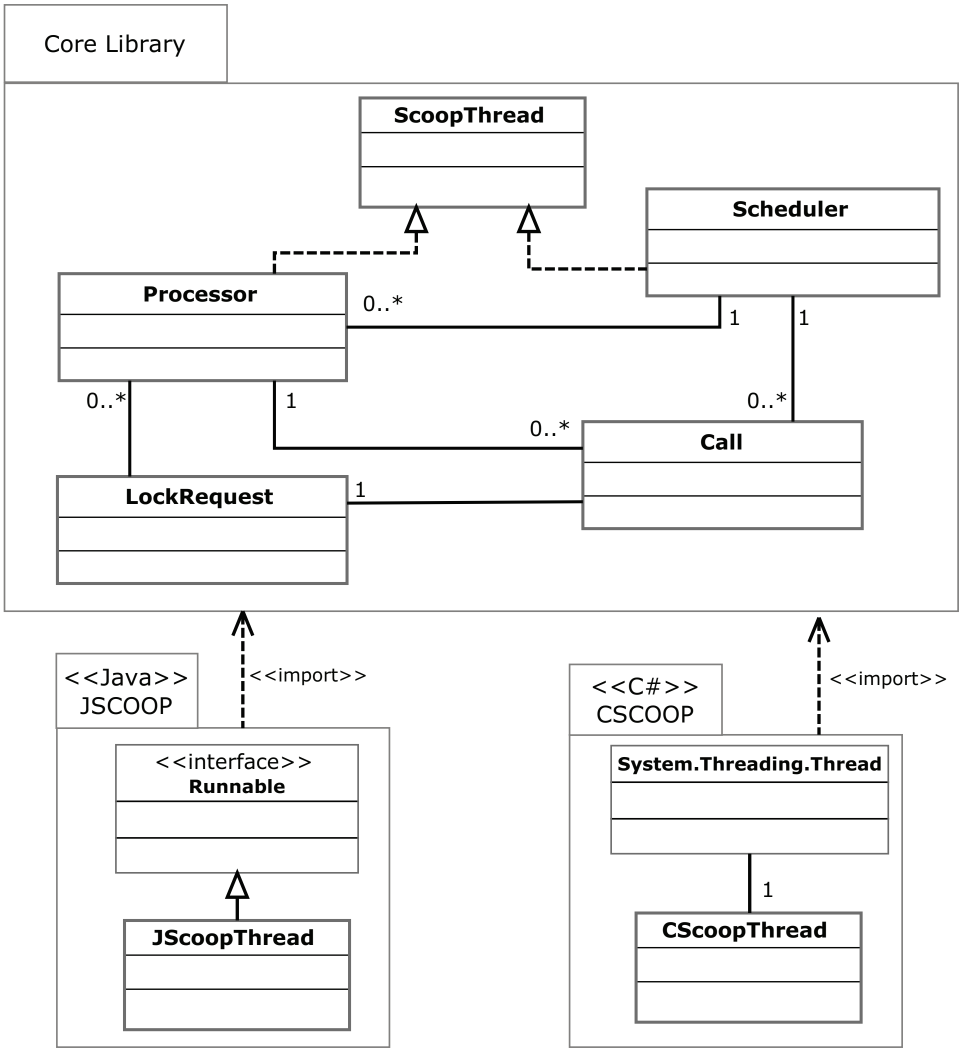


Figure 1. Core library classes associated with the SCOOP design pattern (see Fig. 9 in the appendix for more details)

Method calls on an object are executed by its processor. Processors are instances of the `Processor` class. Every processor has a *local call stack* and a *remote call queue*. The local stack is used for storing non-separate calls and the remote call queue is used for storing calls made by other processors. A processor can add calls to the remote call queue of another processor only when it has a lock on the receiving processor. All calls need to be parsed to extract method names, return type, parameters, and parameter types. This information is stored in the `Call` objects (acting as a wrapper for method calls). The queue and the stack are implemented as lists of `Call` elements.

The `ScoopThread` interface is implemented by classes whose instances are intended to be executed by a thread, e.g., `Processor` (whose instances run on a single thread acting as the "processor" executing operations) or the global scheduler `Scheduler`. This interface allows the rest of the core library to rely on certain methods being present in the translated code. Depending on the language SCOOP is implemented, `ScoopThread` can be redefined to obey the thread creation rules of the supporting language. For example, a Java based implementation of this interface (`JScoopThread`) extends Java's `Runnable` interface.

The `Scheduler` should be instantiated once for every SCOOP application. This instance acts as the global resource scheduler, and is responsible for checking await conditions and acquiring locks on supplier processors on behalf of client processors. The scheduler manages a global lock request queue where locking requests are stored. The execution of a method by a processor may result in creation of a call request (an instance of `Call`) and its addition to the global request queue. This is the job of SCOOP scheduler to atomically lock all the arguments (i.e., processors associated with arguments) of a routine. For example, the execution of `eat` (listing 3) blocks until both processors handling left and right forks have been locked. The maximum number of locks acquired atomically is the same as the maximum number of formal arguments allowed in the language.

Every class that implements the `ScoopThread` interface must define a `run()` method. Starting the thread causes the object's `run()` method to be called in that thread. In the `run()` method, each `Processor` performs repeatedly the following actions:

1. If there is an item on the call stack which has a wait condition or a separate argument, the processor sends a lock request (`LockRequest`) to the global scheduler `Scheduler` and then blocks until the scheduler sends back the "go-ahead" signal. A lock request maintains a list of processors (corresponding to separate arguments of the method) that need to be locked as well as a `Semaphore` object which allows this processor to block until it is signaled by the scheduler.
2. If the remote call queue is not empty, the processor 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, the processor waits for new requests to be enqueued by other processors.

3.1. Scheduling

When a processor `p` is about to execute a non-separate routine `r` that has parameters, it creates a call request and adds this request to the global request queue (handled by the scheduler processor). A call request contains (a) the identity of the requesting processor `p`, (b) list of resources (processors) requested to be locked, (c) the await condition, and (d) routine `r` itself. When the request is added to the global queue, `p` waits (blocks) until the request is granted by the SCOOP scheduler. Requests are processed by the SCOOP scheduler in FIFO order. The scheduler first tries to acquire locks on the requested processors on behalf of the requesting processor `p`. If all locks are acquired, then the scheduler checks the await condition. According to the original description of the algorithm in [3] if either of (a) acquiring locks on processors or (b) wait condition fail, the scheduler moves to the next request, leaving the current one intact. If both locking and wait condition checking are successful, the request is granted giving the client processor the permission to execute the body of the associated routine. The client processor releases the locks on resources (processors) as soon as it terminates executing the routine.

3.2. Dynamic Behaviour

In order to demonstrate what happens during a separate or non-separate call, let's consider the JSCOOP code in Listing 6. We assume that processor *A* (handling `this` of type `ClassA`) is about to execute line 6. Since method `m` involves two separate arguments (`arg-B` and `arg-C`), *A* needs to acquire the locks on both processors handling objects attached to these arguments (i.e., processors *B* and *C*).

Fig. 2 is a sequence diagram illustrating actions that happen when executing `m`. First, *A* creates a request asking for a locks (*lock-semaphore-B* and *lock-semaphore-C*) on both processors handling objects attached to arguments and sends this request to the global scheduler. *A* then *blocks* on another semaphore (*call-semaphore-A*) until it receives the "go-ahead" signal from the scheduler.

The scheduler acquires both *lock-semaphore-B* and *lock-semaphore-C* in a fair manner and safely evaluates the await condition. If the await condition is satisfied, then the scheduler

```
 1 public class ClassA
 2 {
 3    private @separate ClassB b;
 4    private @separate ClassC c;
 5      ...
 6    this.m (b, c);
 7      ...
 8    @await(pre="arg-B.check1()&&arg-c.check2()")
 9    public void m(@separate ClassB arg-B, @separate ClassC arg-C){
10       arg-B.f(arg-C); // separate call (no waiting)
11       arg-C.g(arg-B); // separate call (no waiting)
12       ...
13    }
14      ...
15 }
```

Listing 6: Example of a non-separate method `m` and separate methods `f` and `g`

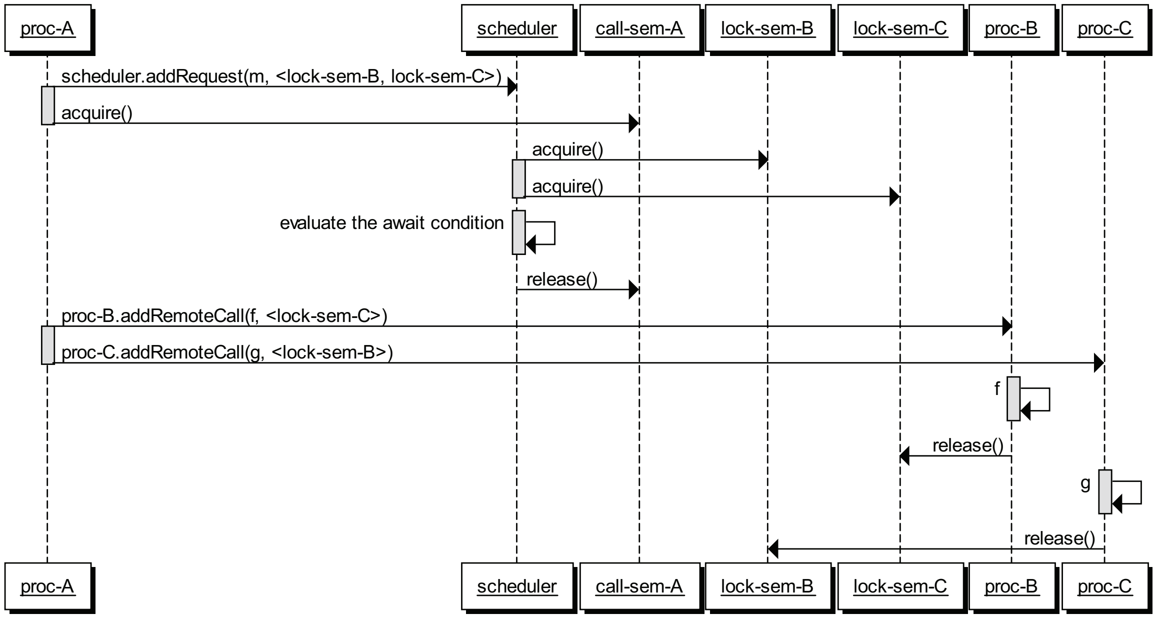


Figure 2. Sequence diagram showing a non-separate call followed by two separate calls involving three processors A, B and C (processor is abbreviated as *proc* and semaphore as *sem*)

releases *call-semaphore-A* signaling processor *A* to continue to the body of `m`. *A* can then safely add separate calls `f` and `g` (lines 10 and 11) to the remote call queue of *B* and *C* respectively and continue to the next instructions without waiting for `f` and `g` to return.

The call semaphore described above is also used for query calls, where wait-by-necessity is needed. After submitting a remote call to the appropriate processor, the calling class blocks on the call semaphore. The call semaphore is released by the executing object after the method has terminated and its return value has been stored in the `call` variable. Once the calling object has been signaled, it can retrieve the return value from the `Call` object.

What happens if the body of method `f` has a separate call on object attached to `arg-C` (i.e., processor *B* invoking a call on *C*)? Since *A* already has a lock on *C*, *B* cannot get that lock and has to wait until the end of method `m`. In order to avoid cases like this, the caller processor "passes" the needed locks to the receiver processor and let the receiver processor release those locks. Therefore, *A* passes the lock that it holds on *C* to *B* allowing *B* to continue safely. Locks are passed down further (if needed) and are released at the end of the call.

4. Translation Rules

In this section we describe some of the SCOOP translation rules for Java-like languages. These rules take the syntax-checked annotated code (with **separate** and **await**) as their input and generate pure multi-threaded code (without the annotation) as their output. The output uses the core library facilities to create the dynamic behavior required by the SCOOP model.

All method calls in SCOOP objects must be evaluated upon translation to determine whether a call is non-separate, separate, requires locks, or has an await condition. If the call is non-separate and requires no locks (i.e., does not have any separate arguments) and has no await condition, it can be executed as-is. However, if the call is separate, requires locks, or has an await, it is flagged for translation. All classes that use separate variables or have await conditions, or are used in a separate context are marked for translation. In addition, the main class (entry point of the application) is marked for translation. For each marked class `A`, a corresponding SCOOP class `SCOOP_A` is created which implements the `ScoopThread` interface.

4.1. Non-separate Method

Fig. 3 shows the rule for translating non-separate methods taking separate argument(s). To make the rule more generic, we use variables (entities starting with %). The method call (*%Method*) is invoked on an entity (*%VAR2*) which is not declared as separate. This call is therefore a non-separate call and should be executed by the current processor. The call takes one or more separate arguments (indicated as *%SepArg0* to *%SepArgN*) and zero or more non-separate arguments (indicated as *%Arg0* to *%ArgN*). The method may have the return value of type (*%ReturnType*).

The SCOOP model requires that all formal separate arguments of a method have their processors locked by the scheduler before the method can be executed. In order to send the request to the scheduler, this processor needs to wrap the call and the required resources in a `Call` object. The creation of a `Call` wrapper `scoop_call` is done in line 20. In order to create such a wrapper we need to (1) create a `LockRequest` object that contains the list of locks required by this call, i.e., processors handling objects attached to the arguments (lines 3–8), (2) capture the arguments and their types (lines 12–18), and (3) capture the return type of this call. We can assume that the classes corresponding to separate arguments are marked for translation to *%SCOOP_Type0* to *%SCOOP_TypeN*. This allows us to access the remote call queues of the processors associated with these arguments.

The new `Call` object is added to the stack of this processor (line 23) to be processed by the `run()` method (not shown here). Finally, the current processor blocks on its call semaphore (line 24) until it receives the go-ahead signal (`release()`) from the scheduler. The thread can safely enter the body of the method (line 26) after the scheduler issues the signal (the body of the method is translated by the separate-call rule). At the end of the method, the lock semaphores on all locked processors are released (lines 28–32).

4.2. Await Condition

Fig. 4 shows the translation rule that deals with await conditions. The await conditions for all methods of a class are encapsulated in a single boolean method `checkPreconditions` located in the generated *SCOOP_%ClassName*. This method is executed only by the scheduler thread (right after the lock semaphores on all requested processors are acquired). The scheduler sets the attribute `call` of this class and then runs the `checkPreconditions`. In this method, each of the await conditions is translated and suited to the *SCOOP_%ClassName* context i.e., separate variables in await clause should be casted to the corresponding SCOOP types. As an example, if we have the await condition `a.m1() == b.m2()` where `a`

Annotated code

```
// Method call on a non-separate object, including method
// calls on 'this'.
Class %ClassName {
  ...
  %ReturnType %VAR1;
  %TargetType %VAR2;
  ...
  [%VAR1 =] %VAR2.%Method(%Type0 %SepArg0,[...,%TypeN %SepArgN,
                          %Type00 %Arg0,...,%TypeNN %ArgN]);
  ...
}
```

Translation

```
Class SCOOP_%ClassName {
  ...
    List<Processor> locks;
    //loop
    locks.add(%SepArg0.getProcessor());
    ... // do this for all separate args
    locks.add(%SepArgN.getProcessor());
    //end loop
    lock_request := new LockRequest(
     %VAR2, locks, this.getProcessor().getLockSemaphore());
    List<Object> args_types, args;
    //loop
    args_types.add(SCOOP_%Type0);
    args.add(%SepArg0);
    ...//do this for all args
    args_types.add(%TypeNN);
    args.add(%ArgN);...
    //end loop
    Semaphore call_semaphore = new Semaphore(0);
    scoop_call := new Call(
     "%Method",  args_types, args,  %ReturnType,
      lock_request, %VAR2, call_semaphore);
    getProcessor().addLocalCall(scoop_call);
    call_semaphore.acquire();
    //call the translated version of %Method (see Rule 3)
    [%VAR1 = ] %VAR2.translated_%Method(%SepArg0,[...,%SepArgN,
                                              %Arg0,...,%ArgN]);
    //loop
    locks[0].unlockProcessor();
    ...//do this for all locks processors
    locks[N].unlockProcessor();
    //end loop
...
}
```

Figure 3. Translation rule for invoking non-separate method taking separate arguments

is a separate variable of type `A` and `b` is a non-separate variable of type `B`, the corresponding await condition translation looks like `((SCOOP_A) a).m1() == b.m2()`. `checkPreconditions` returns true iff the await condition associated with the call evaluates to true.

4.3. Separate Call

This rule describes the translation of a separate method (*%SepMethod*) invoked on a separate target object *%SepArg0* in the body of a non-separate call *%Method*. *%Method* has at least one argument that is declared to be separate (i.e., *%SepArg0*), we can assume that the corresponding classes are marked for translation (i.e., *%SCOOP_Type0*). This allows us to access the remote call queues of the processors associated with those arguments. As in the first rule, we first create a list of processors that needs to be locked for (*%SepMethod*) (lines 8–12),

Annotated code

```
Class %ClassName {
   ...
   //method signature
   %AwaitCondition0
   ... %Method0(%Args0...%ArgsA)
   { ... }
   ...
   %AwaitConditionN
   ... %MethodN(%Args0...%ArgsB)
   { ... }
}
```

Line | Translation

```
 1   Class SCOOP_%ClassName {
 2    {
 3       ...
 4     public boolean checkPreconditions() {
 5       Call this_call = call; //set by the scheduler
 6       String method_name = this_call.getMethodName();
 7       int num_args = this_call.getNumArgs();
 8
 9       //"Reflection" method to check wait condition
10       if(method_name.equals("%Method0")) {
11           if(num_args == A) {
12                   if(Translated_%AwaitCondition0)
13                           return true;
14               else return false;
15           }
16       }
17       ...// this is done for all methods
18       } else if(method_name.equals("%MethodN")) {
19           if(num_args == B) {
20                   if(Translated_%AwaitConditionN)
21                           return true;
22               else return false;
23           }
24       }
25       ...
26   }
```

Figure 4. Translation rule for methods with await condition

and create a lock request from that information (line 13). We collect the arguments and their types in lines 15–22. The `Call` object is finally created in line 23. In order to schedule the call for execution, we add the call object to the end of the remote call queue of the processor responsible for *%SepArg0* (line 25). After the request has been sent, the current thread can safely continue to the next instruction without waiting.

This rule only deals with void separate calls. If the call returns a value needed by the current processor, the current processor has to wait for the result (wait-by-necessity). This is achieved by waiting on the call semaphore. The supplier processor releases the call semaphore as soon as it is done with the calls.

In this section we only showed three of the main translation rules. The rest of the translation rules are available online[1]. A sample translation of a JSCOOP code to Java can be found in the appendix.

[1] http://www.cs.toronto.edu/~faraz/jscoop/rules.pdf

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)
  { ... }
  ...
}
```

⇩

Line	Translation

```
 1  Class SCOOP_%ClassName {
 2    ...
 3    //translated method
 4    %ReturnType %Method(SCOOP_%Type0 %SepArg,...);
 5    { ...
 6      Call call;
 7      List<Processor> locks;
 8      //loop
 9      locks.add(%SepArg0.getProcessor());
10      ...//do this for all separate arguments of %SepMethod
11      locks.add(%SepArgN.getProcessor());
12      //end loop
13      lock_request = new LockRequest(%SepArg, locks,
14         %SepArg.getProcessor().getLockSemaphore());
15      List<Object> args_types, args;
16      //loop
17      args_types.add(%Type00); //use the SCOOP_ for separate types
18      args.add(%Arg0);
19      ...//do this for all arguments of %SepMethod
20      args_types.add(%TypeNN);
21      args.add(%ArgN);...
22      //end loop
23      call = new Call("%SepMethod", arg_types, args,
24                     void, lock_request, %SepArg0, void);
25      %SepArg0.getProcessor().addRemoteCall(call);
26      ...//move on to the next operation without waiting
27    }
28    ...
29  }
```

Figure 5. Translation rule for separate methods

5. Tool Support

JSCOOP is supported by an Eclipse plug-in[2] that provides well-formedness checking, as well as syntax highlighting and other typical features of a development environment. Overall, the plug-in consists of two packages:

1. `edu.jscoop`: This package is responsible for well-formedness and syntax checking and GUI support for JSCOOP Project creation. A single visitor class is used to parse JSCOOP files for problem determination.
2. `edu.jscoop.translator`: This package is responsible for the automated translation of JSCOOP code to Java code. Basic framework for automated translation has

[2] http://www.cs.toronto.edu/~faraz/jscoop/jscoop.zip

been designed with the anticipation of future development to fully implement this feature.

The plug-in reports errors to the editor on incorrect placement of annotations and checks for violation of SCOOP consistency rules. We use the Eclipse AST Parser to isolate methods decorated by an **await** annotation. Await conditions passed to **await** as strings are translated into a corresponding assert statement which is used to detect problems. As a result, all valid assert conditions compile successfully when passed to **await** in the form of a string, while all invalid assert conditions are marked with a problem. For example, on the `@await (pre="x=10")` input, our tool reports "Type mismatch: cannot convert from Integer to boolean." All **await** statements are translated into assert statements, and rewritten to a new abstract syntax tree, which is in turn written to a new file. This file is then traversed for any compilation problems associated with the assert statements, which are then reflected back to the original SCOOP source file for display to the developer.

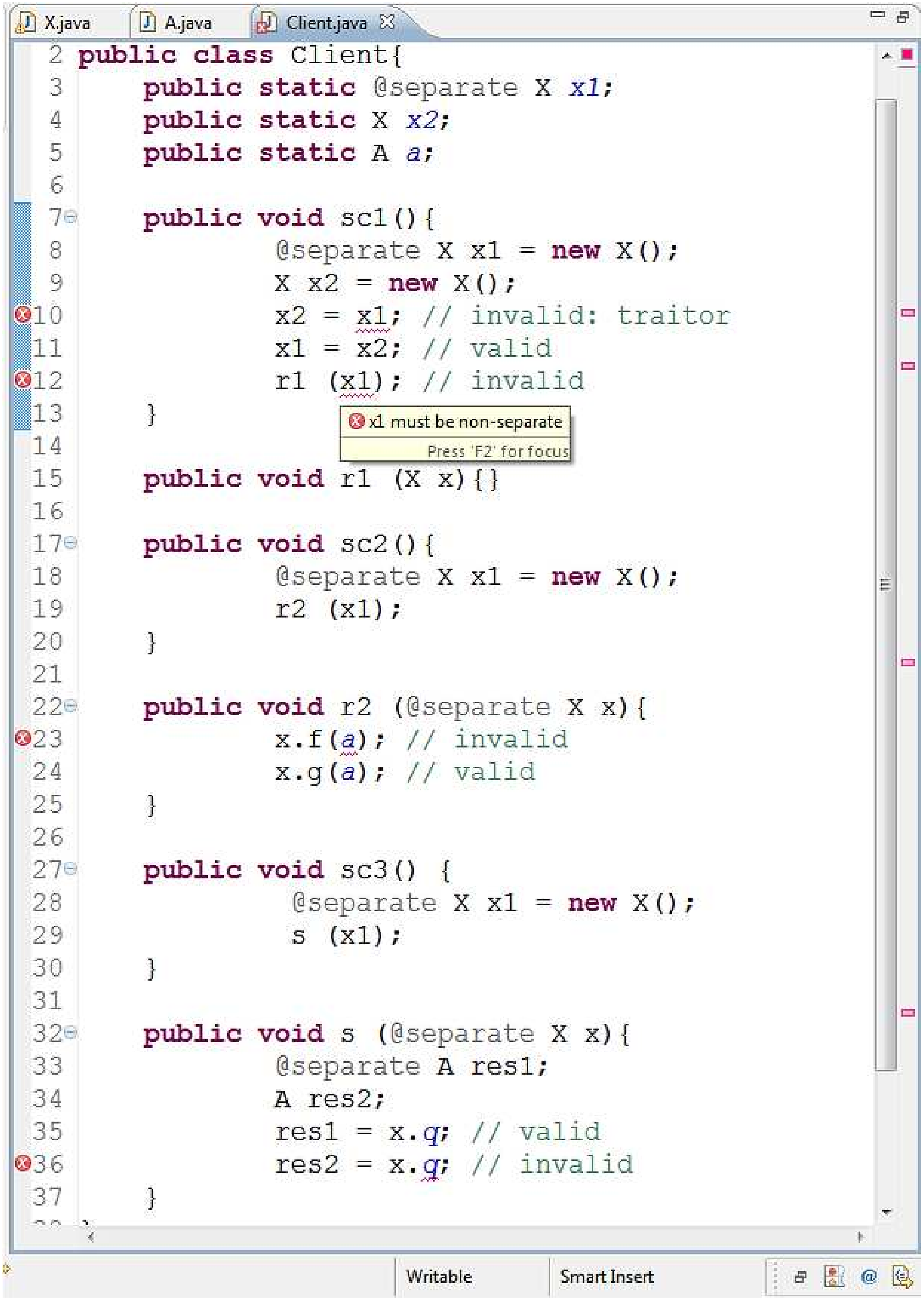
The Eclipse AST Parser is also able to isolate **separate** annotations. We are able to use the Eclipse AST functionalities to type check the parameters passed in the method call by comparing them to the method declaration. Separateness consistency rules SC1 and SC2 are checked when the JSCOOP code is compiled using this plug-in. Fig. 6 is an illustration of these compile time checks. In order to process the JSCOOP annotations, we have created a hook into the Eclipse's JDT Compilation Participant. The `JSCOOP_CompilationParticipant` acts on all JSCOOP Projects, and uses the `JSCOOP_Visitor` to visit and process all occurrences of **separate** and **await** in the source code.

6. Related work

The Java with Annotated Concurrency (JAC) system [6] is similar in intent and principle to JSCOOP. JAC provides concurrency annotations — specifically, **controlled** and **compatible** — that are applicable to sequential program text. JAC is based on an active object model [7]. Unlike JSCOOP, JAC does not provide a wait/precondition construct, arguing instead that both waiting and exceptional behaviour are important for preconditions. Also, via the **compatible** annotation, JAC provides means for identifying methods whose execution can safely overlap (without race conditions), i.e., it provides mutual exclusion mechanisms. JAC also provides annotations for methods, so as to indicate whether calls are synchronous or asynchronous, or autonomous (in the sense of active objects). The former two annotations are subsumed by SCOOP (and JSCOOP)'s **separate** annotation. Overall, JAC provides additional annotations to SCOOP and JSCOOP, thus allowing a greater degree of customisability, while requiring more from the programmer in terms of annotation, and guaranteeing less in terms of safety (i.e., through the type safety rules of [3]).

Similar annotation-based mechanisms have been proposed for JML [8,9]; the annotation mechanisms for JML are at a different level of abstraction than JSCOOP, focusing on method-level locking (e.g., via a new *locks* annotation). Of note with the JML extension of [8] is support for model checking via the Bogor toolset. The RCC toolset is an annotation-based extension to Java designed specifically to support race condition detection [10]. Additionally, an extension of Spec# to multi-threaded programs has been developed [11]; the annotation mechanisms in this extension are very strong, in the sense that they provide exclusive access to objects (making it local to a thread), which may reduce concurrency.

The JCSP approach [12] supports a different model of concurrency for Java, based on the process algebra CSP. JCSP also defines a Java API and set of library classes for CSP primitives, and does not make use of annotations, like JSCOOP and JAC.



```
public class Client{
    public static @separate 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() {
             @separate X x1 = new X();
             s (x1);
    }

    public void s (@separate X x){
            @separate A res1;
            A res2;
            res1 = x.q; // valid
            res2 = x.q; // invalid
    }
}
```

x1 must be non-separate

Press 'F2' for focus

Figure 6. Snapshot of the Eclipse plug-in

Polyphonic C# is an annotated version of C# that supports synchronous and asynchronous methods [13]. Polyphonic C# makes use of a set of private messages to underpin its communication mechanisms. Polyphonic C# is based on a sound theory (the join calculus), and is now integrated in the Cω toolset from Microsoft Research.

Morales [14] presents the design of a prototype of SCOOP's **separate** annotation for Java; however, preconditions and general design-by-contract was not considered, and the support for type safety and well-formedness was not considered.

More generally, a modern abstract programming framework for concurrent or parallel programming is Cilk [15]; Cilk works by requiring the programmer to specify the parts of the program that can be executed safely and concurrently; the scheduler then decides how to allocate work to (physical) processors. Cilk is also based, in principle, on the idea of annotation, this time of the C programming language. There are a number of basic annotations, including mechanisms to annotate a procedure call so that it can (but doesn't have to) operate in parallel with other executing code. Cilk is not yet object-oriented, nor does it provide design-by-contract mechanisms (though recent work has examined extending Cilk to C++). It has a powerful execution scheduler and run-time, and recent work is focusing on minimising and eliminating data race problems.

Recent refinements to SCOOP, its semantics, and its supporting tool have been reported. Ostroff et al [16] describe how to use contracts for both concurrent programming and rich formal verification in the context of SCOOP for Eiffel, via a virtual machine, thus making it feasible to use model checking for property verification. Nienaltowski [17] presents a refined access control policy for SCOOP. Nienaltowski also presents [4] a proof technique for concurrent SCOOP programs, derived from proof techniques for sequential programs. An interesting semantics for SCOOP using the process algebra CSP is provided in [18]. Brooke [19] presents an alternative concurrency model with similarities to SCOOP that theoretically increases parallelism. Some of the open research questions with regards to SCOOP are addressed in [20].

Our approach allows us to take advantage of the recent SCOOP developments and re-use them in Java and C# settings.

7. Conclusion and Future Work

In this paper, we described a design pattern for SCOOP, which enables us to transfer concepts and semantics of SCOOP from its original instantiation in Eiffel to other object-oriented programming languages such as Java and C#. We have instantiated this pattern in an Eclipse plug-in, called JSCOOP, for handling Java. Our initial experience with JSCOOP has been very positive, allowing us to achieve clean and efficient handling of concurrency independently from the rest of the program.

The work reported in this paper should be extended in a variety of directions. First of all, we have not shown correctness of our translation. While the complete proof would indicate that the translation is correct for every program, we can get partial validation of correctness by checking the translation on *some* programs. For example, Chapter 10 of [3] includes many examples of working SCOOP code. We intend to rewrite these programs in JSCOOP and check that they have the same behaviour (i.e., are bi-similar) as programs written directly in Java. These experiments will also enable us to check efficiency (e.g., time to translate) and effectiveness (lines of the resulting Java code and performance of this code) of our implementation of the Eclipse plug-in.

We also need to extend the design pattern and add features to our implementation. Full design-by-contract includes postconditions and class invariants, and we intend to extend our design pattern to allow for these. Further, we need to implement support for inheritance and the full lock passing mechanism of SCOOP [17]. And creating support for handling C# remains future work as well.

SCOOP is free of atomicity violations and race conditions by construction but it is still prone to user-introduced deadlocks (e.g., an **await** condition that is never satisfied). Since JSCOOP programs are annotated Java programs, we can re-use existing model-checkers and theorem-provers to reason about them. For example, we intend to verify the executable code produced by the Eclipse plug-in for the presence of deadlocks using Java Pathfinder [21].

Acknowledgements

We would like to thank Kevin J. Doyle, Jenna Lau and Cameron Gorrie for their contributions to the JSCOOP project. This work was conducted under a grant from Natural Sciences and Engineering Research Council of Canada.

References

[1] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice*. Addison-Wesley, May 2006. ISBN-10: 0321349601 ISBN-13: 978-0321349606.

[2] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.

[3] Piotr Nienaltowski. *Practical framework for contract-based concurrent object-oriented programming, PhD thesis 17031*. PhD thesis, Department of Computer Science, ETH Zurich, 2007.

[4] Piotr Nienaltowski, Bertrand Meyer, and Jonathan Ostroff. Contracts for concurrency. *Formal Aspects of Computing*, 21(4):305–318, 2009.

[5] Stephen J. Hartley. *Concurrent programming: the Java programming language*. Oxford University Press, Inc., New York, NY, USA, 1998.

[6] Klaus-Peter Lohr and Max Haustein. The JAC system: Minimizing the differences between concurrent and sequential Java code. *Journal of Object Technology*, 5(7), 2006.

[7] Oscar Nierstrasz. Regular types for active objects. In *OOPSLA*, pages 1–15, 1993.

[8] Edwin Rodríguez, Matthew B. Dwyer, Cormac Flanagan, John Hatcliff, Gary T. Leavens, and Robby. Extending JML for modular specification and verification of multi-threaded programs. In *ECOOP*, pages 551–576, 2005.

[9] Wladimir Araujo, Lionel Briand, and Yvan Labiche. Concurrent contracts for Java in JML. In *ISSRE '08: Proceedings of the 2008 19th International Symposium on Software Reliability Engineering*, pages 37–46, Washington, DC, USA, 2008. IEEE Computer Society.

[10] Cormac Flanagan and Stephen Freund. Detecting race conditions in large programs. In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 90–96, New York, NY, USA, 2001. ACM.

[11] Bart Jacobs, Rustan Leino, and Wolfram Schulte. Verification of multithreaded object-oriented programs with invariants. In *Proc. Workshop on Specification and Verification of Component Based Systems*. ACM, 2004.

[12] Peter H. Welch, Neil Brown, James Moores, Kevin Chalmers, and Bernhard H. C. Sputh. Integrating and extending JCSP. In *CPA*, pages 349–370, 2007.

[13] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for C#. *ACM Trans. Program. Lang. Syst.*, 26(5):769–804, 2004.

[14] Francisco Morales. Eiffel-like separate classes. *Java Developer Journal*, 2000.

[15] Robert Blumofe, Christopher Joerg, Bradley Kuszmaul, Charles Leiserson, Keith Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *Proc. Principles and Practice of Programming*. ACM Press, 1995.

[16] Jonathan S. Ostroff, Faraz Ahmadi Torshizi, Hai Feng Huang, and Bernd Schoeller. Beyond contracts for concurrency. *Formal Aspects of Computing*, 21(4):319–346, 2009.

[17] Piotr Nienaltowski. Flexible access control policy for SCOOP. *Formal Aspects of Computing*, 21(4):347–362, 2009.

[18] 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.

[19] Phillip J. Brooke and Richard F. Paige. Cameo: an alternative concurrency model for eiffel. *Formal Aspects of Computing*, 21(4):363–391, 2009.

[20] Richard F. Paige and Phillip J. Brooke. A critique of SCOOP. In *First International Symposium on Concurrency, Real-Time, and Distribution in Eiffel-like Languages (CORDIE)*, 2006.

[21] Klaus Havelund and Tom Pressburger. Model checking Java programs using JavaPathfinder. *Software Tools for Technology Transfer (STTT)*, 2(4):72–84, 2000.

Appendix

JSCOOP

```
...
public void act(){
        ...
        eat(leftFork, rightFork);
        ...
}
```

Java

```
...
public void act()
{
        Semaphore j_call_semaphore = new Semaphore(0);
        JSCOOP_Call j_call;
        JSCOOP_Call[] j_wait_calls;
        ...
        //eat(leftFork, rightFork);
        j_req_locks = new LinkedList<JSCOOP_Processor>();
        j_req_locks.add(lefFork.getProcessor());
        j_req_locks.add(rightFork.getProcessor());
        j_lock_request = new JSCOOP_LockRequest(this, j_req_locks,
                this.getProcessor().getLockSemaphore());

        j_arg_types = new Class[2];
        j_arg_types[0] = leftFork.getClass();
        j_arg_types[1] = rightFork.getClass();
        j_args = new Object[2];
        j_args[0] = leftFork;
        j_args[1] = rightFork;
        j_call = new JSCOOP_Call("eat", j_arg_types, j_args,
                null, j_lock_request, this, j_call_semaphore);

        getProcessor().addLocalCall(j_call);

        j_call_semaphore.acquire();

        eat(leftFork, rightFork);
        scheduler.releaseLocks(j_call);
         ...
}
```

Figure 7. Mapping from `Philosopher` to `JSCOOP_Philosopher`: calling `eat` method

JSCOOP

```
@await(pre="!l.isInUse()&&!r.isInUse()")
public void eat(@separate Fork l, @separate Fork r){
        l.pickUp();
        r.pickUp();
        if(l.isInUse() && r.isInUse()) {
            status = 2;
        }...
}...
```

Java

```
public boolean checkPreconditions()
{       ...
        if(method_name.equals("eat")) { ...
                if((((JSCOOP_Fork) args[0]).isInUse()==false) &&
                        (((JSCOOP_Fork) args[1]).isInUse()==false))
                        return true;
                else return false;
        }} else if(method_name.equals(...)) { ...
}

public void eat(JSCOOP_Fork l, JSCOOP_Fork r)
{       JSCOOP_Call[] j_wait_calls;
        Semaphore j_call_semaphore = new Semaphore(0);
        JSCOOP_Call j_call;
        //l.pickUp();
        j_req_locks = new LinkedList<JSCOOP_Processor>();
        j_lock_request = new JSCOOP_LockRequest(l, j_req_locks,
                  l.getProcessor().getLockSemaphore());
        j_arg_types = null; j_args = null;
        j_call = new JSCOOP_Call("pickUp", j_arg_types, j_args,
                         null, j_lock_request, l, null);
        l.getProcessor().addRemoteCall(j_call);
        //r.pickUp(); ...
        //if(l.isInUse() && r.isInUse())
         j_wait_calls = new JSCOOP_Call[2];
         j_req_locks = new LinkedList<JSCOOP_Processor>();
         j_lock_request = new JSCOOP_LockRequest(l, j_req_locks,
                l.getProcessor().getLockSemaphore());
         j_arg_types = null;
         j_args = null;
         j_wait_calls[0] = new JSCOOP_Call("isInUse", j_arg_types,
                j_args, Boolean.class, j_lock_request, l,
                j_call_semaphore);
         j_req_locks = new LinkedList<JSCOOP_Processor>();
         j_lock_request = new JSCOOP_LockRequest(r, j_req_locks,
                r.getProcessor().getLockSemaphore());
         j_arg_types = null;
         j_args = null;
         j_wait_calls[1] = new JSCOOP_Call("isInUse", j_arg_types,
                j_args, Boolean.class, j_lock_request, r,
                j_call_semaphore);
         l.getProcessor().addRemoteCall(j_wait_calls[0]);
         r.getProcessor().addRemoteCall(j_wait_calls[1]);
         //(wait-by-necessity)
         j_call_semaphore.acquire(2);
         //Execute the if statement with returned values
         if((Boolean)j_wait_calls[0].getReturnValue() &&
                (Boolean)j_wait_calls[0].getReturnValue())
                status = 2; ...
}
```

Figure 8. Mapping from `Philosopher` to `JSCOOP_Philosopher`

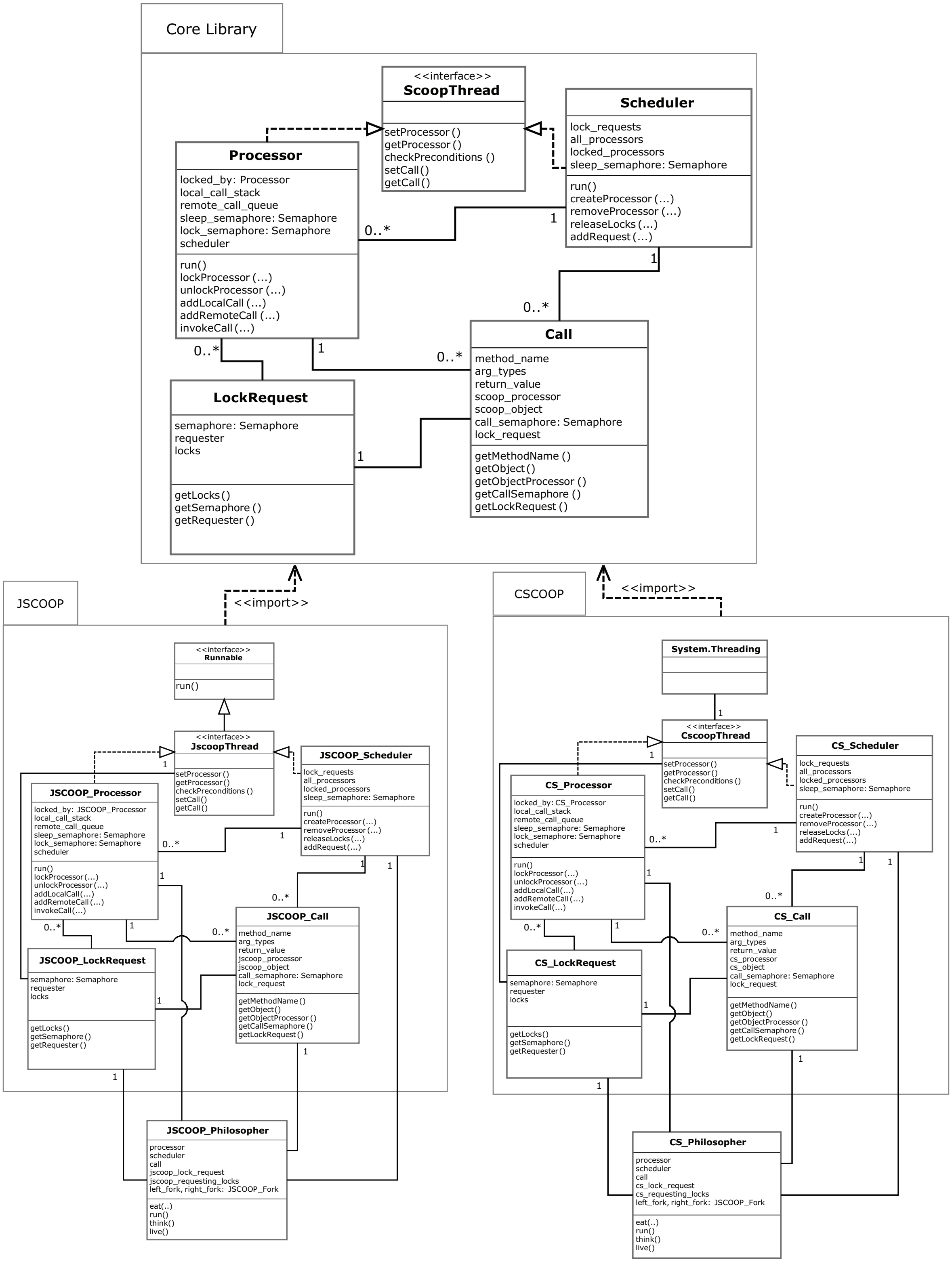


Figure 9. Core library classes