JSCOOP: A High-Level Concurrency Framework for Java

Faraz Ahmadi Torshizi

Jonathan S. Ostroff

Richard F. Paige

Kevin J. Doyle

Jenna Lau

Technical Report CSE-2008-09

December 22, 2008

Department of Computer Science and Engineering

4700 Keele Street Toronto, Ontario M3J 1P3 Canada

JSCOOP: A High-Level Concurrency Framework for Java

Faraz Ahmadi Torshizi[1], Jonathan S. Ostroff[2], Richard F. Paige[3], Kevin J. Doyle[4], and Jenna Lau[4]

[1] University of Toronto, Toronto, Canada, M5S 1A1
faraz@cs.toronto.edu
[2] York University, Toronto, Canada, M3J 1P3
jonathan@cse.yorku.ca
[3] University of York, Heslington, UK, York, YO10 5DD
paige@cs.york.ac.uk
[4] IBM, Toronto, Canada
[kjdoyle,jennalau]@ca.ibm.com

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. It attempts to guarantee fairness via use of a global scheduler. We present a new implementation of SCOOP for Java, called JSCOOP. JSCOOP introduces a new set of annotations modeled after SCOOP keywords, as well as several core library classes which provide the support necessary to implement the SCOOP semantics. A prototype Eclipse plug-in allows for the creation of JSCOOP projects. The plug-in does syntax checking and detects consistency problems at compile time. The use of JSCOOP is demonstrated by an example.

1 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. For example, in multi-core processing, hardware designers are able to produce advanced designs and implementations. The tools available to software designers for writing advanced code that exploits multi-core technology is limited; much of this language and library support is still very low-level, and does not allow programmers to easily exploit, for example, object-oriented (OO) programming techniques – the very techniques that are widely taught and understood by programmers. We argue that there is a gap between the capabilities and needs of modern programmers, and the facilities available for concurrent programming.

The existence of this gap motivates the need for sophisticated languages, tools and techniques for making it easier to build concurrent applications, par-

ticularly by abstracting away from platform details. A general principle for accomplishing this makes use of a *layered* architecture for concurrent applications: the top layer provides programmer-accessible features that abstract away from run-time issues; the bottom layer provides run-time functionality needed for supporting the low-level mechanisms needed for concurrent programming.

Simple Concurrent Object-Oriented Programming (SCOOP) [13, 9] has been developed over the past fifteen years as a high-level framework for concurrent OO programming that supports this conceptual layered architecture. Specifically, SCOOP abstracts from technical details of concurrent programming (e.g., explicit locking when accessing shared objects), while providing a lightweight syntactic extension to sequential object-oriented programming. In particular, SCOOP allows existing sequential code to be used seamlessly in combination with concurrent code while guaranteeing freedom from atomicity and race conditions by construction. This in part reduces the conceptual gap that programmers have to cross when they migrate to concurrent programming, and also allows integration of new code with legacy applications.

SCOOP can be used with a variety of run-time models as illustrated in Fig. 1. Application developers mostly see the higher-level programming layer. Automatic translation occurs under the hood to the underlying runtime. The mapping from conceptual processors to actual CPUs or to distributed processors appears in a configuration file. The configuration data is used to configure the run-time, e.g., by describing the number of logical and physical processors used.

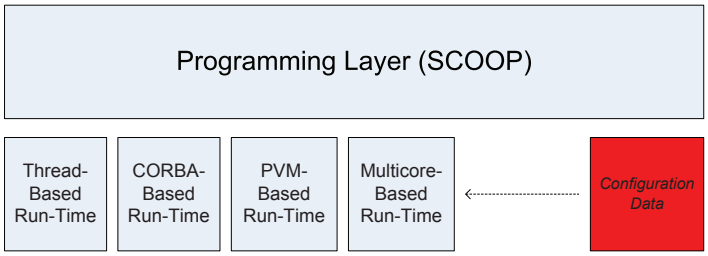

Fig. 1. Layered architecture for concurrent programming

In addition to offering a high-level concurrent programming model, SCOOP has also been designed to provide infrastructure that will help programmers and verification teams establish that their concurrent programs satisfy desirable *correctness* properties – e.g., deadlock freedom, weak/strong fairness, etc. While current SCOOP implementations do not currently guarantee that such properties are exhibited by programs through construction, the SCOOP design and underlying theory has been developed to allow verification of these properties, either through intrinsic capabilities in SCOOP tooling (e.g., the compiler/interpreter) or through interfacing with external verification tools such as model checkers.

1.1 Contribution of this paper

Until now, the SCOOP model has been implemented solely for the Eiffel programming language, and has built on Eiffel's powerful support for correctness-by-construction [8]. This paper contributes a new implementation of SCOOP for Java, thus demonstrating that the SCOOP model is more generally applicable. Moreover, by implementing SCOOP for Java, we provide a more abstract concurrency model to Java programmers, thus ideally making concurrent programming easier to carry out as well as leaving room for application to new hardware designs such as as multi-core.

Specifically, the contributions of this paper are:

- A prototype implementation of the SCOOP concurrency model for Java, called JSCOOP. JSCOOP uses two new keywords **@separate** and **@await** (in the form of annotations) at the programming layer.
- A new library hides the low-level implementation details from the application programmer. A prototype translator maps JSCOOP programs into low-level pure Java programs (using Java's thread mechanism) to execute the JSCOOP program.
- A prototype Eclipse plug-in allows programmers to create JSCOOP projects. The JSCOOP plug-in supports syntax highlighting and compile time consistency checking (for "traitors" that break the concurrency model). The Eclipse plugin uses the library to translate JSCOOP programs to executable code. Compile time errors are reported at the JSCOOP level using the Eclipse API and GUI.

2 Motivating JSCOOP example and SCOOP background

In this section, we provide a motivating example of JSCOOP, and explain the SCOOP model more precisely. The following sections present the technical design and implementation of JSCOOP and its supporting tools.

A snippet of a JSCOOP program for the dining philosophers is shown in Listing 1. Attributes `leftFork` and `rightFork` are declared **separate**, meaning that each fork object is handled by its own processor (i.e., thread of control) separate from the current processor (in this case the thread handling the philosopher object). Method calls from a philosopher to a fork object (e.g., to pick the fork up) are handled by the fork's dedicated processor in the order that the calls are received.

Philosophers deadlock when forks are picked up one at a time by competing philosophers. Application programmers may avoid deadlock by recognizing that two forks must 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 underlying runtime ensures that all separate arguments of methods are reserved before proceeding to the body of the method. In addition, the runtime will check that the **await** condition holds

before allocating the requested resources. In our case, the wait condition asserts that both forks must not be in use.

```
1  public class Philosopher {
2     private @separate Fork rightFork;
3     private @separate Fork leftFork;
4     private int status; // 0:idle, 1:thinking, 2:eating
5
6     public Philosopher (@separate Fork l, @separate Fork r) {
7        left_fork = l;
8        right_fork = r;
9     }
10
11    @await(pre="!l.isInUse()&&!r.isInUse()")
12    public void eat(@separate Fork l, @separate Fork r){
13       l.pickUp(); //separate call
14       r.pickUp();
15       if(l.isInUse() && r.isInUse()) {
16          status = 2;
17       }
18       l.putDown();
19       r.putDown();
20       if(!l.isInUse() && !r.isInUse()) {
21          status = 0;
22       }
23    }
24
25    public void live(){
26       while(true) {
27          status = 1;
28          eat(left_fork, right_fork); //non-separate call
29       }
30    }
31 }
```

Listing 1. Example of a JSCOOP program: `Philosopher`

A scheduling algorithm hidden from the application programmer ensures that resources are fairly allocated. Thus, philosophers must 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.

The example in Listing 1 thus illustrates some of the simplicity of the programming layer while providing guidance in reasoning about deadlock avoidance. The code in the listing does not need to use lower level thread and synchronization mechanisms such as monitors or semaphores. The underlying JSCOOP runtime scheduling algorithm also removes the burden on programmers of producing fair scheduling code themselves. A fair textbook example [5, p137] of the dining philosophers example is shown in the appendix which allows for comparsion between ordinary Java and JSCOOP.

JSCOOP makes it easier to reuse of sequential libraries. One of the problems that makes it hard to develop multithreaded applications is the limited ability to reuse sequential libraries. A naive reuse of library classes that have not

been designed for concurrency often leads to data races and atomicity violations. The mutual exclusion guarantees offered under the hood makes it possible to assume a correct synchronisation of client calls and focus on solving the problem without bothering about the exact context in which a class will be used [13]. The `Fork` class (Listing 2) is an example of a sequential library, i.e., it has no `@await` or `@separate` annotations and is used by the `Philosopher` class. The main class of this system is shown in Listing 3. In line 4 we used the specialized separate annotation for arrays to create an array object. In SCOOP, arrays can be treated as separate objects. Likewise, their contents can also be declared as separate. This results in the following options: a non-separate array with non-separate elements, a non-separate array with separate elements, a separate array with non-separate elements, and a separate array with separate elements.

In order to provide the flexibility of declaring one of the four array specifications listed above, a *directive* (dir) has been added to the `@separate` annotation. Developers are then free to use standard array notation with this added directive. The directive corresponding the above options is as follows:

1. Object[] o = new Object[#]
2. @separate(dir="Object[@separate] o") Object[] o = new Object[#]
3. @separate(dir="@separate Object[] o") Object[] o = new Object[#]
4. @separate(dir="@separate Object[@separate] o") Object[] o = new Object[#]

Therefore, Line 4 creates an array object (handled by the current processor) that contains five separate fork elements (handled by other processors). Line 7 creates a separate fork object causing creation of a new thread in the system. Similarly, lines 10–16 create an array that contains five separate philosophers. Method `startPhilosopher` starts individual philosophers.

```
1  public class Fork {
2     private Boolean in_use;
3
4     public Fork() {
5        in_use = false;
6     }
7
8     public void putDown() {
9        in_use = false;
10    }
11
12    public void pickUp() {
13       in_use = true;
14    }
15
16    public boolean isInUse() {
17       return in_use;
18    }
19 }
```

Listing 2. The `Fork` class

```
1  public class DiningPhilosophers {
2     public static void main(String[] args) {
3
4        @separate(dir="Fork[@separate] forks") Fork[] forks = new Fork[5];
5        //creating separate forks
6        for(int i = 0; i < 5; i++) {
7           @separate Fork fork = new Fork();
8           forks[i] = fork;
9        }
10       @separate(dir="Philosopher[@separate] philosophers")
11          Philosopher[] philosophers = new Philosopher[5];
12       //creating separate philosophers
13       for(int i = 0; i < 5; i++) {
14          @separate Philosopher phil =
15                          new Philosopher(forks[i],forks[(i+1)%5]);
16          philosophers[i] = phil;
17          startPhilosopher(phil); //starting the philosophers
18       }
19    }
20
21    private static void startPhilosopher(@separate Philosopher p) {
22       p.live();
23    }
24 }
```

Listing 3. JSCOOP main class: `DiningPhilosophers`

2.1 Background: the SCOOP model

The SCOOP model was described in detail by Meyer [9], and refined (with a prototype implementation) by Nienaltowski [13]. The overall model presents several useful properties: (a) it guarantees freedom from race conditions and atomicity violations by construction; (b) it guarantees fairness with respect to resource locking; and (c) it provides an easy way to program concurrent object-oriented application by hiding many of the cumbersome lower level implementation details.

The SCOOP model is based on the basic concept of object oriented computation: *the method call* of the form `o.m(a)`, where `m` is a method called on an object `o` passing argument(s) `a`. SCOOP adds the notion of a *processor* (handler) to the above concept. A processor is an abstract notion used to define behavior; operations defined by `m` are executed on `o` by a processor `p`. Processors can be mapped to virtual machine threads, OS threads, or even physical CPUs (in case of multi-core systems). 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 method 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 root (`this`) 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 `o` (e.g., an attribute), a new processor `p` will be created as soon as an instance

of o is created. From that point on, all actions on o will be handled by processor p. Assignment of objects to processors does not change over time.

In SCOOP, method calls are divided into two categories:

– A *non-separate call* where the target of the call is not declared as separate. This type of call will be handled by the current processor. If arguments contain separate objects, locks will be acquired automatically on all separate arguments before the method is executed. Locks are kept during the execution of the method and are released after method is finished. Therefore, atomicity is guaranteed at the level of methods.
– A *separate call*. In the body of a non-separate method, one can safely call another method on an object declared as separate in the arguments of the method. This type of call where the target of the call is separate, is referred to as a separate call. Calls on separate objects will be handled by the processor other than the one handling the current object.

In order to guarantee atomicity and freedom of data races, SCOOP implements automatic locking on all formal arguments of methods. The SCOOP compiler enforces certain rules to achieve this. There are three types of synchronization in SCOOP programs:

1. **Acquiring locks:** SCOOP locks all separate arguments before executing the method. There are different interpretations on what the scheduler should do when only a partial set of locks can be acquired [3].
2. **Wait conditions:** the original SCOOP model in [9] was described in terms of Eiffel [4] which has strong support for Design by Contract [8]. In this model, preconditions become wait conditions.
3. **Wait by necessity:** separate calls execute asynchronously with respect to the client, i.e., the client of a separate call does not need to wait for the call to return and can continue to its next instruction. However, as soon as the client makes a *query call* (i.e. method that does not return void) on the separate object, it must wait for all previous separate calls to finish.

Recent refinements to SCOOP, its semantics, and its supporting tools have been reported. Ostroff et al [17] describe how to use contracts for both concurrent programming and rich formal verification in the context of SCOOP for Eiffel, via a specialised virtual machine, thus making it feasible to use model checking for property verification. Nienaltowski [14] presents a refined access control policy for SCOOP that potentially enable more parallelism and reduce the chances of deadlock. Nienaltowski also presents [12] a proof technique for concurrent SCOOP programs, derived from proof techniques for sequential programs. Brooke [2] presents an alternative concurrency model with similarities to SCOOP that theoretically increases parallelism.

2.2 Related work

The Java with Annotated Concurrency (JAC) system [7] is very similar in intent and principle to our work on JSCOOP. JAC provides concurrency annotations – specifically, the **@controlled** and **@compatible** – that are applicable

to sequential program text. JAC is based on an active object model [16]. 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 [13]).

Morales [11] 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.

SCOOP and JAC differently adopt ideas from active objects, which provide a general purpose form of multitasking. An object can request services asynchronously, but once the request has been made, control returns to the requester. Once the task has completed, the requesting active object is sent the result bu the operating system.

A modern abstract programming framework for concurrent or parallel programming is Cilk [1]; 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 for annotate a procedure call so that it can (but doesn't have to) operate in parallel with other executing code. As well, barrier methods can be provided through annotations so as to provide synchronisation points. 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.

3 JSCOOP Design and Implementation

In this section we focus on the design and implementation of the JSCOOP core library classes shown in Fig. 2; these form the foundation of the JSCOOP 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. The additional part of the implementation is the translation of JSCOOP code to threaded Java;

this is done by JSCOOP's supporting Eclipse plugin (described in the next section) so that the complexity is hidden from the user.

In SCOOP, method calls on each object is executed by its processor. Processors are instances of the `JSCOOP_Processor` class. Every processor has a *local-call stack* and a *remote-call queue*. The local stack is used for storing non-separate calls made by this processor 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. Due to the lack of an agent [10] mechanism in Java, all calls need to be parsed to extract method names, return type, arguments, and argument types. This information is stored in the `JSCOOP_Call` objects (acting as a wrapper for method calls). The queue and the stack are implemented as linked lists of `JSCOOP_Call` elements.

In JSCOOP the `Runnable` interface is implemented by classes whose instances are intended to be executed by a thread, e.g., `JSCOOP_Processor` (whose instances run on a single thread acting as the "processor" executing operations) or the global scheduler `JSCOOP_Scheduler`. In the core library there is also the `JSCOOP_Runnable` interface which extends the `Runnable` interface. This interface allows the rest of the JSCOOP core library to rely on certain methods being present in the translated code, while working around the lack of support for multiple inheritance [10] in Java.

The `JSCOOP_Scheduler` should be instantiated once for every JSCOOP application. This instance acts as the global resource scheduler, and is responsible for checking `@await` conditions and acquiring locks on 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 `JSCOOP_Call`) and its addition to the global request queue. The scheduling algorithm used in the `JSCOOP_Scheduler` is strongly fair with respect to locking resources [13].

Every class that implements the `Runnable` 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 `JSCOOP_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 (`JSCOOP_LockRequest`) to the global scheduler `JSCOOP_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 `JSCOOP_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.

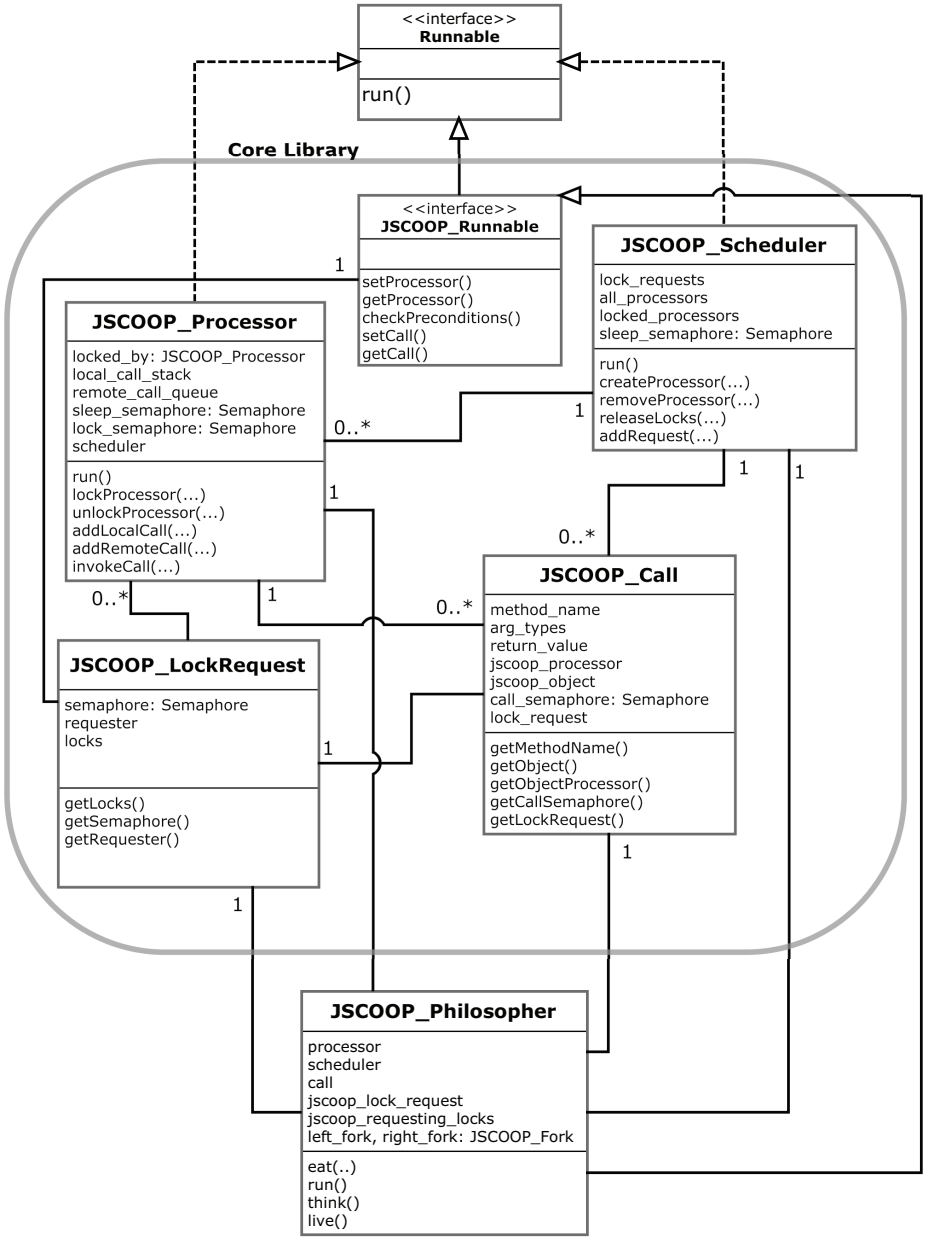

Fig. 2. UML class diagram showing the main elements of JSCOOP

3.1 Synchronization Mechanism

A synchronization mechanism is key to ensuring that the JSCOOP translation is sound. Java provides the **synchronized** keyword to allow synchronization of methods or code fragments on a variable and ensure atomicity of synchronized actions. This is extremely useful, however, synchronizing individual code fragments is not enough to provide full synchronization across a JSCOOP application. Features of the `java.util.concurrent` package were utilized in order to provide the additional synchronization needed. The `Semaphore` class provided the mechanism necessary to signal blocking classes across threads without the need to obtain a lock on the blocking object. Objects are able to block by calling `acquire()` on a `Semaphore` object. Objects are released when some other thread calls `release()` on that semaphore.

Each `JSCOOP_Processor` maintains a *lock semaphore* that is used to protect access to the remote call queue by other processors. A "call request" to the scheduler contains a list of lock semaphores that are needed by the client processor. The required locks are stored in the `JSCOOP_LockRequest` object which is passed to the `JSCOOP_Call` wrapper. The scheduler then grabs the requested lock semaphores on behalf of the client processor and sends the "go-ahead" signal to the client processor. The client continues its execution and releases the lock semaphores as soon as the call is finished.

In addition to the lock semaphore, each `JSCOOP_Runnable` maintains a *call semaphore*. The call semaphore is passed to all method calls made by this processor, and serves multiple purposes. When a call is separate, the client processor is not required to wait for the call to be completed. The call is wrapped and sent to the remote call queue of the supplier processor. The client processor can then continue its execution. However, when a call is non-separate or contain a wait condition, the client processor is required to wait for locks to be granted or for the condition to become true before it can continue. In order to do this, the client processor calls `acquire()` on its call semaphore after submitting a call request. This causes waiting on the client processor side. The "go-ahead" signal by the scheduler releases the call semaphore of the waiting processor.

In order to show what happens in the system during a separate or non-separate call let's consider the JSCOOP code in Listing 4. We assume that *processor-A* (handling an object of type `ClassA`) is about to execute line 6. Since method `m` involves two separate arguments (`arg-B` and `arg-C`), *processor-A* needs to acquire the lock semaphores of both processors handling objects attached to these arguments (e.g. *processor-B* and *processor-C*). Figure 3 is a sequence diagram illustrating actions that happen under the hood to execute `m`. First, *processor-A* creates a request asking for *lock-semaphore-B* and *lock-semaphore-C* (i.e. locks associated with the above arguments) and sends this request to the global scheduler. *processor-A* then *blocks* on its *call-semaphore-A* until it receives the go-ahead signal from the scheduler. Next, the scheduler acquires both *lock-semaphore-B* and *lock-semaphore-C* (if both are available at the same time) and checks the wait condition. If this is successful, then it releases *call-semaphore-A* therefore signaling *processor-A* to continue to the body of `m`.

```
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.checkCondition&&arg-c.checkCondition")
9     public void m(@separate ClassB arg-B, @separate ClassC arg-C){
10       arg-B.f(); // separate call
11       arg-C.g(); // separate call
12       ...
13    }
14      ...
15 }
```

Listing 4. Example of a non-separate method `m` and separate methods `f` and `g`

processor-A can then safely add separate calls `f` and `g` (lines 9 and 10) to the remote call queue of *processor-B* and *processor-C*, respectively. At the end of method call, *processor-A* releases both *lock-semaphore-B* and *lock-semaphore-C* allowing other processors to access the remote call queues.

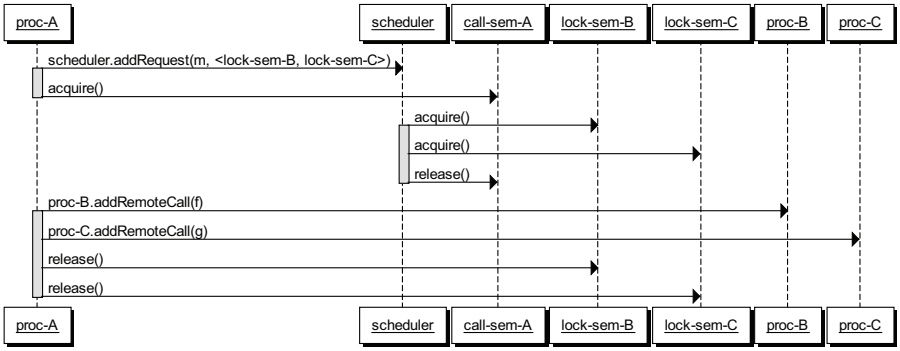

Fig. 3. Sequence diagram showing a non-separate call followed by two separate calls involving three processors A, B and C (processor is abbreviated as *proc* and semaphore as *sem*)

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 `JSCOOP_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 has been stored in the call variable. Once the calling object has been signaled, it can retrieve the return value from the `JSCOOP_Call`.

Although processors do not have to wait for the completion of remote calls, some synchronization is needed to ensure the object performing the execution of the method call has the required call information before the processor con-

tinues to process calls from its queue. With no synchronization, it is possible to skip over the execution of some calls if the next queued call is processed before the first call's execution has taken place. In order to provide synchronization in these cases, `JSCOOP_Processor` can set a *processor semaphore* in the `JSCOOP_Call` it is processing. In the `run()` method of `JSCOOP_Runnable` classes, the `JSCOOP_Call` is pulled from the global variable and stored in a local variable. Once this action takes place, the running object signals the `JSCOOP_Processor` to continue.

Semaphores were also used to eliminate busy waiting from several classes in the core JSCOOP library. Objects such as the `JSCOOP_Processor` and the `JSCOOP_Scheduler` run continuously, processing `JSCOOP_Calls` stored in a private `LinkedList`. When these lists are empty, a *sleep semaphore* is used to block. The sleep semaphore is released with status change (i.e., A new `JSCOOP_Call` is available for processing).

Line	JSCOOP
1	...
2	**public void** live(){
3	...
4	eat(left_fork, right_fork);
5	...
6	}

Line	Java
7	...
8	**public void** live()
9	{
10	Semaphore j_call_semaphore = **new** Semaphore(0);
11	JSCOOP_Call j_call;
12	JSCOOP_Call[] j_wait_calls;
13	...
14	*//eat(left_fork, right_fork);*
15	j_req_locks = **new** LinkedList<JSCOOP_Processor>();
16	j_req_locks.add(left_fork.getProcessor());
17	j_req_locks.add(right_fork.getProcessor());
18	j_lock_request = **new** JSCOOP_LockRequest(**this**, j_req_locks,
19	**this**.getProcessor().getLockSemaphore());
20	
21	j_arg_types = **new** Class[2];
22	j_arg_types[0] = left_fork.getClass();
23	j_arg_types[1] = right_fork.getClass();
24	j_args = **new** Object[2];
25	j_args[0] = left_fork;
26	j_args[1] = right_fork;
27	j_call = **new** JSCOOP_Call("eat", j_arg_types, j_args,
28	**null**, j_lock_request, **this**, j_call_semaphore);
29	
30	getProcessor().addLocalCall(j_call);
31	
32	j_call_semaphore.acquire();
33	
34	eat(left_fork, right_fork);
35	scheduler.releaseLocks(j_call);
36	...
37	}

Fig. 4. Mapping from `Philosopher` to `JSCOOP_Philosopher`: calling `eat` method

3.2 Sample translation

In this section we show samples of the translation of the dining philosopher example (shown in Listing 1) from JSCOOP source code to threaded Java. Figs. 4 and 5 show line by line translation snippets of the `eat` method of the `Philosopher` class to the corresponding Java code `JSCOOP_Philosopher`. The UML class diagram in Fig. 2 shows the relationship between this class and the core library. Due to the space limits we only show the important parts of this translation. For full translation see the appendix.

All method calls in JSCOOP Objects must be evaluated upon translation to determine whether a call is non-separate, separate (remote), requires locks, or has a wait condition. If the call is non-separate, requires no locks, and has no wait condition, it can be executed as-is. However, if the call is separate, requires locks, or has an `@await`, it must be wrapped in a `JSCOOP_Call` object and passed to the appropriate `JSCOOP_Processor`. If the call is non-separate, or requires wait-by-necessity the client processor must be blocked on the call semaphore.

Line 4 in Fig. 5 is an example of a non-separate call `eat` which takes two separate arguments `left_fork` and `right_fork`. The call is non-separate since the processor handling the target of the call (`this`) is the current processor. In order to continue to the body of the `eat` method, the current processor needs to (a) have explicit locks on processors that handle objects attached to `left_fork` and `right_fork` and (b) the wait condition of `eat` must be true. The above two operations are performed by the global scheduler `JSCOOP_Scheduler`. The method call therefore needs to be wrapped as a `JSCOOP_Call` object and passed to the global scheduler. Also this call will eventually executed by the current processor, therefore it needs to be added to the call stack of the current processor.

The creation of a `JSCOOP_Call` wrapper `j_call` is done in line 27 in Fig. 5. In order to create such a wrapper we need to (a) capture the argument types and the return type of this call (lines 21–26) because of the lack of agent mechanism in Java as mentioned above, (b) create a lock request object that contains the list of locks required by this call, i.e., processors handling objects attached to `left_fork` and `right_fork` (lines 15–18), and (c) the call semaphore associated with this processor. The creation of the call request is done at lines 27–28. The call is then added to the stack of this processor at line 30. After this, the current processor blocks on its call semaphore (line 32) to receive the "go-ahead" signal from the scheduler. The method body can safely be executed (line 34) after the scheduler issues the signal. After execution of the `eat` method, the lock semaphores of both arguments are released at line 35.

Fig. 4 shows the translation of the body of `eat` method. As mentioned in the previous section, the `eat` method will only be executed when lock semaphores on both processors handling the left and right forks are acquired by the scheduler and the wait condition is satisfied. The await annotations is translated into a boolean method `checkPreconditions` (lines 10–21 in Fig. 4) which returns true iff the condition is satisfied. This boolean method is called by the global

```
Line | JSCOOP
 1   @await(pre="!l.isInUse()&&!r.isInUse()")
 2   public void eat(@separate Fork l, @separate Fork r){
 3           l.pickUp();
 4           r.pickUp();
 5           if(l.isInUse() && r.isInUse()) {
 6               status = 2;
 7           }
 8           ...
 9   }...
```

```
Line | Java
10   public boolean checkPreconditions()
11   {       ...
12           if(method_name.equals("eat")) {
13                   ...
14                   if((((JSCOOP_Fork) args[0]).isInUse()==false) &&
15                           (((JSCOOP_Fork) args[1]).isInUse()==false))
16                           return true;
17                   else
18                           return false;
19           }} else if(method_name.equals(...)) {
20           ...
21   }
22
23   public void eat(JSCOOP_Fork l, JSCOOP_Fork r)
24   {
25           JSCOOP_Call[] j_wait_calls;
26           Semaphore j_call_semaphore = new Semaphore(0);
27           JSCOOP_Call j_call;
28
29           //l.pickUp();
30           j_req_locks = new LinkedList<JSCOOP_Processor>();
31           j_lock_request = new JSCOOP_LockRequest(l, j_req_locks,
32                    l.getProcessor().getLockSemaphore());
33
34           j_arg_types = null; j_args = null;
35           j_call = new JSCOOP_Call("pickUp", j_arg_types, j_args,
36                           null, j_lock_request, l, null);
37
38           l.getProcessor().addRemoteCall(j_call);
39
40           //r.pickUp();
41            ...
42            //if(l.isInUse() && r.isInUse())
43            j_wait_calls = new JSCOOP_Call[2];
44            j_req_locks = new LinkedList<JSCOOP_Processor>();
45            j_lock_request = new JSCOOP_LockRequest(l, j_req_locks,
46                   l.getProcessor().getLockSemaphore());
47
48            j_arg_types = null;
49            j_args = null;
50            j_wait_calls[0] = new JSCOOP_Call("isInUse", j_arg_types,
51                   j_args, Boolean.class, j_lock_request, l,
52                   j_call_semaphore);
53
54            j_req_locks = new LinkedList<JSCOOP_Processor>();
55            j_lock_request = new JSCOOP_LockRequest(r, j_req_locks,
56                   r.getProcessor().getLockSemaphore());
57
58            j_arg_types = null;
59            j_args = null;
60            j_wait_calls[1] = new JSCOOP_Call("isInUse", j_arg_types,
61                   j_args, Boolean.class, j_lock_request, r,
62                   j_call_semaphore);
63
64            l.getProcessor().addRemoteCall(j_wait_calls[0]);
65            r.getProcessor().addRemoteCall(j_wait_calls[1]);
66
67            //(wait by necessity)
68            j_call_semaphore.acquire(2);
69
70            //Execute the if statement with returned values
71            if((Boolean)j_wait_calls[0].getReturnValue() &&
72                   (Boolean)j_wait_calls[0].getReturnValue())
73                   status = 2;
74            ...
75   }
```

Fig. 5. Mapping from `Philosopher` to `JSCOOP_Philosopher`

scheduler right after the lock semaphores on all arguments are acquired. The reason for using the ad hoc if-else clause in the `checkPreconditions` method (lines 15 and 20) is that the translation code tries to avoid using reflection as much as possible. Using the reflection capabilities of Java makes it harder to use a model checker such as Java pathfinder [6].

The arguments `l` and `r` of type `Fork` are declared to be separate (i.e. they will be running on a different processor). The `Fork` class is therefore translated to the `JSCOOP_Fork` (which implements the `JSCOOP_Runnable` interface). This allows us to access the remote call queues of the processors associated with the forks. The separate call `l.pickUp()` is translated in lines 30–38. A `JSCOOP_Call` is created (in this case with empty list of required locks) and is added to the remote call queue of the processor associated with the left fork (line 38). Since `l.pickUp()` does not return any values and is separate, the current processor does not need to wait for its finish and can continue to the next instruction. The translation of the `r.pickUp()` is not shown for brevity.

Line 5 shows an example of two separate calls to the left and right `Fork` objects `l.isInUser()` and `r.isInUser()` that require wait-by-necessity since they return boolean values. The client processor therefore needs to wait until calls are finished and then evaluate the condition of the `if` statement. Since both calls are separate, their translation (lines 43–65) is very similar to `l.pickUp()` (i.e., call requests are created and added to the remote call queues of the processors handling `l` and `r`). The main difference is that the client processor is forced to wait for the results. This is achieved by calling acquire(2) (line 68) on the call semaphore causing this processor to wait. The supplier processors call `release()` on the call semaphore as soon as they are done with the calls causing this processor to continue its execution. The `if` statement is translated in lines 70–73 where the current processor can safely check the returned values. The returned value is stored in the `JSCOOP_Call` object.

4 Eclipse plug-in

Due to the difference 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 executed except through the official locking mechanism that guaranteed freedom from atomicity and race violations. These checks for traitors and other syntax checking are done by the Eclipse plug-in.

To eliminate the traitors, the SCOOP model provides the following separateness consistency rules (SC for short) [15].

– **SC1**: If the source of an attachment (assignment instruction or argument passing) is separate, its target entity must be separate too. This rule makes

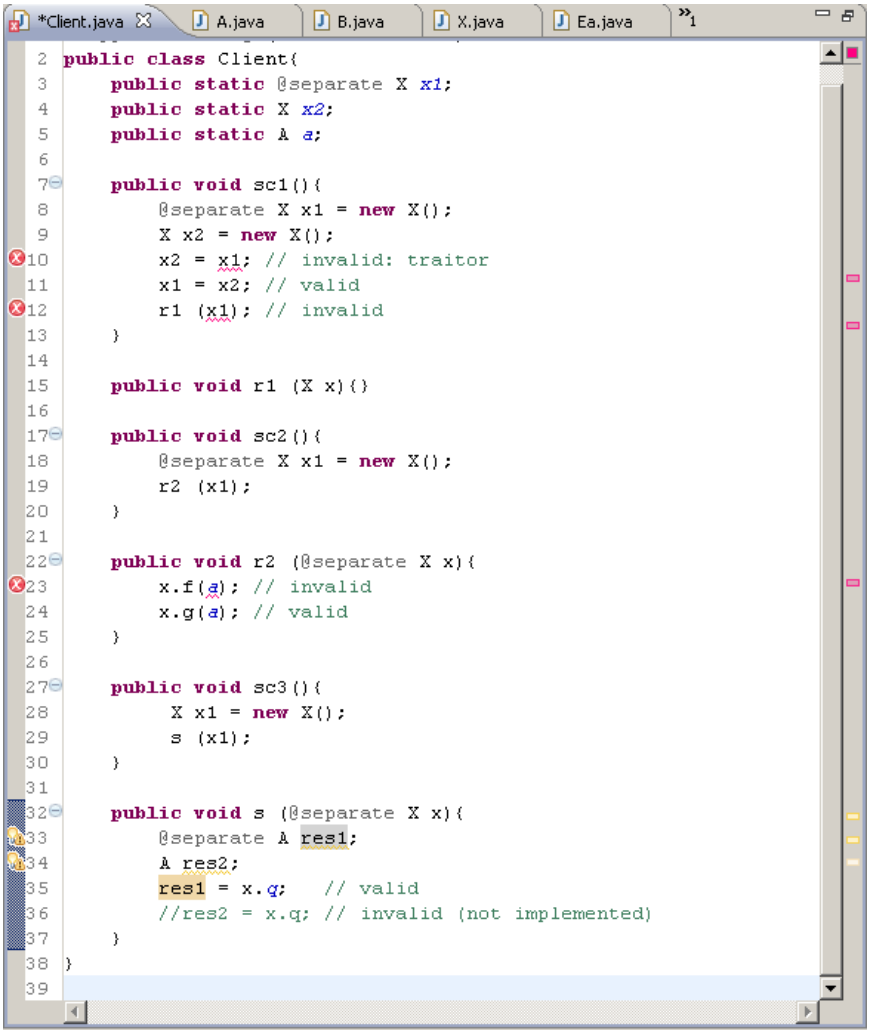

Fig. 6. Consistency rules

sure that the information regarding the processor of a source entity is preserved in the assignment. As an example, line 10 in Fig. 6 is an invalid assignment because its source `x1` is separate but its target is not. Similarly, the call in line 12 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 nonseparate to separate entities (e.g. line 11 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 23 in Fig. 6) takes a non-separate argument and method `g` takes a separate argument (line 24). The client is not allowed to use its non-separate attribute a as 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 function `q` of class `X` returns a reference, that result should be considered as separate with respect to the client object. Therefore, assignment in line 35 is valid while the assignment in line 36 is invalid.
- **SC4**: If an actual argument or result of a separate call is of an expanded type, its base class may not include, directly or indirectly, any non-separate attribute of a reference type. This rule is not applicable to Java since there is no way for the user to define an expanded class.

A prototype Eclipse plug-in provides syntax checking for the input JSCOOP code. The plug-in consists of two packages.

1. `edu.yorku.jscoop`: This package is responsible for correctness checking and GUI support for JSCOOP Project creation. A single visitor class is used to parse JSCOOP files for problem determination
2. `edu.yorku.jscoop.translator`: This package is responsible for the automated translation of JSCOOP code to Java code. Basic framework for automated translation has 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, incorrect use of directive (dir), and partial checks for violation of SCOOP consistency rules. We use the Eclipse AST Parser to isolate methods decorated by an `@await` annotation. Wait 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."

The Eclipse AST Parser is also able to isolate `@separate` annotations. We are able to use the Eclipse AST functionalities to type check the arguments passed in the method call by comparing them to the method declaration. Separateness correctness rules SC1 and SC2 are checked when the JSCOOP code is compiled using this plug-in. One of the following four error messages is displayed in the Eclipse environment depending on the context of the problem: (a) arg_name must be non-separate, (b) arg_name must be non-separate to caller (c) Type mismatch: cannot convert from separate to non-separate (d) Type mismatch: non-separate return is separate to this class. Fig. 6 is an illustration of these compile time checks.

In order to process the JSCOOP annotations, we have created a hook into the 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. In order to process `@await`, we need to ensure that all preconditions passed as a string to the annotation are valid. In order to do this, we first obtain the abstract syntax tree from the original JSCOOP source files. From there, 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 JSCOOP source file for display to the developer (see Fig. 7).

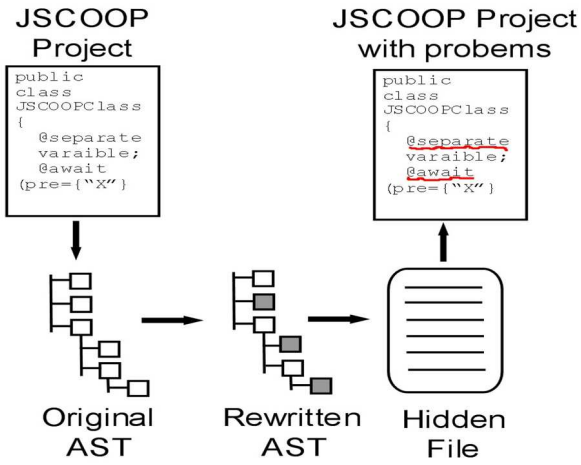

Fig. 7. Processing of `@await` annotations

5 Conclusion and future work

We have presented a new implementation of SCOOP for Java, based on the use of Java annotations. A set of library classes and a preprocessor serve as an implementation, and allow co-existence of annotated and concurrent JSCOOP classes with unannotated and sequential Java classes. We demonstrated the usefulness and simplicity of the JSCOOP annotation approach, and the overall translation from annotated Java to pure Java, that forms the basis of our implementation. We also briefly discussed a set of simple Eclipse-based development tools that help to support the developer in using JSCOOP.

Our implementation of JSCOOP needs further extension to resolve two key fundamental issues: deadlock detection and fairness. We aim to support these

through connections to model checkers and theorem provers. The fact that JSCOOP programs are annotated Java programs means we can exploit existing Java formal methods tools, particularly JML and Java Pathfinder. We intend to provide better integrated Eclipse support for connecting JSCOOP, JML, and Java Pathfinder so that the results of analysis are presented in terms of JSCOOP and its annotations, and the analysis engines are hidden from the user.

A further technical extension is to fully work through the relationship between JSCOOP and class extension in Java; this is not fully implemented in the JSCOOP toolset so far.

References

1. 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.
2. Phillip J. Brooke and Richard F. Paige. Cameo: an alternative concurrency model for eiffel. *Formal Aspects of Computing*, 2009. to appear.
3. 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.
4. ECMA. Eiffel: Analysis, design and programming language. Standard ECMA-367 (2nd edition), June 2006.
5. Stephen J. Hartley. *Concurrent programming: the Java programming language*. Oxford University Press, Inc., New York, NY, USA, 1998.
6. K. Havelund and T. Pressburger. Model checking Java programs using Java pathfinder. *Software Tools for Technology Transfer (STTT)*, 2(4):72–84, 2000.
7. Klaus-Peter L̈ohr and Max Haustein. The JAC system: Minimizing the differences between concurrent and sequential Java code. *Journal of Object Technology*, 5(7), 2006.
8. Bertrand Meyer. Design by Contract. Technical Report TR-EI-12/CO, Interactive Software Engineering Inc., 1986.
9. Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.
10. Bertrand Meyer. Practice To Perfect: The Quality First Model. *Computer*, 1997. Practice To Perfect: The Quality First Model.
11. Francisco Morales. Eiffel-like separate classes. *Java Developer Journal*, 2000.
12. P. Nienaltowski, B. Meyer, and J.S. Ostroff. Contracts for concurrency. *Formal Aspects of Computing*, 2008.
13. Piotr Nienaltowski. *Practical framework for contract-based concurrent object-oriented programming, PhD thesis 17031*. PhD thesis, Department of Computer Science, ETH Zurich, 2007.
14. Piotr Nienaltowski. Flexible access control policy for SCOOP. *Formal Aspects of Computing*, 2008.
15. Piotr Nienaltowski and Bertrand Meyer. Contracts for concurrency. *Formal Aspects of Computing (to appear)*, 2007.
16. Oscar Nierstrasz. Regular types for active objects. In *OOPSLA*, pages 1–15, 1993.
17. Jonathan S. Ostroff, Faraz Torshizi, Hai Feng Huang, and Bernd Schoeller. Beyond contracts for concurrency. *Formal Aspects of Computing*, 2008.

Appendix

A textbook example [5, p137] of a fair Java solution to the dining philosophers is provided below. The example below deals with lower level threads, synchonized blocks, and wait and notify constructs. It uses a dining server and an array of philosopher states to maintain fairness. In JSCOOP, fairness happens "under the hood" with a global synchornizer.

```
1  public class DiningPhilosophers {
2    public static void main(String[] args) {
3      int numPhilosophers = 5;
4      boolean checkStarving = true;
5      DiningServer ds = new DiningServer(numPhilosophers, checkStarving);
6      Philosopher[] philosophers = new Philosopher[numPhilosophers];
7      for (int i = 0; i < numPhilosophers; i++) {
8        philosophers[i] = new Philosopher("Philosopher", i, ds);
9        philosophers[i].start();
10     }
11   }
12 }
```

Listing 5. Deadlock-free version of the main Java class for dining philosophers

```
1  public class Philosopher extends Thread {
2    private int id = 0;
3    private DiningServer ds = null;
4    public Philosopher(String name, int id, DiningServer ds) {
5      this.id = id; this.ds = ds;
6    }
7
8    private void think();
9    private void eat();
10
11   public void run() {
12     while (true) {
13       think();
14       ds.takeForks(id);
15       eat();
16       ds.putForks(id);
17     }
18   }
19 }
```

Listing 6. Deadlock-free version of the Java class Philoşopher

```
class DiningServer {
  private boolean checkStarving = false;
  private int numPhils = 0;
  private int[] state = null;
  private static final int
     THINKING = 0, HUNGRY = 1, STARVING = 2, EATING = 3;

  public DiningServer(int numPhils, boolean checkStarving) {
    this.numPhils = numPhils;
    this.checkStarving = checkStarving;
    state = new int[numPhils];
    for (int i = 0; i < numPhils; i++) state[i] = THINKING;
    System.out.println("DiningServer: checkStarving="
      + checkStarving);
  }

  private final int left(int i) { return (numPhils + i - 1) % numPhils; }
  private final int right(int i) { return (i + 1) % numPhils; }

  private void seeIfStarving(int k) {
    if (state[k] == HUNGRY && state[left(k)] != STARVING &&
        state[right(k)] != STARVING) {
      state[k] = STARVING;
      System.out.println("philosopher " + k + "is STARVING");
    }
  }

  private void test(int k, boolean checkStarving) {
    if (state[left(k)] != EATING && state[left(k)] != STARVING &&
        (state[k] == HUNGRY || state[k] == STARVING) &&
        state[right(k)] != STARVING && state[right(k)] != EATING)
      state[k] = EATING;
    else if (checkStarving)
      seeIfStarving(k); // simplistic naive check for starvation
  }

  public synchronized void takeForks(int i) {
    state[i] = HUNGRY;
    test(i, false);
    while (state[i] != EATING)
      try {wait();} catch (InterruptedException e) {}
  }

  public synchronized void putForks(int i) {
    state[i] = THINKING;
    test(left(i), checkStarving);
    test(right(i), checkStarving);
    notifyAll();
  }
}
```

Listing 7. Deadlock-free version of the Java monitor class