Sequential Object Monitors

Denis Caromel[1] Luis Mateu[1,2] Éric Tanter[2,3]

[1] OASIS project, Université de Nice – CNRS – INRIA
2004, Rt. des Lucioles, Sophia Antipolis, France
denis.caromel@sophia.inria.fr
[2] University of Chile, Computer Science Dept.
Avenida Blanco Encalada 2120, Santiago, Chile
{lmateu,etanter}@dcc.uchile.cl
[3] OBASCO project, École des Mines de Nantes – INRIA
4, rue Alfred Kastler, Nantes, France
etanter@emn.fr

Abstract. Programming with Java monitors is recognized to be difficult, and potentially inefficient due to many useless context switches induced by the `notifyAll` primitive. This paper presents SOM, *Sequential Object Monitors*, as an alternative to programming with Java monitors. Reifying monitor method calls as requests, and providing full access to the pending request queue, gives rise to fully sequential monitors: the SOM programmer gets away from any code interleaving. Moreover, useless context switches are avoided. Finally, from a software engineering point of view, SOM promotes separation of concerns, by untangling the synchronization concern from the application logic.
This paper illustrates SOM expressiveness with several classical concurrency problems, and high-level abstractions like *guards* and *chords*. Benchmarks of the implementation confirm the expected efficiency.

1 Introduction

Programming with Java monitors is hard because the semantics of the operations `wait/notifyAll` is difficult to understand for most programmers, and, even when understood, getting the correct expected behavior can be cumbersome. Moreover, the resulting programs are inefficient because `notifyAll` awakes all waiting threads, triggering lots of thread context switches which are expensive in terms of execution time. Finally, from a software engineering point of view, using Java monitors enforces a *tangling* of the synchronization concern with the application logic.

In this paper we introduce a new concurrency abstraction called SOM, *Sequential Object Monitor*, as an alternative to Java monitors. We developed a 100% pure Java library providing powerful and efficient sequential object monitors. A SOM is a *sequential* monitor in the sense that the execution of a method cannot be interleaved with that of another method: once a method starts executing, it is guaranteed to complete before starting the execution of another method.

We show that SOMs are *(i)* powerful because other high-level synchronization abstractions (e.g., guards, chords) are easily expressed with SOMs, *(ii)* easier to understand and use due to their sequential nature and finally, *(iii)* efficient because they require less thread context switches than standard Java monitors. Performance measurements are provided to support our proposal. Finally, since it is based on a reflective infrastructure, SOM makes it possible to completely separate synchronization specification from application logic, thus achieving a clean separation of concerns [12], promoting reuse of both synchronization and application code.

Section 2 discusses related work in the area of concurrency and establishes the main motivation of our proposal. Section 3 presents SOM, through its main principles, API, and some canonical examples. Section 4 exposes how concurrency abstractions such as guards [6, 13, 17] and chords [4] can be expressed in SOM. Section 5 explores implementation issues, such as the SOM reflective infrastructure, how efficient scheduling is obtained, and finally some benchmarks validating our approach. Section 6 concludes with future work.

2 Related Work and Motivation

Two threads accessing simultaneously a shared data structure can lead the data structure to an inconsistent state. Such a programming error is called a *data race*. To avoid data races, programmers must *synchronize* the access to the shared data structure. In this section we describe the different mechanisms that have been proposed to allow programmers to write thread-safe programs (i.e., programs where data races do not occur).

2.1 Classical Synchronization Mechanisms

Monitors. A monitor is a language-level construct similar to a class declaration. In a monitor, private variables and public operations are declared. The semantics of the monitor ensures that concurrent invocations of operations are executed in mutual exclusion, hence avoiding data races. Monitors were invented by Brinch Hansen [7] and Hoare [16]. These monitors avoid thread context switches by introducing condition variables (thread queues) to explicitly resume only one thread instead of all threads. However, Brinch Hansen states in [8] that such monitors are *baroque and lack the elegance that comes from utter simplicity only.*

Guards. Guards are a simple concept, easy to understand and reason about. The idea of associating a boolean expression to indicate when a given operation may be executed was first introduced for the critical region construct [6]. These boolean expressions evolved to become the guarded commands of [13] and [17]. The main problem with guards is to implement them efficiently, that is, without requiring lots of thread context switches.

Schedulers. The scheduler approach relies on having an entity, called a *scheduler*, that is responsible for determining the order in which concurrent requests to a shared object are performed, similarly to the way an operating system scheduler manages the access to the CPU by concurrent processes. The scheduler approach relates to the *actor* and *active object* models[1], which focus on the separation of coordination and computation (see for instance [1, 15, 3]). They introduce the concept of a *body*: *"a distinguished centralized operation, which explicitly describes the types and the sequence of requests that the object might accept during its activity"* [9]. Such an approach originated in Simula-67 [5], and has been used in several distributed object systems like POOL [2], Eiffel// [10] and, in Java, ProActive [11].

Such approaches are usually in the framework of active entities which implies at least an extra thread for synchronization and extra context switches: a scheduler runs in its own thread of control in an infinite loop. The cost of context switches is not really an issue for systems aiming at parallel programming of distributed memory systems, since the overhead of thread context switches is hidden by network latency. On the other hand such an overhead is a concern for concurrent programming of shared memory multiprocessors.

2.2 Java Monitors

Java is one of the first massively-used languages that includes multi-threaded programming as an integral part of the language. For synchronization, Java offers a flavor of monitors which we will refer to as *Java monitors*. They are inspired from the *critical region* concept invented by Brinch Hansen [6]. The main idea behind the original critical regions is to support the *guard* programming pattern: each operation has an associated guard, a boolean expression which must be true before executing the operation. If the guard is false, the critical region transparently delays the operation until the guard becomes true.

Java monitors are somehow lower level than critical regions because the programmer must *explicitly* test the guard condition before each operation, and must *explicitly* notify waiting threads when guards must be evaluated. Fig. 1 shows the typical code of a guard-like implementation of the `get` method of a bounded buffer.

In Java, a monitor is a normal class definition that includes methods declared with the `synchronized` modifier (1). Concurrent invocations of synchronized methods are executed in mutual exclusion. Conversely, a guard does not have a special syntax construct in Java. It is implemented by a `while` statement where the boolean expression is the (negated) guard condition (2). The programmer must *explicitly* call `wait` (3) to suspend a thread until `notifyAll` is invoked by another thread (5).

This simple example clearly highlights the main disadvantages of the standard Java synchronization mechanism:

[1] The actor model is in a functional setting, while the active object model is rather in imperative object languages.

```
(1) public synchronized Object get()
       throws InterruptedException {
(2)   while (!bufarray.size() > 0)
(3)     wait();
(4)   Object o = bufarray.get();
(5)   notifyAll();
(6)   return o;
    }
```

Fig. 1. Guard-like code for a bounded buffer in Java.

- Java monitors are a low-level abstraction and therefore, programmers are prone to introduce many bugs in their programs: e.g., forgetting to specify the `synchronized` modifier, using an `if` statement instead of a `while` for evaluating guards, wrongly using `notify` instead of `notifyAll`[2] or not invoking `notifyAll` when needed, etc.
- The application functional code (4-6) is tangled with the synchronization concern (1-2-3-5). Tangling non-functional concerns with application code is a violation of the Separation of Concerns (SOC) principle [12], and leads to less understandable, reusable and maintainable code.
- From an efficiency point of view, calling `notifyAll` is inefficient because it awakes all waiting threads. When many threads are waiting on the same lock, this entails a lot of useless, expensive, thread context switches. For instance, in the bounded buffer problem, if many consumers are waiting for an item to be produced, putting a single item in the buffer will awake *all* consumers, although only one of them can get the item (see Sect. 5.5 for benchmarks).
- Programmers frequently disregard multi-threading when defining classes. This entails that there are plenty of useful libraries with classes which are not thread-safe. Making such classes thread-safe, if at all possible, is hard and error prone.

2.3 Recent Proposals

We now review two recent proposals in the area of concurrency, chords [4] and the Java Specification Request 166 [21].

Chords were first introduced in Polyphonic C♯, an extension of the C♯ language. Chords are join patterns inspired by the join calculus [14]. Functional Nets [22] is another example of join-inspired calculus. The vision of Functional Nets as a combination of Functional and Petri-Nets explains well the intrinsic nature of a join pattern: function applications conditioned by the presence of

[2] Recall that `notify` only awakes a single thread, but using it is not recommended because in most cases it introduces subtle race conditions which are very hard to track down.

several inputs. Developed in a functional setting, the absence of state is somehow hidden away by the memorization of tokens (function application) within pending continuations.

Within Polyphonic C♯, a chord consists of a header and a body. The header is a set of method declarations, which may include at most one synchronous method name. All other method declarations are asynchronous events. The chord body is executed only when the chord has been *enabled*. A chord is enabled once all the methods in its header have been called. Method calls are implicitly queued up until they are matched up.

Chords do not address the mutual exclusion problem per se: multiple enabled chords are triggered simultaneously. Although mutual exclusion can be achieved with chords, it must be implemented explicitly and is error-prone. Indeed, to implement mutual exclusion of chord bodies, the programmer must include an additional asynchronous event to represent the idle state, adding it to the header of each chord requiring mutual exclusion and calling it at the end of constructors and chord bodies. Furthermore, there are some classical problems which are difficult to solve with chords, such as implementing a buffer which ensures servicing of get requests in order of arrival.

The Java Specification Request 166 [21] is a proposal for new standard Java abstractions for concurrency. It basically standardizes medium-level constructions, such as synchronizers and concurrent collections, and adds a few native lower-level constructions, such as locks and conditions. The aim behind the JSR 166 is to provide a wide set of constructions so that people can use the appropriate abstractions for a given problem, and hence does not promote any concurrent paradigm in particular. The basic synchronization facility are Hoare's style monitors. These monitors can be more fragile than current Java monitors, because programmers are responsible of explicitly asking and releasing monitors. In fact, the JSR 166 favors flexibility and efficiency at the expense of increased verbosity, with a risk of fragility.

2.4 Motivation

Overall, this paper proposes an alternative for programming concurrency which aims at solving the problems mentioned above:

- **easy**: it should be a high-level and easy-to-use concurrency mechanism, ensuring thread safety;
- **powerful**: it should be expressive enough to support any concurrency abstraction;
- **efficient**: it should avoid useless threads and useless thread context switches;
- **modular**: synchronization code should be specified separately from application code, in order to achieve a clean separation of concerns, and making it easy to "plug" synchronization onto existing, not thread-safe, classes;
- **portable**: the system should be a standard Java library, not requiring a specific virtual machine.

3 Sequential Object Monitors

Sequential Object Monitors, SOMs, are a high-level abstraction inspired by the scheduler approach (Sect. 2.1), intended as an alternative to Java monitors at the programming level. We do not modify the Java language, but instead provide an optional library, so that one can use the right abstraction depending on the problem to tackle and the actual programmer skills.

3.1 Main Ideas

A sequential object monitor, SOM, is a standard object to which a low-cost, thread-less, *scheduler* is attached (Fig. 2a). A SOM does not have its own thread of control, i.e. it is a *passive object*. The functional code of a SOM is a standard Java class in which synchronization does not need to be considered at all. The synchronization code is localized in a separate entity, a scheduler, which implements a *scheduling method* responsible for specifying how concurrent requests should be scheduled. A SOM system makes it possible to define schedulers and to specify which schedulers should be attached to which objects in an application.

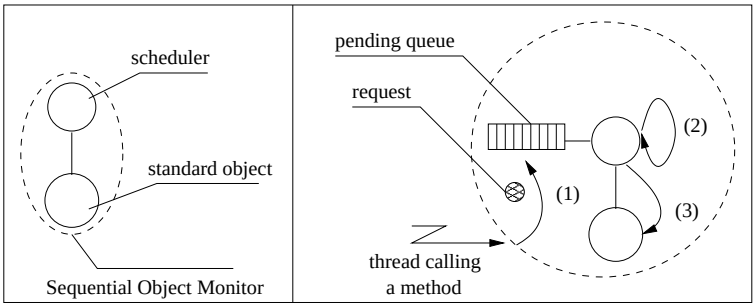


Fig. 2. Structure and operational sketch of a Sequential Object Monitor.

When a thread invokes a method on a monitor, this invocation is reified and turned into a *request* object (Fig. 2b(1)). Requests are then queued in a *pending queue* until they get scheduled by the scheduling method (Fig. 2b(2)). The scheduling method can mark several requests for scheduling. A scheduled request is safely executed (Fig. 2b(3)), in mutual exclusion with other scheduled requests (and the scheduling method).

A scheduling method simply states how requests should be scheduled. Fig. 3 is an example, in pseudo-code, of a scheduling method specifying a classic strategy for a bounded buffer.

A SOM is a *sequential* monitor since considering thread interleaving is not necessary when writing the functional code; a method body is always executed

```
schedule method:
  if buffer empty then schedule oldest put
  elseif buffer full then schedule oldest get
  else schedule oldest
```

Fig. 3. Pseudo-code of a fair scheduling strategy for a bounded buffer. (The equivalent code in SOM is shown later, in Fig. 8)

atomically from begin to end with regards to other invocations. Conversely, in a *quasi-parallel* monitor [18, 9] (e.g., Hoare's, Java monitors) although only one thread can be active at a time, several method activations may coexist. It can make it complex to reason about the program. Fig. 4 summarizes the main principles of SOM.

1. *Any method invocation on a SOM is reified as a* request *and delayed in a pending queue until scheduled.*
2. *The scheduling method is guaranteed to be executed if a request may be scheduled.*
3. *A request is executed atomically with respect to other requests.*

Fig. 4. Main SOM principles.

SOM provides certain guarantees, as listed in Fig. 5. Some of these guarantees are functional, like monitor reentrancy and execution order of scheduled requests. Others have more to do with the thread management strategy of SOM, presented in detail in section 5.3. Recall that it aims at avoiding useless thread context switches.

The SOM model is indeed close to the active object model. The fundamental difference is that a sequential object monitor is a *passive object*, meaning it has no autonomous behavior, no additional scheduling thread: a SOM is much more lightweight than an active object. Nevertheless, the synchronization mechanism of both entities are similar. Compared to existing work carried out in the context of actors and active objects, the specific contribution of SOM rather relates to two specific original points: the sequential nature of synchronizations, for simplicity, and the absence of a synchronization thread, for efficiency.

3.2 Main Entities and API

We now present some elements concerning the entities and the API of the SOM library, in order to go through concrete examples afterwards. In SOM, a scheduler is defined in a class extending from the base abstract class `Scheduler` (Fig. 6). A scheduler must simply define the no-arg `schedule()` method. In this method,

1. *A SOM is reentrant, meaning that any self send on a monitor is executed immediately, without calling the scheduling method.*
2. *Given that the scheduling method can schedule several requests at a time:*
   - *After execution of the scheduling method, the scheduled requests are executed by their calling thread, in the scheduling order.*
   - *The scheduling method will not be called again before* all *scheduled requests complete.*
3. *There is no infinite busy execution of the scheduling method.*
4. *The scheduling method is executed by caller threads, in mutual exclusion. The exact thread executing this method is unspecified.*
5. *After a caller thread has executed its request, it is guaranteed to return at most after one execution of the scheduling method.*
6. *Whenever a SOM is free, if a request arrives and is scheduled by the scheduling method, the request is executed without any context switch.*

Fig. 5. Main SOM guarantees.

the scheduling strategy is defined. The basic idea is that a scheduler can *mark for execution* one or more pending requests, stored in a pending queue. This is called *scheduling (a) request(s)*. Such scheduling decision may be based on requests characteristics as well as the state of the associated base object (passed to the scheduler as a constructor parameter) or any other external criteria.

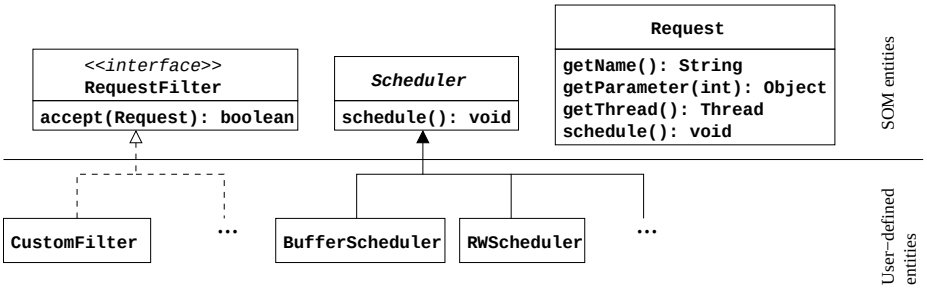


Fig. 6. Main entities provided by SOM, and some potential user-defined extensions.

Various methods are provided for the scheduler to express its scheduling strategies (Fig. 7). For instance, `scheduleOldest("put")` schedules the oldest pending request on method `put`, if any (otherwise it does nothing). Requests that are not scheduled remain in the queue to be scheduled later, on a future invocation of `schedule`. Requests are represented as `Request` objects. A scheduler can obtain an iterator on the pending queue using the `iterator()` method,

scheduling	queue management
`void schedule(Request)`	`Iterator iterator()`
`void scheduleAll`[1]	`boolean hasRequest`[1]
`void scheduleOldest`[1]	`int requestCount`[1]
`void scheduleYoungest`[1]	
`void scheduleOlderThan`[2]	
`void scheduleAllOlderThan`[2]	
`void scheduleYoungerThan`[2]	
`void scheduleAllYoungerThan`[2]	

- [1] Method available in various overloaded versions:
  - `()`: apply to all requests in the queue
  - `(String)/(String[])`: apply to request(s) with given name(s)
  - `(RequestFilter)`: apply to request(s) accepted by filter
- [2] Method available in 2 overloaded versions, taking either two `String` or two `RequestFilter` parameters.

Fig. 7. Scheduler API for scheduling and queue management.

and can then introspect request objects, in arrival order, to determine which one(s) to schedule. Once a request is scheduled, it is removed from the pending queue. Request objects encapsulate the name of the requested method, its actual parameters, and a reference to the calling thread (Fig. 6).

To express elaborated selection scheme, most scheduling methods accept *filters* as an alternative to simple request names. A request filter implements the `RequestFilter` interface (Fig. 6), defining the `accept()` method. For instance, `scheduleAll(rf)` will schedule *all* requests in the queue that are accepted by the `rf` filter, while `scheduleOldest(rf)` will only schedule *one* request, the oldest accepted by `rf` (if any).

Recall that scheduled requests are executed in the scheduling order. To execute requests in the original arriving order, they should simply be scheduled in that order. For instance, ensuring FIFO mutual exclusion with SOM is trivial: it is enough to attach a scheduler whose scheduling method simply calls `scheduleAll()`.

3.3 Canonical Examples

We now briefly present SOM solutions to some classical concurrency problems: bounded buffer, readers and writers, and dining philosophers.

Bounded buffer. Fig. 8 presents the implementation in SOM of a scheduler for the bounded buffer example. It is a straightforward mapping of the pseudo-code shown previously in Fig. 3. Class `Buffer` is a trivial, unsynchronized, implementation of a buffer (not presented). In this implementation, when the buffer is neither full nor empty, the oldest request is scheduled (`scheduleOldest`). Now, imagine we use the following schedule method instead:

```
public class BufferScheduler extends Scheduler {
  Buffer buffer;
  public BufferScheduler(Buffer b) {
    super(b);
    buffer = b;
  }
  public void schedule() {
    if (buffer.isEmpty()) scheduleOldest("put");
    else if (buffer.isFull()) scheduleOldest("get");
    else scheduleOldest();
} }
```

Fig. 8. Scheduler for the bounded buffer example.

```
public void schedule() {
  if (!buffer.isEmpty()) scheduleOldest("get");
  if (!buffer.isFull())  scheduleOldest("put");
}
```

In this case, when the buffer is neither full nor empty, it alternates serving `get` and `put` requests, not respecting the order. This calls for several first comments. The SOM abstraction provides the user with the ability to finely control and tune the synchronization if needed. Of course, higher-level abstractions, potentially with good non-determinism, are also needed. They will be expressed on top of the basic SOM primitives (see Section 4 for guards and chords).

Readers and writers. The readers and writers is another classical problem of concurrent programming. Readers are threads that query a given data structure and writers are threads that modify it. A coordinator object `c` is responsible for granting access to the data structure. Readers request access by calling `c.enterRead()` and notify when they stop accessing data with `c.exitRead()`, while writers use `c.enterWrite()` and `c.exitWrite()` respectively. This problem is easily solved by making the coordinator a sequential object monitor. The code of the solution is presented in Fig. 9. The functional part is the coordinator implementation, which is self-explaining. The code of the scheduler specifies the following strategy. First, `exitRead` and `exitWrite` requests are scheduled immediately and unconditionally, because they are just notifications, not requests for access – similarly for `getReaders` and `isWriting` requests.

If a writer is currently modifying the data structure, the scheduler does not grant other permissions for access. If there are readers accessing the data structure, it grants permission to another `enterRead`, if any. Finally, if there is currently no writer nor reader accessing the data structure, it schedules the oldest request.

```
public class RWCoordinator {
  int readers = 0;
  boolean write = false;

  void enterRead(){readers++;}
  void exitRead(){readers--;}

  void enterWrite(){write = true;}
  void exitWrite(){write = false;}

  int getReaders(){return readers;}
  boolean isWriting(){return write;}
}
```

```
public FairRWScheduler
       extends Scheduler {
  RWCoordinator c;
  // initialized in constructor

  public void schedule() {
    scheduleAll(new String[]{
       "exitRead", "exitWrite",
       "getReaders", "isWriting"});

    if(!c.isWriting()) {
      if(c.getReaders() > 0)
        scheduleOlderThan(
         "enterRead","enterWrite");
      else scheduleOldest();
} } }
```

Fig. 9. The coordinator and its associated scheduler.

Note that readers are scheduled by calling `scheduleOlderThan`, not `scheduleOldest`. This is to ensure that writers may not starve: an `enterRead` request is scheduled *only if it is older* than the first `enterWrite` in the pending queue.

Also, `schedule` only schedules one pending `enterRead` at a time (call to `scheduleOlderThan`). This does not mean that two or more readers cannot work in parallel. Indeed, when finishing the execution of `enterRead`, `schedule` will be reinvoked and another `enterRead` may be scheduled for execution, even if current readers have not called `exitRead`.

Dining philosophers. In this problem, several philosophers (concurrent threads) spend their time thinking and eating. To eat, they first need to get two forks. Fig. 10 shows the code of a philosopher.

A table monitor is used for granting access to two consecutive forks. The solution presented here is fair, meaning no philosopher may starve. Moreover, this solution ensures that forks are granted to philosophers in the same order as they request them. To avoid deadlocks, the table provides a method to atomically request two forks simultaneously (Fig. 11)

The table scheduler (Fig. 12) schedules all non `pick` requests, and all `pick` requests for which both requested forks are free and *none* have been *previously requested* by another philosopher. In the scheduling method, the local variable array `reservedFork`, created every time an iteration over the request queue begins, is used for ensuring that forks are granted in the desired order. When a fork is requested and cannot be granted because it is still busy, it is tagged as "reserved". A request including a previously reserved fork is rejected immediately

```
public class Philosopher implements Runnable {
  int id1; Table table;
  public void run(){
    int id2 = (id1+1)%5;
    for(;;){
      think();
      table.pick(id1, id2); eat(id1, id2); table.drop(id1, id2);
  } }
  void think(){...}  void eat(int id1, int id2){...}
}
```

Fig. 10. The philosopher class.

```
public class Table {
  boolean[] forks = new boolean[5];
  public void pick(int id1, int id2) { forks[id1] = forks[id2] = true; }
  public void drop(int id1, int id2) { forks[id1] = forks[id2] = false; }
  boolean mayEat(int id1, int id2) { return !forks[id1] && !forks[id2]; }
}
```

Fig. 11. The table.

```
public class TableScheduler extends Scheduler {
  Table table; // initialized in constructor

  public void schedule() {
    boolean[] reservedFork = new boolean[5]; // all start false
    Iterator it = iterator();
    while (it.hasNext()) {
      Request req = (Request) it.next();
      if (!req.is("pick")) req.schedule();
      else {
        int id1 = req.getIntParameter(0);
        int id2 = req.getIntParameter(1);
        if (!reservedFork[id1] && !reservedFork[id2] &&
            table.mayEat(id1, id2))
          req.schedule();
        reservedFork[id1] = reservedFork[id2] = true;
  } } } }
```

Fig. 12. The table scheduler.

in the current scan, even if such a fork is free, because the fork must first be granted to the philosopher that first requested it. Of course, less fair strategies can also be easily expressed.

3.4 Modularity and Reuse of Synchronization Policies

SOM makes it easy to define various synchronization policies, thanks to the full access given in the scheduling method to the queue of pending requests. For instance, in the case of the readers and writers problem, several fairness policies can be devised. We already exposed (Fig. 9) a fair policy, where both writers and readers are ensured not to starve. Alternative policies can easily be provided, for instance giving priority to readers or writers (Fig. 13).

```
public WriterPriorityRWScheduler
       extends Scheduler {
  RWCoordinator c;
  // initialized in constructor

  public void schedule() {
    // schedule all notifications

    if(!c.isWriting()) {
     if(hasRequest("enterWrite")){
      if(c.getReaders() == 0)
       scheduleOldest("enterWrite");
     }
     else scheduleAll();
} } }
```

```
public ReaderPriorityRWScheduler
       extends Scheduler {
  RWCoordinator c;
  // initialized in constructor

  public void schedule() {
    // schedule all notifications

    if(!c.isWriting()) {
     if(hasRequest("enterRead"))
       scheduleAll("enterRead");
     else if(c.getReaders()==0)
       scheduleOldest();
} } }
```

Fig. 13. Alternative fairness policies for the readers and writers problem.

Reuse of synchronization policies in different contexts depends on their genericity. As of now, the schedulers we have exposed all depend on string names (e.g., `"put"`). A scheduler class can be made independent from actual method names through configuration. For instance, considering buffer-like containers, reusable policies just need to be configured in order to know which methods are to be considered *put* methods and which ones are *get* methods. Then, the `schedule` method can be made independent of method names, for instance (`putMethod` is an instance variable configured to hold the name of the *put* method):

```
void schedule() { if (buffer.isEmpty()) scheduleOldest(putMethod); ... }
```

Determining emptiness and fullness of the synchronized data structure can also be made generic: the reflection API can be used to invoke emptiness and fullness methods according to configuration. Apart from being reusable, generic synchronization classes are more robust with regards to changes.

4 Concurrency Abstractions with SOM

SOM is equivalent in expressiveness to the classic synchronization mechanisms like locks, semaphores, Hoare's monitors, Java monitors, guards, etc. This means that if a synchronization problem can be solved with the classic mechanisms it can also be solved with SOM and vice versa. To prove it, it is enough to show an implementation of a classic mechanism in terms of SOM and vice versa, because all classic mechanisms are equivalent. This proof is trivial as SOM is implemented in terms of Java monitors and implementing a lock with SOM is easy:

```
class Lock {
  boolean busy = false;
  void ask() {
    busy = true;
  }
  void release() {
    busy = false;
  }
}
```

```
class LockScheduler extends Scheduler {
  Lock lock;
  // initialized in constructor

  void schedule() {
    scheduleAll("release");
    if (!lock.busy)
      scheduleOldest("ask");
} }
```

However, such an equivalence is not enough. It is also important to show that solutions that are easily expressed with other synchronization mechanisms are also easily expressed with SOM. Although effectively proving this property is hard, a good approximation consists in showing how other synchronization mechanisms are easily expressed with SOM. In this section, we have chosen to present the concise implementation of two synchronization mechanisms with SOM: guards and chords.

4.1 Guards

In SOM Guards, a guard scheduler contains method guards and is responsible for scheduling concurrent requests. We provide an abstract class for guard schedulers, `GuardScheduler`, with a method for registering method guards, `addGuard`. A guard is defined by attaching a request filter, that indicates when a method fulfills the conditions to be executed, to a method name. It is worthwhile to highlight that the guard system presented here avoids unnecessary context switches (see Section 5.3 for an explanation of the efficient scheduling strategy of SOM). Fig. 14 illustrates an implementation of the bounded buffer based on SOM Guards. This code simply associates a request filter for `get` and one for `put`.

The expressiveness of SOM is illustrated by the simplicity of the base class `GuardScheduler` (Fig. 15), that completely implements the guard system. The scheduler simply iterates over the request queue and schedules the oldest request whose associated guard evaluates to true. Note that this scheduler is not optimized: the actual implementation of the guard scheduler avoids evaluating all guards upon each invocation of the scheduling method. If an invocation of `schedule` does not schedule any request, the scheduler will not re-evaluate the corresponding guards until a new request arrives *and* is scheduled.

```
public class GuardedBufferScheduler extends GuardScheduler {
  public GuardedBufferScheduler(final GuardedBuffer buf) {
    super(buf);
    addGuard("get",
      new RequestFilter() { public boolean accept(Request req) {
          return !buf.isEmpty();
      }});
    addGuard("put",
      new RequestFilter() { public boolean accept(Request req) {
          return !buf.isFull();
      }});
} }
```

Fig. 14. Code of a guard scheduler for the bounded buffer example.

```
public abstract class GuardScheduler extends Scheduler {
  HashMap guardMap = new HashMap();

  public GuardScheduler(Object o){ super(o); }
  public void addGuard(String name, RequestFilter guard){
    guardMap.put(name, guard);
  }

  public void schedule(){
    Iterator it = iterator();
    while (it.hasNext()){
      Request req = (Request) it.next();
      RequestFilter guard = (RequestFilter) guardMap.get(req.getName());
      if (guard == null || guard.accept(req)){ req.schedule(); break; }
    }
} }
```

Fig. 15. The (non-optimized) guard scheduler implemented with SOM.

4.2 Chords

Chords were first introduced in Polyphonic C♯ [4], a C♯ dialect offering dedicated syntax to define join patterns (see section 2.3).

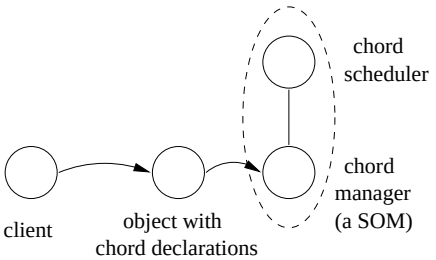
Fig. 16 shows the implementation of a multiple-reader, single-writer lock with chords as exposed in [4]. It consists of just five chords and illustrates pretty well the kind of concise definition enabled by chords. The chords 1 and 2 are *alternative chords*, meaning that the actual `shared` chord being executed depends on the previous asynchronous events that were fired. Chords 3 and 4 are *simple*

chords, i.e. made of a synchronous method that only appears in one chord, and of some asynchronous events (in this case just one), while chord 5 is a standard synchronous method (*trivial chord*).

```
class ReaderWriter {
(1)  public void shared() & async idle() { sharedRead(1); }
(2)  public void shared() & async sharedRead(int n) { sharedRead(n+1); }
(3)  public void releaseShared() & async sharedRead(int n) {
       if(n == 1) idle(); else sharedRead(n-1);
     }
(4)  public void exclusive() & async idle() {}
(5)  public void releaseExclusive() { idle(); }
}
```

Fig. 16. Solution to the readers and writers problem with chords (Polyphonic C♯ code).

We implemented a chord system for Java based on SOM, presenting the same semantics as the chords of Polyphonic C♯. However, SOM Chords are implemented as a library, not as a language extension. Implications of this fact are discussed at the end of this section. The aim of this section is to show how SOM can be simply used to implement other synchronization mechanisms, thereby illustrating its expressiveness.




```
class ChordManager {
  AsyncEvent defineEvent();
  Chord defineChord(..);
  ChordSet defineChordSet(..);

  void trigger(AsyncEvent e, ..);
  void await(ChordSet c);
  boolean isEnabled(ChordSet c);
}
```

Fig. 17. Operational sketch of SOM Chords and public interface of the chord manager.

The main principle of SOM Chords is that an instance of a class declaring chords is associated to a *chord manager* (Fig. 17). A chord manager, instance of `ChordManager`, *is* a sequential object monitor (scheduled by a `ChordScheduler`),

```
class ReaderWriter {
  ChordManager mgr = new ChordManager();
  AsyncEvent idle = mgr.defineEvent();
  AsyncEvent sharedRead = mgr.defineEvent();
  ChordSet idleOrSharedRead = mgr.defineChordSet(idle, sharedRead);

  // public void shared() & async idle() { sharedRead(1); }
  // public void shared() & async sharedRead(int n) { sharedRead(n+1); }
  public void shared() {
    EnabledChord ch = mgr.await(idleOrSharedRead);
    if(ch.is(idle)) mgr.trigger(sharedRead, new Integer(1));
    else {
      int n = ch.getIntParameter(sharedRead);
      mgr.trigger(sharedRead, new Integer(n+1));
  } }

  // public void releaseShared() & async sharedRead(int n) {
  //     if(n == 1) idle(); else sharedRead(n-1); }
  public void releaseShared() {
    EnabledChord ch = mgr.await(sharedRead);
    int n = ch.getIntParameter(sharedRead);
    if(n == 1) mgr.trigger(idle);
    else mgr.trigger(sharedRead, new Integer(n-1));
  }

  // public void exclusive() & async idle() {}
  public void exclusive() { mgr.await(idle); }

  // public void releaseExclusive() { idle(); }
  public void releaseExclusive() { mgr.trigger(idle); }
}
```

Fig. 18. Solution to the readers and writers problem with SOM Chords (Polyphonic C♯ code is given in commentaries).

whereas the object using it *is not* (recall that mutual exclusion of chord bodies is not guaranteed in chords as formulated in [4]).

There are three types of chords in SOM Chords: *asynchronous events* (class `AsyncEvent`); *chords*, which are enabled when a set of asynchronous events is matched up (class `Chord`); and *chord sets*, which are an alternative set of chords, enabled whenever one chord in the set is enabled (class `ChordSet`). Since a chord is indeed a chord set made of a single chord, and similarly an asynchronous event is a chord made of a single event, `AsyncEvent` is a subclass of `Chord` which is a subclass of `ChordSet`.

```
public class ChordScheduler extends Scheduler {
  ChordManager mgr;
  public ChordScheduler(ChordManager mgr) {
    super(mgr); this.mgr= mgr;
  }
  public void schedule() {
    scheduleOldest(new RequestFilter(){
      public boolean accept(Request req){
        // schedule non await methods
        if (!req.is("await")) return true;

        // the chord specified as parameter in the await call
        ChordSet c = (ChordSet) req.getParameter(0);

        // true if all required events have been triggered
        return mgr.isEnabled(c);
      }
    });
} }
```

Fig. 19. The chord scheduler implemented with SOM.

The chord manager makes it possible to define asynchronous events and chords (Fig. 17). The methods `await` and `trigger` make it possible to respectively wait for a chord to be enabled, and to trigger an asynchronous event. The method `isEnabled` checks if a given chord (or chord set) is enabled.

Fig. 18 shows the `ReaderWriter` class implemented with SOM Chords. First of all, a chord manager is associated with each instance. Asynchronous events, chords and chord sets are declared as instance variables, in order to be able to refer to them. Then, a simple chord (i.e., whose synchronous method appears only once in all chords, such as `exclusive`) is expressed as a standard synchronous method that starts by waiting for the set of events to be matched up (`mgr.await(..)`). An alternative set of chords (i.e., that share the same synchronous method, such as `shared-idle` and `shared-sharedRead`) is expressed as one synchronous method that first waits for one of the chords to be enabled. Then, depending on which chord was actually enabled, the appropriate body is executed. An `EnabledChord` object represents an enabled chord instance, and stores the parameters associated with each event of the chord. Finally, method bodies are changed so that they trigger asynchronous events on the manager (e.g., `idle()` is replaced by `mgr.trigger(idle)`).

Fig. 19 shows the straightforward implementation of the chord scheduler controlling the chord manager. The scheduler uses a filter that accepts non-`await` requests, and accepts `await` requests only if all required events have been

previously triggered. To this end, the chord manager uses bit masks to determine if a chord is ready to be processed (`isEnabled`), just as explained in [4].

A major benefit of Polyphonic C♯ is that it is a *language extension*, not a simple API. This brings a number of benefits, in particular compact syntax and compiler support. Also, providing chords as a language extension results in clearer, more manageable code. In this regard, our claim here is that the underlying system supporting chords is concisely expressed with SOM, as validated by the code of the chord scheduler shown in Fig. 19. Obviously, the advantages of a language extension are not met by the SOM Chords library: however the set of transformations from a chord syntax to calls to the library is straightforward and linear. In other words, code of Fig. 18 can preferably be seen as code generated by a chords compiler using the SOM Chords library as its back end, rather than as hand-written code.

5 Implementation

5.1 Reifying and synchronizing method calls

In order to remain portable while being able to transparently reify method calls as requests, the SOM library is based on a reflective infrastructure operating at load time. Using computational reflection also brings a clean separation of concerns between the application logic (at the base level) and the synchronization concern (at the metalevel). The SOM metaobject protocol (MOP) is defined within Reflex, an open behavioral reflective extension of Java [24]. When creating a sequential object monitor, the reflective infrastructure ensures interception of its method invocations. Doing so, a controller metaobject will be invoked *(a)* before requesting a method, and *(b)* just after returning from the method. The controller metaobject, provided by the SOM system, ensures scheduling and mutual exclusion of concurrent requests.

5.2 Initialization and Configuration

There are several alternatives to obtain sequential object monitors. Especially, there is a need for a means to specify the association between *standard objects* and *schedulers*.

First of all, at instantiation time, one can use:

```
Buffer b = (Buffer) SOM.newMonitor(Buffer.class, aScheduler);
```

This runtime approach however suffers from some limitations with `final` classes and methods, due to the fact that it is based on dynamic generation of implicit subclasses, and requires explicit use by programmers in client code (replacing standard `new` calls).

A transparent approach is also available. One can specify in a configuration file the desired associations $baseClass \rightarrow schedulerClass$, e.g.:

```
schedule: Buffer with: BufferScheduler
```

The SOM system offers two means to apply configuration files to an application: load time and compile time. Class `SOMGenerator` is provided as a compile-time utility to generate modified class files:

```
% java SOMGenerator <configuration-file> <target-directory>
```

will generate transformed class files for all classes which have to be scheduled, as specified by the given configuration file. Load-time transformation is also available, thanks to the `SOMRun` class:

```
% java SOMRun <configuration-file> <application>
```

will run the application by applying, at load time, the configuration specified in the configuration file.

5.3 Efficient Scheduling

This section explains the inner working of SOM thread management that makes it possible to avoid unnecessary context switches.

The controller mentioned in the section above handles two queues:

- The *wait queue* holds pending requests that have not been scheduled for execution by the scheduler.
- The *ready queue* keeps pending requests that have just been scheduled (during the last execution of the scheduling method), but have not been executed yet because the monitor is busy, either still executing the scheduling method, or executing another request.

A sequential object monitor M works as follows. Let T be a thread that has begun the execution of a request R on a method of M. Suppose the ready queue already contains requests made by other threads. T is said to *own* M and the other threads *wait*. M is in a *busy* state. When T finishes the execution of the method, the controller takes control and extracts the oldest request R' in the ready queue. Thread T thereby passes the ownership of M directly to thread T', the thread requesting R'. Finally, T wakes T' up and returns to the caller of M. T' starts the execution of its own request, R'.

When the ready queue is empty, the controller makes thread T automatically invoke the `schedule` method of the user-provided scheduler. Recall that this method will schedule one or several requests; these requests will be transferred from the wait queue to the ready queue. Making T invoke the scheduling method implies that T spends some time scheduling requests for other threads. Thus programmers should preferably write simple scheduling methods. If after invoking the scheduling method, the ready queue is still empty, the sequential object monitor is said to be *free*.

Let us now consider a thread T requesting a method of M. First, the request is put in the wait queue. If M is busy, T is blocked, provoking a context switch. If M is free, the controller makes T invoke the scheduling method. If the request is scheduled, T takes ownership of M and executes the method immediately, i.e. no context switch occurs. If the request is not scheduled, M remains free and T is blocked.

5.4 Limitations

The current implementation of SOM presents some limitations. In particular, constructors of classes that are to be scheduled should never expose a reference to `this` to another thread, otherwise the thread-safety guarantee will be broken.

Furthermore, a SOM is *sequential* and hence potentially entails a loss of parallelism. However, it must be clear that the sequential constraint relates to the synchronization code: the overall program indeed remains parallel. This constraint makes it possible to simplify the task of writing, maintaining and reasoning about concurrent programs, thanks to a clear semantics and high expressiveness. Furthermore, our conjecture is that there are no problems that cannot be solved with this constraint. One could indeed compare this constraint to the absence of a `goto` statement in modern programming languages. In cases where the loss of parallelism is a critical issue, an approach similar to the one we took for SOM Chords should be adopted: the considered class (e.g., the class with chord declarations) is not converted to a SOM, but rather uses an auxiliary class (e.g., the chord manager), converted to a SOM, that is responsible of the coordination and synchronization.

5.5 Micro-Benchmarks

We argued at the beginning of the paper that the main inefficiency of Java monitors comes from the fact that `notifyAll` wakes up all waiting threads.

This section presents measurements of the execution time of five different buffer implementations; the typical solution using legacy Java monitors, as advised in the Sun Java tutorial [23], a "smart" solution using condition variables and mutexes as presented in [19], and three SOM-based solution: the direct solution using SOM (as in Fig. 8), the solution using SOM Guards (as in Fig. 14), and a solution with SOM Chords.

The measurements are given for a buffer of one slot, with one producer, and with different number of consumers. As the interest is measuring the cost of the synchronization, the time to produce and consume items in the tests is marginal. The results (Tables 1 and 2) were obtained by performing five measurements – each of which consists in the production of 100,000 items– discarding the best and worst cases and taking the average of the remaining three measurements. The benchmarks were executed on a single processor Athlon XP 2600+ machine with 512 MB of memory, with Java 1.4.2 with native threads. We allocated a large heap size to the JVM in order to limit the number of garbage collections. We run the benchmarks under Windows 2000 (Table 1) and Linux, kernel 2.4 (Table 2).

The case with one consumer is a best case for the Java monitors solution, because when the producer puts an item in the buffer, `notifyAll` always wakes up one thread only, and this thread will get the item. Hence no useless context switches occur. SOM solutions, as well as the solution based on condition variables, are slower in this case because they are implemented with multiple Java monitors, and therefore there is an associated overhead.

number of consumers	*Java monitors*	*Condition Variables*	*SOM*	*SOM Guards*	*SOM Chords*
1	390	1057	796	802	1203
2	510	1088	864	885	1229
4	771	1114	942	948	1265
8	1416	1120	1010	1026	1317
16	2823	1213	1166	1208	1541
32	7317	1375	1604	1593	1958
64	23479	2010	2322	2270	2708
128	80422	3234	3604	3442	4083

Table 1. Benchmark results under Windows 2000 with JDK 1.4.2 (time in ms).

number of consumers	*Java monitors*	*Condition Variables*	*SOM*	*SOM Guards*	*SOM Chords*
1	1006	1905	1656	1642	1954
2	1225	2018	1708	1690	2029
4	1918	2276	1891	1839	2148
8	5723	2412	2125	1982	2276
16	16005	2451	2435	2199	2488
32	49767	2659	3156	2766	3123
64	133612	2946	4407	3771	4196
128	358218	3049	6653	5259	5934

Table 2. Benchmark results under Linux 2.4 with JDK 1.4.2 (time in ms).

Increasing the number of consumers while keeping a single producer is greatly disadvantageous for the Java monitors solution, because when the producer puts an item in the buffer, several consumers must be waken up, although only one will get the item. The others will be put to wait again. Each failed wake up is a useless context switch, which is expensive in terms of execution time. We can easily see that SOM solutions scale much better with regards to the number of consumers, because *(i)* only one thread is waken up, and *(ii)* the evaluation of which thread to wake up is done by the thread leaving the monitor: no useless context switches occur.

The solution based on condition variables scales similarly well, and performs slightly better. With SOM, increasing the number of consumers lowers performance because the evaluation of the scheduling method depends on the size of the pending request queue (due to iterations over the queue). In contrast, the condition variables implementation is independent from the number of consumers, but still its performance slightly decreases because context switches seem to cost more when more threads are running.

number of consumers	*Java monitors*	*Condition Variables*	*SOM*	*SOM Guards*	*SOM Chords*	*Cond. Vars JDK1.5*
1	531	1279	1199	1157	1425	537
2	732	1234	1225	1196	1518	586
4	1131	1293	1333	1309	1573	556
8	2195	1281	1495	1378	1660	581
16	4312	1276	1851	1549	1916	592
32	9714	1350	2543	1969	2371	645
64	31637	1587	4305	2885	3601	850
128	95391	1762	7331	4414	5683	1062

Table 3. Benchmark results under Linux 2.6 with JDK 1.5 beta (time in ms).

An interesting point is that a straightforward implementation of the buffer with SOM (recall the simplicity of Fig. 8) brings better performance than the standard Java monitors solution, and comparable performance to the one with condition variables, which is more complex to correctly design and program.

The micro-benchmarks presented here do not take into account the cost of program transformation for SOM: classes were transformed statically before the benchmarks. If SOM was supported at the virtual machine level, no transformation cost would be incurred. We could also expect SOM to be at least as efficient as Java monitors even in the worst case. Hence, the micro-benchmarks presented here validate the interest of the SOM approach: although SOM implies an overhead at start, it scales very well. An efficient, VM-based, implementation of SOM would further reduce that overhead and make the approach even more competitive.

Finally, we have run the same benchmarks under Linux with the new 2.6 kernel, and with the beta version of JDK 1.5 (Table 3). Clearly, the results show that Linux is more competitive with the new optimized kernel. Furthermore, we have included another implementation: that of condition variables as included in the JDK 1.5, coming from the JSR 166 library [21]. Recall that normal condition variables are implemented with standard Java monitors, while condition variables of the JDK 1.5 are implemented with very efficient locks. The results, which are globally better than with JDK 1.4.2, confirm the previous analysis, and open new optimization perspective for SOM. Improvements similar to those that can be observed for condition variables should be obtainable in a new version of SOM implemented over the efficient locks of JDK 1.5.

6 Conclusion and Future Work

We have presented Sequential Object Monitors as an alternative to programming directly with standard Java monitors. Due to its sequential nature, a SOM is easier to reason about and maintain. We have illustrated the expressiveness of SOM through several examples, in particular through the implementation

of high-level abstractions like guards and chords. Furthermore, SOM promotes good modularization properties by untangling the synchronization concern from the application logic. Programmers can concentrate on programming functional code without worrying too much about concurrency. SOM provides a means to turn sequential classes into thread-safe classes without modifying them. Finally, SOM seems more efficient and scalable than the standard Java monitors due to its explicit control over which thread is woken up and its efficient scheduling strategy, as opposed to the untargeted and context-switch expensive `notifyAll` primitive. SOM can be characterized by the use of *run-to-completion* methods. It also relies on the packaging of small closures (reified method calls) as the building blocks of practical concurrent programming constructions, in the line of Lea's Java fork/join framework [20].

As future work, it would be interesting to reengineer an existing, non-trivial, concurrent application with SOM in order to study the benefits of our approach on large-scale software. Future work also includes studying the possibility to provide alternative scheduler base classes, such as a non-systematically reentrant scheduler in which some self sends can also reified as requests to be scheduled, upon user choice. Other alternatives include a scheduler able to dispatch in parallel a set of requests (e.g., all read requests for the readers and writers problem). With these features, we then plan to study several alternatives of join patterns with SOM.

Acknowledgements. We would like to thank Jacques Noyé and the anonymous ECOOP reviewers for their constructive comments.

This work was partially funded by the CONICYT-INRIA project ProXiMoS, and the Millenium Nucleous Center for Web Research, Grant P01-029-F, Mideplan, Chile.

References

[1] Gul Agha. *ACTORS: a model of concurrent computation in distributed systems.* The MIT Press: Cambridge, MA, 1986.

[2] P. H. M. America and F. Van Der Linden. A parallel object-oriented language with inheritance and subtyping. In N. Meyrowitz, editor, *Proceedings of the OOPSLA/ECOOP'90 Conference on Object-Oriented Programming Systems, Languages and Applications.* ACM Press, October 1990. ACM SIGPLAN Notices, 25(10).

[3] Colin Atkinson, Andrea Di Maio, and Rami Bayan. Dragoon: An object-oriented notation supporting the reuse and distribution of ada software. In *International Workshop on Real-Time Ada Issues*, 1990.

[4] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for C♯. In B. Magnusson, editor, *ECOOP 2002 - Object-Oriented Programming: 16th European Conference*, volume 2374, Málaga, Spain, June 2002. Springer-Verlag.

[5] G. M. Birtwistle, O.-J. Dahl, B. Myhrhaug, and K. Nygaard. *Simula Begin.* Petrocelli Charter, 1973.

[6] Per Brinch Hansen. Structured multiprogramming. *Communications of the ACM*, 15(7):574–578, July 1972.

[7] Per Brinch Hansen. A programming methodology for operating system design. In *Proceedings of the IFIP Congress 74*, pages 394–397, Amsterdam, Holland, August 1974. North-Holland.

[8] Per Brinch Hansen. Monitors and concurrent pascal, a personal history. *ACM SIGPLAN Notices*, 28(3):1–35, March 1993.

[9] Jean-Pierre Briot, Rachid Guerraoui, and Klaus-Peter Löhr. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys*, 30(3):291–329, September 1998.

[10] Denis Caromel. Towards a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, 1993.

[11] Denis Caromel, Wilfried Klauser, and Julien Vayssière. Towards seamless computing and metacomputing in Java. *Concurrency Practice and Experience*, 10(11-13):1043–1061, September 1998.

[12] Edsger W. Dijkstra. The structure of *THE* - multiprogramming system. *Communications of the ACM*, 11(5):341–346, May 1968.

[13] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[14] Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of POPL'96*, pages 372–385. ACM, January 1996.

[15] Svend Frolund and Gul Agha. A language framework for multi-object coordination. In O. Nierstrasz, editor, *Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP'93)*, volume 952 of *Lecture Notes in Computer Science*, pages 346–360, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[16] Charles A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–577, October 1974.

[17] Charles A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[18] W.H. Kaubisch, R.H. Perrott, and C.A.R. Hoare. Quasi-parallel programming. *Software: Practice and Experience*, 6(3):341–356, 1976.

[19] Doug Lea. *Concurrent Programming in Java, Design Principles and Patterns.* Addison Wesley, Reading, Massachusetts, 1997.

[20] Doug Lea. A Java fork/join framework. In *Proceedings of the ACM 2000 Conference on Java Grande*, pages 36–43, San Francisco, California, USA, 2000.

[21] Doug Lea. Java Specification Request 166: Concurrency utilities, 2003. `www.jcp.org/en/jsr/detail?id=166`.

[22] Martin Odersky. Functional nets. In Gert Smolka, editor, *ESOP*, volume 1782 of *Lecture Notes in Computer Science*, pages 1–25. Springer, 2000.

[23] Sun Microsystems, Inc. The producer/consumer example, from Java tutorials, 2003. `java.sun.com/docs/books/tutorial/essential/threads`.

[24] Eric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In *Proceedings of the 18th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2003)*, pages 27–64, Anaheim, CA, USA, October 2003. ACM Press. ACM SIGPLAN Notices, 38(11).