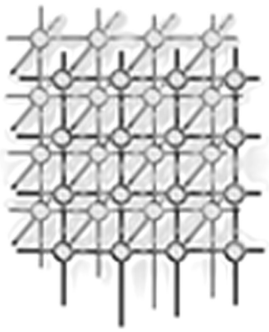
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE

Concurrency Computat.: Pract. Exper. 2000; **00**:1–7 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*

Parallel object monitors



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

[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

SUMMARY

Coordination of parallel activities on a shared memory machine is a crucial issue for modern software, even more with the advent of multi-core processors. Unfortunately, traditional concurrency abstractions force programmers to tangle the application logic with the synchronization concern, thereby compromising understandability and reuse, and fall short when fine-grained and expressive strategies are needed. This paper presents a new concurrency abstraction called POM, *Parallel Object Monitor*, supporting expressive means for coordination of parallel activities over one or more objects, while allowing a clean separation of the coordination concern from application code. Expressive and reusable strategies for concurrency control can be designed, thanks to a full access to the queue of pending requests, parallel execution of dispatched requests together with after actions, and complete control over reentrancy. A small domain-specific aspect language is provided to adequately configure pre-packaged, off-the-shelf synchronizations.

KEY WORDS: Synchronization, Concurrent Activities, Parallel Execution, Scheduler

*Correspondence to: University of Chile, Computer Science Dept.
Avenida Blanco Encalada 2120, Santiago, Chile
Contract/grant sponsor: This work is partially funded by the Conicyt-INRIA project OSCAR, and the CoreGRID EU Network of Excellence.
É. Tanter is partially funded by the Millennium Nucleus Center for Web Research, Grant P04-067-F of Mideplan, Chile.

1. Introduction

Synchronization of parallel activities in a concurrent system is a fundamental challenge. There are several dimensions to this challenge. To avoid *data races* while still allowing shared data, one may need to ensure *mutual exclusion* when accessing the shared structure (*e.g.* a bounded buffer). To enhance parallelism, it is often desirable to let multiple threads access *simultaneously* a shared data structure whenever data races are known not to occur (*e.g.* readers and writers). Furthermore, one may need to coordinate parallel activities in a more complex manner: for instance to achieve temporal ordering properties of requests executed over a set of application objects, depending on application-specific constraints (*e.g.* dining philosophers).

Using traditional concurrency abstractions like locks and monitors for expressing elaborate parallel coordination is cumbersome, and requires intrusive changes in the application code; in particular if coordination constraints over several objects have to be expressed. Actually, coordination of parallel activities is a specific *concern* that should ideally be implemented separately from the application code. Specifying coordination in a clearly modularized manner is important in order to foster understandability, maintainability and reuse: objects are independent of how they are coordinated, and multi-object coordination patterns can be abstracted and reused over different groups of objects [18, 24].

This paper introduces a new concurrency abstraction called *Parallel Object Monitors* for coordination of parallel activities over single objects and groups of objects in shared memory systems. A POM is said to be *parallel* because it is used to control the parallel execution of threads, is said to be *object* as it provides an object-oriented view of concurrent calls via reified requests and request queues, and is said to be a *monitor* because it provides a centralized place for specifying concurrency control in a thread-safe manner. We show that POMs are *(i)* expressive because a POM has full control over the queue of pending requests on coordinated objects and can implement custom reentrancy policies, *(ii)* easier to write and understand than existing techniques because concurrency control is expressed concisely

in a single place, *(iii)* efficient enough because the overhead of POMs in execution time is reasonable, and in some cases they are faster than legacy monitors. We provide POM as a library implemented over Reflex, a versatile kernel for multi-language aspect-oriented programming [29, 31], and configured with a small domain-specific aspect language. Therefore off-the-shelf POM schedulers can be reused and applied to components at the sole cost of proper configuration, while achieving reasonable performance.

Section 2 discusses related work and establishes the main motivation of our proposal. Section 3 presents POM, through its main principles and API, while Section 4 illustrates POM through some canonical examples. Section 5 exposes how high-level concurrency abstractions such as Sequential Object Monitors [10], chords [3], and synchronizers [18], can be expressed in POM. Section 6 presents the implementation of POM over Reflex, and benchmarks validating our proposal. Section 7 discusses some issues and Section 8 concludes with future work.

2. Related Work and Motivation

2.1. Synchronization Mechanisms for Mutual Exclusion

A great number of mechanisms have been proposed in order to address the mutual exclusion issue, starting with monitors as invented by Brinch Hansen [6] and Hoare [20]. A more elaborate kind of synchronization for mutual exclusion, called conditional synchronization, is made possible for instance by guards and guarded commands [13, 21]: a boolean expression is associated to an operation in order to indicate when it may be executed. Another mechanism for concurrency control, that originated in Simula-67 [4] and the Dragoon language [2], is the concept of *schedulers* [7], related to the *actor* [1] and *active object* [9] models. In such approaches, a separate entity called a *scheduler* is responsible for determining which and when concurrent requests to a shared object are performed. Schedulers enable separation of concerns, because the scheduler is defined apart from the application logic. However, like monitors and

guards, schedulers are commonly used to ensure mutual exclusion on the scheduled object. The recently proposed sequential object monitors (SOM) [10] also fall into this category.

Modern programming languages like Java and C♯ have adopted a flavor of monitors that is recognized to have a number of drawbacks [10]: these monitors are a low-level, error-prone abstraction that implies tangling functional code with synchronization code, breaking modularization. Also, Java monitors perform poorly in situations with high lock contention due to the `notifyAll` primitive, which may entail a lot of useless context switches.

The new concurrency utilities coming with Java 5 standardize medium-level constructions, such as *semaphores* and *futures*, and add a few native lower-level constructions, such as *locks* and *conditions*, which can be used to create fast new abstractions. The idea here is that people can use the appropriate abstractions for a given problem, and hence no particular concurrent paradigm is promoted. Such basic synchronization facilities are Hoare's style monitors. Still, the lowest-level Java 5 lock can be more fragile than before, because programmers are responsible for explicitly asking and releasing monitors, while with `synchronized` blocks the releasing of monitors is implicitly triggered at the end of such blocks. In fact, these new utilities favor flexibility and efficiency at the expense of increased verbosity, with a risk of fragility. Actually, programmers are rather expected to use higher-level abstractions whenever possible.

2.2. Parallel Coordination with Mutual Exclusion Mechanisms

Although the above mechanisms especially target the mutual exclusion problem, they can be used for coordination of parallel activities, such as in the classical readers and writers or dining philosopher problems. These solutions introduce a *controller* object where coordination is defined. The methods called on the controller are executed under mutual exclusion, ensuring that coordination constraints are not violated. The problem of these approaches is that clients have to be modified to explicitly communicate with the controller. For instance in the reader and writers problem, clients need to first ask for read or write access to a

controller (*e.g.* `enterRead`, `enterWrite`) before proceeding on the shared, unsynchronized structure. Furthermore, they need to notify the controller when they exit the shared structure (*e.g.* `exitRead`, `exitWrite`).

Therefore, from a software engineering viewpoint, mechanisms for mutual exclusion do not make it possible to achieve a clean separation of the synchronization concern when parallel coordination is needed. Even scheduler-based approaches like SOM [10], which do achieve separation of concern for mutual exclusion scenarios, fall short when dealing with *coordination of parallel activities*.

2.3. Synchronization Mechanisms for Parallel Coordination

Coordination of parallel activities refers to the synchronization of methods potentially executing simultaneously (*e.g.* read methods in the reader-writers problem). We name such a case *parallel coordination*, and review below two frameworks allowing its expression.

2.3.1. Synchronizers.

The synchronizers of Frølund and Agha [18] are the proposal that is most related to ours as it aims at separate specification and high-level expression of multi-object coordination. *Coordination patterns* are expressed in the form of *constraints* that restrict invocation of a group of objects. Invocation constraints enforce properties, such as temporal ordering and atomicity, that hold when invoking objects in a group. Synchronizers generalize the ideas of per-object coordination by means of synchronization constraints [27, 17] to the case of object groups. Furthermore, synchronizers not only involve the state of coordinated objects, but as well the invocation *history* of these objects. Although the declarativeness of synchronizers is appealing, a number of limitations still exist: fairness cannot be specified at the application level, but rather rely on implementation fairness; history-based strategies must be manually constructed; reentrancy cannot be customized; and finally, although synchronizers encapsulate

coordination, their usage has to be explicit in the application, requiring intrusive changes to existing code. Furthermore, the proposed implementation compiles away synchronizers, making them different from normal objects: other objects cannot interact with them via message passing. We will come back on synchronizers to show how they can be expressed with POM (Sect. 5).

2.3.2. Chords.

Chords were first introduced in Polyphonic C♯, an extension of the C♯ language. Chords are join patterns inspired by the join calculus [15]. 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. Invocations of asynchronous methods is non-blocking, while an invocation of a synchronous method blocks until the chord is *enabled*. A chord is enabled (and its body consequently executed) once all the methods in its header have been called. Method calls are implicitly queued until they are matched up. Chords enable coordination of parallel activities because multiple enabled chords are triggered simultaneously. However, although chords make it possible to concisely and elegantly express several concurrent programming problems, there are some classical problems which are difficult to solve with chords, such as implementing a buffer which ensures servicing of requests in order of arrival. Also, the use of chords over a group of objects has not been considered. Finally, chords as a language extension are not meant to achieve separation of the synchronization concern: a class definition with chords is a mixture of functional and synchronization code.

2.4. Motivation

This paper proposes an abstraction for concurrent programming that aims at solving the problems mentioned above: supporting expressive means for coordination of parallel activities

over one or more objects, while allowing a clean separation of the coordination concern from application code.

3. Parallel Object Monitors

Parallel Object Monitors, POMs, are a high-level abstraction for controlling synchronization of parallel threads. POM is inspired by the scheduler approach, in particular SOM and the synchronizers of Frølund and Agha. POM retains from SOM the notion of explicit access to the queue of pending requests and expressive means to specify scheduling strategies. But conversely to SOM, POM does not ensure mutual exclusion of requests themselves: a POM can dispatch several requests in *parallel*. Still, *concurrency control* is specified in a single place –a scheduler– and executed *sequentially*, in mutual exclusion.

3.1. Main Ideas

A parallel object monitor, POM, is a low-cost, thread-less, *scheduler* controlling parallel invocations on one or more standard, unsynchronized, objects. A POM is therefore a *passive object*. A POM controls the synchronization aspect of objects in which functional code is not tangled with the synchronization concern. A POM is a monitor defining a *scheduling method* responsible for specifying how concurrent requests should be scheduled, possibly in parallel. A POM also defines a *leaving method* which is executed by each thread once it has executed its request. Such methods are essential to the proposed abstraction as it makes it possible to reuse functional code as it is, adding necessary synchronization actions externally. A POM system makes it possible to define schedulers in plain Java, and to configure the binding of schedulers to application objects either in plain Java or using a convenient lightweight domain-specific aspect language.

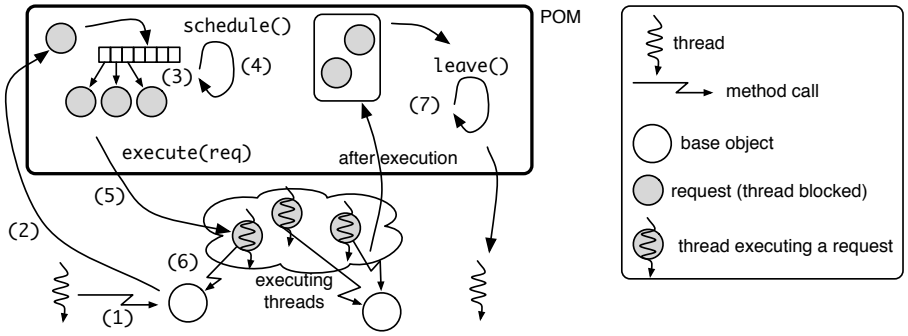


Figure 1. Operational sketch of a Parallel Object Monitor.

Fig. 1 illustrates the working of a POM. When a thread invokes a method on a object controlled by a POM (1), the thread is blocked and the invocation is reified and turned into a *request* object (2). Requests are then queued in a *pending queue* (3) until the scheduling method (4) grants them permission to execute (5). The scheduling method can trigger the execution of several requests. All selected requests are then free to execute *in parallel*, run by the thread that originated the call (6). Note that, if allowed by the scheduler, new requests can be dispatched before a first batch of selected request has completed. Parallel execution of selected requests in POM is in sharp contrast with SOM, where scheduled requests are executed in mutual exclusion with other scheduled requests [10]. Finally, when a thread has finished the execution of its associated request, it has to run the leaving method before leaving the POM (7). To run the leaving method, a thread may have to wait for the scheduler monitor to be free (a POM *is* a monitor), since invocations of the scheduling method and the leaving method are always safely executed, in *mutual exclusion*. Before leaving the monitor, a thread may have to execute the scheduling method again. The fact that a thread spends some time scheduling requests for other threads (recall that the scheduler is a passive object) implies that programmers should preferably write simple scheduling methods.

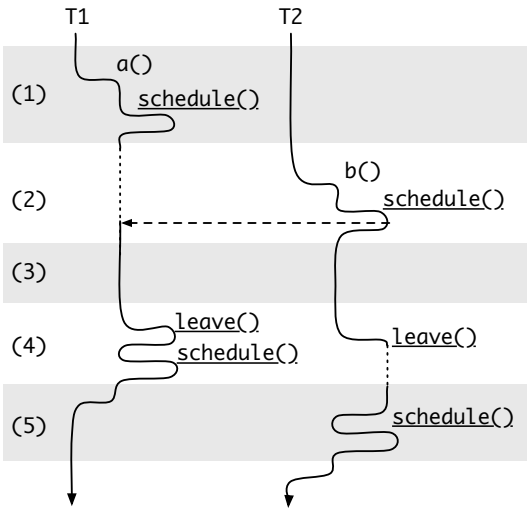


Figure 2. Two threads coordinated by a POM *(underlined method calls are performed in mutual exclusion within the scheduler monitor).*

Fig. 2 shows a *thread diagram* of two threads `T1` and `T2` coordinated by a POM. A thread diagram pictures the execution of threads according to time by picturing the call stack, and showing when a thread is active (plain line) or blocked waiting (dash line). Let us consider that the POM ensures the following coordination constraint: two operations `a` and `b` must be dispatched in pair. `T1` and `T2` are two threads invoking `a` and `b`, respectively. First, let us consider that the POM is free when `T1` calls `a`. `T1` directly executes the scheduling method, but its associated request is not granted permission to execute `(1)`. While `T1` is blocked, `T2` calls `b` on an object controlled by the POM. It executes `schedule`: this execution of the scheduling method results in both requests being selected `(2)`. Hence `T1` and `T2` execute their respective request in parallel `(3)`. When `T1` completes the execution of its associated request, it executes the leaving method and the scheduling method. These executions are performed *within* the scheduler monitor, hence in mutual exclusion with other invocations of the scheduling and

```
scheduling method:
if no writer executing then
  execute all readers, older than oldest writer
  if no reader executing then execute oldest writer
```

Figure 3. Pseudo-code of a fair scheduling strategy for readers and writers.

leaving methods: when `T2` finishes, it is blocked until `T1` releases the monitor `(4)`, before effectively executing in turn the leaving and scheduling methods `(5)`.

Defining a POM consists in specifying when requests should be granted permission to execute in the scheduling method, and optionally, specifying code that must be executed each time a request is completed, in the leaving method. Fig. 3 is an example, in pseudo-code, of a scheduling method specifying a fair strategy for the readers and writers problem. The leaving method is not shown, as it is just used to update the state of the scheduler (the complete POM implementation is however shown later, in Fig. 8).

A POM is a *parallel* monitor because it allows several threads to execute concurrently. Also, an executing request may not be run-to-completion: a POM is *by default* non-reentrant[†], therefore a thread already executing a request may be blocked if it calls a method controlled by the same POM. Fig. 4 summarizes the main principles and guarantees of POM.

3.2. Main Entities and API

We now present the main elements and API of a POM library (Fig. 5), in order to go through concrete examples afterwards.

[†]Rather, control is given over custom reentrancy policies, as discussed later in Sect. 4.4.

POM Principles

1. *Any method invocation on an object controlled by a POM 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. *All scheduled requests are executed by their calling thread, in parallel.*
4. *A POM is by default not reentrant: reentering calls are blocked and subject to scheduling. Reentrancy can be control on a per-case basis.*
5. *The scheduling and leaving methods are executed in a thread-safe manner within the scheduler monitor, but in parallel with executing requests.*

POM Guarantees

1. *Given that the scheduling method can schedule several requests at a time:*
   - *As soon as a request is given permission to execute, it starts execution in parallel with already-executing requests. If the thread owning such a request is the actual thread executing the scheduling method, then the request starts execution as soon as the thread exits from the scheduling method.*
   - *If a new request arrives, the scheduling method is called, even if all scheduled requests did not complete execution.*
2. *There is no unbounded busy execution of the scheduling method.*
3. *When a request ends execution, the leaving method is executed by its calling thread.*
4. *The scheduling method is executed by one of the caller threads, in mutual exclusion with the leaving method. The caller thread executing this method is unspecified.*
5. *After a caller thread has executed its request, it is guaranteed to return after one execution of the leaving method and* at most *one execution of the scheduling method.*
6. *Whenever a POM is idle, if a request arrives and is granted permission to execute by the scheduling method, the request is executed without any context switch.*

Figure 4. Main POM principles and guarantees.

3.2.1. Defining a POM.

A POM is defined in a class extending from the base abstract class `POMScheduler`. A POM must define the no-arg `schedule` method, in which the scheduling strategy is specified. The basic idea is that a scheduler can *grant permission to execute* to one or more pending requests, stored in a pending queue. Requests are represented as `Request` objects (Fig. 5). The scheduling

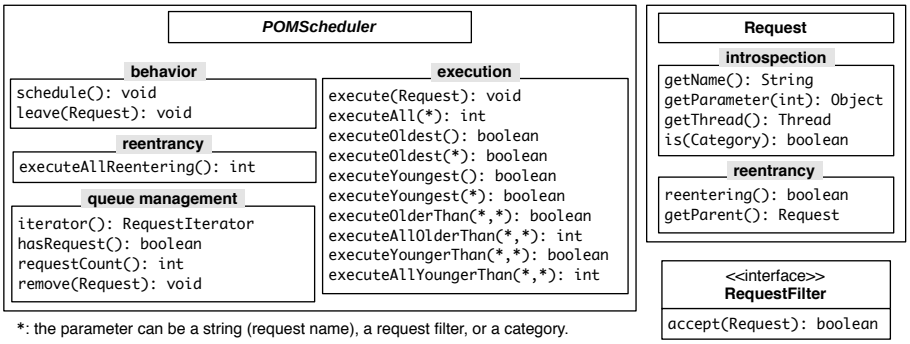


Figure 5. Main entities provided by a POM framework.

decision may be based on requests characteristics as well as the state of the scheduler itself, or any other external criteria. A scheduler can obtain an iterator on the pending queue using the `iterator()` method, and can then examine request objects in order to decide which ones should be granted permission to execute. A request object offers a protocol for introspection, which gives access to, *e.g.*, the actual parameters of the invocation and its calling thread. Once a request is selected for execution, it is removed from the pending queue.

3.2.2. Specifying After Behavior.

In some cases, the scheduler may maintain a state for determining which requests shall be granted permission to execute; such a state may have to be updated when requests finish executing. In POM, this is done in the leaving method, `leave`, which is executed by caller threads each time they complete the execution of their associated request. This method receives as parameter the request that has been executed. Since this method is defined in the scheduler and is typically used to update its state, it is executed within the monitor of the scheduler, in mutual exclusion with other invocations of the leaving and scheduling methods.

3.2.3. Defining a Scheduling Strategy.

A set of methods is available to grant permissions to execute pending requests. The most basic, `execute`, triggers the execution of the request given as parameter, by waking up its associated thread. More expressive methods are also provided. Some methods may result in a *single* request being executed, like `executeOldest()`, which triggers the execution of the oldest request in the pending queue. These methods return a boolean that indicates whether a request was effectively given permission to execute or not. For instance, `executeOldest()` returns `false` if the queue is empty. Other methods may grant permission to *several* requests at a time: they return the number of requests that have been effectively granted permission to execute. An example is `executeAllOlderThan`, which takes as parameters two criteria for selecting requests, and grants permission to all requests in the pending queue that meet the first criterion and that are older than the first pending request meeting the second criterion.

3.2.4. Selecting a Request.

There are basically three means to specify a criterion for selecting a request. First, one can pass the name of the requested method as a string. Second, a *request filter* can be given. A request filter implements the `RequestFilter` interface (Fig. 5), which declares the `accept` method. For instance, `executeAll(rf)` triggers the execution of all requests in the queue that are accepted by the `rf` filter.

Finally, one can specify a *method category*. Method categories are used to partition the set of methods in a given class into meaningful subsets, in order to enhance the potential of reuse and robustness with regards to change of off-the-shelf POM schedulers. We illustrate method categories in Section 4.2.

The protocols of `POMScheduler` and `Request` dealing with reentrancy (Fig. 5) are discussed in Section 4.4.

```
SCHED_DECL := "schedule:" APP_CLASS
              [ "on:" METHOD_LIST ]? "with:" SCHEDULER_CLASS
              [ CAT_DECL ]* [ "bygroup:" GROUP_CLASS ] ";"
CAT_DECL := "category:" CAT_ID "is:" METHOD_LIST
METHOD_LIST := METHOD_ID | METHOD_ID "," METHOD_LIST
```

Figure 6. BNF syntax of the POM configuration language.

3.2.5. Configuring POM.

POM is implemented on top of Reflex, a versatile kernel for multi-language AOP (Section 6.1). Similarly to SOM, we designed a small language to configure POM. The syntax of this language is overviewed on Fig. 6. We illustrate its use along the various examples in the paper.

POM can also be configured via a Java API exposed by the `POM` class. However, for clarity, we adopt the POM-specific language in the examples.

4. Canonical Examples

This section illustrates POM via a number of small cases: ensuring mutual exclusion, managing parallel dispatch, performing group synchronization, and controlling reentrancy. We refer to a simple `Dictionary` data type, with operations `query`, `define`, `size` and `delete` – in which concurrency is not dealt with.

4.1. Mutual Exclusion

We show how POM can easily be used to ensure a simple synchronization concern: the mutual exclusion of threads executing the dictionary methods (Fig. 7).

The state of the `MutualExclusionSched` consists of a boolean variable (`working`) that is true if and only if there is *one* thread executing a dictionary method. The scheduling method only grants permission to execute a request when `working` is false. Such a permission is

```
public class MutualExclusionSched extends POMScheduler {
  private boolean working = false;
  public void schedule(){
    if(!working) working = executeOldest();  (*)
  }
  public void leave(Request req){ working = false; }
}
```

Figure 7. Ensuring mutual exclusion with POM.

granted by calling `executeOldest` (Fig. 7`(*)`): if the pending queue is not empty, the oldest pending request is executed; otherwise, nothing happens. The method `executeOldest` returns a boolean value indicating whether a request actually received permission to execute. This boolean value is used to update the state of the scheduler. The `leave` method updates the value of `working` to false in order to state that no request is in execution. There is no need to deal with granting permissions to execute in the leaving method because the scheduling method is called immediately after the leaving method returns. The following POM declaration using the language of Fig. 6 associates a `MutualExclusionSched` instance per instance of `Dictionary`:

```
schedule: Dictionary with: MutualExclusionSched;
```

The `MutualExclusionSched` suffers from an important issue: reentrant method invocations result in a deadlock. For instance, if `define` invokes `query` for checking if a key is already defined, the scheduling method is called twice: once for `define` and once for `query`. The first execution of `schedule` grants permission for execution, but not the second one. Because `define` has not finished execution, `schedule` finds the monitor in a working state, and hence both requests are suspended forever. We address reentrancy in Section 4.4.

4.2. Parallel Dispatch

Considering a usage pattern of dictionaries where queries are performed much more frequently than definitions, and both operations are time consuming, it becomes interesting to dispatch all queries in parallel. This scenario is known as the readers and writers problem. Readers are

```
public class RWScheduler extends POMScheduler {
  public static Category READER = category();
  public static Category WRITER = category();
  private int readers = 0;
  private boolean writing = false;

  public void schedule(){
    if(!writing) {
      readers += executeAllOlderThan(READER, WRITER); (*)
      if(readers == 0) writing = executeOldest(WRITER);
  } }
  public void leave(Request req){
    if(req.is(READER)) readers--;
    else if(req.is(WRITER)) writing = false;
} }
```

Figure 8. Safely dispatching readers in parallel with POM.

invocations of *observer* methods (*e.g.* `query`). Executing them in parallel is possible because they do not produce data races. Writers are invocations of *mutator* methods (*e.g.* `define`). They may produce data races when executed in parallel, so they must be executed in mutual exclusion with other readers and other writers.

The POM scheduler safely dispatches readers in parallel while ensuring mutual exclusion when a writer executes (Fig. 8). The implemented strategy is fair: only readers that are older than the first writer in the pending queue are granted permission to execute `(*)`. This ensures that writers never starve.

As mentioned earlier, method categories make it possible to enhance reuse of synchronization policies. For instance, the `RWScheduler` of (Fig. 8) is a generic off-the-shelf scheduler that can be reused to deal with any class having multiple observer and mutator methods. It is not restricted to a dictionary class, because it does not rely on particular method names. Rather, the scheduler relies on two method categories, a `READER` category for observer methods and a `WRITER` category for mutator ones. Categories in a scheduler are simply defined as static variables of type `Category`, and can then be used to select requests.

The binding between categories in a scheduler and actual methods in base classes can be done declaratively using the POM configuration language introduced in Fig. 6. Note that a scheduler can also explicitly specify the methods that belong to a given category, but then reuse is lost[‡]. For instance, configuring the off-the-shelf `RWScheduler` to coordinate parallel activities over instances of `Dictionary` is done as follows:

```
schedule: Dictionary with: RWScheduler
   category: READER is: query, size
   category: WRITER is: define, delete ;
```

Like the scheduler of Fig. 7, `RWScheduler` is not reentrant and hence deadlocks on reentrant calls. A full version is presented in Section 4.4.

4.3. Group Synchronization

A typical example of parallel coordination among several objects is the dining philosophers problem. The dining philosophers problem consists of five philosophers sitting around a table, with five available sticks to eat Chinese food. Philosophers are active objects that spend their time looping over thinking and then eating. The philosopher at seat i needs the two sticks around him to eat (i and $(i+1)\%5$). A typical implementation of the `Philosopher` class is given in Fig. 9.

The synchronization constraints of this problem are that two philosophers cannot eat with the same stick at the same time, philosophers must not deadlock and must not starve. Using classical monitors, if we do not want to manually handle low-level synchronization in the `eat` method itself, the solution to this problem relies on the introduction of a controller object to which philosophers explicitly request and release their sticks. This is similar to the readers and

[‡]The use of annotations in base code, such as `@Reader`, is an easily-implementable and interesting alternative (Reflex supports annotations).

```
public class Philosopher extends Thread {
  int seat; Table table;
  public Philosopher(Table t){
    table = t; seat = table.getSeat(); start(); }
  public void run(){
    for(;;){ eat(); think(); } }
  // eat, think, getSeat, getTable, ...
}
```

Figure 9. A non-intrusive implementation of philosophers with POM.

writers problem discussed previously. Therefore, without POM, the code of the `for` loop of Fig. 9 has to be rewritten as follows:

```
int id1 = seat; int id2 = (seat+1)%5;
for(;;){  controller.pick(id1, id2); eat();
          controller.drop(id1, id2); think(); }
```

Although the `eat` method is not rewritten and the synchronization logic is encapsulated in the controller, such an intrusive approach is problematic: existing client code has to be manually updated to add synchronization code. Conversely, with POM, the code of philosophers is left untouched without any extra code.

`PhiloSched` (Fig. 10) implements a *fair* solution to the philosophers problem: a request is granted execution if both requested sticks are free and *none* have been *previously requested* by another philosopher. In the scheduling method, the local variable array `reserved`, created every time an iteration over the request queue begins, is used for ensuring that sticks are granted in the desired order. When a stick is requested and cannot be granted because it is still busy, it is tagged as "reserved". A request including a previously-reserved stick is not granted permission even though the stick may be free, because the stick must first be granted to the philosopher that first requested it. This is to ensure fairness, otherwise starvation may occur. POM allows fairness to be expressed at the application level, according to application-

```
public class PhiloSched extends POMScheduler {
  boolean[] busy = new boolean[5]; // false

  public void schedule(){
    boolean reserved[] = new boolean[5]; // false
    RequestIterator it = iterator();
    while(it.hasNext()){
      Request req = it.next();
      Philosopher p = (Philosopher) req.getReceiver();
      int id1 = p.getSeat(); int id2 = (id1+1)%5;
      if(!busy[id1] && !busy[id2] && !reserved[id1] && !reserved[id2]){
        busy[id1] = busy[id2] = true; execute(req); // granted
      }
      else { // not granted but reserved
        reserved[id1] = reserved[id2] = true;
  } } }
  public void leave(Request req){
    Philosopher p = (Philosopher) req.getReceiver();
    int id1 = p.getSeat(); int id2 = (id1+1)%5;
    busy[id1] = busy[id2] = false;
} }
```

Figure 10. Scheduler for the philosophers.

specific policies. Not that a more efficient solution can be devised if the constraint on stick reservation is relaxed.

To have a single scheduler controlling the concurrent activities of a group of philosophers, POM uses the grouping facility provided by Reflex. Basically, a group can be defined intentionally by an *association function*:

```
public class TableGroup implements GroupDefinition {
  public Object getGroup(Object obj){
    return ((Philosopher) p).getTable();
} }
```

The `getGroup` method returns an object whose identity corresponds to the group to which the object passed as parameter belongs. In the case of philosophers, a group corresponds to a table. Reflex associates one instance of scheduler per group, making it possible for different groups of philosophers to be controlled by their own specific POM instance. As a matter of fact, only

requests for `eat` in `Philosopher` must be subject to scheduling. In the small configuration language of POM (Fig. 6), the whole setting is expressed as follows:

```
schedule: Philosopher on: eat with: PhiloSched bygroup: TableGroup;
```

This code states that `eat` requests on `Philosopher`s are scheduled by a `PhiloSched` instance attached to each group as defined by `TableGroup`. It uses two extensions to the POM configuration language: `on:`, to restrict control by the scheduler to some method(s); and `bygroup:`, to specify that the association between a base object and a scheduler is neither per instance nor per class, but rather defined by a group.

4.4. Reentrancy Control

As mentioned earlier, a POM is not intrinsically reentrant. Rather, the programmer can have explicit control over reentrancy. When custom reentrancy policies are needed, POM exposes an API via `Request` objects. It is possible to determine if a request is reentrant by calling its `reentering` method. A convenience method `executeAllReentering` automatically triggers the execution of all reentering requests currently in the pending queue. Also, POM maintains the complete nesting structure of reentering requests on a per-thread basis: a reentering request has a reference to its parent request (accessed with `getParent`).

The control given over reentrancy makes it possible to express various reentrancy policies. Reentrancy can depend on both the name of the requested method and the identity of the calling thread, *e.g.* to allow only recursive method calls to reenter, or on some parameter of the requested method. Using both thread identifier and invocation parameters to determine reentrancy can be useful when considering a resource allocation system: when a resource `r` is asked for by calling `grant(r)`, then if a thread owning a resource asks again for the same resource, this last request should be reentrant and not block the thread. Conversely, asking for another resource may block.

```
public class ReentrantRWSched extends RWScheduler {
  public void schedule(){
    RequestIterator it = iterator();
    while(it.hasNext()){
      Request req = it.next();
      if(req.reentering()){                          (1)
        checkNoWriteAfterRead(req);                  (2)
        execute(req);                                (3)
    } }
    super.schedule();                                (4)
  }
  public void leave(Request req){
    if(!req.reentering()) super.leave(req);          (5)
  }
  void checkNoWriteAfterRead(Request req){
    if(req.is(WRITER) && req.getParent().is(READER)){
      remove(req);
      throw new RuntimeException("reader cannot invoke writer!");
} }
```

Figure 11. A reentrant scheduler for readers and writers.

In the case of readers and writers, requests should be reentrant, except in one forbidden case: a reader should never invoke a writer, as this would break the data race free property. `ReentrantRWScheduler` (Fig. 11) is a reentrant extension of `RWScheduler` (Fig. 8). If a request is reentering `(1)`, it is executed `(3)`. When all pending reentering requests have been granted execution, the scheduling method of the superclass is invoked `(4)`. The leaving method does nothing for reentrant requests, and reuses the original leaving method for non-reentrant ones `(5)`. Instead of relying on the assumption that a reader never calls a writer, `ReentrantRWSched` actually *checks* that this constraint is not violated `(2)`. The `checkNoWriteAfterRead` method throws an exception when the anomaly is detected: the parent of a reentrant writer cannot be a reader[§]. Being able to eagerly detect incorrect reentrancy patterns is particularly interesting to avoid mysterious deadlocks and data races, which are always hard to debug.

[§]It would even be possible to program a specific policy to handle this case: waiting for all other current reader requests to complete, and then execute the writer at hand. In the current version, we choose to throw a runtime exception, hence it has to be handled by callers.

```
public abstract class SOMScheduler extends POMScheduler {
  private boolean working = false;                        (1)
  private RequestQueue readyQueue = new RequestQueue();   (2)

  public final void schedule(){
    if(working) executeAllReentering();                   (3)
    else {
      working = true;
      if(readyQueue.isEmpty()) somSchedule();             (4)
      working = !readyQueue.isEmpty();  // queue may have changed
      if(working) execute(readyQueue.remove(0));          (5)
  }

  public abstract void somSchedule(); // defined by SOM user
  public final void leave(Request r){
    if(!r.reentering()) working = false;                  (6)
  }
  protected final void schedule(Request r){               (7)
    remove(r); readyQueue.add(r);
} }
```

Figure 12. Implementation of the SOM scheduler in POM.

Finally, when a scheduler needs to be completely reentrant, it is enough to define it as a subclass of `ReentrantPOMScheduler`, a subclass of `POMScheduler`.

5. Concurrency Abstractions in POM

We now discuss the implementation of three high-level concurrency abstractions with POM: sequential object monitors [10], chords [3], and synchronizers [18].

5.1. Sequential Object Monitors

Sequential Object Monitors (SOM) [10] are a high-level abstraction offering *fully sequential* monitors: the SOM programmer gets away from any code interleaving, because requests are always executed in mutual exclusion and *run to completion*. The latter means that when a request starts executing, other requests cannot start executing until it completes. SOM offers an API similar to that of POM: a scheduler implements a scheduling method where

the scheduling strategy is defined. But instead of granting requests the permission to start executing in parallel with others (with `execute`), the scheduling method of a SOM is used to mark requests (with `schedule`) that should be executed, in their scheduling order, in *mutual exclusion* with other requests and the scheduling method.

Because POM is more general, lower level, than SOM, it is feasible to express SOM with POM. Fig. 12 shows the POM implementation of the base scheduler class of SOM, `SOMScheduler`. This exercise highlights the very difference between both approaches, summarized by the following equation:

$$SOM = POM + mutual\ exclusion + reentrancy$$

- To ensure *mutual exclusion*, a scheme like that of Fig. 7 is used: a boolean `working` variable indicates whether the monitor is busy or not `(1)`. Because the scheduling method of SOM, renamed `somSchedule` for clarity, can schedule several methods at a time, a queue of scheduled requests is maintained; it is called the *ready queue* `(2)`. In the scheduling method of SOM, a request is scheduled via calls to `schedule(Request) (7)`. This method moves the given request from the pending queue (common to POM and SOM) to the ready queue (specific to SOM). The `somSchedule` method is invoked only when all previously-scheduled requests have been executed, that is to say, when the ready queue is empty `(4)`. Otherwise, scheduled methods are executed in their scheduling order `(5)`.
- To ensure *reentrancy*, when the monitor is busy, reentering requests are immediately executed `(3)` and do not free the monitor when leaving `(6)`. Reentrancy is *compulsory* to achieve run-to-completion of scheduled requests.

The very concise implementation of SOM in POM illustrates well the fact that SOM is a higher-level abstraction than POM. This makes SOM easier to use and understand, while

POM is a more versatile abstraction, where the programmer has explicit control over the form and degree of mutual exclusion and reentrancy that are required in a particular situation.

5.2. POM Chords

POM makes it possible to formulate an elegant *variation* of the chords of Polyphonic C♯ [3]. First of all, in POM Chords, chord-related logic is defined in the scheduler, leaving the base code intact (for instance, a dictionary class), as opposed to the language extension approach of Polyphonic C♯. In POM Chords, events can be triggered before and after invocations of methods on an object controlled by a chord scheduler. Before events correspond to requests before execution, and are *synchronous* because they may block; conversely, after events are *asynchronous*. Before and after events can be conveniently defined to correspond to method categories (Section 4.2). In addition, the chord scheduler can manage *internal* events, which are used to encode the internal state of the scheduler. Internal events are necessarily *asynchronous* because otherwise the scheduler is bound to deadlock. A chord is defined similarly to Polyphonic C♯, that is, by defining a body whose execution is conditioned to the occurrence of a given set of events. In POM Chords however, a chord body can only contain logic to trigger asynchronous events.

The solution to the readers and writers problem with POM Chords is given in Fig. 13. Similarly to Fig. 8, this scheduler relies on two method categories, `READER` for reader methods `(1)` and `WRITER` for writer methods `(2)`. The events `shared` and `exclusive` are defined as before events of these categories `(3,4)`, while `releaseShared` and `releaseExclusive` are after events `(5,6)`. Finally, the state of the scheduler is encoded with the use of two internal asynchronous events, `sharedRead` and `idle (7,8)`.

Chords themselves are defined in the `defineChords` method. The five chords of the Polyphonic C♯ implementation [3] in POM Chords are shown. The mapping from Polyphonic C♯ syntax to POM Chords is straightforward. Defining a chord consists in:

```
public class RWChordScheduler extends ChordScheduler {
  static Category READER, WRITER = category();              (1)(2)
  Event shared = before(READER);                               (3)
  Event exclusive = before(WRITER);                            (4)
  Event releaseShared = after(READER);                         (5)
  Event releaseExclusive = after(WRITER);                      (6)
  Event sharedRead, idle = event();                         (7)(8)

  void defineChords() {
   chord(shared).and(idle).body(new Body(){ void exec(){       (9)
       sharedRead.trigger(1);}});                             (10)

   chord(shared).and(sharedRead).body(new Body(){ void exec(){
       int n = sharedRead.getIntParam();                      (11)
       sharedRead.trigger(n+1);}});

   chord(releaseShared).and(sharedRead).body(new Body(){ void exec(){
       int n = sharedRead.getIntParam();
       if (n == 1) idle.trigger();
       else sharedRead.trigger(n-1);}});

   chord(exclusive).and(idle);
   chord(releaseExclusive).and(idle).body(new Body(){ void exec(){
       idle.trigger();}});
} }
```

Figure 13. Scheduler for the readers and writers problem with POM Chords.

- creating a new chord object by calling `chord`, specifying the first event of the chord, and aggregating other events in the chord via the `and` method (*e.g.* `(9)`);
- setting the chord body with `body(b)`, where `b` is an object implementing the `Body` interface. This interface declares a single `exec` method in which the chord body is specified. In the body, events are triggered by invoking `trigger` on them, possibly specifying parameters (*e.g.* `(10)`). Furthermore, the body can access the parameters of the event occurrences that enabled the chord (*e.g.* `(11)`).

A sketch of the implementation of the chord scheduler in POM is given in Fig. 14: we only show the scheduling and leaving methods. A chord scheduler maintains a mapping of categories to associated events (if any), which is filled by the `before` and `after` methods (recall Fig. 13`(3-6)`).

```
public abstract class ChordScheduler extends POMScheduler {
  ...
  public void schedule() {
    Request req;
    while((req = getOldest()) != null){
      BeforeEvent e = getBeforeEvent(req);              (1)
      if (e == null) execute(req);                      (2)
      else {
        e.triggerWith(req);                             (3)
        remove(req);                                    (4)
    } }
    while(!candidateChords.isEmpty()){                  (5)
      for(Chord c : candidateChords){
        if(isEnabled(c)) c.play();                      (6)
        candidateChords.remove(c);                      (7)
  } } }
  public void leave(Request req) {
    Event e = getAfterEvent(req);                       (8)
    if (e != null) e.trigger(req.getParams());          (9)
} }
```

Figure 14. Sketch of the chord scheduler in POM.

The scheduling method proceeds in two phases: first, all requests that have no associated before event `(1)` are directly executed `(2)`; otherwise, the before event is triggered `(3)`, passing the request as parameter, and the request is removed from the pending queue `(4)`. Triggering an event consists in publishing a token that can be consumed by chords, and updating the set of potentially-enabled chords, `candidateChords`, *i.e.* chords for which at least one token of each event is available. Second, the scheduler repeatedly iterates over this set until it is empty `(5)`. If a chord is enabled, it is "played" `(6)`: the associated event tokens are effectively consumed, and the chord body is executed. A chord present in this set may not be enabled because another chord may have just consumed some tokens needed by this chord. Then, the chord is removed `(7)`. The chord scheduler uses bit masks to efficiently determine if a chord is enabled, as explained in [3]. The leaving method is trivial: if a request completing execution has an associated after event `(8)`, this event is triggered, with the parameters of the request `(9)`.

This exercise shows the expressiveness of POM: a chord-like abstraction is concisely expressed with POM. The variation of chords we have exposed differs from the chords of Benton *et al.* by the fact that a chord may be made up of different synchronous events. This is made possible because the synchronization strategy is expressed outside the functional code, hence there is no issue about which return value to take into account. Actually, the chords of Polyphonic C$^\sharp$ and POM Chords are two different, though similar, abstractions, with potentially different application scenarios. Finally, it has to be noted that our approach does not rely on an extended base language with its own extended compiler. Therefore, a number of guarantees cannot be provided at the compiler level; for instance, compilation cannot ensure that the asynchronous methods have a void return type. In our implementation, we can ensure such guarantees at runtime, via dynamic checks (*e.g.* relying on the Java reflection API): whenever a binding between a base object and POM is done, checks are performed and runtime warnings/errors generated in case of violations.

5.3. Synchronizers

The synchronizers of Frølund and Agha offer a declarative interface to specify multi-object coordination [18]. We now explain how the features of synchronizers are supported by POM. The synchronizers definition language supports three operators of interest to us here: `updates` to update the state of the synchronizer, `disables` to disallow execution of certain methods (in a guard-like manner), and `atomic` to trigger several methods in parallel in an atomic manner.

The `updates` operator is trivially supported by POM because a POM is an object, and is always accessed in mutual exclusion. This also ensures that guard-like conditions associated to the `disables` operator are evaluated in mutual exclusion. In case where history-based coordination is required, the state of the synchronizer must be manually updated; this is similar in POM, but POM goes further by providing access to the queue of *pending* requests: therefore one can also base coordination on the state of the pending queue and the relative order

of request arrival. This is not feasible with synchronizers. Expressing guard-like conditions for the `disables` operator is expressed in POM as request filters operating on the pending queue, as in [10].

Finally, the `atomic` operator is but one pattern of coordination that can be expressed with POM. Given a batch of requests to be executed atomically (none or all at once), a POM proceeds as follows: all requests from the atomic batch are blocked (*i.e.* not granted permission to execution) until the batch is complete; when this occurs, the POM switches to a blocking state in which all new incoming requests are blocked; all requests from the batch are granted permission to execute, and the POM monitors the end of their execution in the leaving method; once all requests in the atomic batch finish execution, the POM goes back to its original state.

This shows that synchronizers are easily expressible in POM. Concrete syntax for declarative specification can be provided via the multi-language support of Reflex. We only focused here on the core semantics of synchronizers in POM.

6. Implementation

6.1. POM as a Reflex Plugin

The POM implementation for Java is based on Reflex, a versatile kernel for multi-language AOP [29, 31]. Reflex is designed to be a powerful back-end to implement possibly domain-specific aspect languages. It is based on a reflective model that makes it possible to define customized metaobject protocols, with expressive selection means to precisely configure where and when reification occurs [32].

A POM scheduler is a metaobject that takes control *before* and *after* a method is invoked. The base class `POMScheduler` implements the POM metaobject protocol and exposes the interface of the POM system to subclasses defined by users, as presented in this paper.

The POM configuration language provides concrete syntax for the declarative deployment of off-the-shelf POM schedulers, making it possible to specify the binding between base objects and schedulers, the methods that should be controlled, and the association strategy (per instance, per group, etc.).

Reflex supports multiple (domain-specific) aspect languages via a simple plugin architecture. A major interest of Reflex for this purpose is that it automatically detects and reports on interactions between aspects defined in different languages during weaving, and provides expressive means for specifying aspect composition. SOM [10] is also implemented as a Reflex plugin. As a consequence, unexpected interactions between SOM and POM, such as a class that should be both a sequential object monitor and an object controlled by a POM, can be detected and forbidden. Finally, Reflex is a reasonably efficient bytecode transformer based on Javassist [11]. It can operate either offline, as a compile-time utility, or online, as a load-time transformer.

6.2. Micro-Benchmarks

We now report on several micro-benchmarks comparing the cost of POM and other synchronization tools. In the two first benchmarks, we use a bounded buffer scenario with producers and consumers. The implementation of the POM scheduler for a bounded buffer is shown in Fig. 15. The scheduler makes use of method categories, `PUT` and `GET`, which are defined when configuring POM. Also, this scheduler uses the possibility to perform set operations on categories: the category `NOTPUT` is defined as comprising any method that does not belong to the `PUT` category `(*)`. Introducing these categories makes it possible to express the scheduling in a very concise manner.

```
public class BufferSched extends POMScheduler {
  static final Category PUT, GET = category();
  static final Category NOTPUT = PUT.not(); (*)
  static final Category NOTGET = GET.not();
  private int maxsize; // init in constructor
  private int size = 0;
  private boolean working = false;

  public void schedule() {
   if(!working) {
    if(size == 0) working = executeOldest(NOTGET);
    else if(size == maxsize) working = executeOldest(NOTPUT);
    else working = executeOldest();
  }}
  public void leave(Request req) {
   if (req.is(PUT)) size++;
   else if (req.is(GET)) size--;
   working = false;
}}
```

Figure 15. POM scheduler for a bounded buffer.

Table I. 1 producer/1 consumer on a dual processor (in ms).

buffer size	*C.V.*	*Fair C.V.*	*Java mon.*	*SOM*	*POM*
10000	187	2052	1568	1448	1750
1000	166	2016	1562	1724	1776
100	156	2046	1578	1802	1776
10	219	2052	1578	1818	1854
1	2114	2073	1698	1807	1807

6.2.1. Producer and consumer on a dual processor.

This micro-benchmark aims at comparing the overhead of POM for synchronization. In this scenario there is one producer sending 100,000 integer objects to a buffer while in parallel a consumer sums up the numbers taken from the buffer. The test machine is a dual Xeon 2.8 GHz with hyperthreading disabled, running Windows XP.

Table II. 1 producer/multiple consumers, buffer size 1, on a dual processor (in ms).

number of consumers	*C.V.*	*Java mon.*	*SOM*	*POM*
1	2156	1708	1818	1849
2	2156	3286	1822	1880
4	2166	3614	1849	1932
8	2213	7104	1849	1880
16	2187	14807	1854	1916
32	2187	29890	1849	2000
64	2260	62510	1833	2093
128	2426	136406	1921	2359

Results are shown in Table I. The first column shows the maximal size of the buffer, while other columns show the execution time for these implementations:

- *C.V.* (for *Condition Variables*): a smart buffer using the reentrant locks and condition variables introduced in Java 5 (from JSR 166), as described in [23].
- *Fair C.V.*: the same buffer than C.V. but enforcing a fair allocation of locks.
- *Java mon.*: a buffer using legacy Java monitors, as advised in the Sun Java tutorial.
- *SOM*: the buffer synchronized with SOM, as described in [10].
- *POM*: the buffer synchronized with POM, as shown in Fig. 15.

To analyze these results it is important to consider that the scenario is a worst-case scenario for synchronization tools because the time to effectively produce and consume items is marginal compared to the time to synchronize access to the buffer: hence in such a situation, two processors work slower than a single one; moreover a single thread executing both producer and consumer is a lot faster than two threads. This scenario is just targeted at measuring the overhead of synchronization in presence of high contention for the different monitors.

The first implementation is by far the most efficient. The other four implementations are much slower because they are victim of *processor oscillation* phenomena, which the

first implementation avoids due to its unfair allocation policy. To better understand this phenomena, consider a processor executing the producer, owning the monitor, while the second processor executes the consumer and is waiting to get the monitor. When the first processor releases the monitor, it quickly produces another item and asks again for the monitor, before the second processor wakes up.

With the unfair allocation policy of the Java 5 locks, the first processor wins the monitor immediately and continues without pauses: it can work continuously until completely filling the buffer. Then the second processor gets the processor and works continuously until the buffer is empty. Conversely, with a fair allocation policy, when the first processor asks again for the monitor, it does not get it and has to wait until the second processor wakes up and executes its own operation. This situation is reproduced for the production and consumption of each and every item. All this sleeping and waking up of processors greatly degrades performance: actually, having a buffer of more than one slot is useless in this case. The unfair locks of Java 5 only face this processor oscillation phenomena when the buffer is of size 1, as clearly reported in Table I.

6.2.2. Producer and multiple consumers on a dual processor.

This benchmark aims at measuring the overhead of synchronization in situations with extremely high contention on a monitor. One thread produces 100,000 integer objects and puts them in a buffer of a single slot. A varying number of consumer threads compete to get items from the buffer and sum them up. Since the time for producing an item is as short as that for consuming, consumers are almost always waiting to get an item from the buffer. Since the buffer is of size 1, the producer is constrained to give up the processor each time it puts an item (dually for a consumer). As a consequence, in this setting there are thread context switches for each put and each get operation.

We excluded the fair locks of Java 5 because the timings in this scenario are identical to the unfair ones. The results (Table II) show that both SOM and POM are slightly more efficient than the Java 5 solution. Moreover, legacy Java monitors become severely inefficient for high numbers of consumers (two orders of magnitude for one hundred consumers). This is due to the semantics of the `notifyAll` operation: when the producer puts an item, it calls `notifyAll` and awakes *all* waiting consumers. Consumers awake and take one by one the monitor. The first one succeeds in getting an item, but others then find the buffer empty and hence go back to wait. Therefore lots of expensive and useless thread context switches occur. In all cases, the overall time increases with the number of threads due to thread overhead. Having a dual processor is useless in this scenario because of the high contention of the monitor.

6.2.3. Readers and writers on a dual processor.

We now compare POM with other synchronization tools when parallel execution of threads is required, with different degrees of monitor contention. For this purpose we consider a readers and writers problem: a dictionary that allows queries (implemented as linear search) to proceed in parallel. We consider three implementations:

- *RWLock*: using the read/write lock of Java 5 (`ReadWriteReentrantLock`).
- *Monitor*: a read and write lock implemented with legacy Java monitors.
- *POM*: a POM scheduler implementing readers and writers synchronization as explained in Section 4.2.

During the experiments, we progressively increase the size of the dictionary, so that the work done per read request gets higher. For each dictionary size and implementation, we measure the performance with a *single* thread that makes one million unsuccessful queries, and with *two* threads making each half a million queries. In the latter case, the two processors of the machine are exploited, but there should be some contention to access the monitor: indeed, even if both threads can make queries in parallel, they still need to access the internal data of

Table III. Benchmark results of readers and writers on a dual processor (time in ms).

dict.	*RWLock*			*Monitor*			*POM*		
size	1 thrd	2 thrds	speedup	1 thrd	2 thrds	speedup	1 thrd	2 thrds	speedup
1	130	416	(0.31)	468	15442	(0.03)	1802	3838	(0.47)
2	141	437	(0.32)	463	15234	(0.03)	1781	3421	(0.52)
4	187	442	(0.42)	526	15677	(0.03)	1854	3718	(0.50)
8	239	328	(0.72)	583	15104	(0.04)	1921	4083	(0.47)
16	349	245	**(1.42)**	687	15125	(0.05)	2021	4521	(0.45)
32	562	338	(1.66)	890	15176	(0.06)	2224	4937	(0.45)
64	1000	557	(1.80)	1333	14885	(0.09)	2661	4240	(0.62)
128	1859	984	(1.89)	2203	15025	(0.15)	3583	4495	(0.80)
256	3646	1875	(1.94)	3974	14114	(0.28)	5536	3578	**(1.55)**
512	7838	3969	(1.97)	8078	5354	**(1.50)**	10041	6125	(1.64)
1024	16390	8250	(1.99)	16370	8792	(1.86)	18208	9885	(1.84)
2048	32271	16267	(1.98)	32250	17734	(1.81)	33578	17942	(1.87)
4096	65297	32759	(1.99)	64140	36416	(1.76)	67432	35135	(1.92)

the monitors in mutual exclusion. We do not expect any performance improvement in having two threads for small dictionaries, because of the high monitor contention. On the other hand, for big dictionaries, we expect a speedup approaching 2x when using two threads compared to only one. Also, overall timing should increase while increasing the dictionary size.

Table III shows the results of this experiment. For each case, we give the execution time for one thread, two threads, and the calculated speedup (dividing the time for one thread by the time for two threads). The measurements basically confirm our expectations: having two threads for small dictionaries degrades performance. This is dramatically true for the solution based on legacy Java monitors. For the Java 5 locks solution, having two threads becomes beneficial for a dictionary of size 16, while for legacy monitors the breaking point is for a size of 512. POM performs much better with two threads than the solution with legacy Java monitors, and profits from two threads starting at a dictionary of size 256. For big dictionaries, the three implementations perform similarly, because the predominant work is the linear search.

With a very small dictionary and only one thread, POM is one order of magnitude slower than the Java 5 read-write locks. With two threads, the worst case is for legacy monitors, where the mean cost of a query is 15 microseconds (μs). Hence, a parallel solution starts to be beneficial with the Java 5 read-write locks if the job done by a query is around 0.3 μs (for a Xeon 2.8 GHz), while the threshold is around 6 μs for POM, and around 8 μs for legacy Java monitors.

6.2.4. Evaluation.

As a result of these benchmarks, we applaud the performance of the new locks coming with Java 5. We actually used them to implement POM, and reimplement SOM. The new version of SOM is much more efficient than the one presented in [10], which relied on legacy Java monitors. The benchmarks show that POM (and SOM) globally perform better than legacy Java monitors, in particular in cases with true parallelism and cases with high contention. Also, POM exhibits a reasonable overhead compared to the very efficient locks of Java 5. We believe that this overhead is acceptable when considering the gains in expressiveness, simplicity, and modularity brought by our approach. Furthermore, considering a VM-based implementation (rather than bytecode transformation-based) would further reduce the overhead of POM in terms of interception and reification of method calls, hence making the approach even more competitive while preserving the software engineering benefits.

7. Discussion

Generality vs. Specificity. POM makes it possible to tune the generality of a given scheduler. A scheduler can be highly generic and reusable (like the scheduler of Fig. 8, which relies on method categories), or it can be very application-specific (like the scheduler for philosophers in Fig. 10). In addition to allowing this fine-tuning, POM allows for modular

definition of synchronization. A POM scheduler encapsulates the synchronization logic, while the POM configuration language is used to specify the binding of a scheduler to base code. Not that since Reflex is based on the MetaBorg approach and tools for embedding and assimilation of domain-specific languages [5], and although we have not elaborated on this aspect in the paper, the specifications are not necessarily defined in external files.

Inheritance Anomaly. The interaction of inheritance and concurrency control is well-known to be difficult and raise a number of *inheritance anomalies* [8, 25]. It has been shown that AOP approaches to concurrency control are more adequate than standard concurrent object-oriented languages to handle anomalies, although not all equally [26]. By allowing separate specification of synchronizations and giving access to the queue of pending requests, POM helps addressing anomalies related to history sensitiveness and partitioning of states. Finally, the capacity to program reentrancy is also an asset for fighting reuse difficulties without changing base code, as strategies can be changed by switching to different schedulers. Still, an in-depth analysis of inheritance anomalies in POM should be addressed in future work.

Transactional Memory. Lately, the research community is putting a lot of attention on software transactional memory (STM) [28]. Adapting the well-known concept of database transactions to concurrent programming, with STM the programmer signals the start of a transaction (instead of asking a lock), performs some data accesses and finally commits the operation (instead of releasing a lock). The STM system detects any data race when threads access shared data and aborts inconsistent transactions. The main advantage of STM is that deadlocks or priority inversion problems cannot occur. In contrast, in lock-based systems avoiding those problems is the responsibility of programmers and is generally difficult. Some implementations of STM also avoid livelocks but with extra overhead.

Software transactional memory is more light-weight than data base transactions because in the former case only memory accesses are concerned while the latter implicates disk operations. On the other hand, for general concurrent programming, STM is more expensive than locks in terms of processor and/or memory usage, although STM performance is approaching the perfomance of locks-based systems [28] and sometimes exceeding them [19]. One problem associated to the STM is that the programmer must access data through a special STM API [16, 28], hence requiring extensive changes to software. A Java language extension has been proposed to alleviate this issue, relying on an extended compiler[19], but is still intrusive.

An interesting research perspective is to combine POM and STM into a system that, like POM, uses schedulers to specify the synchronization concern, and runs methods into transactions, as in STM. However, the implementation of such a system will be harder than POM, because it will require complex bytecode changes, as explained in[19].

Relation to AOP. The approach we use is indeed aspect oriented [14]: the aspect of coordination of parallel activities is separated from the base code through a mechanism similar to other AOP approaches. In this regard, our implementation of POM consists of a framework for defining the coordination semantics (the scheduler) as well as a small language for the binding between base entities and schedulers. The gain with respect to directly using a general-purpose AOP approach is twofold: first, the configuration language only exposes the joinpoints that do make sense in the context of POM, at a higher-level of abstraction; second, the framework for defining schedulers hides the low-level logic needed to actually realize synchronization appropriately. This said, it is completely feasible to devise an implementation of POM using a general-purpose aspect language like AspectJ [22], although some features may be more cumbersome to provide. We have chosen to include POM in our more general research artefact for multi-language AOP, in order to be able to study related issues such as

aspect interactions across languages [31, 30].

Limitations. Our approach to concurrency control fosters expressiveness. This is why schedulers are defined imperatively, in plain Java. As a consequence, this makes it very difficult, if not impossible, to develop analysis and optimizations of the scheduling code. As mentioned before, we focused on core semantics in this work. This however does not preclude the provide of a restricted expressiveness on top of POM using a domain-specific language, which would then be amenable to analysis and optimizations. A first direction would be to extend the POM configuration language so that the scheduler is expressed directly in this extended language, rather than just using the language for specifying the binding between Java schedulers and base entities.

Another limitation of our approach is that there is no parallel execution of the scheduler itself. This means that threads may have to wait for executing the scheduling and leaving methods. However, it has to be highlighted that threads are only blocked during the execution of scheduling and leaving methods (*not* during execution of other requests), and that these methods are meant to be short-running. For instance, in the philosophers example, the scheduling method is only called to determine if a philosopher can eat; while the philosopher is eating, the scheduler monitor is not blocked, so other executions of the scheduling or leaving methods can proceed. Similarly, calls to the leaving methods are only used to update the state of the scheduler. As a matter of fact, the performance issue with POM does not come from blocking threads during the execution of the scheduling and leaving methods, because the probability of contention there is marginal. The true issue is the fact of having to ask a lock for this. So a very promising approach would actually be to use transactional memory for the POM schedulers, in order to avoid requesting locks for scheduler execution.

8. Conclusion and Future Work

We have presented Parallel Object Monitors as a new abstraction for synchronizing parallel activities. The main contribution of POM is to combine the power of the multi-object coordination of Frølund and Agha [18], with the expressiveness provided by an explicit access to the request queue, as in scheduler approaches [9, 10]. In addition, POM gives complete control over reentrancy strategies. Furthermore, following an aspect-oriented approach, POM promotes separation of concerns by untangling the synchronization concern from the application code. Based on the Reflex AOP kernel, the Java implementation of POM supports reuse of off-the-shelf synchronization policies thanks to method categories. A lightweight domain-specific aspect language is provided for configuration of existing schedulers. We have illustrated the expressiveness of POM through several examples, in particular through the implementation of high-level abstractions like sequential object monitors [10] and chords [3], as well as the synchronizers of [18]. Finally, several benchmarks validate the applicability of the proposal.

As future work, several trends shall be pursued. First, the aspect language for POM can be made more expressive, both in terms of configuration control and scheduling specification. Second, the POM Chords abstraction presented here represents a powerful starting point for several alternatives of join patterns, such as n-way rendez-vous. Third, the relation with software transactional memory should be further explored. STM has several advantages over lock-based approaches, but is still an intrusive approach to concurrency specification. It is therefore particularly interesting to try to reconcile the transparency and fine-grained control of our approach with the efficiency of STM. Finally, composability of SOMs and POMs should be studied, taking advantage of the aspect composition facilities of Reflex, including both detection and resolution of interactions [31].

ACKNOWLEDGEMENTS

We thank Oscar Nierstrasz and Sébastien Vaucouleur for their comments on a draft of this paper.

REFERENCES

1. G. Agha. *ACTORS: a model of concurrent computation in distributed systems.* The MIT Press: Cambridge, MA, 1986.
2. C. Atkinson, A. D. Maio, and R. Bayan. Dragoon: An object-oriented notation supporting the reuse and distribution of Ada software. In *Proceedings of the 4th International Workshop on Real-Time Ada Issues*, pages 50–59, Pitlochry, Perthshir, Scotland, 1990.
3. N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C♯. *ACM Transactions on Programming Languages and Systems*, 26(5):769–804, Sept. 2004.
4. G. M. Birtwistle, O.-J. Dahl, B. Myhrhaug, and K. Nygaard. *Simula Begin.* Petrocelli Charter, 1973.
5. M. Bravenboer and E. Visser. Concrete syntax for objects. In *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2004)*, Vancouver, British Columbia, Canada, Oct. 2004. ACM Press. ACM SIGPLAN Notices, 39(11).
6. P. Brinch Hansen. A programming methodology for operating system design. In *Proceedings of the IFIP Congress 74*, pages 394–397, Amsterdam, Holland, Aug. 1974. North-Holland.
7. J.-P. Briot, R. Guerraoui, and K.-P. Löhr. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys*, 30(3):291–329, Sept. 1998.
8. J.-P. Briot and A. Yonezawa. Inheritance and synchronization in concurrent oop. In *Proceedings of the 1st European Conference on Object-Oriented Programming (ECOOP 87)*, volume 276 of *Lecture Notes in Computer Science*, pages 32–40. Springer-Verlag, 1987.
9. D. Caromel. Towards a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, 1993.
10. D. Caromel, L. Mateu, and É. Tanter. Sequential object monitors. In M. Odersky, editor, *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP 2004)*, number 3086 in Lecture Notes in Computer Science, pages 316–340, Oslo, Norway, June 2004. Springer-Verlag.
11. S. Chiba and M. Nishizawa. An easy-to-use toolkit for efficient Java bytecode translators. In F. Pfenning and Y. Smaragdakis, editors, *Proceedings of the 2nd ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2003)*, volume 2830 of *Lecture Notes in Computer Science*, pages 364–376, Erfurt, Germany, Sept. 2003. Springer-Verlag.
12. R. Crocker and G. L. Steele, Jr., editors. *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2003)*, Anaheim, CA, USA, Oct. 2003. ACM Press. ACM SIGPLAN Notices, 38(11).

13. E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.
14. T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming. *Communications of the ACM*, 44(10), Oct. 2001.
15. C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of POPL'96*, pages 372–385. ACM, Jan. 1996.
16. K. Fraser. *Practical Lock-Freedom.* PhD thesis, King's College, University of Cambridge, Sept. 2003.
17. S. Frølund. Inheritance and synchronization constraints in concurrent object-oriented programming languages. In O. Lehrmann Madsen, editor, *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP 92)*, volume 615 of *Lecture Notes in Computer Science*, pages 185–196, Utrecht, The Netherlands, July 1992. Springer-Verlag.
18. S. Frølund and G. 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 707 of *Lecture Notes in Computer Science*, pages 346–360, Kaiserslautern, Germany, July 1993. Springer-Verlag.
19. T. Harris and K. Fraser. Language support for lightweight transactions. In Crocker and Steele, Jr. [12]. ACM SIGPLAN Notices, 38(11).
20. C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–577, October 1974.
21. C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.
22. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, number 2072 in Lecture Notes in Computer Science, pages 327–353, Budapest, Hungary, June 2001. Springer-Verlag.
23. D. Lea. *Concurrent Programming in Java.* The Java Series. Addison Wesley, second edition, 1999.
24. S. Matsuoka, T. Watanabe, and A. Yonezawa. Hybrid group reflective architecture for object-oriented concurrent reflective programming. In P. America, editor, *Proceedings of the 5th European Conference on Object-Oriented Programming (ECOOP 91)*, volume 512 of *Lecture Notes in Computer Science*, pages 231–250, Geneva, Switzerland, July 1991. Springer-Verlag.
25. S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.
26. G. Milicia and V. Sassone. The inheritance anomaly: ten years after. In *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC'04)*, pages 1267–1274, New York, NY, USA, 2004. ACM Press.

27. O. Nierstrasz. Active objects in Hybrid. In N. Meyrowitz, editor, *Proceedings of the 2nd International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 87)*, pages 243–253, Orlando, Florida, USA, Oct. 1987. ACM Press. ACM SIGPLAN Notices, 22(12).
28. N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of PODC'95*, pages 204–213, Ottawa, Ontario, Canada, Aug. 1995.
29. É. Tanter. *From Metaobject Protocols to Versatile Kernels for Aspect-Oriented Programming*. PhD thesis, University of Nantes and University of Chile, Nov. 2004.
30. É. Tanter. Aspects of composition in the Reflex AOP kernel. In *Proceedings of the 5th International Symposium on Software Composition (SC 2006)*, volume 4089 of *Lecture Notes in Computer Science*, Vienna, Austria, Mar. 2006. Springer-Verlag.
31. É. Tanter and J. Noyé. A versatile kernel for multi-language AOP. In R. Glück and M. Lowry, editors, *Proceedings of the 4th ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2005)*, volume 3676 of *Lecture Notes in Computer Science*, pages 173–188, Tallinn, Estonia, Sept./Oct. 2005. Springer-Verlag.
32. É. Tanter, J. Noyé, D. Caromel, and P. Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In Crocker and Steele, Jr. [12], pages 27–46. ACM SIGPLAN Notices, 38(11).