Parallel object monitors

Denis Caromel[1], Luis Mateu[2,3], Guillaume Pothier[2]
and Éric Tanter[2,*,†]

[1]*OASIS Project, Université de Nice-CNRS-INRIA 2004, Rt. des Lucioles, Sophia Antipolis, France*
[2]*PLEIAD Lab, Computer Science Department (DCC), University of Chile, Avenida Blanco Encalada 2120, Santiago, Chile*
[3]*Synopsys Chile R&D Center, Av. Vitacura 5250. Piso 7, Of. 708, Santiago, Chile*

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

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

*Correspondence to: Éric Tanter, PLEIAD Lab, Computer Science Department (DCC), University of Chile, Avenida Blanco Encalada 2120, Santiago, Chile.
†E-mail: etanter@dcc.uchile.cl

Contract/grant sponsor: Conicyt-INRIA project OSCAR
Contract/grant sponsor: CoreGRID EU Network of Excellence
Contract/grant sponsor: Millennium Nucleus Center; contract/grant number: P04-067-F
Contract/grant sponsor: FONDECYT Project; contract/grant number: 11060493

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 such as 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 [1,2].

This paper introduces a new concurrency abstraction called *parallel object monitor* (POM) 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 [3], 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 (SOMs) [4], chords [5], and synchronizers [1] 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 [7]. A more elaborate kind of synchronization for mutual exclusion, called conditional synchronization, is made possible, for instance, by guards and guarded commands [8,9]: a boolean expression is associated to an operation in order to indicate when it may be executed. Another mechanism for concurrency control, which originated in Simula-67 [10] and the Dragoon language [11], is the concept of *schedulers* [12], related to the *actor* [13] and *active object* [14] 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, similar to monitors and guards, schedulers are commonly used to ensure mutual exclusion on the scheduled object. The recently proposed SOMs [4] also fall into this category.

Modern programming languages such as Java and C$^\sharp$ have adopted a flavor of monitors that is recognized to have a number of drawbacks [4]; 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 number 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 philosophers 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 readers 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 such as SOM [4], 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 [1] 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 [15,16] to the case of object groups. Furthermore, synchronizers involve not only the state of coordinated objects but also 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 the 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 to synchronizers to show how they can be expressed with POM (Section 5).

2.3.2. Chords

Chords were first introduced in Polyphonic C$^\sharp$, an extension of the C$^\sharp$ language. Chords are join patterns inspired by the join calculus [17]. Within Polyphonic C$^\sharp$, 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 are 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 that 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 codes.

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

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 POM is a low-cost, threadless *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 illustrates the working of a POM. When a thread invokes a method on an 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 [4]. 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*.

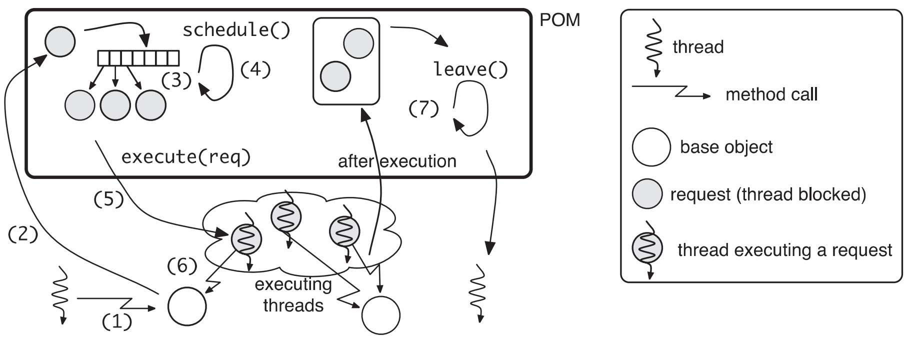


Figure 1. Operational sketch of a parallel object monitor.

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 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 requests 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 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 a code that must be executed each time a request is completed, in the leaving method. Figure 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 only used to update the state of the scheduler (the complete POM implementation is, however, shown later in Figure 8).

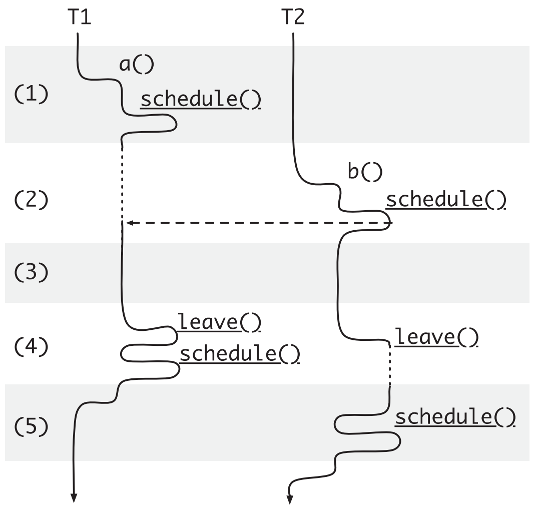


Figure 2. Two threads coordinated by a POM (*underlined method calls are performed in mutual exclusion within the scheduler monitor*).

```
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.

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.

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. Figure 4 summarizes the main principles and guarantees of POM. The principles are elements

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

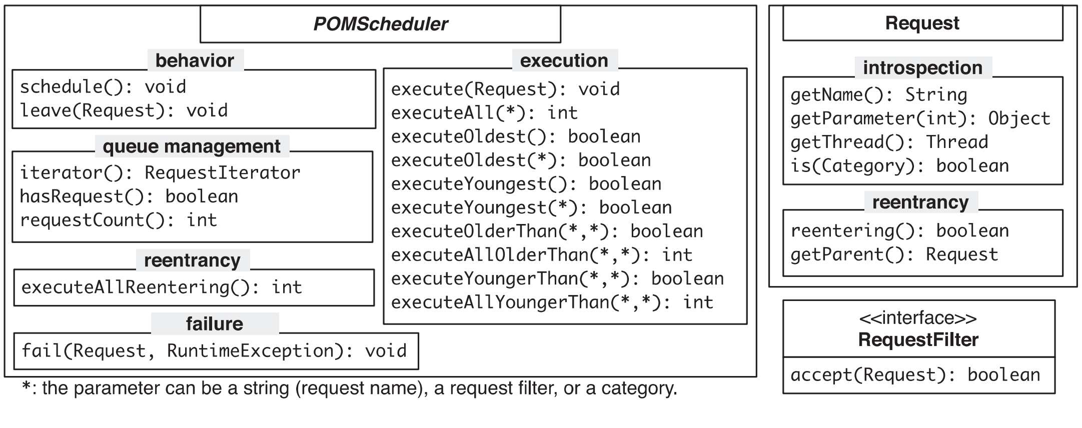
describing the fundamental concepts to consider when programming with POMs; they have to be taken into account to define correct POM synchronizations. The guarantees, in contrast, come with a given implementation: they describe extra properties that are desirable to obtain an efficient implementation, especially minimizing context switches. Recall that our approach is based on *cooperative scheduling*, as in SOM [4], where the scheduling job is performed by client threads (the scheduler object is itself threadless). Therefore, the guarantees clarify assumptions that could or should not be made by programmers, such as the exact thread executing the scheduling or leaving methods.

3.2. Main Entities and API

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

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 (Figure 5). The scheduling decision may be based on requests characteristics as well as on the state of the scheduler itself, or on 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, and which ones should fail, causing the thread that originated the request to throw a runtime exception. 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.



POMScheduler
behavior
schedule(): void
leave(Request): void
queue management
iterator(): RequestIterator
hasRequest(): boolean
requestCount(): int
reentrancy
executeAllReentering(): int
failure
fail(Request, RuntimeException): void
execution
execute(Request): void
executeAll(*): int
executeOldest(): boolean
executeOldest(*): boolean
executeYoungest(): boolean
executeYoungest(*): boolean
executeOlderThan(*,*): boolean
executeAllOlderThan(*,*): int
executeYoungerThan(*,*): boolean
executeAllYoungerThan(*,*): int
*: the parameter can be a string (request name), a request filter, or a category.
Request
introspection
getName(): String
getParameter(int): Object
getThread(): Thread
is(Category): boolean
reentrancy
reentering(): boolean
getParent(): Request
<<interface>>
RequestFilter
accept(Request): boolean

Figure 5. Main entities provided by a POM framework.

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, such as `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.

If no request is scheduled for execution at a given time, all the threads that called a method supervised by a POM are blocked. Our implementation of POM supports a debug mode in which a warning is issued whenever no request is scheduled after the execution of the `schedule()` method. Although not necessarily an error in itself, this can help in identifying potentially abnormal situations.

3.2.4. Selecting a request

There are basically three ways of specifying 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 (Figure 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 (Figure 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. Using POM

POM is implemented on top of Reflex, a versatile kernel for multi-language aspect-oriented programming (Section 6.1). Similar to SOM, we designed a small language to configure POM, which can be seen as a very simple domain-specific aspect language. The syntax of this language is overviewed in Figure 6. As an alternative, it is possible to place Java 5 annotations directly in front of the classes and methods to synchronize. We illustrate the use of both the configuration language and the annotations along the various examples in the paper. Finally, POM can also be configured via a Java API exposed by the `POM` class, which we omit for clarity 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 (Figure 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 granted by calling `executeOldest` (Figure 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 Figure 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 whether a key is already

```
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.

```
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.

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 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; hence, 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 (Figure 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. Similar to the scheduler of Figure 7, `RWScheduler` is not reentrant and hence deadlocks on reentrant calls. A full version is presented in Section 4.4.

As mentioned earlier, method categories make it possible to enhance reuse of synchronization policies. For instance, the `RWScheduler` (Figure 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 carried out declaratively using the POM configuration language introduced in Figure 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 ;
```

We can reuse the category-based scheduler `RWScheduler` to synchronize another data structure by simply defining categories appropriately. For instance, the following is the definition of a `Stack` class, which uses the annotation-style definition of POM:

```
@POMSyncClass(scheduler = RWScheduler.class)
public class Stack {
  @POMSync(category = "WRITER") public void push(Object o){...}
  @POMSync(category = "WRITER") public Object pop(){...}
  @POMSync(category = "READER") public Object peek(){...}
  @POMSync(category = "READER") public boolean isEmpty(){...}
}
```

The class to synchronize is marked with the `POMSyncClass` annotation, which also indicates the class that acts as the scheduler (here, `RWScheduler`). The `POMSync` annotation for methods is used to selectively mark methods to synchronize, and is used here with the optional category attribute to define the `READER` and `WRITER` categories.

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 Figure 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,

[§]Note that a `schedule` declaration does not define any namespace of its own: method names (e.g. `query`, `size`, etc.) are searched in the scheduled class (e.g. `Dictionary`), and category names (e.g. `READER`) are searched in the public static fields of the scheduler class (e.g. `RWScheduler`). It is currently not possible to disambiguate between overloaded methods; hence all the methods that have the same name are selected at once.

```
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.

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 writers problem discussed previously. Therefore, without POM, the code of the `for`-loop of Figure 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` (Figure 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-specific policies. Note 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

```
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.

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

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

This code states that `eat` requests on `Philosophers` 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. The same effect can be achieved using annotations, using the `group` attribute of the `POMSyncClass` annotation to specify the group definition:

```
@POMSyncClass(scheduler = PhiloSched.class, group = TableGroup.class)
public class Philosopher { @POMSync public void eat() {...} ... }
```

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.

```
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)){
      fail(req, new RuntimeException("reader cannot invoke writer!"));
} }
```

Figure 11. A reentrant scheduler for readers and writers.

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.

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` (Figure 11) is a reentrant extension of `RWScheduler` (Figure 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 causes the thread that issued an illegal request to throw an exception, because 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.

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: SOMs [4], chords [5], and synchronizers [1].

[¶]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.

5.1. Sequential object monitors

SOMs [4] 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. Figure 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 similar to that in Figure 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)`.

```
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.

- 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$^\sharp$ [5]. 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$^\sharp$. 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$^\sharp$, 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 Figure 13. Similar to Figure 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$^\sharp$ implementation [5] in POM chords are shown. The mapping from Polyphonic C$^\sharp$ syntax to POM chords is straightforward. Defining a chord consists in:

- 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 Figure 14: we show only 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 Figure 13`(3)–(6)`).

```
public class RWChordScheduler extends ChordScheduler {
  static Category READER = category(), 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.

```
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 whether a chord is enabled, as explained in [5]. The leaving method is trivial: if a request completing the 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, although 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 are generated in case of violations.

5.3. Synchronizers

The synchronizers of Frølund and Agha offer a declarative interface to specify multi-object coordination [1]. 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 with the `disables` operator are evaluated in mutual exclusion. In the 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 on the relative order of request arrival. This is not feasible with synchronizers. Guard-like conditions for the `disables` operator is expressed in POM as request filters operating on the pending queue, as in [4].

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 execute) 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 focused here only on the core semantics of synchronizers in POM.

6. IMPLEMENTATION

We now discuss elements of our implementation of parallel object monitors and report on performance evaluation.

6.1. POM as a Reflex plugin

The POM implementation for Java is based on Reflex, a versatile kernel for multi-language AOP [3]. Reflex is designed to be a powerful back-end to implement both general-purpose and domain-specific aspect languages. It is based on a reflective model that makes it possible to define customized metaobject protocols (MOPs), with expressive selection means to precisely configure where and when reification occurs [18].

A POM scheduler is a metaobject that takes control *before* and *after* a method is invoked. The base class `POMScheduler` implements the POM MOP, which consists of the `callTrap` and `returnTrap` methods as discussed below, and exposes the interface of the POM system to subclasses defined by users, in particular the `schedule` and `leave` methods.

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.). The same can be done alternatively using Java 5 annotations, as illustrated in this paper.

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 [19]. SOM [4] 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 SOM and an object controlled by a POM, can be detected and forbidden. Finally, Reflex is based on the efficient bytecode transformation library Javassist [20] and can operate either offline, as a compile-time utility, or online, as a Java VM agent.

6.2. Scheduler implementation

The POM Reflex plugin sets up a MOP so that the `callTrap` and `returnTrap` of the appropriate `SOMScheduler` metaobject are called whenever scheduled methods are entered and exited.

The `callTrap` method receives two parameters: the name of the invoked method and its category. The ability of Reflex to define customized MOPs permits to determine the value of these parameters statically at weaving time for each metaobject call location, thus limiting the overhead of reifying request data. The `callTrap` method wraps the method name and category, as well as the parent request (for reentrancy control) in a `Request` object. It then acquires a lock on the scheduler object, using the `ReentrantLock` of Java 5, adds the request to the request queue and calls the user-provided `schedule` method. Once the `schedule` method terminates, the lock is released and the current thread waits for the active request to be granted permission to execute; this can occur immediately if the recently executed `schedule` method already gave permission to the request, or postponed until the `schedule` method running in another thread gives permission to the request. Threads waiting for the permission to execute requests are blocked and woken up using the low-level Java 5 `LockSupport` primitives.

Once the request has been executed, the `returnTrap` method acquires the lock to call the user-provided `leave` method, and if there are pending requests in the queue, the `schedule` method is run.

Note the cooperative scheduling strategy: a POM scheduler (which defines the `schedule` and `leave` methods) is a *threadless* object. The job of scheduling requests is shared by client threads, as they enter and exit their requested methods. This is why, upon exiting from a requested method, a client thread spends some time running the `leave` method and possibly the `schedule` method, during which it can enable other client threads to proceed. This is the same cooperative scheduling strategy as that used in SOM [4], and has the benefit of avoiding attaching a thread to the scheduler and paying for many context switches, as is the case with traditional active objects approaches.

6.3. 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 Figure 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 = category(), 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.

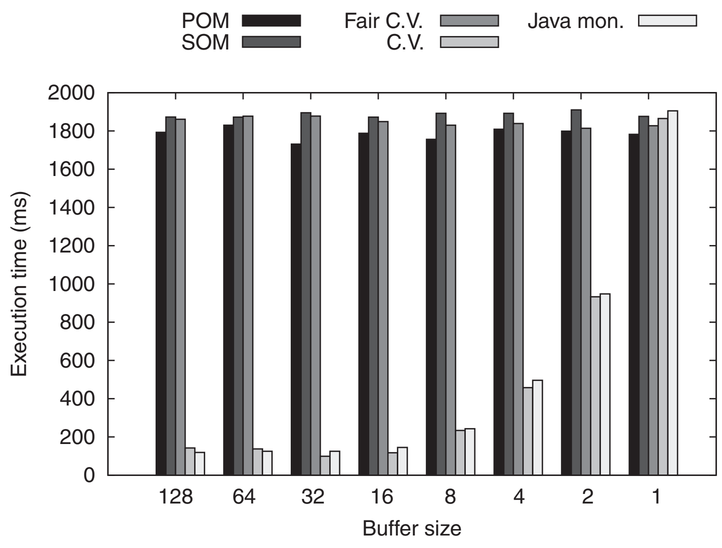


Figure 16. One producer/1 consumer on four CPUs.

6.3.1. *Producer and consumer on four CPUs*

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 4-CPU machine (two-way Dual-Core Xeon 2.33 GHz), running Red Hat Enterprise Linux ES release 4 and the JDK 1.6.0 virtual machine from Sun.

Results are shown in Figure 16. The horizontal axis represents the maximal size of the buffer and the vertical axis represents the execution times 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 [21].
- *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 [4].
- *POM*: The buffer synchronized with POM, as shown in Figure 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 with the time to synchronize access to the buffer: hence in such a situation, four 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 the presence of high contention for the different monitors.

The *C.V.* and *Java mon.* implementations are by far the most efficient. The other implementations are much slower because they are victims of *processor oscillation* phenomena caused by their fair allocation policy. To better understand this phenomenon, consider a processor executing the producer, owning the monitor, while another 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 legacy monitors and 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 monitor 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 the performance: actually, having a buffer of more than one slot is useless in this case. The unfair legacy monitors and Java 5 locks face only this processor oscillation phenomenon when the buffer is of size 1, as clearly reported in Figure 16.

6.3.2. Producer and multiple consumers on four CPUs

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.

The results (Figure 17) show that both SOM and POM are slightly more efficient than the Java 5 solution. Moreover, legacy Java monitors (Figure 17(b)) become severely inefficient for high numbers of consumers (almost 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, a number 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 four CPUs is useless in this scenario because of the high contention of the monitor.

6.3.3. Readers and writers on four CPUs

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`).
- *Java mon*: 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.

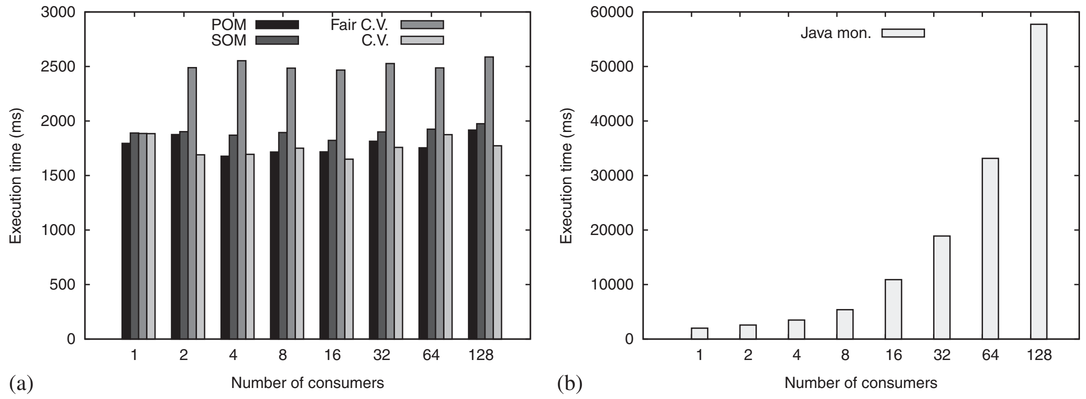


Figure 17. One producer/multiple consumers, buffer size 1, on four CPUs.

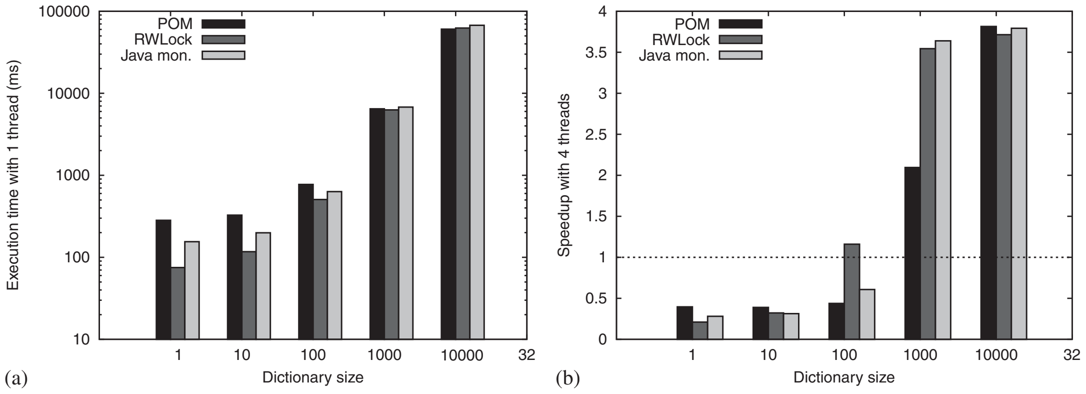


Figure 18. Readers and writers on four CPUs: (a) execution time with 1 thread and (b) speedup with four threads.

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 *four* threads making each 250 000 queries. In the latter case, the four CPUs of the machine are exploited, but there should be some contention to access the monitor: indeed, even if all threads can make queries in parallel, they still need to access the internal data of the monitors in mutual exclusion. We do not expect any performance improvement in having four threads for small dictionaries, because of the high monitor contention. On the other hand, for big dictionaries, we expect a speedup approaching $4\times$ when using four threads compared with only one. Also, overall timing should increase while increasing the dictionary size.

Figure 18 shows the results of this experiment. For each case, we give the execution time for one thread (Figure 18(a)) and the calculated speedup (dividing the time for one thread by the time for four threads, Figure 18(b)). The measurements basically confirm our expectations: having four threads for small dictionaries degrades performance. For the Java 5 locks solution, having four threads becomes beneficial for a dictionary of size between 10 and 100, while for legacy monitors and POM the breaking point is between 100 and 1000. For big dictionaries, the three implementations perform similarly, because the predominant work is the linear search.

With very small dictionary sizes and only one thread, Java 5 locks perform twice as fast as legacy monitors and four times as fast as POM. With four threads, the results are more balanced but the order is maintained.

6.3.4. Evaluation

These benchmarks show that POMs (and SOM) are competitive with existing, low-level synchronization techniques. In cases with high contention, POM is on par with Java 5 locks, and is significantly faster than legacy Java monitors. When parallel execution is required, POM exhibits a reasonable overhead, which we believe is acceptable when considering the gains in expressiveness, simplicity, and modularity brought by our approach.

Indeed, the different concurrency tools coming with Java 5 are considerably more complex to use correctly than POM. As an illustration, the implementation of the efficient synchronized buffer with Java 5 that we used in the benchmarks is given in Figure 19. It ought to be contrasted with the POM implementation of Figure 15: the synchronization logic is spread out in various places and is complex due to explicit lock manipulation. Conversely, in the POM solution, the bounded buffer is a simple, unsynchronized class (not presented here) and the synchronization logic is localized in the scheduler and easy to understand.

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

7.1. Generality vs Specificity

POM makes it possible to tune the generality of a given scheduler. A scheduler can be highly generic and reusable (similar to the scheduler of Figure 8, which relies on method categories), or it can be very application-specific (similar to the scheduler for philosophers in Figure 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.

7.2. Relation to AOP

The approach we use *is* indeed aspect oriented [22]: 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

```
public class CVBuffer extends Buffer {
  private final ReentrantLock mutex;
  private final Condition notFull, notEmpty;
  private int maxsize;
  private List buf = new LinkedList();

  public CVBuffer(int capacity) {
    maxsize = capacity;
    mutex = new ReentrantLock();
    notFull = mutex.newCondition();
    notEmpty = mutex.newCondition();
  }

  public void put(Object x) {
    try {
      mutex.lock();
      while (maxsize == buf.size()) { notFull.await(); }
      buf.add(x);
      notEmpty.signal();
    }
    catch (InterruptedException e) { ... }
    finally { mutex.unlock(); } }

  public Object get() {
    Object x = null;
    try {
      mutex.lock();
      while (buf.size() == 0) { notEmpty.await(); }
      x = buf.remove(0);
      notFull.signal();
    }
    catch (InterruptedException e) { ... }
    finally { mutex.unlock(); }
    return x;
} }
```

Figure 19. Implementation of the bounded buffer synchronized with a reentrant lock and condition objects of Java 5.

(the scheduler) as well as a small domain-specific language for the binding between base entities and schedulers. We have also illustrated how this binding specification can be done explicitly in source code using annotations. The success of the annotation style of AspectJ aspects shows that although using annotations directly in the code compromises syntactic obliviousness, it can be more convenient in cases where syntactic obliviousness is not required.

The gain with respect to directly using a general-purpose aspect language such as AspectJ is twofold: first, the configuration language exposes only the joinpoints that do make sense in the context of POM (method executions), 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 feasible to devise an implementation of POM using a general-purpose aspect language such as AspectJ [23], although some features may be more cumbersome to provide:

- Implementing categories (Section 4.2) requires passing metadata from the pointcut to the advice. In AspectJ it would be necessary, on *each* invocation, to use reflection through `thisJoinPoint` to obtain the name of the invoked method, and then perform a lookup of the corresponding category. With Reflex, the category metadata is determined at weaving time and can therefore be passed directly to the metaobject, without using reflective features.

- Group synchronization (Section 4.3) requires associating a particular aspect instance with an arbitrary set of objects, a feature that is not provided by AspectJ. This limitation of AspectJ was noted by Sakurai *et al.* [24], who developed an extension of it, which is however not included in the standard AspectJ distribution.

Furthermore, we have chosen to include POM in our more general research artifact for multi-language AOP, in order to be able to study related issues such as aspect interactions across languages [3,19].

7.3. Inheritance anomaly

The interaction between inheritance and concurrency control is well known to be difficult and raises a number of *inheritance anomalies* [25,26]. According to the taxonomy given in [26], the different occurrences of the inheritance anomaly can be categorized into three classes:

History-sensitiveness of acceptable states: Extending a class with methods that require a synchronization based on past activity implies redefining all methods of the base class to manually keep track of the bits of history that are relevant.

Partitioning of states: Introducing a new logical state in the synchronization strategy implies pervasive changes to the original synchronization specifications.

Modification of acceptable states: This last class of anomaly refers to the impossibility of using mixin-like extensions dealing with synchronization out of the box: classes that are extended via a synchronization mixin have to be manually modified.

Recently, Milicia and Sassone [27] have studied the impact of new modularization approaches such as composition filters [28] and aspect-oriented programming [22] on the inheritance anomaly. They showed that decoupling the synchronization concern from the business logic of a module improves the situation, in particular with respect to the third class of anomaly. However, proper modularization of synchronization does not by itself eliminate the inheritance anomaly [27]. In particular, the history sensitiveness class of anomaly generally requires tedious rewriting, extension, and book-keeping code. Jeeg [29] is an aspect language extending guards with linear temporal logic (LTL); this way, Jeeg provides history-sensitive expressiveness that allows a straightforward tackling of these kinds of anomalies (at least, for constraints that can be described as star-free regular languages over the execution traces of objects). However, none of the modern languages discussed in [27] are capable of tackling all cases of inheritance anomaly; as discussed, the very nature of the anomaly makes it hard to make any strong claim in this regard, and most likely further experience with OOP and concurrency will unveil new kinds of anomalies.

POM clearly falls in the category of modern approaches to concurrency that aim at decoupling the synchronization concern from the rest of the application. This by itself helps in certain kind of anomalies, but not all. Still, since POM chooses to reify incoming requests; the pending queue of requests, the scheduling strategy, reentrancy, and leaving requests; all elements are at hand of the programmer. For instance, if one wishes to tackle history sensitiveness, one can provide a base-scheduler class that delegates to subclasses the task of returning all requests to be executed (rather than executing them directly). This base-scheduler class then triggers execution of the requests in that list. This indirection provides the necessary hook for a scheduler subclass to ‘remember’ the requests that were previously dispatched, and hence makes it possible to provide abstractions inspired by LTL like in Jeeg, such as `previous`, `since`, etc. Therefore, the powerful imperative framework of POM can serve as a base to devise different abstractions to address particular classes of the anomaly.

7.4. Transactional memory

Lately, the research community is putting a lot of attention on software transactional memory (STM) [30]. 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 these problems is the responsibility of programmers and is generally difficult. Some implementations of STM also avoid livelocks but with extra overhead.

STM is more light-weight than database 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 performance of lock-based systems [30] and sometimes exceeding them [31,32]. One problem associated with the STM is that the programmer must access data through a special STM API [30,33], hence requiring extensive changes to software. A Java language extension has been proposed to alleviate this issue, relying on an extended compiler [31,32], 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 turns 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 [31,32].

7.5. 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 provision 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 only 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 blocked only 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 called only to determine whether a philosopher can eat; while the philosopher is eating, the scheduler monitor is not blocked; hence, other executions of the scheduling or leaving methods can proceed. Similarly, calls to the leaving methods are used only 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. Hence, a very promising approach would actually be to use transactional memory for the POM schedulers, in order to avoid requesting locks for scheduler execution.

Finally, in this work we do not analyze and address the issues raised by the interactions between various POMs. For now, we consider that the user must take care not to introduce such interactions, exactly as is the case with other concurrency abstractions, and leave the further study of this issue for future work. In particular, recent work on omniscient debugging can bring new perspectives on detecting subtle concurrency issues [34].

8. CONCLUSION AND FUTURE WORK

We have presented parallel object monitor (POM) 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 [1] with the expressiveness provided by an explicit access to the request queue, as in scheduler approaches [4,14]. 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. On the basis of 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 such as sequential object monitors (SOMs) [4] and chords [5], as well as the synchronizers of [1]. 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, in terms of both configuration control and scheduling specification. Second, the POM chord abstraction presented here represents a powerful starting point for several alternatives of join patterns, such as n-way rendezvous. Third, the relation with software transactional memory (STM) 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 [3].

Availability: POM and SOM are available at http://reflex.dcc.uchile.cl/POM under the MIT open source license.

ACKNOWLEDGEMENTS

We thank Oscar Nierstrasz and Sébastien Vaucouleur for their comments on a draft of this paper, as well as the anonymous reviewers for suggesting various improvements to this article. 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, as well as by the FONDECYT Project 11060493.

REFERENCES

1. Frølund S. Agha G. A language framework for multi-object coordination. *Proceedings of the 7th European Conference on Object-oriented Programming* (*ECOOP 93*), Kaiserslautern, Germany, July 1993 (*Lecture Notes in Computer Science*, vol. 707), Nierstrasz O (ed.). Springer: Berlin, 1993; 346–360.
2. Matsuoka S, Watanabe T, Yonezawa A. Hybrid group reflective architecture for object-oriented concurrent reflective programming. *Proceedings of the 5th European Conference on Object-oriented Programming* (*ECOOP 91*), Geneva, Switzerland, July 1991 (*Lecture Notes in Computer Science*, vol. 512), America P (ed.). Springer: Berlin, 1991; 231–250.
3. Tanter É. Noyé J. A versatile kernel for multi-language AOP. *Proceedings of the 4th ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering* (*GPCE 2005*), Tallinn, Estonia, September/October 2005 (*Lecture Notes in Computer Science*, vol. 3676), Glück R, Lowry M (eds.). Springer: New York, 2005; 173–188.
4. Caromel D, Mateu L, Tanter É. Sequential object monitors. *Proceedings of the 18th European Conference on Object-oriented Programming* (*ECOOP 2004*), Oslo, Norway, June 2004 (*Lecture Notes in Computer Science*, vol. 3086), Odersky M (ed.). Springer: Berlin, 2004; 316–340.
5. Benton N, Cardelli L, Fournet C. Modern concurrency abstractions for C$^\sharp$. *ACM Transactions on Programming Languages and Systems* 2004; **26**(5):769–804.
6. Brinch Hansen P. A programming methodology for operating system design. *Proceedings of the IFIP Congress 74*, Amsterdam, Holland, August 1974. North-Holland: Amsterdam, 1974; 394–397.
7. Hoare CAR. Monitors: An operating system structuring concept. *Communications of the ACM* 1974; **17**(10):549–577.
8. Dijkstra EW. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM* 1975; **18**(8):453–457.
9. Hoare CAR. Communicating sequential processes. *Communications of the ACM* 1978; **21**(8):666–677.
10. Birtwhistle GM, Dahl OJ, Myhrhaug B, Nygaard K. *Simula Begin*. Chartwell-Bratt Ltd., 1979. ISBN: 086238009X.
11. Atkinson C, Maio AD, Bayan R. Dragoon: An object-oriented notation supporting the reuse and distribution of Ada software. *Proceedings of the 4th International Workshop on Real-time Ada Issues*, Pitlochry, Perthshir, Scotland, 1990; 50–59.
12. Briot J-P, Guerraoui R, Löhr K-P. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys* 1998; **30**(3):291–329.
13. Agha G. *ACTORS*: *A Model of Concurrent Computation in Distributed Systems*. The MIT Press: Cambridge, MA, 1986.
14. Caromel D. Towards a method of object-oriented concurrent programming. *Communications of the ACM* 1993; **36**(9): 90–102.
15. Nierstrasz O. Active objects in hybrid. *Proceedings of the 2nd International Conference on Object-oriented Programming Systems*, *Languages and Applications* (*OOPSLA 87*), Orlando, FL, U.S.A., October 1987 (*ACM SIGPLAN Notices*, vol. 22(12)), Meyrowitz N (ed.). ACM Press: New York, 1987; 243–253.
16. Frølund S. Inheritance synchronization constraints in concurrent object-oriented programming languages. *Proceedings of the 6th European Conference on Object-oriented Programming* (*ECOOP 92*), Utrecht, The Netherlands, July 1992 (*Lecture Notes in Computer Science*, vol. 615), Madsen OL (ed.). Springer: Berlin, 1992; 185–196.
17. Fournet C, Gonthier G. The reflexive chemical abstract machine and the join-calculus. *Proceedings of POPL'96*. ACM: New York, 1996; 372–385.
18. Tanter É, Noyé J, Caromel D, Cointe P. Partial behavioral reflection: Spatial and temporal selection of reification. *Proceedings of the 18th ACM SIGPLAN Conference on Object-oriented Programming Systems*, *Languages and Applications* (*OOPSLA 2003*), Anaheim, CA, U.S.A., October 2003 (*ACM SIGPLAN Notices*, vol. 38(11)), Crocker R, Steele GL Jr (eds.). ACM Press: New York, 2003; 27–46.
19. Tanter É. Aspects of composition in the Reflex AOP kernel. *Proceedings of the 5th International Symposium on Software Composition* (*SC 2006*), Vienna, Austria, March 2006 (*Lecture Notes in Computer Science*, vol. 4089), Löwe W, Südholt M (eds.). Springer: Berlin, 2006; 98–113.
20. Chiba S, Nishizawa M. An easy-to-use toolkit for efficient Java bytecode translators. *Proceedings of the 2nd ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering* (*GPCE 2003*), Erfurt, Germany, September 2003 (*Lecture Notes in Computer Science*, vol. 2830), Pfenning F, Smaragdakis Y (eds.). Springer: Berlin, 2003; 364–376.
21. Lea D. *Concurrent Programming in Java* (2nd edn) (*The Java Series*). Addison Wesley: New York, 1999.
22. Elrad T, Filman RE, Bader A. Aspect-oriented programming: Introduction. *Communications of the ACM* 2001; **44**(10): 29–32.

23. Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J, Griswold W. An overview of AspectJ. *Proceedings of the 15th European Conference on Object-oriented Programming* (*ECOOP 2001*), Budapest, Hungary, June 2001 (*Lecture Notes in Computer Science*, vol. 2072), Knudsen JL (ed.). Springer: Berlin, 2001; 327–353.
24. Sakurai K, Masuhara H, Ubayashi N, Matsuura S, Komiya S. Association aspects. *Proceedings of the 3rd International Conference on Aspect-oriented Software Development* (*AOSD 2004*), Lancaster, U.K., March 2004, Lieberherr K (ed.). ACM Press: New York, 2004; 16–25.
25. Briot J-P, Yonezawa A. Inheritance and synchronization in concurrent oop. *Proceedings of the 1st European Conference on Object-oriented Programming* (*ECOOP 87*) (*Lecture Notes in Computer Science*, vol. 276). Springer: Berlin, 1987; 32–40.
26. Matsuoka S, Yonezawa A. Analysis of inheritance anomaly in object-oriented concurrent programming languages. *Research Directions in Concurrent Object-oriented Programming*, Agha G, Wegner P, Yonezawa A (eds.). MIT Press: Cambridge, MA, 1993; 107–150.
27. Milicia G, Sassone V. The inheritance anomaly: Ten years after. *Proceedings of the 2004 ACM Symposium on Applied Computing* (*SAC'04*), New York, NY, U.S.A., 2004. ACM Press: New York, 2004; 1267–1274.
28. Bergmans L, Akşit M. Composing crosscutting concerns using composition filters. *Communications of the ACM* 2001; **44**(10):51–57.
29. Milicia G, Sassone V. Jeeg: A programming language for concurrent objects synchronization. *Proceedings of Joint ACM Java Grande—ISCOPE Conference (JGI 2002)*, Seattle, WA, November 2002; 212–221.
30. Shavit N, Touitou D. Software transactional memory. *Proceedings of PODC'95*, Ottawa, Ont., Canada, August 1995; 204–213.
31. Harris T. Fraser K. Language support for lightweight transactions. *Proceedings of the 18th ACM SIGPLAN Conference on Object-oriented Programming Systems*, *Languages and Applications* (*OOPSLA 2003*), Anaheim, CA, U.S.A., October 2003 (*ACM SIGPLAN Notices*, vol. 38(11)), Crocker R, Steele GL Jr (eds.). ACM Press: New York, 2003.
32. Crocker R, Steele GL Jr (eds). *Proceedings of the 18th ACM SIGPLAN Conference on Object-oriented Programming Systems*, *Languages and Applications* (*OOPSLA 2003*), Anaheim, CA, U.S.A., October 2003 (*ACM SIGPLAN Notices*, vol. 38(11)), ACM Press: New York, 2003.
33. Fraser K. Practical lock-freedom. *PhD Thesis*, King's College, University of Cambridge, September 2003.
34. Pothier G, Tanter É. Piquer J. Scalable omniscient debugging. *Proceedings of the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems*, *Languages and Applications* (*OOPSLA 2007*), Montreal, Canada, October 2007. ACM Press: New York, 2007.