Parallel Actor Monitors

Christophe Scholliers *[1], **Éric Tanter**[2], **Wolfgang De Meuter**[1]

[1] Software Languages Lab
Vrije Universiteit Brussel
Pleinlaan 2, Elsene, Belgium

[2]PLEIAD Laboratory
DCC University of Chile
Avenida Blanco Encalada 2120, Santiago, Chile

cfscholl@vub.ac.be, etanter@dcc.uchile.cl, wdmeuter@vub.ac.be

Abstract. *While the actor model of concurrency is well appreciated for its ease of use, its scalability is often criticized. Indeed, the fact that execution* within *an actor is sequential prevents certain actor systems to take advantage of multi-core architectures. In order to combine scalability and ease of use, we propose Parallel Actor Monitors (PAM), as a means to relax the sequentiality of intra-actor activity in a structured and controlled way. A PAM is a modular, reusable scheduler that permits to introduce intra-actor parallelism in a local and abstract manner. PAM allows the stepwise refinement of local parallelism within a system on a per-actor basis, without having to deal with low-level synchronization details and locks. We present the general model of PAM and its instantiation in the AmbientTalk language. Benchmarks confirm the expected scalability gain.*

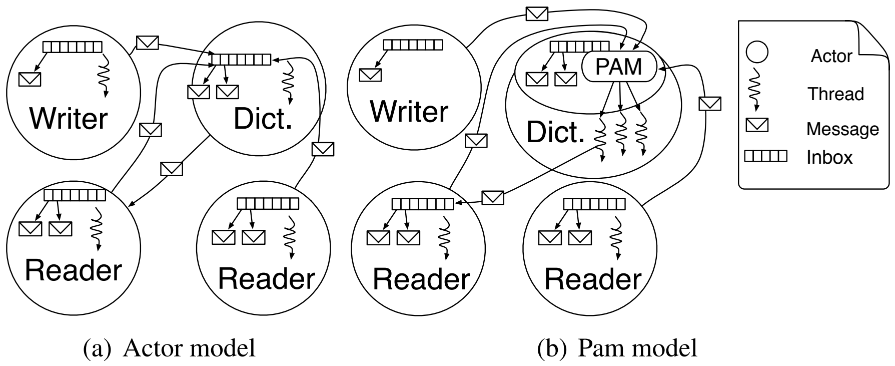
1. Introduction

The actor model of concurrency [Agha 1986] is well recognized for the benefits it brings for building concurrent systems. Actors are *strongly encapsulated* entities that communicate with each other by means of asynchronous message passing. Because data cannot be shared among actors, and actors process messages sequentially, there can be no data races in an actor system. However, these strong guarantees come at a cost: efficiency.

The overall concurrency obtained in an actor system stems from the concurrent execution of multiple actors. Each actor has only *one* thread of execution which dequeues the pending messages of its inbox and processes them *one by one*. Message passing between actors is purely asynchronous. Responding to a message is done either by sending another asynchronous message, or, if supported by the actor system, by resolving a future [Chatterjee 1989].

Let us further illustrate the actor model by means of an example shown in Figure 1(a). The example consists of four actors, one of which is a dictionary actor. The three other actors are clients of the dictionary: one actor does updates (writer), while the others only consult the dictionary (readers). The implementation of the dictionary example with actors is easy because the programmer does not need to be concerned with data races: reads *and* writes to the dictionary are ensured to be executed in mutual exclusion.

The programmer is sure that no other actor could have a reference to the data encapsulated in the dictionary actor. The asynchronous nature of the actor model also brings distribution transparency as the dictionary code is independent from the physical location of the actors.



(a) Actor model (b) Pam model

Figure 1. Dictionary Reader/Writer Example

However, when the number of readers increases the resulting application performs badly precisely because of the benefits of serial execution of requests to the dictionary actor: there are no means to process read-only messages in parallel and thus the dictionary actor becomes the bottleneck.

In order to obtain scalability in such a scenario, some form of *intra-actor concurrency* is needed. This has been acknowledged by current actor implementations however their solutions are both unsafe and ad-hoc as we show in the next section. In order to provide the programmer with intra-actor concurrency in a structured and high-level manner we propose the use of *parallel actor monitors* (PAM). In essence, a PAM is a *scheduler* that expresses a coordination strategy for the parallel execution of requests within a single actor. Figure 1(b) sketches the implementation of the dictionary example with a PAM. The dictionary actor has been changed by plugging in a generic, reusable PAM that implements the typical multiple-reader/single-writer coordination strategy.

There are four main contributions in applying the PAM model in order to overcome the bottleneck introduced by traditional actor systems.

1. **Efficiency.** A PAM makes it possible to take advantage of parallel computation for improved scalability. For the dictionary example, benchmarks of our prototype implementations suggest speedups that are almost linear to the number of processors available (Section 5).
2. **Modularity.** A PAM is a modular, reusable scheduler that can be parameterized and plugged into an actor to introduce intra-actor concurrency without modification of the original code. This allows generic well-defined scheduling strategies to be implemented in libraries and reused as needed. By using a PAM programmers can separate the coordination concern from the rest of the code.
3. **Locality.** Binding a PAM to an actor *only* affects the concurrency of that single actor. The scheduling strategy applied by one actor is completely transparent to other actors. This is because a PAM preserves the strong encapsulation boundaries between actors.

4. **Abstraction.** A PAM is expressed at the same level of abstraction as actors: the scheduling strategy realized by a PAM is defined in terms of a message queue, messages, and granting permissions to execute. A PAM programmer does *not* refer explicitly to threads and locks. It is the underlying PAM system that takes responsibility to hide the complexity of allocating and handling threads and locks for the programmer.

The next section gives an overview of the closest related work and shows why current approaches are not sufficient. Section 3 presents our parallel actor monitors in a general way, independent of a particular realization. Section 4 then overviews how our implementation of PAM on top of AmbientTalk is used to express canonical examples as well as a more complex coordination strategy. Section 5 details the implementation of PAM in AmbientTalk, and provides an initial assessment of the implementation through a set of benchmarks. Section 6 concludes.

2. Related Work

While a large amount of variations on the actor model exist nowadays one of the first stateful actor-like languages was ABCL[Yonezawa 1990]. When introducing state in the previously functional actor model one of the major design decisions for synchronization was the following:

> "*One at a time: An object always performs a single sequence of actions in response to a single acceptable message. It does not execute more than one sequence of actions at the same time.*"

Since then it has been one of the main rules in stateful actor languages. This is reflected in that main actor languages today have adopted this design decision including Erlang[Armstrong et al. 1996], ProActive[Baduel et al. 2006], E[Stiegler. 2004], Salsa[Varela and Agha 2001] and AmbientTalk[Dedecker et al. 2006]. In all these languages execution of parallel messages within an actor is disallowed by construction, i.e. every actor has only one thread of control and data cannot be shared between actors. As seen in the introduction this leads to scalability issues when resources have to be shared among a number of actors.

With this wild growth of actor languages it is not surprising that we are not the first ones to observe that the actor model is too strict [Hickey 2010]. In the following we discuss two typical alternatives used in actor-based systems to overcome this limitation. However, we argue that they are unsatisfying for being too ad-hoc and unsafe.

First, actor languages that are built on top of a thread-based concurrency system can allow an "escape" to the implementing substrate. For instance, AmbientTalk [Dedecker et al. 2006] supports symbiosis with Java [Gosling et al. 1996], which can be used to take advantage of multi-threading as provided by the Java language. However, such a backdoor reintroduces the traditional concurrency problems and forces the programmer to think in two different paradigms (actor model and thread based model).

Another approach is to introduce heterogeneity in the system by allowing actors to coexist with non-encapsulated, shared data structures. This is the case for instance in the ProActive actor-based middleware for Java [Baduel et al. 2006] A ProActive implementation of the dictionary example[1] makes the dictionary a "naked" data structure, that actors

[1]http://proactive.inria.fr/index.php?page=reader_writers

```
public void runActivity(org.objectweb.proactive.Body body) {
 while(!done) {
  coordinator.enterRead();
  //Read
  coordinator.exitRead();
 }
}
```

Listing 1. Reader Actor Coordination in ProActive

can access freely, concurrently. Avoiding data races is then done by *suggesting* client actors to request access through a coordinator actor (with methods such as `enterRead`, `exitWrite`, etc.), which implements a typical multiple-readers/single-writer strategy. An example of this approach is shown in Listing 1. In this example there is one reader actor, which first sends a message to the coordinator actor in order to get permission to read from the shared data structure. Once this message is processed by the coordinator actor the reader is sure that it can safely read from the shared data. After reading from this shared data the reader actor needs to signal the coordinator that it has finished using the shared data.

A major issue with this approach is that the model does not *enforce* clients to use the coordinator actor: nothing prevents an actor to access the shared data structure directly, thereby compromising thread safety. Readers and writers themselves have to notify the coordinator when they are finished using the shared resource; of course, failing to do so is possible and results in faulty programs. The introduction of shared data violates both locality and modularity, because the use of shared data influences the whole program; it basically reintroduces the traditional difficulties associated with threads and locks.

Next to actor-based systems other related work deals with the coordination and synchronization of messages in object-oriented systems. Join methods [Fournet and Gonthier 1996] are defined by a set of method fragments and their body is only executed when all method fragments are invoked. Implementation of Join methods can be found in C# [Benton et al. 2004] and Join Java[Itzstein and Jasiunas 2003]. Synchronizers [Frølund and Agha 1993] is a declarative modular synchronization mechanism to coordinate the access to one or a group of objects by enabling and disabling groups of method invocations. We come back to synchronizers and show their implementation in PAM in Section 4.

In summary, while the actor model has proven to be an appropriate mechanism for concurrent and distributed applications current approaches to deal with intra-actor concurrency are both ad-hoc and unsafe. As a solution we present PAM an abstraction which allows programmers to modularize their intra-actor coordination code in a local and abstract manner.

3. Parallel Actor Monitor

A parallel actor monitor, PAM, is a low-cost, thread-less *scheduler* controlling parallel execution of messages within an actor. Recall that an actor encapsulates a number of passive objects, accessed from other actors through asynchronous method calls. A PAM is therefore a *passive object* that controls the synchronization aspect of objects living within an actor, whose functional code is not tangled with the synchronization concern.

It is our objective to bring the benefits of the model of *parallel object monitors* (POM) [Caromel et al. 2008] to actor programming. It is therefore unsurprising that the operational description and guarantees of a parallel actor monitors closely resemble those of POM. POM is formulated in a *thread-based*, synchronous world: PAM is its adaptation to an actor-based, purely asynchronous *message-passing* model.

3.1. Operational Description

A PAM is a monitor defining a *schedule method* responsible for specifying how messages in an actor queue should be scheduled, possibly in parallel. A PAM also defines a *leave method* that is executed by each thread once it has executed a message. These methods are essential to the proposed abstraction, making it possible to reuse functional code as it is, adding necessary synchronization constraints externally. An actor system that supports PAM allows the definition of schedulers and the binding of schedulers to actors.

Figure 2 illustrates the operation of a PAM in more detail. The figure displays an actor runtime system hosting a single actor and its associated PAM, as well as a thread pool, responsible for allocating threads to the processing of messages. Several actors and their PAMs can live within the runtime system but at any moment in time a PAM can only be bound to one actor. When an asynchronous call is performed on an object hosted by an actor (1), it is put in the actor queue as a message object (2). Messages remain in the queue until the scheduling method (3) grants them permission to execute (4).

The scheduling method can trigger the execution of several messages. All selected messages are then free to execute *in parallel*, each one run by a thread allocated by the thread pool of the runtime system (5). Note that, if allowed by the scheduler, new messages can be dispatched before a first batch of selected messages has completed. In contrast to the traditional actor model, where messages are executed sequentially [Yonezawa 1990], a PAM enables the parallel execution of multiple messages.

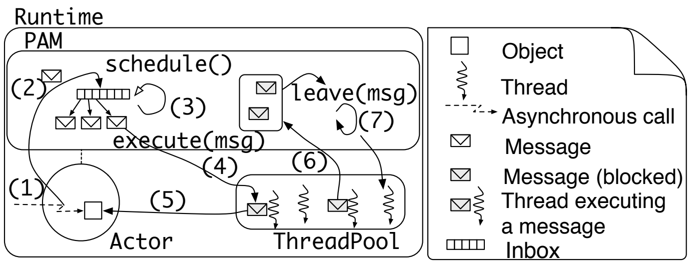


Figure 2. Operational sketch of PAM

Finally, when a thread has finished the execution of its associated message (6), the thread calls the leave method of the PAM (7). To run the leave method, a thread may have to wait for the scheduler monitor to be free (a PAM *is* a monitor): invocations of the scheduling and leaving methods are always safely executed, in *mutual exclusion*. A thread about to leave the monitor will first execute the scheduling method again in case the inbox of the actor is not empty. The fact that a thread spends some time scheduling requests for other threads (recall that the scheduler itself is a passive object) allows for a more efficient scheduling by avoiding unnecessary thread context switches.

One of the main operational differences between PAM and POM is a direct consequence of the underlying paradigm. In POM there are several threads in the system which are being coordinated by the POM while a PAM coordinates messages. Moreover in PAM, the thread of the caller can not be reused, as done in POM, because this would block the caller and violate the asynchronous nature of the model.

3.2. PAM at runtime

To further illustrate the working of a PAM, let us consider an actor whose PAM implements a simple join pattern coordination: when both a message a and a message b have been received by the actor, both can proceed in parallel. Otherwise, the messages are left in the queue. Figure 3 shows a *thread diagram* of the scenario. 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). The diagram shows two threads T1 and T2 (from the threadpool), initially idle, available for the activity of the considered actor. The state of the actor queue is initially empty.

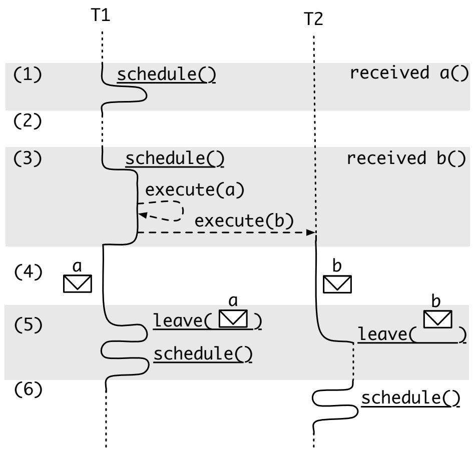


Figure 3. A simple join pattern coordinated by a PAM. ***(underlined method calls are performed in mutual exclusion within the scheduler)***

When a message a is received in the queue, T1 runs the schedule method (1). Since there is no message b in the queue, nothing happens, T1 remains idle (2). When a message b is received, T1 runs again the schedule method (3). This time, both messages a and b are found in the queue, so they are both dispatched in parallel. First a is dispatched, then b. T1 finishes the execution of the schedule method, while T2 starts processing the b message. Then, both T1 and T2 are executing, in parallel, their respective messages (4). When T1 finishes processing a, T1 calls the leave and then the schedule method, in mutual exclusion (5). Meanwhile, T2 also finishes processing its message but has to wait until the PAM is free in order to execute both methods itself (6). Note that the schedule method is only called at this point if there are pending messages in the queue.

3.3. PAM Guarantees

An actor system supporting PAM should support the following guarantees to the programmers.

1. *An asynchronous message sent to an object encapsulated by an actor that is bound to a PAM is guaranteed to be scheduled by this PAM.*

2. *The schedule and leave method of a PAM are guaranteed to be executed in mutual exclusion within the PAM, but in parallel with the messages being executed.*
3. *The schedule method is guaranteed to be executed if a message may be scheduled and guaranteed not to be executed when there are no pending messages.*
4. *When the schedule method instructs the execution of a message, the message is executed in parallel with the PAM or as soon as the schedule method is finished.*
5. *When a message has been processed, it is guaranteed that the leave method is called once, and the schedule method at most once.*

3.4. Binding and Reuse

A PAM is a passive object that defines a scheduling strategy. Upon creation it is unbound and does not schedule any message. To be effective, a PAM must be *bound* to an actor. After binding a PAM to a specific actor, all the guarantees listed above hold. In order to enhance reuse of schedulers, it is important for a scheduler to be as independent as possible from the actors it can be bound to. More precisely, a PAM has to abstract over the actual message *names* to coordinate. In order to support the definition of abstract, reusable schedulers, PAM adopts the notion of *message categories*, also found in POM. Actual message names are associated to specific categories at *scheduler binding time*. For example, in the case of a dictionary, a query message would belong to the reader category, while a put message would belong to the writer category. By making use of message categories, a reader-writer PAM works not only with dictionaries whose methods have specific names, but can also be used by actors encapsulating other data structures. The use of categories is further illustrated in the following section.

4. Canonical examples

In this section, we first introduce our implementation of PAM in AmbientTalk, from the programmers point of a view. To illustrate its use, we then give the implementation of two classical schedulers and how to bind them to actors in a program. The first example is purely didactic, since it re-introduces sequentiality within an actor: a standard mutual exclusion scheduler. The second example is the classical multiple-reader/single-writer scheduler. Although we have implemented more canonical examples with PAM (*e.g.* dining philosophers) these two problems clearly show the typical use of PAM. Finally we show the PAM implementation of a more elaborate coordination abstraction called Synchronizers[Frølund and Agha 1993]. Because PAM is a reincarnation of POM within actors, it allows the implementation of advanced coordination mechanisms, like guards [Dijkstra 1975] and chords [Benton et al. 2004, Caromel et al. 2008].

4.1. PAM in AmbientTalk

AmbientTalk [Dedecker et al. 2006] is an actor language with an event loop concurrency model adopted from E [Stiegler. 2004]. In this model, an actor encapsulates a number of passive objects, which can be referenced by objects living in other actors. Communication between objects can only be conducted by sending asynchronous messages. Note that AmbientTalk is a *prototype-based* language, that is, objects are created ex-nihilo or by cloning existing objects, rather than by instantiating classes. Like E, Ambienttalk differentiates between synchronous message sends (`o.m()`) and asynchronous messages (`o←m()`). Synchronous messages can only be sent between objects enclosed by the same actor while objects enclosed by different actors communicate asynchronously.

executeLetter(L)	executeAll(F)	executeOldestLetter(F)
executeYougest(F)	executeAllOlderThan(F)	executeAllYougerThan(F)
executeOlderThan(F_a, F_b)	executeYoungerThan(F_a, F_b)	
scheduler: *codeblock*	listIncomingLetters()	Category()
bindScheduler: S on: A	tagMethod: M with: C_{tag} on: O	contains(F)

Table 1. PAM API

We extended AmbientTalk in order to allow intra-actor concurrency according to the PAM model presented in the previous sections. The core API of PAM (Table 1) supports abstractions to create and bind a PAM to an actor as well as to coordinate the asynchronous messages sent to an actor. A PAM is created by using the `scheduler:` constructor function, which expects a block of code that can contain variable and method definitions just like any other object. Every scheduler in AmbientTalk has to implement at least the `schedule` and `leave` methods. The scheduler constructor returns a passive object (the PAM), which can be bound to an actor using the `bindScheduler` construct. After a PAM has been bound to an actor all asynchronous messages sent to this actor will be scheduled by the PAM.

Inside a PAM the programmer can examine the inbox of the actor, which contains `letters`. Access to the inbox is granted by using `listIncomingLetters()`, which returns the list of messages in the inbox of the actor. A letter contains a `message` and a `receiver`. The receiver is the (passive) object that lives inside the actor to which the message was sent. It is possible to execute a single letter or a group of letters. To start the execution of a specific letter, the programmer can pass the letter to `executeLetter`.

For a scheduler to be reusable it has to abstract over the actual message names in order to coordinate the access to the actor. Programmers define PAM in abstract terms by making use of *message categories*. Message categories are constructed by calling the `Category` constructor which will create a category C with its unique category tag C_{tag}. The set of messages belonging to a category is specified by tagging methods with the category tag. A message M belongs to a message category C when it is targeted to invoke a method which is tagged with the category tag C_{tag}. For programmers convenience a method category is also a *filter*. A filter is an object that defines a `pass` predicate to select letters. A message category can be used as a filter that only passes letters that contain a message belonging to the message category.

Most PAM abstractions expect a filter as argument, for example `executeAll` initiates the execution of all letters in the inbox that pass the filter F. Similarly the `excuteOlderThan` function expects two filters and executes all the messages that pass filter F_b, only if they are older than the first letter that passed F_a. Functions from the API which can only execute one letter return a boolean indicating that a matching letter was executed or not. All the other methods, which can potentially instruct the execution of multiple letters, return an integer indicating the number of letters they triggered for execution. We further illustrate the use of the PAM abstractions along with the examples in the following sections.

4.2. Mutual Exclusion

In this section we show how the normal mutual exclusion behavior of traditional actor systems can be implemented in PAM. Note that this is purely for didactical reasons as mutual exclusion is the default behavior of an actor when no PAM is bound to it. The implementation of such a PAM is shown in Listing 2. The mutual exclusion PAM is created by means of the `mutex` constructor function which returns a new mutex PAM when applied.

The PAM implements the methods `schedule` and `leave` providing mutual exclusion guarantees to the actor. The PAM has one instance variable `working`, initialized to `false` to keep track of whether a message is currently being executed. The `schedule` method only triggers the execution of a message if there are no executing message already. If so, it executes the oldest letter in the inbox, if any. Its state is updated to the result of invoking `executeOldestLetter`, which indicates if a message was actually triggered or not. The `leave` method changes the state of the scheduler accordingly after a message has been processed. Finally the scheduler is instantiated and bound to the dictionary actor by the `bindScheduler` method as shown in Listing 3.

```
def mutex() { scheduler: {
 def working := false;
 def schedule() {
  if: !(working) then: {
   working := executeOldestLetter();
  };
 };
 def leave(letter) { working := false; };
 };
};
```

Listing 2. Mutual exclusion PAM

```
def dictionaryActor := actor: {
 def dictionary := object: {
  def put() { ... };
  def get() { ... };
 }
}


bindScheduler: mutex() on: dictionaryActor;
```

Listing 3. PAM Binding

Note that the definition of the scheduler is simple and does not deal with low-level synchronization and notification details. This is in contrast with conventional monitors where one has to explicitly notify waiting threads. In PAM the underlying system guarantees that after the leave method, the schedule method is automatically invoked if there are waiting letters. Also, note that the actor definition did not suffer any intrusive change.

4.3. Parallel dispatch

The reader-writer scheduler for coordinating the parallel access to a shared data structure is shown in Listing 4.

```
def RWSched() { scheduler: {
 def R := Category();
 def W := Category();
 def writing := false;
 def readers := 0;
 def schedule() {
  if: !(writing) then: {
   def executing :=
   super.executeAllOlderThan(R,W);
   readers := readers + executing;
   if: (readers == 0) then: {
       writing := super.executeOldest(W);
   };
  };
 };
  def leave(letter) {
   dispatch: letter as:
   [[R, { readers := readers - 1 }],
    [W, { writing := false }]];
 };
};};
```

Listing 4. Schedule and Leave methods of the Reader/Writer PAM

```
annotateMethod: `get with: RWSched.R on: dictionary;
annotateMethod: `put with: RWSched.W on: dictionary;
bindScheduler: RWSched() on: dictionaryActor;
```

Listing 5. Instantiating and binding of a PAM to an actor.

Like before, `RWSched` is a constructor function that, when applied, returns a fresh scheduler. The scheduler defines two method categories readers (`R`) and writers (`W`). In order to keep track of how many readers and writers are executing, the scheduler maintains two variables `writing` (boolean) and `readers` (integer). When the scheduler is executing a write letter, no further message can be processed. In case the scheduler is not executing a write letter, the scheduling method triggers the parallel execution of all the read letters that are older than the oldest write letter, if any, using `executeAllOlderThan`. This call returns the number of dispatched readers, used to update the `readers` state. If there were no reader letters to process (older than the oldest writer), the scheduler dispatches the oldest writer using `scheduleOldest`. This method returns true if the processing of a letter was actually dispatch, false otherwise. Note that this scheduler uses a fair strategy but could easily be modified to give priority to writers. Finally, the `leave` method updates its state according to which message has finished executing, by either decreasing the number of readers, or turning the `writer` flag to false. To do so it makes use of the `dispatch` construct which re-uses the message category filters in order to decide which code block to execute. In order to use this scheduler with a dictionary actor, one should just annotate the methods of the dictionary with the appropriate categories, and then instantiate and bind the scheduler to the Actor (Listing 5).

4.4. Synchronizers

A related coordination abstraction, called synchronizers, invented by Frølund and Agha offers a declarative mechanism to deal with multi-object coordination for thread based systems [Frølund and Agha 1993]. As an actor can encapsulate multiple passive objects, a PAM can be used to coordinate the access to this group of objects. In this section we show that the features of synchronizers can easily be expressed in PAM to coordinate the messages sent to the objects encapsulated by an actor. The three main synchronization abstractions defined by synchronizers are: `update` to update the state of the synchronizer, `disable` to disallow the execution of certain methods (in a guard-like manner), and `atomic` to trigger several methods in parallel in an atomic manner.

To illustrate the use of these constructs consider Listing 6, which shows an example synchronizer. The aim of this synchronizer is to restrict the total amount of simultaneous requests on two objects `adm1` and `adm2`. To do so the synchronizer keeps track of the amount of simultaneous requests performed on those objects with the `prev` integer variable. Every time the method `request` is called either on `adm1` or `adm2` the `prev` value is incremented. Similarly when the method `release` is called on one of both objects the `prev` is decreased. This is implemented in the synchronizer with the `when:` $MethodPattern$ `update:` $Codeblock$ construct. This construct registers the $CodeBlock$ to be executed whenever a method that matches the $MethodPattern$ is executed. Finally, requests to both objects are disabled when `prev` equals or exceeds `max` with the `when:` $Predicate$ `disable:` $MethodPattern$ construct.

```
def collectiveBound(adm1, adm2, max) { synchronizer: {
 def prev := 0;
 when: ((MethodCall: `request on: adm1).or(MethodCall: `request on: adm2)) update: {
  prev := prev + 1;
 };
 when: ((MethodCall: `release on: adm1) or (MethodCall: `release on: adm2)) update: {
  prev := prev - 1;
 };
 when: { prev >= max } disable:
 ((MethodCall: `request on: adm1).or(MethodCall: `request on: adm2));
 }; };
```

Listing 6. Synchronizer example from [Frølund and Agha 1993] with a PAM

The implementation of these registration constructs is shown in Listing 7. The update and disable constructs (Lines 6–7) keep track of their registrations in the `updateTable` and `disableTable` respectively (Lines 1–2). The `executeUpdates` method performs the actual invocation block given a letter that matches one of its registrations (Line 10). The `isDisabled(letter)` function returns a boolean indicating whether the given letter is currently disabled (Line 11). Similar to the `when:update:` and `when:disable:` construction the `atomic:` construction adds the method patterns to the `atomicTable` (Line 8). The `executeAtomic` method goes over the `atomicTable` and checks for each group of method patterns whether the inbox has a matching message for each of these patterns (Lines 13–18). If this is the case all these messages are executed in parallel and the executing variable is updated to keep track of the amount of messages which are being executed (Lines 15-16).

```
1   def updateTable := [];
2   def disableTable := [ default: {false} ];
3   def atomicTable := [];
4   def executing := 0;
5
6   def when: pattern update: block  { updateTable := updateTable + [[pattern, block]] }
7   def when: block disable: pattern { disableTable := [[pattern, lambda]] + disableTable; }
8   def atomic: patterns  { atomicTable := atomicTable + [patterns] }
9
10  def executeUpdates(letter) { dispatch: letter as: updateTable; }
11  def isDisabled(letter) { dispatch: letter as: disableTable;  }
12  def executeAtomic() {
13   atomicTable.each: { |MethodPatterns|
14     if: containsAll(MethodPatterns) then: {
15      executing :=  executeAll(MethodPatterns);
16      return true;
17     }
18   }
19   return false;
20  }
```

Listing 7. Synchronizers registration constructs implemented in AmbientTalk

The implementation of the schedule and leave methods of the PAM is shown in Listing 8. The execution of individual messages should be prevented when the actor is executing a group of atomic messages. Similarly starting the execution of a group of atomic messages should be prevented when there are currently executing individual messages in the actor. In order to keep track of which kind of messages and how many are executing the scheduler maintains two variables. `Atomic` and `executing` which respectively indicate the execution of a group of atomic messages and the number of currently executing messages. When the schedule method is called the scheduler makes sure that it is not

```
def schedule() {
 if: !atomic then: {
  if: (executing == 0 & executeAtomic()) then: {
   atomic := true;
  } else: {
   listIncomingLetters().each: { |letter|
    if: ( (!isDisabled(letter)).and: { !inAtomic(letter) }) then: {
     executeUpdates(letter);
     executing := executing + 1;
     super.executeLetter(letter);};};};};};};

def leave() {
 executing := executing - 1;
 if: (execucting == 0) then: { atomic := false}
};
```

Listing 8. Schedule method for implementing Synchronizers with PAM

executing an atomic block. If this is not the case the scheduler stops in order to prevent the execution of concurrent messages during the execution of an atomic block, otherwise the scheduler proceeds. When there are no currently executing messages (indicated by the executing variable) and the scheduler can start the execution of an atomic block the variable `atomic` is changed accordingly. Otherwise the scheduler walks over the mailbox of incoming letters and executes all the letters which are not disabled. For each letter which is not disabled the scheduler starts it's execution, updates the executing variable by incrementing it by one and executes all the updates. The leave method of the scheduler decrements the executing variable by one for each leaving letter and sets the atomic variable to false when the executing counter equals 0. Synchronizers offer the programmer a declarative mechanism for coordinating multiple objects, however they are more limited than our PAM abstraction: fairness cannot be specified at the application level; history-based strategies must be manually constructed; and finally, although synchronizers encapsulate coordination, their usage has to be explicit in the application, requiring intrusive changes to the existing code. Also, synchronizers explicitly reference method names: their reuse potential is therefore more limited than PAM, where the method categories allow for more abstraction.

5. Micro-benchmarks

We now report on micro-benchmarks of our PAM implementations. First we report on the AmbientTalk implementation in Java. The aim is to measure both the overhead of PAM by contrasting plain AmbientTalk with PAM using a mutual exclusion scheduler (Listing 2), and to measure the speedup obtained by using the reader-writer PAM scheduler (Listing 4). The results depicted on Figure 4 were obtained on an Intel Core 2 Duo with a processor speed of 1.8 GHz running Mac OS X (10.5.8). We measured the processing time of reading one hundred items from a dictionary actor of varying size. For each measurement, we take the average of 30 tests, discarding the extremes. These results show the low overhead of PAM ($<$ 6%) and the expected speedup (1.9x for the 32K and 150K dictionary), taking almost full advantage of the 2 cores. Our implementation uses standard threads and a standard threadpool mechanism which makes us believe that the same optimisations can be applied for a range of JVM based actor languages.

Actor languages implemented in C can be optimised similarly. We have developed a minimal actor system in C and added support for the PAM constructs and measured the throughput of messages in the actor system. The test consists of sending a number of

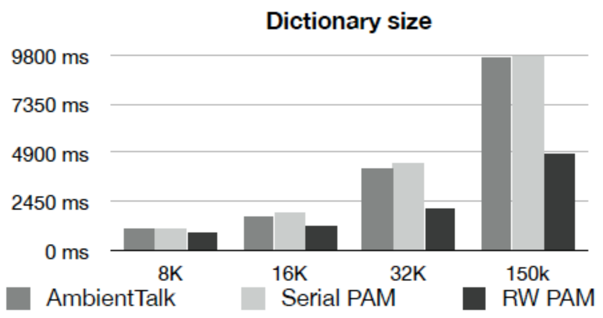
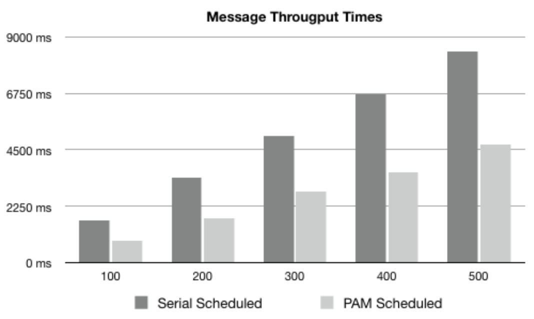
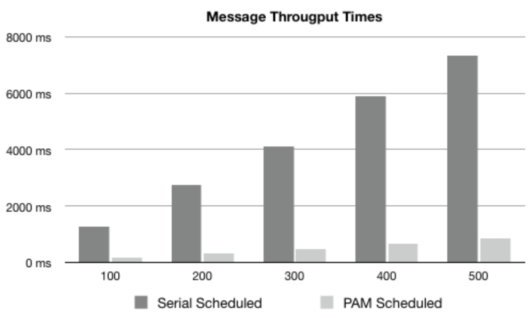


Figure 4. Speedup for the dictionary in the Java AmbientTalk implementation.



(a) Dual Core Laptop



(b) Eight Core MacPro

Figure 5. Throughput of messages in the C actor implementation of PAM

query messages from a sender actor to the PAM controlled dictionary actor which sends its final result to a receiving actor. The test is finished when all the results of the queries from the sender actor are received by the receiving actor. As the throughput highly depends on the speedup of the dictionary (with a higher speedup the dictionary actor can send the final result faster) this test actually measures both. The comparison between the PAM controlled reader/writer dictionary and a traditional serial actor scheduling for an increasing number of messages is shown in Figure 4. The left hand side shows the throughput speedup for a the dual core machine (1.8 for 500 messages). The right hand side shows the throughput speedup on a MacPro4 with two Quad-Core Intel Xeon processors. We have seen that our implementation again shows linear speedups of the throughput according to the number of cores (7.9 for 500 messages).

6. Conclusion

In order to address the strict restriction of sequentiality inside actors, we have proposed the model of Parallel Actor Monitors. Using PAM, there can be intra-actor concurrency, thereby leading to better scalability by making it possible for a single actor to take advantage of real concurrency offered by the underlying hardware. PAM offers a particularly attractive alternative to introduce concurrency inside actors, because it does so in a modular, local, and abstract manner: modular, because a PAM is a reusable scheduler, specified separately; local, because only the internal activity of an actor is affected by using a PAM; abstract, because the scheduler is expressed in terms of (categories of) messages, queues and granting permission to execute. Benchmarks on dual and oct core machines confirm the expected speedups both for computation as message throughput.

References

Agha, G. (1986). *Actors: a Model of Concurrent Computation in Distributed Systems*. MIT Press.

Armstrong, J., Virding, R., Wikström, C., and Williams, M. (1996). *Concurrent Programming in ERLANG*. Prentice Hall.

Baduel, L., Baude, F., Caromel, D., Contes, A., Huet, F., Morel, M., and Quilici, R. (2006). *Grid Computing: Software Environments and Tools*, chapter Programming, Deploying, Composing, for the Grid. Springer-Verlag.

Benton, N., Cardelli, L., and Fournet, C. (2004). Modern concurrency abstractions for c#. *ACM Trans. Program. Lang. Syst.*, 26(5):769–804.

Caromel, D., Mateu, L., Pothier, G., and Tanter, É. (2008). Parallel object monitors. *Concurrency and Computation—Practice and Experience*, 20(12):1387–1417.

Chatterjee, A. (1989). Futures: a mechanism for concurrency among objects. In *Proc. of the 1989 ACM/IEEE conf. on Supercomputing*, pages 562–567. ACM Press.

Dedecker, J., Van Cutsem, T., Mostinckx, S., D'Hondt, T., and De Meuter, W. (2006). Ambient-oriented Programming in Ambienttalk. In Thomas, D., editor, *ECOOP*, volume 4067 of *Lecture Notes in Computer Science*, pages 230–254. Springer.

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

Fournet, C. and Gonthier, G. (1996). The reflexive cham and the join-calculus. In *POPL '96*, pages 372–385, New York, NY, USA. ACM.

Frølund, S. and Agha, G. (1993). A language framework for multi-object coordination. In *ECOOP '93*, pages 346–360, London, UK. Springer-Verlag.

Gosling, J., Joy, B., and Steele, G. (1996). *The Java Language Specification*. GOTOP Information Inc.

Hickey, R. (2010). Message passing and actors. *Clojure online documentation, http://clojure.org/state*.

Itzstein, G. S. and Jasiunas, M. (2003). On implementing high level concurrency in java. In *In Proceedings of the Eighth Asia-Pacific Computer Systems Architecture Conference*, pages 151–165. Springer.

Stiegler., M. (2004). The E language in a walnut. *www.skyhunter.com/marcs/ewalnut.html*.

Varela, C. and Agha, G. (2001). Programming dynamically reconfigurable open systems with SALSA. *ACM SIGPLAN Notices. OOPSLA'2001*, 36(12):20–34.

Yonezawa, A., editor (1990). *ABCL: An Object-Oriented Concurrent System*. Computer Systems Series. MIT Press.