# Actors that Unify Threads and Events

Philipp Haller, Martin Odersky

LAMP-REPORT-2007-001
École Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland

## 1 Introduction

Concurrency issues have lately received enormous interest because of two converging trends: First, multi-core processors make concurrency an essential ingredient of efficient program execution. Second, distributed computing and web services are inherently concurrent. Message-based concurrency is attractive because it might provide a way to address the two challenges at the same time. It can be seen as a higher-level model for threads with the potential to generalize to distributed computation. Many message passing systems used in practice are instantiations of the actor model [1,11,12]. A popular implementation of this form of concurrency is the Erlang [3] programming language. Erlang supports massively concurrent systems such as telephone exchanges by using a very lightweight implementation of concurrent processes.

On mainstream platforms such as the JVM [16], an equally attractive implementation was as yet missing. Their standard concurrency constructs, shared-memory threads with locks, suffer from high initialization and context-switching overhead as well as high memory consumption. Therefore, the interleaving of independent computations is often modelled in an event-driven style on these platforms. However, programming in an explicitly event-driven style is complicated and error-prone, because it involves an inversion of control.

In previous work [10], we developed *event-based actors* which let one program event-driven systems without inversion of control. Event-based actors support the same operations as thread-based actors, except that the receive operation cannot return normally to the thread that invoked it. Instead the entire continuation of such an actor has to be a part of the receive operation. This makes it possible to model a suspended actor by a closure, which is usually much cheaper than suspending a thread.

One remaining problem in this work was that the decision whether to use event-based or thread-based actors was a global one. Actors were either event-based or thread-based and it was difficult to mix actors of both kinds in one system.

In this paper we present a unification of thread-based and event-based actors. There is now just a single kind of actor. An actor can suspend with a full stack frame (`receive`) or it can suspend with just a continuation closure (`react`). The first form of suspension corresponds to thread-based, the second form to event-based programming. The new system combines the benefits of both models. Threads support blocking operations such as system I/O, and can be executed on multiple processor cores in parallel. Event-based computation, on the other hand, is more lightweight and scales to large numbers of actors. We also present a set of combinators that allows a flexible composition of these actors.

The scheme has been implemented in the Scala *actors* library[1]. It requires neither special syntax nor compiler support. A library-based implementation has the advantage that it can be flexibly extended and adapted to new needs. In fact, the presented implementation is the result of several previous iterations. However, to be easy to use, the library draws on several of Scala's advanced abstraction capabilities; notably partial functions and pattern matching [7].

The user experience gained so far indicates that the library makes concurrent programming in a JVM-based system much more accessible than previous techniques. The reduced complexity of concurrent programming is influenced by the following factors.

- Message-based concurrency with pattern matching is at the same time more convenient and more secure than shared-memory concurrency with locks.
- Actors provide monitoring constructs which ensure that exceptions in sub-threads do not get lost.
- Actors are lightweight. On systems that support 5000 simultaneously active VM threads, over 1,200,000 actors can be active simultaneously. Users are thus relieved from writing their own code for thread-pooling.
- Actors provide good scalability on multiple processor cores. Speed-ups are competitive with high-performance fork/join frameworks.
- Actors are fully inter-operable with normal VM threads. Every VM thread is treated like an actor. This makes the advanced communication and monitoring capabilities of actors available even for normal VM threads.

*Related work.* Our library was inspired to a large extent by Erlang's elegant programming model. Erlang [3] is a dynamically-typed functional programming language designed for programming real-time control systems. The combination of lightweight isolated processes, asynchronous message passing with pattern matching, and controlled error propagation has been proven to be very effective [2,17]. One of our main contributions lies in the integration of Erlang's programming model into a full-fledged OO-functional language. Moreover, by lifting compiler magic into library code we achieve compatibility with standard, unmodified JVMs. To Erlang's programming model we add new forms of composition as well as *channels*, which permit strongly-typed and secure inter-actor communication.

Termite Scheme [9] integrates Erlang's programming model into Scheme. Scheme's first-class continuations are exploited to express process migration. However, their system apparently does not support multiple processor cores. All published benchmarks were run in a single-core setting.

The actor model has also been integrated into various Smalltalk systems. Actalk [6] is a library for Smalltalk-80 that does not support multiple processor cores. Actra [18] extends the Smalltalk/V VM to provide lightweight processes. In contrast, we implement lightweight actors on unmodified virtual machines.

SALSA (Simple Actor Language, System and Architecture) [19] extends Java with concurrency constructs that directly support the notion of actors. A preprocessor translates SALSA programs into Java source code which in turn is linked to a custom-built actor library. As SALSA implements actors on the JVM, it is somewhat closer related to our

[1] Available as part of the Scala distribution at http://scala.epfl.ch/.

work than Smalltalk-based actors. We compare performance of Scala actors with SALSA in section 6.

Timber [4] is an object-oriented and functional programming language designed for real-time embedded systems. It offers message passing primitives for both synchronous and asynchronous communication between concurrent *reactive objects*. In contrast to our programming model, reactive objects cannot call operations that might block indefinitely.

Frugal objects [8] (FROBs) are distributed reactive objects that communicate through typed events. FROBs are basically actors with an event-based computation model, just as our actors. The approaches are orthogonal, though. The former provide a *computing model* suited for resource-constrained devices, whereas our library offers a *programming model* (i.e. a convenient syntax) for event-based actors including FROBs.

Li and Zdancewic [15] propose a language-based approach to unify events and threads. By integrating events into the implementation of language-level threads, they achieve impressive performance gains. However, their approach is conceptually different from ours, as we build a unified abstraction *on top* of threads and events.

The rest of this paper is structured as follows. In the next section we introduce our programming model and explain how it can be implemented as a Scala library. In section 3 we introduce a larger example that is revisited in later sections. Our unified programming model is explained in section 4. Section 5 introduces channels as a generalization of actors. Experimental results are presented in section 6. Section 7 concludes.

## 2 Programming with actors

An actor is a process that communicates with other actors by exchanging messages. There are two principal communication abstractions, namely *send* and *receive*. The expression `a!msg` sends message `msg` to actor `a`. Send is an *asynchronous* operation, i.e. it always returns immediately. Messages are buffered in an actor's *mailbox*. The receive operation has the following form:

```
receive {
  case msgpat_1 => action_1
  ...
  case msgpat_n => action_n
}
```

The first message which matches any of the patterns $msgpat_i$ is removed from the mailbox, and the corresponding $action_i$ is executed. If no pattern matches, the actor suspends.

The expression `actor { body }` creates a new actor which runs the code in `body`. The expression `self` is used to refer to the currently executing actor. Every Java thread is also an actor, so even the main thread can execute `receive`[2].

The example in Figure 1 demonstrates the usage of all constructs introduced so far. First, we define an `orderManager` actor that tries to receive messages inside an infinite loop. The `receive` operation waits for two kinds of messages. The `Order(sender, item)` message handles an order for `item`. An object which represents

[2] Using `self` outside of an actor definition creates a dynamic proxy object which provides an actor identity to the current thread, thereby making it capable of receiving messages from other actors.

```
// base version
val orderManager = actor {
  while (true)
    receive {
      case Order(sender, item) =>
        val o = handleOrder(sender, item)
        sender ! Ack(o)
      case Cancel(sender, o) =>
        if (o.pending) {
          cancelOrder(o)
          sender ! Ack(o)
        } else sender ! NoAck
      case x => junk += x
    }
}
val customer = actor {
  orderManager ! Order(self, myItem)
  receive {
    case Ack(o) => ...
  }
}
```

```
// simplified version with reply and !?
val orderManager = actor {
  while (true)
    receive {
      case Order(item) =>
        val o = handleOrder(sender, item)
        reply(Ack(o))
      case Cancel(o) =>
        if (o.pending) {
          cancelOrder(o)
          reply(Ack(o))
        } else reply(NoAck)
      case x => junk += x
    }
}
val customer = actor {
  orderManager !? Order(myItem) match {
    case Ack(o) => ...
  }
}
```

**Fig. 1.** Example: orders and cancellations.

the order is created and an acknowledgment containing a reference to the order object is sent back to the sender. The `Cancel(sender, o)` message cancels order `o` if it is still pending. In this case, an acknowledgment is sent back to the sender. Otherwise a `NoAck` message is sent, signaling the cancellation of a non-pending order.

The last pattern `x` in the `receive` of `orderManager` is a variable pattern which matches any message. Variable patterns allow to remove messages from the mailbox that are normally not understood ("junk"). We also define a customer actor which places an order and waits for the acknowledgment of the order manager before proceeding. Since spawning an actor (using `actor`) is asynchronous, the defined actors are executed concurrently.

Note that in the above example we have to do some repetitive work to implement request/reply-style communication. In particular, the sender is explicitly included in every message. As this is a frequently recurring pattern, our library has special support for it. Messages always carry the identity of the sender with them. This enables the following additional operations:

| | |
|---|---|
| `a !? msg` | sends `msg` to `a`, waits for a reply and returns it. |
| `sender` | refers to the actor that sent the message that was last received by `self`. |
| `reply(msg)` | replys with `msg` to `sender`. |
| `a forward msg` | sends `msg` to `a`, using the current `sender` instead of `self` as the sender identity. |

With these additions, the example can be simplified as shown on the right-hand side of Figure 1.

Looking at the examples shown above, it might seem that Scala is a language specialized for actor concurrency. In fact, this is not true. Scala only assumes the basic thread model of the underlying host. All higher-level operations shown in the examples are defined as classes and methods of the Scala library. In the rest of this section, we look "under the covers" to find out how each construct is defined and implemented. The implementation of concurrent processing is discussed in section 4.

The send operation `!` is used to send a message to an actor. The syntax `a ! msg` is simply an abbreviation for the method call `a.!(msg)`, just like `x + y` in Scala is an abbreviation for `x.+(y)`. Consequently, we define `!` as a method in the `Actor` trait[3]:

```
trait Actor {
  private val mailbox = new Queue[Any]
  def !(msg: Any): unit = ...
  ...
}
```

The method does two things. First, it enqueues the message argument in the actor's mailbox which is represented as a private field of type `Queue[Any]`. Second, if the receiving actor is currently suspended in a `receive` that could handle the sent message, the execution of the actor is resumed.

The `receive { ... }` construct is more interesting. In Scala, the pattern matching expression inside braces is treated as a first-class object that is passed as an argument to the `receive` method. The argument's type is an instance of `PartialFunction`, which is a subclass of `Function1`, the class of unary functions. The two classes are defined as follows.

```
abstract class Function1[-a,+b] {
  def apply(x: a): b
}
abstract class PartialFunction[-a,+b] extends Function1[a,b] {
  def isDefinedAt(x: a): boolean
}
```

Functions are objects which have an `apply` method. Partial functions are objects which have in addition a method `isDefinedAt` which tests whether a function is defined for a given argument. Both classes are parameterized; the first type parameter `a` indicates the function's argument type and the second type parameter `b` indicates its result type[4].

A pattern matching expression `{ case` $p_1$ `=>` $e_1$`; ...; case` $p_n$ `=>` $e_n$ `}` is then a partial function whose methods are defined as follows.

- The `isDefinedAt` method returns `true` if one of the patterns $p_i$ matches the argument, `false` otherwise.

[3] A trait in Scala is an abstract class that can be mixin-composed with other traits.

[4] Parameters can carry `+` or `-` variance annotations which specify the relationship between instantiation and subtyping. The `-a, +b` annotations indicate that functions are contravariant in their argument and covariant in their result. In other words `Function1[X1, Y1]` is a subtype of `Function1[X2, Y2]` if `X2` is a subtype of `X1` and `Y1` is a subtype of `Y2`.

```
class InOrder(n : IntTree) extends Producer[int] {
  def produceValues = traverse(n)
  def traverse(n : IntTree) {
    if (n != null) {
      traverse(n.left)
      produce(n.elem)
      traverse(n.right)
    }}}
```

**Fig. 2.** A producer which generates all values in a tree in in-order.

– The `apply` method returns the value $e_i$ for the first pattern $p_i$ that matches its argument. If none of the patterns match, a `MatchError` exception is thrown.

The two methods are used in the implementation of `receive` as follows. First, messages in the mailbox are scanned in the order they appear. If `receive`'s argument `f` is defined for a message, that message is removed from the mailbox and `f` is applied to it. On the other hand, if `f.isDefinedAt(m)` is `false` for every message `m` in the mailbox, the receiving actor is suspended.

The `actor` and `self` constructs are realized as methods defined by the `Actor` *object*. Objects have exactly one instance at run-time, and their methods are similar to static methods in Java.

```
object Actor {
  def self: Actor ...
  def actor(body: => unit): Actor ...
  ...
}
```

Note that Scala has different name-spaces for types and terms. For instance, the name `Actor` is used both for the object above (a term) and the trait which is the result type of `self` and `actor` (a type). In the definition of the `actor` method, the argument `body` defines the behavior of the newly created actor. It is a closure returning the unit value. The leading `=>` in its type indicates that it is an unevaluated expression (a *thunk*).

There is also some other functionality in Scala's actor library which we have not covered. For instance, there is a method `receiveWithin` which can be used to specify a time span in which a message should be received allowing an actor to timeout while waiting for a message. Upon timeout the action associated with a special `TIMEOUT` pattern is fired. Timeouts can be used to suspend an actor, completely flush the mailbox, or to implement priority messages [3].

## 3 Example

In this section we present a larger example that will be revisited in later sections. We are going to write an abstraction of *producers* which provide a standard iterator interface to retrieve a sequence of produced values.

```
class Producer[T]
extends Iterator[T] {
  protected def produceValues
  private val producer = actor {
    produceValues
    coordinator ! None
  }
  def produce(x: T) {
    coordinator !? Some(x)
  }
  ...
}
```

```
private val coordinator = actor {
  val q = new Queue[Option[Any]]
  loop {
    receive {
      case HasNext if !q.isEmpty =>
        reply(q.front != None)
      case Next if !q.isEmpty =>
        q.dequeue match {
          case Some(x) => reply(x)
        }
      case x: Option[_] =>
        q += x; reply()
    }}}
```

**Fig. 3.** Implementation of the producer and coordinator actors.

Specific producers are defined by implementing an abstract `produceValues` method. Individual values are generated using the `produce` method. Both methods are inherited from class `Producer`. As an example, Figure 2 shows the definition of a producer which generates the values contained in a tree in in-order.

Producers are implemented in terms of two actors, a *producer* actor, and a *coordinator* actor. Figure 3 shows their implementation. The producer runs the `produceValues` method, thereby sending a sequence of values, wrapped in `Some` messages, to the coordinator. The sequence is terminated by a `None` message. `Some` and `None` are the two cases of Scala's standard `Option` class. The coordinator synchronizes requests from clients and values coming from the producer. The implementation in Figure 3 yields maximum parallelism through an internal queue that buffers produced values.

## 4 Unified actors

Concurrent processes such as actors can be implemented using one of two implementation strategies:

- Thread-based implementation: The behavior of a concurrent process is defined by implementing a thread-specific method. The execution state is maintained by an associated thread stack.
- Event-based implementation: The behavior is defined by a number of (non-nested) event handlers which are called from inside an event loop. The execution state of a concurrent process is maintained by an associated record or object.

Often, the two implementation strategies imply different programming models. Thread-based models are usually easier to use, but less efficient (context switches, memory requirements), whereas event-based models are usually more efficient, but very difficult to use in large designs [14].

Most event-based models introduce an *inversion of control*. Instead of calling blocking operations (e.g. for obtaining user input), a program merely registers its interest to be

resumed on certain *events* (e.g. signaling a pressed button). In the process, *event handlers* are installed in the execution environment. The program never calls these event handlers itself. Instead, the execution environment dispatches events to the installed handlers. Thus, control over the execution of program logic is "inverted". Because of inversion of control, switching from a thread-based to an event-based model normally requires a global re-write of the program.

In our library, both programming models are unified. As we are going to show, this unified model allows to trade-off efficiency for flexibility in a fine-grained way. We present our unified design in three steps. First, we review a thread-based implementation of actors. Then, we show an event-based implementation that avoids inversion of control. Finally, we discuss our unified implementation. We apply the results of our discussion to the case study of section 3.

*Thread-based actors.* Assuming a basic thread model is available in the host environment, actors can be implemented by simply mapping each actor onto its own thread. In this naïve implementation, the execution state of an actor is maintained by the stack of its corresponding thread. An actor is suspended/resumed by suspending/resuming its thread. On the JVM, thread-based actors can be implemented by subclassing the `Thread` class:

```
trait Actor extends Thread {
  private val mailbox = new Queue[Any]
  def !(msg: Any): unit = ...
  def receive[R](f: PartialFunction[Any, R]): R = ...
  ...
}
```

The principal communication operations are implemented as follows.

- *Send*. The message is enqueued in the actor's mailbox. If the receiver is currently suspended in a `receive` that could handle the sent message, the execution of its thread is resumed.
- *Receive*. Messages in the mailbox are scanned in the order they appear. If none of the messages in the mailbox can be processed, the receiver's thread is suspended. Otherwise, the first matching message is processed by applying the argument partial function `f` to it. The result of this application is returned.

*Event-based actors.* The central idea of event-based actors is as follows. An actor that waits in a receive statement is not represented by a blocked thread but by a closure that captures the rest of the actor's computation. The closure is executed once a message is sent to the actor that matches one of the message patterns specified in the receive. The execution of the closure is "piggy-backed" on the thread of the sender. When the receiving closure terminates, control is returned to the sender by throwing a special exception that unwinds the receiver's call stack.

A necessary condition for the scheme to work is that receivers never return normally to their enclosing actor. In other words, no code in an actor can depend on the termination or the result of a receive block. This is not a severe restriction in practice, as programs can always be organized in a way so that the "rest of the computation" of an actor is executed

from within a receive. Because of its slightly different semantics we call the event-based version of the receive operation `react`.

In the event-based implementation, instead of subclassing the `Thread` class, a private field `continuation` is added to the `Actor` trait that contains the rest of an actor's computation when it is suspended:

```
trait Actor {
  private var continuation: PartialFunction[Any, unit]
  private val mailbox = new Queue[Any]
  def !(msg: Any): unit = ...
  def react(f: PartialFunction[Any, unit]): Nothing = ...
  ...
}
```

At first sight it might seem strange to represent the rest of an actor's computation by a partial function. However, note that only when an actor suspends, an appropriate value is stored in the `continuation` field. An actor suspends when `react` fails to remove a matching message from the mailbox:

```
def react(f: PartialFunction[Any, unit]): Nothing = {
  mailbox.dequeueFirst(f.isDefinedAt) match {
    case Some(msg) => f(msg)
    case None      => continuation = f; suspended = true
  }
  throw new SuspendActorException
}
```

Note that `react` has return type `Nothing`. In Scala's type system a method has return type `Nothing` iff it never returns normally. In the case of `react`, an exception is thrown for all possible argument values. This means that the argument `f` of `react` is the last expression that is evaluated by the current actor. In other words, `f` always contains the "rest of the computation" of `self`[5]. We make use of this in the following way.

A partial function, such as `f`, is usually represented as a block with a list of patterns and associated actions. If a message can be removed from the mailbox (tested using `dequeueFirst`) the action associated with the matching pattern is executed by applying `f` to it. Otherwise, we remember `f` as the "continuation" of the receiving actor. Since `f` contains the complete execution state we can resume the execution at a later point when a matching message is sent to the actor. The instance variable `suspended` is used to tell whether the actor is suspended. If it is, the value stored in the `continuation` field is a valid execution state. Finally, by throwing a special exception, control is transferred to the point in the control flow where the current actor was started or resumed.

An actor is started by calling its `start()` method. A suspended actor is resumed if it is sent a message that it waits for. Consequently, the `SuspendActorException` is handled in the `start()` method and in the send method. Let's take look at the send method.

[5] Not only this, but also the complete execution state, in particular, all values on the stack accessible from within `f`. This is because Scala automatically constructs a *closure* object that lifts all potentially accessed stack locations into the heap.

```
def !(msg: Any): unit =
  if (suspended && continuation.isDefinedAt(msg))
    try { continuation(msg) }
    catch { case SuspendActorException => }
  else mailbox += msg
```

If the receiver is suspended, we check whether the message `msg` matches any of the patterns of the partial function stored in the `continuation` field of the receiver. In that case, the actor is resumed by applying `continuation` to `msg`. We also handle `SuspendActorException` since inside `continuation(msg)` there might be a nested `react` that suspends the actor. If the receiver is not suspended or the newly sent message does not enable it to continue, `msg` is appended to the mailbox.

Note that the presented event-based implementation forced us to modify the original programming model: In the thread-based model, the `receive` operation returns the result of applying an action to the received message. In the event-based model, the `react` operation never returns normally, i.e. it has to be passed explicitly the rest of the computation. However, we present below combinators that hide these explicit continuations. Also note that when executed on a single thread, an actor that calls a blocking operation prevents other actors from making progress. This is because actors only release the (single) thread when they suspend in a call to `react`.

The two actor models we discussed have complementary strengths and weaknesses: Event-based actors are very lightweight, but the usage of the `react` operation is restricted since it never returns. Thread-based actors, on the other hand, are more flexible: Actors may call blocking operations without affecting other actors. However, thread-based actors are not as scalable as event-based actors.

*Unifying actors.* A unified actor model is desirable for two reasons: First, advanced applications have requirements that are not met by one of the discussed models alone. For example, a web server might represent active user sessions as actors, and make heavy use of blocking I/O at the same time. Because of the sheer number of simultaneously active user sessions, actors have to be very lightweight. Because of blocking operations, pure event-based actors do not work very well. Second, actors should be composable. In particular, we want to compose event-based actors and thread-based actors in the same program.

In the following we present a programming model that unifies thread-based and event-based actors. At the same time, our implementation ensures that most actors are lightweight. Actors suspended in a `react` are represented as closures, rather than blocked threads.

Actors can be executed by a pool of worker threads as follows. During the execution of an actor, *tasks* are generated and submitted to a thread pool for execution. Tasks are implemented as instances of classes that have a single `run()` method:

```
class Task extends Runnable {
  def run() { ... }
}
```

A task is generated in the following three cases:

1. Spawning a new actor using `actor { body }` generates a task that executes `body`.

2. Calling `react` where a message can be immediately removed from the mailbox generates a task that processes the message.
3. Sending a message to an actor suspended in a `react` that enables it to continue generates a task that processes the message.

All tasks have to handle the `SuspendActorException` which is thrown whenever an actor suspends inside `react`. Handling this exception transfers control to the end of the task's `run()` method. The worker thread that executed the task is then free to execute the next pending task. Pending tasks are kept in a task queue inside a global scheduler object.[6]

The basic idea of our unified model is to use a thread pool to execute actors, and to *resize* the thread pool whenever it is necessary to support general thread operations. If actors use only operations of the event-based model, the size of the thread pool can be fixed. This is different if some of the actors use blocking operations such as `receive` or system I/O. In the case where every worker thread is occupied by a suspended actor and there are pending tasks, the thread pool has to grow.

In our library, system-induced deadlocks are avoided by increasing the size of the thread pool whenever necessary. It is necessary to add another worker thread whenever there is a pending task and all worker threads are blocked. In this case, the pending task(s) are the only computations that could possibly unblock any of the worker threads (e.g. by sending a message to a suspended actor.) To do this, a scheduler thread (which is separate from the worker threads of the thread pool) periodically checks if there is a task in the task queue and all worker threads are blocked. In that case, a new worker thread is added to the thread pool that processes any remaining tasks.

Unfortunately, on the JVM there is no safe way for library code to find out if a thread is blocked. Therefore, we implemented a conservative heuristic that approximates the predicate "all worker threads blocked". The approximation uses a time-stamp of the last "library activity". If the time-stamp is not recent enough (i.e. it has not changed since a multiple of scheduler runs), the predicate is assumed to hold, i.e. it is assumed that all worker threads are blocked. We maintain a global time-stamp that is updated on every call to send, receive etc.

*Example.* Revisiting our example of section 3, it is possible to economize one thread in the implementation of `Producer`. As shown in Figure 4, this can be achieved by simply changing the call to `receive` in the coordinator process into a call to `react`. By calling `react` in its outer loop, the coordinator actor allows the scheduler to detach it from its worker thread when waiting for a `Next` message. This is desirable since the time between client requests might be arbitrarily long. By detaching the coordinator, the scheduler can re-use the worker thread and avoid creating a new one.

*Composing actor behavior.* Without extending the unified actor model, defining an actor that executes several given functions in sequence is not possible in a modular way.

For example, consider the two methods below:

```
def awaitPing = react { case Ping => }
def sendPong = sender ! Pong
```

[6] Implementations based on work-stealing let worker threads have their own task queues, too. As a result, the global task queue is less of a bottle-neck.

```
private val coordinator = actor {
  val q = new Queue[Option[Any]]
  loop {
    react {
      // ... as in Figure 3
      }}}
```

**Fig. 4.** Implementation of the coordinator actor using `react`.

It is not possible to sequentially compose `awaitPing` and `sendPong` as follows:

```
actor { awaitPing; sendPong }
```

Since `awaitPing` ends in a call to `react` which never returns, `sendPong` would never get executed. One way to work around this restriction is to place the continuation into the body of `awaitPing`:

```
def awaitPing = react { case Ping => sendPong }
```

However, this violates modularity. Instead, our library provides an `andThen` combinator that allows actor behavior to be composed sequentially. Using `andThen`, the body of the above actor can be expressed as follows:

```
{ awaitPing } andThen { sendPong }
```

`andThen` is implemented by installing a hook function in the first actor. This hook is called whenever the actor terminates its execution. Instead of exiting, the code of the second body is executed. Saving and restoring the previous hook function permits chained applications of `andThen`.

The `Actor` object also provides a `loop` combinator. It is implemented in terms of `andThen`:

```
def loop(body: => unit) = body andThen loop(body)
```

Hence, the body of `loop` can end in an invocation of `react`.

## 5 Channels

In the programming model that we have described so far, actors are the only entities that can send and receive messages. Moreover, the receive operation ensures *locality*, i.e. only the owner of the mailbox can receive messages from it. Therefore, race conditions when accessing the mailbox are avoided by design. Types of messages are flexible: They are usually recovered through pattern matching. Ill-typed messages are ignored instead of raising compile-time or run-time errors. In this respect, our library implements a dynamically-typed embedded domain-specific language.

However, to take advantage of Scala's rich static type system, we need a way to permit strongly-typed communication among actors. For this, we use channels which are parameterized with the types of messages that can be sent to and received from it, respectively.

Moreover, the visibility of channels can be restricted according to Scala's scoping rules. That way, communication between sub-components of a system can be hidden. We distinguish input channels from output channels. Actors are then treated as a special case of output channels:

```
trait Actor extends OutputChannel[Any] { ... }
```

*Type-based selective communication.* The possibility for an actor to have multiple input channels raises the need to selectively communicate over these channels. Up until now, we have shown how to use `receive` to remove messages from an actor's mailbox. We have not yet shown how messages can be received from multiple input channels. Instead of adding a new construct, we generalize `receive` to work over multiple channels.

For example, a model of a component of an integrated circuit can receive values from both a control and a data channel using the following syntax:

```
receive {
  case DataCh ! data => ...
  case CtrlCh ! cmd => ...
}
```

Our library also provides an `orElse` combinator that allows reactions to be composed as alternatives. For example, using `orElse`, our electronic component can inherit behavior from a superclass:

```
receive {
  case DataCh ! data => ...
  case CtrlCh ! cmd => ...
} orElse super.reactions
```

An important use case of channels is strongly-typed communication. For this, channels are parameterized with the type of messages that may be sent to and/or received from them. In the following example, a channel of integers is created and afterwards a value is received from it and further processed:

```
object IC extends Channel[int]
receive { case IC ! x => val y = 2*x; ... }
```

In the above pattern `IC ! x`, the type of the pattern variable `x` is inferred to be `int` since it represents a value that is received from a `Channel[int]`. Therefore, any subsequent use of `x` in a position that is incompatible with `int` is rejected by the compiler.

## 6 Experimental results

*Message Passing.* In the first benchmark we measure throughput of blocking operations in a queue-based application. The application is structured as a ring of $n$ producers/consumers (in the following called *processes*) with a shared queue between each of them. Initially, $k$ of these queues contain tokens and the others are empty. Each process loops removing an item from the queue on its right and placing it in the queue on its left.

The following tests were run on a 1.80GHz Intel Pentium M processor with 1024 MB memory, running Sun's Java HotSpot™VM 1.5.0 under Linux 2.6.15. We set the JVM's maximum heap size to 512 MB to provide for sufficient physical memory to avoid any disk activity. In each case we took the median of 5 runs.

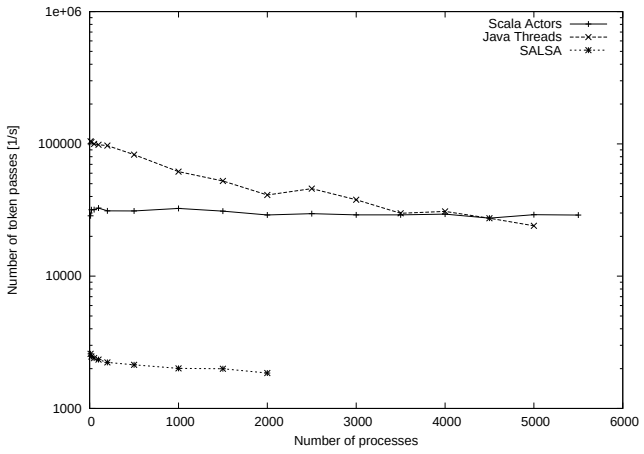


**Fig. 5.** Throughput (number of token passes per second) for a fixed number of 10 tokens.

The execution times of three equivalent implementations written using (1) our actor library, (2) threads and locks, and (3) SALSA [19], a state-of-the-art Java-based actor language, respectively, are compared. Figure 5 shows the number of token passes per second (throughput) depending on the ring size. Since the differences in throughput are very large, we chose a logarithmic scale.

For less than 3000 processes, pure Java threads are between 3.7 (10 processes) and 1.3 (3000 processes) times faster than Scala actors. We expect a certain overhead since in our library every message implicitly carries a reference to the sender. The senders are maintained on an actor-local stack. Moreover, whenever an actor is resumed, `self` has to be updated to point to the currently executing actor. This is done by setting (and restoring) a thread-local reference. Finally, resuming an actor involves creating and submitting a task to the worker thread pool. Interestingly, throughput of Scala actors remains basically constant (at about 30,000 tokens per second), regardless of the number of processes. In contrast, throughput of pure Java threads constantly decreases as the number of processes increases. This behavior is likely to stem from the increasing overhead of context switches. The VM is unable to create a ring with 5500 threads as it runs out of heap memory. In contrast, using Scala actors the ring can be operated with as many as 600,000 processes (since every queue is also an actor this amounts to 1,200,000 simultaneously active actors.) Throughput of Scala actors is on average over 13 times higher than that of SALSA.

*Multi-core scalability.* Scala actors are executed on multiple threads to utilize modern multi-core processors and shared-memory multi-processors. Therefore, we are interested in the speed-up that is gained by adding processor cores to a system.

The following tests were run on a multi-processor with 4 dual-core Opteron 64-Bit processors (2.8 GHz each) with 16 GB memory, running Sun's Java HotSpot™64-Bit Server VM 1.5.0 under Linux 2.6.16. In each case we took the median of 5 runs.

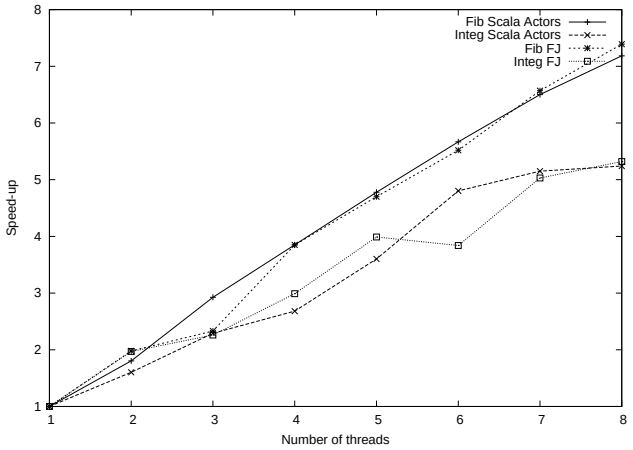


**Fig. 6.** Speed-up for Fibonacci and Integration micro benchmarks.

First, we run benchmark programs that theoretically offer an ideal speed-up, and compare with experimental data. We use direct translations of the Fibonacci (Fib) and Gaussian integration (Integ) programs distributed with Doug Lea's high-performance fork/join framework for Java (FJ) [13]. Figure 6 shows the speed-up when the size of the worker thread pool is increased. In the case of Fib, the speed-up for 8 threads amounts to 7.18. With the same configuration, FJ achieves a speed-up of 7.39. In the case of Integ, the speed-up for 8 threads amounts to 5.24, whereas with FJ a speed-up of 5.32 is achieved.

However, the reported speed-ups are achieved by explicitly tuning the respective granularity threshold for parallel processing. Measurements with a more realistic work-load show that work-stealing strategies for task processing have significant advantages over simple implementations that use a global task queue.

Figure 7 shows the performance running the heat diffusion simulation benchmark that is distributed with Cilk [5] (it uses a Jacobi-type iteration to solve a finite-difference approximation of parabolic partial differential equations.) To measure the effect of local task queues with work-stealing compared to a global task queue we ran Scala actors in two versions. The first version (global) uses a global task queue. The second version (local) uses FJ for task execution, thereby utilizing local task queues for worker threads and work-stealing. For 8 threads, FJ achieves a speed-up of 1.76. Actors using a global task queue achieve the best speed-up for 5 threads (1.25). Contention causes speed-up to decrease down to 1.10 for 8 threads. In contrast, using local task queues and work-stealing, exe-

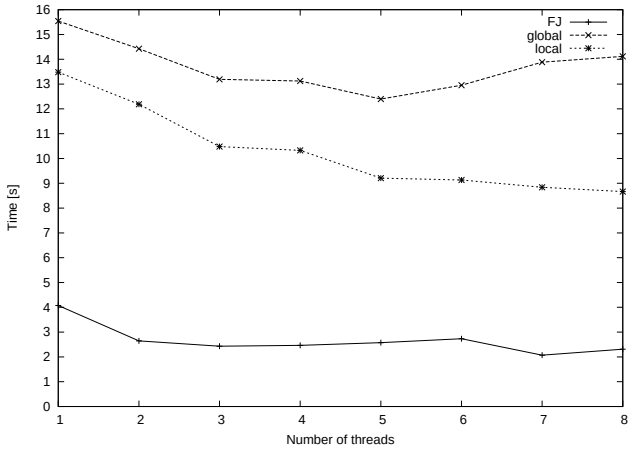


**Fig. 7.** Effect of work-stealing on heat diffusion simulation.

cution time constantly decreases until at 8 threads a speed-up of 1.56 is reached. At that point, absolute performance is over 60% higher compared to the version using a global task queue.

The experimental results given above are preliminary, in that small changes still can have surprisingly large effects. The results show, however, that even by using a simple and robust scheduler implementation, Scala actors are competitive in absolute performance and scale with adding processor cores. Scala actors support a number of simultaneously active actors which is two orders of magnitude higher compared to SALSA. Measured throughput in a queue-based application is over 13 times higher compared to SALSA. A thread-based implementation of one of our benchmarks performs surprisingly well. However, performance constantly decreases with an increasing number of threads. Also, the maximum number of threads is limited due to their memory consumption.

## 7 Conclusion

In this paper we have shown how thread-based and event-based models of concurrency can be unified under a single abstraction of actors. Actors are an attractive structuring method for concurrent systems. Their programming model permits high-level communication through messages and pattern matching. Although races are avoided by design when accessing an actor's mailbox, safety rests on a policy that actors communicate only through mailboxes or channels, not through other shared memory. In Erlang, this is enforced by the language through isolated processes. In the Scala API, we leave this to the programmer. Although this approach is more flexible, it also entails a higher risk. One of our future research directions is therefore to investigate type systems to control memory locality.

## References


1. Gul A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. Series in Artificial Intelligence. The MIT Press, Cambridge, Massachusetts, 1986.
2. Joe Armstrong. Erlang — a survey of the language and its industrial applications. In *INAP*, pages 16–18, Hino, Tokyo, Japan, October 1996.
3. Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang, Second Edition*. Prentice-Hall, 1996.
4. A. Black, M. Carlsson, M. Jones, R. Kieburtz, and J. Nordlander. Timber: A programming language for real-time embedded systems, 2002.
5. Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *PPOPP*, pages 207–216, 1995.
6. Jean-Pierre Briot. Actalk: A testbed for classifying and designing actor languages in the Smalltalk-80 environment. In *ECOOP*, pages 109–129, 1989.
7. Burak Emir, Martin Odersky, and John Williams. Matching objects with patterns. LAMP-Report 2006-006, EPFL, 2006.
8. Benoit Garbinato, Rachid Guerraoui, Jarle Hulaas, Maxime Monod, and Jesper Spring. Frugal Mobile Objects. Technical report, EPFL, 2005.
9. Guillaume Germain, Marc Feeley, and Stefan Monnier. Concurrency oriented programming in Termite Scheme. In *Proc. of the Workshop on Scheme and Functional Programming*, September 2006.
10. Philipp Haller and Martin Odersky. Event-based programming without inversion of control. In *Proc. Joint Modular Languages Conference*, LNCS. Springer, September 2006.
11. Carl E. Hewitt. Viewing controll structures as patterns of passing messages. *Journal of Artificial Intelligence*, 8(3):323–364, 1977.
12. H. G. Baker Jr. and Carl E. Hewitt. ACTORS and continuous functionals. Report A. I. MEMO 436, Massachusetts Institute of Technology, A.I. Lab., Cambridge, Massachusetts, 1977.
13. Doug Lea. A java fork/join framework. In *Java Grande*, pages 36–43, 2000.
14. P. Levis and D. Culler. Mate: A tiny virtual machine for sensor networks. In *International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, USA*, Oct. 2002.
15. Peng Li and Steve Zdancewic. A language-based approach to unifying events and threads. Technical report, CIS Department, University of Pennsylvania, April 2006.
16. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.
17. J. H. Nyström, Philip W. Trinder, and David J. King. Evaluating distributed functional languages for telecommunications software. In Bjarne Däcker and Thomas Arts, editors, *Proceedings of the 2003 ACM SIGPLAN Workshop on Erlang, Uppsala, Sweden, August 29, 2003*, pages 1–7. ACM, 2003.
18. D. A. Thomas, W. R. Lalonde, J. Duimovich, M. Wilson, J. McAffer, and B. Berry. Actra: A multitasking/multiprocessing Smalltalk. *Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming, ACM SIGPLAN Notices*, 24(4):87–90, April 1989.
19. Carlos Varela and Gul Agha. Programming dynamically reconfigurable open systems with SALSA. *SIGPLAN Not.*, 36(12):20–34, 2001.