Actors That Unify Threads and Events

Philipp Haller and Martin Odersky

Programming Methods Lab (LAMP)
École Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland
firstname.lastname@epfl.ch

Abstract. There is an impedance mismatch between message-passing concurrency and virtual machines, such as the JVM. VMs usually map their threads to heavyweight OS processes. Without a lightweight process abstraction, users are often forced to write parts of concurrent applications in an event-driven style which obscures control flow, and increases the burden on the programmer.

In this paper we show how thread-based and event-based programming can be unified under a single actor abstraction. Using advanced abstraction mechanisms of the Scala programming language, we implemented our approach on unmodified JVMs. Our programming model integrates well with the threading model of the underlying VM.

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 [1,19]. 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 [2,27].

On mainstream platforms such as the JVM [26], an equally attractive implementation was as yet missing. Their standard concurrency constructs, shared-memory threads with locks, suffer from high memory consumption and context-switching overhead. Therefore, the interleaving of independent computations is often modeled 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 [32,8].

In previous work [15], 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 continuation closure, which is usually much cheaper than suspending a thread.

A.L. Murphy and J. Vitek (Eds.): COORDINATION 2007, LNCS 4467, pp. 171–190, 2007.

In this paper we present a unification of thread-based and event-based actors. 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 larger numbers of actors. We also present a set of combinators that allows a flexible composition of these actors.

This paper improves on our previous work in several respects. First, the decision whether an actor should be thread-less or not is deferred until run-time. An actor may discard its corresponding thread stack several times. Second, our previous work did not address aspects of composition. Neither a solution for sequential composition of event-based actors, nor an approach for the composition of thread-based and event-based actors in the same program was provided.

The presented 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 [11].

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.

- Since accessing an actor's mailbox is race-free by design, message-based concurrency is potentially more secure than shared-memory concurrency with locks. We believe that message-passing with pattern matching is also more convenient in many cases.
- 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 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. Lauer and Needham [20] note in their seminal work that threads and events are dual to each other. They suggest that any choice of either one of them should therefore be based on the underlying platform. Almost two decades later, Ousterhout [28] argues that threads are a bad idea not only because they often perform poorly, but also because they are hard to use. More recently, von Behren and others [32] point out that even though event-driven programs often outperform equivalent threaded programs, they are too difficult to write. The two main reasons are: first, the interactive logic of a program is fragmented across multiple event handlers (or classes, as in the state design pattern [12]). Second, control flow among handlers is expressed implicitly through manipulation of shared state [6]. In the Capriccio system [33], static

[1] Available as part of the Scala distribution at http://www.scala-lang.org/

analysis and compiler techniques are employed to transform a threaded program into a cooperatively-scheduled event-driven program with the same behavior.

There are several other approaches that avoid the above control inversion. However, they have either limited scalability, or they lack support of blocking operations. Termite Scheme [14] 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. Responders [6] provide an event-loop abstraction as a Java language extension. Since their implementation spends a VM thread per event-loop, scalability is limited on standard JVMs. SALSA [31] is a Java-based actor language that has a similar limitation (each actor runs on its own thread). In addition, message passing performance suffers from the overhead of reflective method calls. 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 are not allowed to call operations that might block indefinitely. Frugal objects [13] (FROBs) are distributed reactive objects that communicate through typed events. FROBs are basically actors with an event-based computation model. Similar to reactive objects in Timber, FROBs may not call blocking operations.

Li and Zdancewic [25] 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, blocking system calls have to be wrapped in non-blocking operations. Moreover, adding new event sources requires invasive changes to the thread library (registering event handlers, adding event loops etc.).

The actor model has also been integrated into various Smalltalk systems. Actalk [5] is an actor library for Smalltalk-80 that does not support multiple processor cores. Actra [30] extends the Smalltalk/V VM to provide lightweight processes. In contrast, we implement lightweight actors on unmodified VMs.

In section 7 we show that our actor implementation scales to a number of actors that is two orders of magnitude larger than what purely thread-based systems such as SALSA support. Moreover, results suggest that our model scales with the number of processor cores in a system. Our unified actor model provides seamless support for blocking operations. Therefore, existing thread-blocking APIs do not have to be wrapped in non-blocking operations. Unlike approaches such as Actra our implementation provides lightweight actor abstractions on unmodified Java VMs.

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 [2,27]. 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.

The idea to implement lightweight concurrent processes using continuations has been explored many times [34,18,7]. However, none of the existing techniques are applicable to VMs such as the JVM because (1) access to the run-time stack is too restricted, and (2) heap-based stacks break interoperability with existing code. However, the approach used to implement thread management in the Mach 3.0 kernel [9] is at least conceptually similar to ours. When a thread blocks in the kernel, either it preserves its register state and stack and resumes by restoring this state, or it preserves a pointer to a continuation function that is called when the thread is resumed. Instead of function pointers we use closures that automatically lift referenced stack variables on the heap avoiding explicit state management in many cases.

There is a rich body of work on building fast web servers, using events or a combination of events and threads (for example SEDA [35]). However, a comprehensive discussion of this work is beyond the scope of this paper.

Our integration of a high-level actor-based programming model, providing strong invariants and lightweight concurrency, with existing threading models of mainstream VM platforms is unique to the best of our knowledge. We believe that our approach offers a qualitative improvement in the development of concurrent software for multi-core systems.

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. By means of a case study (section 6) we show how our unified programming model can be applied to programming advanced web applications. Experimental results are presented in section 7. Section 8 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.

```
// base version
val orderMngr = 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 {
  orderMngr ! Order(self, myItem)
  receive {
    case Ack(o) => ...
  }
}
```

```
// version with reply and !?
val orderMngr = 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 {
  orderMngr !? Order(myItem) match {
    case Ack(o) => ...
  }
}
```

Fig. 1. Example: orders and cancellations

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 `orderMngr` 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 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 `orderMngr` 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

[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.

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)`	replies 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

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

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.
- 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 discuss the benefits of our actor model using a larger example. In the process we dissect three different implementations: an event-driven version, a thread-based version, and a version using Scala actors.

[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)
    }
  }
}
```

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

Fig. 2. Producers that generate all values in a tree in in-order

We are going to write an abstraction of *producers* that provide a standard iterator interface to retrieve a sequence of produced values. Producers are defined by implementing an abstract `produceValues` method that calls a `produce` method to generate individual values. Both methods are inherited from a `Producer` class. As an example, the left-hand side of figure 2 shows the definition of a producer that generates the values contained in a tree in in-order.

In a purely event-driven style, there are basically two approaches to specifying traversals, namely writing traversals in continuation-passing style (CPS), and programming explicit FSMs. The left-hand side of figure 3 shows an event-driven implementation of producers where the traversal is specified using CPS. The idea is that the `produce` method is passed a continuation closure that is called whenever the next value should be produced. For example, the in-order tree producer mentioned earlier is shown on the right-hand side of figure 2. Produced values are exchanged using an instance variable of the producer.

The right-hand side of figure 3 shows a threaded version of the producer abstraction. In the threaded version the state of the iteration is maintained implicitly on the stack of a thread that runs the `produceValues` method. Produced values are put into a queue that is used to communicate with the iterator. Requesting the next value on an empty queue blocks the thread that runs the iterator. Compared to the event-driven version, the threaded version simplifies the specification of iteration strategies. To define a specific iterator, it suffices to provide an implementation for the `produceValues` method that traverses the tree in the desired order.

Figure 4 shows an implementation of producers in terms of two actors, a *producer* actor, and a *coordinator* actor. 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. The coordinator synchronizes requests from clients and values coming from the producer. As in the threaded version, the `produce` method does not take a continuation argument.

The actor-based version improves over the event-driven version by not requiring to specify the traversal in CPS. Moreover, it supports concurrent iterators, since communication using mailboxes is race-free. For the same reason, there is no need for an explicit blocking queue as in the threaded version, since this functionality is subsumed by the

```
abstract class CPSProducer[T] {
  var next: Option[T] = None
  var savedCont: () => unit =
    () => produceValues
  def produce(x: T,
              cont: => unit) {
    next = Some(x)
    savedCont = () => {
      next = None; cont
    }
  }
  ...
}
```

```
abstract class ThreadedProducer[T] {
  val produced = new Queue[Option[T]]
  def next: Option[T] = synchronized {
    while (produced.isEmpty) {wait()}
    produced.dequeue
  }
  new Thread(new Runnable() {
    def run() {
      produceValues
      produced += None
    }
  }).start()
  def produce(x: T) = synchronized {
    produced += Some(x)
    if (produced.length == 1) notify()
  }
  ...
}
```

Fig. 3. Event-driven and threaded producers

```
abstract class ActorProducer[T] {
  def produce(x: T) {
    coordinator ! Some(x)
  }
  private val producer = actor {
    produceValues
    coordinator ! None
  }
  ...
}
```

```
private val coordinator = actor {
  loop { receive {
    case 'next => receive {
      case x: Option[_] => reply(x)
    }
  }}
}
```

Fig. 4. Implementation of the producer and coordinator actors

actors' mailboxes. We believe that the use of blocking queues for communication is so common that it is worth making them generally available in the form of mailboxes for concurrent actors.

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 [24,32,8].

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 programmers 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

[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.

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.

```
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 5, this can be achieved by

[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 {
  loop { react {
    // ... as in Figure 4
  }}}
```

Fig. 5. Implementation of the coordinator actor using `react`

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
```

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] { ... }
```

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
```

6 Case Study

In this section we show how our unified actor model addresses some of the challenges of programming web applications. In the process, we review event- and thread-based solutions to common problems, such as blocking I/O operations. Our goal is then to discuss potential benefits of our unified approach. Advanced web applications typical pose at least the following challenges to the programmer:

– *Blocking operations.* There is almost always some functionality that is implemented using blocking operations. Possible reasons are lack of suitable libraries (e.g. for

non-blocking socket I/O), or simply the fact that the application is built on top of a large code basis that uses potentially blocking operations in some places. Typically, rewriting infrastructure code to use non-blocking operations is not an option.
- *Non-blocking operations.* On platforms such as the JVM, web application servers often provide some parts (if not all) of their functionality in the form of non-blocking APIs for efficiency. Examples are request handling, and asynchronous HTTP requests.
- *Race-free data structures.* Advanced web applications typically maintain user profiles for personalization. These profiles can be quite complex (some electronic shopping sites apparently track every item that a user visits). Moreover, a single user may be logged in on multiple machines, and issue many requests in parallel. This is common on web sites, such as those of electronic publishers, where single users represent whole organizations. It is therefore mandatory to ensure race-free accesses to a user's profile.

Thread-based approaches. VMs overlap computation and I/O by transparently switching among threads. Therefore, even if loading a user profile from disk blocks, only the current request is delayed. Non-blocking operations can be converted to blocking operations to support a threaded style of programming: after firing off a non-blocking operation, the current thread blocks until it is notified by a completion event. However, threads do not come for free. On most mainstream VMs, the overhead of a large number of threads–including context switching and lock contention–can lead to serious performance degradation [35,10]. Overuse of threads can be avoided by using bounded thread pools [21]. Shared resources such as user profiles have to be protected using synchronization operations. This is known to be particularly hard using shared-memory locks [23]. We also note that alternatives such as transactional memory [16,17], even though a clear improvement over locks, do not provide seamless support for I/O operations as of yet. Instead, most approaches require the use of compensation actions to revert the effects of I/O operations, which further complicate the code.

Event-based approaches. In an event-based model, the web application server generates events (network and I/O readiness, completion notifications etc.) that are processed by event handlers. A small number of threads (typically one per CPU) loop continuously removing events from a queue and dispatching them to registered handlers. Event handlers are required not to block since otherwise the event-dispatch loop could be blocked, which would freeze the whole application. Therefore, all operations that could potentially block, such as the user profile look-up, have to be transformed into non-blocking versions. Usually, this means executing them on a newly spawned thread, or on a thread pool, and installing an event handler that gets called when the operation completed [29]. Usually, this style of programming entails an inversion of control that causes the code to loose its structure and maintainability [6,8].

Scala actors. In our unified model, event-driven code can easily be wrapped to provide a more convenient interface that avoids inversion of control without spending an extra thread [15]. The basic idea is to decouple the thread that signals an event from the thread that handles it by sending a message that is buffered in an actor's mailbox. Messages

sent to the same actor are processed atomically with respect to each other. Moreover, the programmer may explicitly specify in which order messages should be removed from its mailbox. Like threads, actors support blocking operations using implicit thread pooling as discussed in section 4. Compared to a purely event-based approach, users are relieved from writing their own ad-hoc thread pooling code. Since the internal thread pool can be global to the web application server, the thread pool controller can leverage more information for its decisions [35]. Finally, accesses to an actor's mailbox are race-free. Therefore, resources such as user profiles can be protected by modeling them as (thread-less) actors.

7 Preliminary Results

We realize that performance across threads and events may involve a number of non-trivial trade-offs. A thorough experimental evaluation of our framework is therefore beyond the scope of this paper, and will have to be addressed in future work. However, the following basic experiments show that performance of our framework is at least comparable to those of both thread-based and event-based systems.

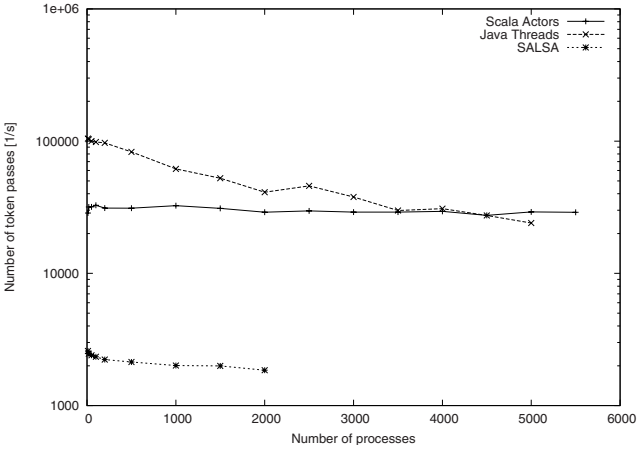

Fig. 6. Throughput (number of token passes per second) for a fixed number of 10 tokens

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. The execution

times of three equivalent implementations written using (1) our actor library, (2) pure Java threads, and (3) SALSA (version 1.0.2), a state-of-the-art Java-based actor language [31], respectively, are compared. Figure 6 shows the number of token passes per second (throughput) depending on the ring size. Note that throughput is given on 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. 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. 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.

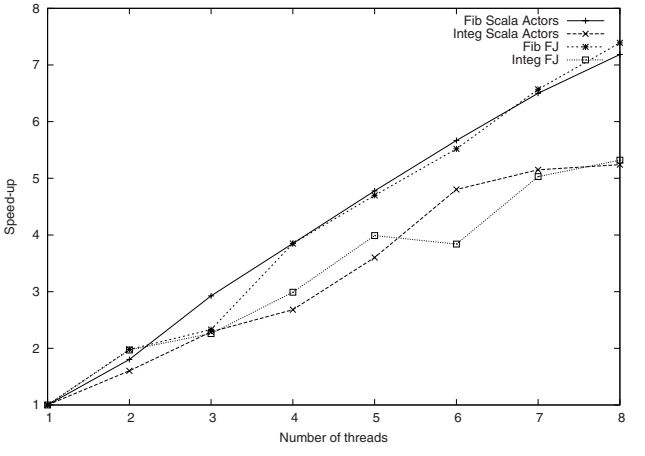

Fig. 7. Speed-up for Fibonacci and Integration micro benchmarks

Multi-core scalability. In the second experiment, 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. We ran direct translations of the Fibonacci (Fib) and Gaussian integration (Integ) programs distributed with Doug Lea's high-performance fork/join framework for Java (FJ) [22]. The speed-ups as shown in figure 7 are linear as expected since the programs run almost entirely in parallel.

8 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. While abstracting commonalities, our approach allows programmers to trade-off efficiency for flexibility in a fine-grained way. Scala's actor library provides a common programming model that permits

high-level communication through messages and pattern matching. We believe that our work closes an important gap between message-passing concurrency and popular VM platforms.

Acknowledgments. We would like to thank our shepherd, Doug Lea, and the anonymous reviewers for their helpful comments.

References

1. Agha, G.A.: ACTORS: A Model of Concurrent Computation in Distributed Systems. MIT Press, Cambridge, Massachusetts (1986)
2. Armstrong, J.: Erlang — a survey of the language and its industrial applications. In: Proc. INAP, pp. 16–18 (October 1996)
3. Armstrong, J., Virding, R., Wikström, C., Williams, M.: Concurrent Programming in Erlang, 2nd edn. Prentice-Hall, Englewood Cliffs (1996)
4. Black, A., Carlsson, M., Jones, M., Kieburtz, R., Nordlander, J.: Timber: A programming language for real-time embedded systems (2002)
5. Briot, J.-P.: Actalk: A testbed for classifying and designing actor languages in the Smalltalk-80 environment. In: Proc. ECOOP, pp. 109–129 (1989)
6. Chin, B., Millstein, T.D.: Responders: Language support for interactive applications. In: Proc. ECOOP, pp. 255–278 (July 2006)
7. Cooper, E., Morrisett, G.: Adding Threads to Standard ML. Report CMU-CS-90-186, Carnegie-Mellon University (December 1990)
8. Cunningham, R., Kohler, E.: Making events less slippery with eel. In: Proc. HotOS. USENIX (June 2005)
9. Draves, R.P., Bershad, B.N., Rashid, R.F., Dean, R.W.: Using continuations to implement thread management and communication in operating systems. Operating Systems Review 25(5), 122–136 (October 1991)
10. Dunkels, A., Grönvall, B., Voigt, T.: Contiki - A lightweight and flexible operating system for tiny networked sensors. In: LCN, pp. 455–462 (2004)
11. Emir, B., Odersky, M., Williams, J.: Matching Objects with Patterns. LAMP-Report 2006-006, EPFL, Lausanne, Switzerland (December 2006)
12. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley, London (1995)
13. Garbinato, B., Guerraoui, R., Hulaas, J., Monod, M., Spring, J.: Frugal Mobile Objects. Technical report, EPFL (2005)
14. Germain, G., Feeley, M., Monnier, S.: Concurrency oriented programming in Termite Scheme. In: Proc. Workshop on Scheme and Functional Programming (September 2006)
15. Haller, P., Odersky, M.: Event-based Programming without Inversion of Control. In: Lightfoot, D.E., Szyperski, C.A. (eds.) JMLC 2006. LNCS, vol. 4228, pp. 4–22. Springer, Heidelberg (September 2006)
16. Harris, T., Fraser, K.: Language support for lightweight transactions. In: OOPSLA, pp. 388–402 (2003)
17. Harris, T., Marlow, S., Jones, S.L.P., Herlihy, M.: Composable memory transactions. In: Proc. PPOPP, pp. 48–60. ACM, New York (June 2005)
18. Haynes, C.T., Friedman, D.P.: Engines build process abstractions. In: Symp. Lisp and Functional Programming, pp. 18–24. ACM, New York (August 1984)
19. Hewitt, C.E.: Viewing controll structures as patterns of passing messages. Journal of Artificial Intelligence 8(3), 323–364 (1977)

20. Lauer, H.C., Needham, R.M.: On the duality of operating system structures. Operating Systems Review 13(2), 3–19 (1979)
21. Lea, D.: Concurrent Programming in Java. Addison-Wesley, London (1996)
22. Lea, D.: A java fork/join framework. In: Java Grande, pp. 36–43 (2000)
23. Lee, E.A.: The problem with threads. Technical Report UCB/EECS-2006-1, University of California, Berkeley (January 2006)
24. Levis, P., Culler, D.: Mate: A tiny virtual machine for sensor networks. In: Proc. ASPLOS (October 2002)
25. Li, P., Zdancewic, S.: A language-based approach to unifying events and threads. Technical report, University of Pennsylvania (April 2006)
26. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. Addison-Wesley, London (1996)
27. Nyström, J.H., Trinder, P.W., King, D.J.: Evaluating distributed functional languages for telecommunications software. In: Proc. Workshop on Erlang, pp. 1–7. ACM, New York (August 2003)
28. Ousterhout, J.: Why threads are A bad idea (for most purposes). Invited talk at USENIX (January 1996)
29. Pai, V.S., Druschel, P., Zwaenepoel, W.: Flash: An efficient and portable Web server. In: Proc. USENIX, pp. 199–212 (June 1999)
30. Thomas, D.A., Lalonde, W.R., Duimovich, J., Wilson, M., McAffer, J., Berry, B.: Actra: A multitasking/multiprocessing Smalltalk. ACM SIGPLAN Notices 24(4), 87–90 (April 1989)
31. Varela, C., Agha, G.: Programming dynamically reconfigurable open systems with SALSA. ACM SIGPLAN Notices 36(12), 20–34 (2001)
32. von Behren, J.R., Condit, J., Brewer, E.A.: Why events are a bad idea (for high-concurrency servers). In: Proc. Hot OS,USENIX, pp. 19–24 (May 2003)
33. von Behren, J.R., Condit, J., Zhou, F., Necula, G.C., Brewer, E.A.: Capriccio: scalable threads for internet services. In: Proc. SOSP, pp. 268–281 (2003)
34. Wand, M.: Continuation-based multiprocessing. In: LISP Conference, pp. 19–28 (1980)
35. Welsh, M., Culler, D.E., Brewer, E.A.: SEDA: An architecture for well-conditioned, scalable internet services. In: Proc. SOSP, pp. 230–243 (2001)