Implementing Joins using Extensible Pattern Matching*

Philipp Haller[1], Tom Van Cutsem[2]**

[1] LAMP-REPORT-2007-004
EPFL
firstname.lastname@epfl.ch
[2] Vrije Universiteit Brussel, Belgium

Abstract. Join patterns are an attractive declarative way to synchronize both threads and asynchronous distributed computations. We explore joins in the context of extensible pattern matching that recently appeared in languages such as F# and Scala. Our implementation supports join patterns with multiple synchronous events, and guards. Furthermore, we integrated joins into an existing actor-based concurrency framework. It enables join patterns to be used in the context of more advanced synchronization modes, such as future-type message sending and token-passing continuations.

Keywords: Concurrent Programming, Join Patterns, Chords, Actors

1 Introduction

Recently, the pattern matching facilities of languages such as Scala and F# have been generalized to allow representation independence for objects used in pattern matching [6,20]. Extensible patterns open up new possibilities for implementing abstractions in libraries which were previously only accessible as language features. More specifically, we claim that extensible pattern matching eases the construction of declarative approaches to synchronization in libraries rather than languages. To support this claim, we show how a concrete declarative synchronization construct, join patterns, can be implemented in Scala, a language with extensible pattern matching.

Join patterns [8,9] offer a declarative way of synchronizing both threads and asynchronous distributed computations that is simple and powerful at the same time. They form part of languages such as JoCaml [7] and Funnel [14]. Join patterns have also been implemented as extensions to existing languages [3,23]. Recently, Russo [17] and Singh [18] have shown that advanced programming language features, such as generics or software transactional memory, make it feasible to provide join patterns as libraries rather than language extensions.

We motivate that our implementation based on extensible pattern matching is an interesting third way to provide join patterns in a library since it has a number of desirable properties. More concretely, we make the following contributions:

* to appear at 10th International Conference on Coordination Models and Languages (COORDINATION 2008).
** supported by a Ph.D. fellowship of the Research Foundation Flanders (FWO).

– We present a novel implementation technique for joins based on extensible pattern matching. We show that it allows programmers to avoid certain kinds of boilerplate code that are inevitable when using existing approaches.
– We discuss a concrete implementation of our approach in Scala. A complete implementation that supports join patterns with multiple synchronous events and a restricted form of guards is available on the web.[3]
– We integrate our library into an existing actor-based concurrency framework. This enables expressive join patterns to be used in the context of more advanced synchronization modes, such as future-type message sending and token-passing continuations.

The rest of this paper is structured as follows. In the following section we briefly highlight join patterns as a declarative synchronization abstraction, how they have been integrated in other languages before, and how combining them with pattern matching can improve this integration. Section 3 shows how to synchronize both threads and actors using our new Scala Joins framework. In section 4 we discuss a concrete implementation of expressive join patterns in Scala. Section 5 discusses related work, and section 6 concludes.

2 Motivation

Background: Join Patterns. A join pattern consists of a body guarded by a linear set of events. The body is executed only when *all* of the events in the set have been signaled to an object. Threads may signal synchronous or asynchronous events to objects. By signaling a synchronous event to an object, threads may implicitly suspend. The simplest illustrative example of a join pattern is that of an unbounded FIFO buffer. In Cω [3], it is expressed as follows:

```
public class Buffer {
  public async Put(int x);
  public int Get() & Put(int x) { return x; }
}
```

A detailed explanation of join patterns is outside the scope of this paper. For the purposes of this paper, it suffices to understand the operational effect of a join pattern. Threads may put values into a buffer `b` by invoking `b.Put(v)`. They may also read values from the buffer by invoking `b.Get()`. The join pattern `Get() & Put(int x)` (called a *chord* in Cω) specifies that a call to `Get` may only proceed if a `Put` event has previously been signaled. Hence, if there are no pending `Put` events, a thread invoking `Get` is automatically suspended until such an event is signaled.

The advantage of join patterns is that they allow a *declarative* specification of the synchronization between different threads. Often, the join patterns correspond closely to a finite state machine that specifies the valid states of an object [3]. In the following, we explain the benefits of our new implementation by means of an example.

[3] See `http://lamp.epfl.ch/~phaller/joins/`.

Example. Consider the traditional problem of synchronizing multiple concurrent readers with one or more writers who need exclusive access to a resource. In Cω, join patterns are supported as a language extension through a dedicated compiler. With the introduction of generics in C# 2.0, Russo has made join patterns available in a C# library called Joins [17]. In that library, a multiple reader/one writer lock can be implemented as follows:

```
public class ReaderWriter {
  public Synchronous.Channel Exclusive, ReleaseExclusive;
  public Synchronous.Channel Shared, ReleaseShared;
  private Asynchronous.Channel Idle;
  private Asynchronous.Channel<int> Sharing;
  public ReaderWriter() {
    Join j = Join.Create(); ... // Boilerplate omitted
    j.When(Exclusive).And(Idle).Do(delegate {});
    j.When(ReleaseExclusive).Do(delegate{ Idle(); });
    j.When(Shared).And(Idle).Do(delegate{ Sharing(1); });
    j.When(Shared).And(Sharing).Do(delegate(int n) {
      Sharing(n+1); });
    j.When(ReleaseShared).And(Sharing).Do(delegate(int n) {
      if (n==1) Idle(); else Sharing(n-1); });
    Idle(); } }
```

In C# Joins, join patterns consist of linear combinations of channels and a delegate (a function object) which encapsulates the join body. Join patterns are triggered by invoking channels which are special delegates.

In the example, channels are declared as fields of the `ReaderWriter` class. Channel types are either synchronous or asynchronous. Asynchronous channels correspond to asynchronous methods in Cω (e.g. `Put` in the previous example). Channels may take arguments which are specified using type parameters. For example, the `Sharing` channel is asynchronous and takes a single `int` argument. Channels are often used to model (parts of) the internal state of an object. For example, the `Idle` and `Sharing` channels keep track of concurrent readers (if any), and are therefore declared as `private`. To declare a set of join patterns, one first has to create an instance of the `Join` class. Individual join patterns are then created by chaining a number of method calls invoked on that `Join` instance. For example, the first join pattern is created by combining the `Exclusive` and `Idle` channels with an empty delegate; this means that invoking the synchronous `Exclusive` channel (a request to acquire the lock in exclusive mode) will not block the caller if the `Idle` channel has been invoked (the lock has not been acquired).

Even though the verbosity of programs written using C# Joins is slightly higher compared to Cω, basically all the advantages of join patterns are preserved. However, this code still has a number of drawbacks: first, the encoding of the internal state is *redun-*

dant. Logically, a lock in idle state can be represented either by the non-empty `Idle` channel or the `Sharing` channel invoked with `0`.[4]

Note that it is impossible in C# (and in Cω) to use only `Sharing`. Consider the first join pattern. Implementing it using `Sharing` instead of `Idle` requires a delegate that takes an integer argument (the number of concurrent readers):

```
j.When(Exclusive).And(Sharing).Do(delegate(int n) {...}
```

Inside the body we have to test whether $n > 0$ in which case the thread invoking `Exclusive` has to block. Blocking without reverting to lower-level mechanisms such as locks is only possible by invoking a synchronous channel; however, that channel has to be different from `Exclusive` (since invoking `Exclusive` does not block when `Sharing` has been invoked) which re-introduces the redundancy.

Another drawback of the above code is the fact that arguments are passed *implicitly* between channels and join bodies: in the third case, the argument `n` passed to the delegate is the argument of the `Sharing` channel. Contrast this with the Cω buffer example in which the `Put` event explicitly binds its argument `x`. Not only are arguments passed implicitly, the order in which they are passed is merely *conventional* and not checked by the compiler. For example, the delegate of a (hypothetical) join pattern with two channels of type `Asynchronous.Channel<int>` would have two `int` arguments. Accidentally swapping the arguments in the body delegate would go unnoticed and result in errors.

In Scala Joins the join patterns of the above example are expressed as follows:

```
join {
  case Exclusive() & Sharing(0) => Exclusive.reply()
  case ReleaseExclusive() => Sharing(0); ReleaseExclusive.reply()
  case Shared() & Sharing(n) => Sharing(n+1); Shared.reply()
  case ReleaseShared() & Sharing(n) if n > 0 =>
    Sharing(n-1); ReleaseShared.reply()
}
```

The internal state of the lock is now represented uniformly using only `Sharing`. Moreover, two formerly separate patterns are unified (patterns 3 and 4 in the C# example) and the `if-else` statement is gone. (Inside join bodies, synchronous events are replied to via their `reply` method; this is necessary since, contrary to C# and Cω, Scala Joins supports multiple synchronous events per pattern, cf. section 3.) The gain in expressivity is due to *nested pattern matching*. In the first pattern, pattern matching constrains the argument of `Sharing` to `0`, ensuring that this pattern only triggers when no other thread is sharing the lock. Therefore, an additional `Idle` event is no longer necessary, which decreases the number of patterns. In the last pattern, a *guard* (`if n > 0`) prevents invalid states (i.e. invoking `Sharing(n)` where $n < 0$).

Joins for Actors. While join patterns have been successfully used to synchronize threads, to the best of our knowledge, join patterns have not yet been applied in the

[4] The above implementation actually ensures that an idle lock is always represented as `Idle` and never as `Sharing(0)`. However, this close relationship between `Idle` and `Sharing` is not explicit and has to be inferred from all the join patterns.

context of an actor-based concurrency model [1]. In Scala, actor-based concurrency is supported by means of a library extension [11]. Because we provide join patterns as a library as well, we have created the opportunity to combine join patterns with the concurrency model offered by actors. We give a more detailed explanation of this combination in section 3. However, in order to understand this integration, we first briefly highlight how to write concurrent programs using Scala's actor framework.

Scala's actors are largely inspired by Erlang's model of concurrent processes communicating by message passing [2]. New actors are defined as classes extending the `Actor` class. An actor's life cycle is defined by its `act` method. The following code shows how to implement the unbounded buffer as an actor:

```
class Buffer extends Actor {
  def act() { loop(List()) }
  def loop(buf: List[Int]) {
    receive {
      case Put(x) => loop(buf ::: List(x)) // append x to buf
      case Get() if !buf.isEmpty =>
        reply(buf.head); loop(buf.tail) }
  } }
```

The `receive` method allows an actor to selectively wait for certain messages to arrive in its mailbox. The actor processes at most one message at a time. Messages that are sent concurrently to the actor are queued in its mailbox. Interacting with a buffer actor occurs as follows:

```
val buffer = new Buffer; buffer.start()
buffer ! Put(42) // asynchronous send, returns nothing
println(buffer !? Get()) // synchronous send, waits for reply
```

Synchronous message sends make the sending process wait for the actor to reply to the message (by means of `reply(value)`). Scala actors also offer more advanced synchronization patterns such as futures [12,25]. `actor !! msg` denotes an asynchronous send that immediately returns a future object. In Scala, a future is a nullary function that, when applied, returns the future's computed result value. If the future is applied before the value is computed, the caller is blocked.

In the above example, the required synchronization between `Put` and `Get` is achieved by means of a *guard*. The guard in the `Get` case disallows the processing of any `Get` message while the `buf` queue is empty. In the implementation, all cases are sequentially checked against the incoming message. If no case matches, or all of the guards for matching cases evaluate to false, the actor keeps the message stored in its mailbox and awaits other messages.

Even though the above example remains simple enough to implement, the synchronization between `Put` and `Get` remains very implicit. The actual *intention* of the programmer, i.e. the fact that an item can only be produced when the actor received both a `Get` *and* a `Put` message, remains implicit in the code. Therefore, even actors can benefit from the added declarative synchronization of join patterns, as we illustrate in section 3.

3 A Scala Joins Library

We discuss a Scala library (called Scala Joins) providing join patterns implemented via extensible pattern matching. First, we explain how Scala Joins enables declarative thread synchronization, postponing joins for actors until the next section.

Joining Threads. Join patterns in Scala Joins are composed of synchronous and asynchronous *events*. Events are strongly typed and can be invoked using standard method invocation syntax. The FIFO buffer example is written in Scala Joins as follows:

```
class Buffer extends Joins {
  val Put = new AsyncEvent[Int]
  val Get = new SyncEvent[Int]
  join { case Get() & Put(x) => Get reply x }
}
```

To enable join patterns, a class inherits from the `Joins` class.[5] Events are declared as regular fields. They are distinguished based on their (a)synchrony and the number and types of arguments they take. For example, `Put` is an asynchronous event that takes a single argument of type `Int`. Since it is asynchronous, no return type is specified (it immediately returns `unit` when invoked). In the case of a synchronous event such as `Get`, the first type parameter specifies the return type. Therefore, `Get` is a synchronous event that takes no arguments and returns values of type `Int`.

Joins are declared using the `join { ... }` construct.[6] This construct enables pattern matching via a list of `case` declarations that each consist of a left-hand side and a right-hand side, separated by `=>`. The left-hand side defines a join pattern through the juxtaposition of a linear combination of asynchronous and synchronous events. As is common in the joins literature, we use `&` as the juxtaposition operator. Arguments of events are usually specified as variable patterns. For example, the variable pattern `x` in the `Put` event can bind to any value (of type `Int`). This means that on the right-hand side, `x` is bound to the argument of the `Put` event when the join pattern matches. Standard pattern matching can be used to constrain the match even further (see section 2).

The right-hand side of a join pattern defines the join body (an ordinary block of code) that is executed when the join pattern matches. Like JoCaml, but unlike Cω and C# Joins, Scala Joins allows any number of synchronous events to appear in a join pattern. Because of this, it is impossible to use the return value of the body to implicitly reply to the single synchronous event in the join pattern. Instead, the body of a join pattern explicitly replies to all synchronous events that are part of the join pattern on the left-hand side. This is done by invoking those events' `reply` method, which wakes up the thread that originally signaled that event.

[5] Actually, `Joins` is a *trait* that can be mixed into any class.

[6] As explained in section 4, `join` is a method of the `Joins` class. In Scala, the body of a class definition serves as the primary constructor of the class which allows this freestanding call to `join`.

Joining Actors. We now describe an integration of our joins library with Scala's actor framework. The following example shows how to re-implement the unbounded buffer example using joins:

```
val Put = new Join1[Int]
val Get = new Join
class Buffer extends JoinActor {
  def act() {
    receive { case Get() & Put(x) => Get reply x }
  } }
```

It differs from the thread-based bounded buffer using joins in the following ways:

- The `Buffer` class inherits from the `JoinActor` class to declare itself to be an actor capable of processing join patterns.
- Rather than defining `Put` and `Get` as synchronous or asynchronous *events*, they are all defined as *join messages* which may support both kinds of synchrony (this is explained in more detail below).
- The `Buffer` actor defines `act` and awaits incoming messages by means of `receive`. It is still possible for the actor to serve regular messages within the `receive` block. Logically, regular messages can be regarded as unary join patterns. However, they don't have to be declared as joinable messages.

We illustrate below how the buffer actor can be used as a coordinator between a consumer and a producer actor. The producer sends an asynchronous `Put` message while the consumer awaits the reply to a `Get` message by invoking it synchronously (using `!?`).

```
val buffer = new Buffer; buffer.start()
val prod = actor { buffer ! Put(42) }
val cons = actor { process(buffer !? Get()) }
```

By applying joins to actors, the synchronization dependencies between `Get` and `Put` can be specified declaratively by the buffer actor. The actor receives `Get` and `Put` messages by queuing them in its mailbox. Only when all of the messages specified in the join pattern have been received is the body executed by the actor. Before processing the body, the actor atomically removes all of the participating messages from its mailbox. Replies may be sent to any or all of the messages participating in the join pattern. This is similar to the way replies are sent to events in the thread-based joins library described previously.

Contrary to the way events are defined in the thread-based joins library, an actor does not explicitly define a join message to be synchronous or asynchronous. We say that join messages are "synchronization-agnostic" because they can be used in different synchronization modes between the sender and receiver actors. However, when they are used in a particular join pattern, the sender and receiver actors have to agree upon a valid synchronization mode. In the previous example, the `Put` join message was sent asynchronously, while the `Get` join message was sent synchronously. In the body of a join pattern, the receiver actor replied to `Get`, but not to `Put`.

The disadvantage of making join messages synchronization-agnostic is that it introduces the possibility for errors. For example, if a receiver does not reply to a synchronously sent message, the sender remains blocked. However, the advantage is that

join messages may be used in many different synchronization modes, including future-type message sending [25] or Salsa's token-passing continuations [22]. Every join message has an associated *reply destination* which is an output channel on which processes may listen for replies to the message. How the reply to a message is processed is determined by the way the message was sent. For example, if the message was sent purely asynchronously, the reply is discarded; if it was sent synchronously, the reply awakes the sender. If it was sent using a future-type message send, the reply resolves the future.

4 Integrating Joins and Extensible Pattern Matching

In this section we present a novel implementation that integrates joins into general language-based pattern matching. We explain our technique using a concrete implementation in Scala. However, we expect that implementations based on, e.g., the active patterns of F# [20] would not be much different.

In the following we first look at pattern matching in Scala; this provides some terminology and background used in subsequent sections. After that we review the essentials of Scala's extensible patterns; the small set of necessary concepts suggests that our approach is readily transferable to languages with similar features. In section 4.1 we outline the core of an implementation of joins that builds on extensible pattern matching. In section 4.2 we highlight how joins have been integrated into Scala's actor framework.

Partial Functions. In the previous section we used the `join { ... }` construct to declare a set of join patterns. It has the following form:

```
join {
  case pat_1 => body_1
  ...
  case pat_n => body_n
}
```

The patterns pat_i consist of a linear combination of events evt_1 `&` `...` `&` evt_m. Threads synchronize over a join pattern by invoking one or several of the events listed in a pattern pat_i. When all events occurring in pat_i have been invoked, the join pattern matches, and its corresponding join $body_i$ is executed.

In Scala, the pattern matching expression inside braces is treated as a first-class value that is passed as an argument to the `join` function. 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 parametrized; the first type parameter `A` indicates the function's argument type and the second type parameter `B` indicates its result type.

In Scala, each pattern matching expression

```
{ case p1 => e1; ...; case pn => en }
```

is compiled into 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.

Note that partial functions are not crucial for our implementation of joins. In fact, Scala's partial functions can be encoded using only higher-order functions as follows. The idea is to define a partial function as a regular function that returns an option;[7] either the partial function is defined at the given value, in which case it returns its body *as a thunk* (i.e. a function with an empty parameter list) wrapped in `Some`. If the partial function is not defined, it returns `None`. Operations for testing whether a partial function is defined at a given value, and for applying it are defined accordingly:

```
type PartFun[A, R] = A => Option[() => R]
def isDefAt[A, R](fun: PartFun[A, R], arg: A) = fun(arg) match {
  case Some(_) => true
  case None    => false }
def apply[A, R](fun: PartFun[A, R], arg: A) = fun(arg) match {
  case Some(res) => res()
  case None      => error("PartFun not defined") }
```

Using this encoding, the native Scala partial function

```
{ case x :: xs => println("head: "+x) }
```

can then be represented as follows:

```
(l: List[Int]) => l match {
  case x :: xs => Some(() => println("head: "+x))
  case _       => None }
```

Join patterns as partial functions. Whenever a thread invokes an event, each join pattern in which `e` occurs has to be checked for a potential match. Therefore, events have to be associated with the set of join patterns in which they participate. As shown before, this set of join patterns is represented as a partial function. Invoking `join(pats)` associates each event occurring in the set of join patterns with `pats`.

When a thread invokes an event, the `isDefinedAt` method of `pats` is used to check whether any of the associated join patterns match. If yes, the corresponding join

[7] The optional value is of parameterized type `Option[T]` that has the two subclasses `Some[T](x: T)` and `None`.

body is executed by invoking the `apply` method of `pats`. A question remains: what argument is passed to `isDefinedAt` and `apply`, respectively? To answer this question, consider the simple buffer example from the previous section. It declares the following join pattern:

```
join { case Get() & Put(x) => Get reply x }
```

Assume that no events have been invoked before, and a thread t invokes the `Get` event to remove an element from the buffer. Clearly, the join pattern does not match, which causes t to block since `Get` is a synchronous event (more on synchronous events later). Assume that after thread t has gone to sleep, another thread s adds an element to the buffer by invoking the `Put` event. Now, we want the join pattern to match since both events have been invoked. However, the result of the matching does not only depend on the event that was last invoked but also on the fact that *other events* have been invoked previously. Therefore, it is *not* sufficient to simply pass a `Put` message to the `isDefinedAt` method of the partial function the represents the join patterns. Instead, when the `Put` event is invoked, the `Get` event has to somehow "pretend" to also match, even though it has nothing to do with the current event. While previous invocations can simply be buffered inside the events, it is non-trivial to make the pattern matcher actually consult this information during the matching, and "customize" the matching results based on this information. To achieve this customization we use extensible pattern matching.

Extensible Pattern Matching. Emir et al. [6] recently introduced *extractors* for Scala that provide representation independence for objects used in patterns. Extractors play a role similar to *views* in functional programming languages [24,15] in that they allow conversions from one data type to another to be applied implicitly during pattern matching. As a simple example, consider the following object that can be used to match even numbers:

```
object Twice {
  def apply(x: Int) = x*2
  def unapply(z: Int) = if (z%2 == 0) Some(z/2) else None }
```

Objects with `apply` methods are uniformly treated as functions in Scala. When the function invocation syntax `Twice(x)` is used, Scala implicitly calls `Twice.apply(x)`. The `unapply` method in `Twice` reverses the construction in a pattern match. It tests its integer argument `z`. If `z` is even, it returns `Some(z/2)`. If it is odd, it returns `None`. The `Twice` object can be used in a pattern match as follows:

```
val x = Twice(21)
x match {
  case Twice(y) => println(x+" is two times "+y)
  case _ => println("x is odd") }
```

To see where the `unapply` method comes into play, consider the match against `Twice(y)`. First, the value to be matched (`x` in the above example) is passed as argument to the `unapply` method of `Twice`. This results in an optional value which is matched subsequently. The preceding example is expanded as follows:

```
val x = Twice.apply(21)
Twice.unapply(x) match {
  case Some(y) => println(x+" is two times "+y)
  case None => println("x is odd") }
```

Extractor patterns with more than one argument correspond to `unapply` methods returning an optional tuple. Nullary extractor patterns correspond to `unapply` methods returning a Boolean.

In the following we show how extractors can be used to implement the matching semantics of join patterns. In essence, we define appropriate `unapply` methods for events which get implicitly called during the matching.

4.1 Matching Join Patterns

As shown previously, a set of join patterns is represented as a partial function. Its `isDefinedAt` method is used to find out whether one of the join patterns matches. In the following we are going to explain the code that the Scala compiler produces for the body of this method. Let us revisit the join pattern that we have seen in the previous section:

```
Get() & Put(x)
```

In our library, the `&` operator is an extractor that defines an `unapply` method; therefore, the Scala compiler produces the following matching code:

```
&.unapply(m) match {
  case Some((Get(), Put(x))) => true
  case None => false }
```

We defer a discussion of the argument m that is passed to the `&` operator. For now, it is important to understand the general scheme of the matching process. Basically, calling the `unapply` method of the `&` operator produces a pair of intermediate results wrapped in `Some`. Nested pattern matching matches the two components of the pair against the `Get` and `Put` events. Only if both of them match, the overall pattern matches. Since the `&` operator is left-associative, matching more than two events proceeds by first calling the unapply methods of all the `&` operators from right to left, and then matching the intermediate results with the corresponding events from left to right.

Since events are objects that have an `unapply` method, we can expand the code further:

```
&.unapply(m) match {
  case Some((u, v)) =>
    Get.unapply(u) match {
      case true => Put.unapply(v) match {
        case Some(x) => true
        case None => false }
      case false => false }
  case None => false }
```

As we can see, the intermediate results produced by the `unapply` method of the `&` operator are passed as arguments to the `unapply` methods of the corresponding events. Since the `Get` event is parameter-less, its `unapply` method returns a Boolean, telling whether it matches or not. The `Put` event, on the other hand, takes a parameter; when the pattern matches, this parameter gets bound to a concrete value that is produced by the `unapply` method.

The `unapply` method of a parameter-less event such as `Get` essentially checks whether it has been invoked previously. The `unapply` method of an event that takes parameters such as `Put` returns the argument of a previous invocation (wrapped in `Some`), or signals failure if there is no previous invocation. In both cases, previous invocations have to be buffered inside the event.

Firing join patterns. As mentioned before, executing the right-hand side of a pattern that is part of a partial function amounts to invoking the `apply` method of that partial function. Basically, this repeats the matching process, thereby binding any pattern variables to concrete values in the pattern body. When firing a join pattern, the events' `unapply` methods have to dequeue the corresponding invocations from their buffers. In contrast, invoking `isDefinedAt` does not have any effect on the state of the invocation buffers. To signal to the events in which context their `unapply` methods are invoked, we therefore need some way to propagate out-of-band information through the matching. For this, we use the argument m that is passed to the `isDefinedAt` and `apply` methods of the partial function. The `&` operator propagates this information verbatim to its two children (its `unapply` method receives m as argument and produces a pair with two copies of m wrapped in `Some`). Eventually, this information is passed to the events' `unapply` methods.

Implementation Details. Events are represented as classes that contain queues to buffer invocations. The `Event` class is the super class of all synchronous and asynchronous events:[8]

```
abstract class Event[R, Arg](owner: Joins) {
  val tag = owner.freshTag()
  val argQ = new Queue[Arg]
  def apply(arg: Arg): R = synchronized { argQ += arg; invoke() }
  def invoke(): R
  def unapply(isDryRun: Boolean): Option[Arg] =
    if (isDryRun && !argQ.isEmpty)
      Some(argQ.front)
    else if (!isDryRun)
      Some(argQ.dequeue())
    else None }
```

The `Event` class takes two type arguments `R` and `Arg` that indicate the result type and parameter type of event invocations, respectively. Events have a unique `owner` which is

[8] In our actual implementation the fact whether an event is parameter-less is factored out for efficiency. Due to lack of space, we show a simplified class hierarchy.

passed as argument of the primary constructor of the `Event` class.[9] An event can appear in several join patterns declared by its `owner`. The `tag` field holds an identifier which is unique with respect to a given `owner` instance; it is used to check the linearity of patterns (i.e. ensuring that an event occurs at most once in a pattern).

Whenever the event is invoked via its `apply` method, we append the provided argument to the `argQ`. The abstract `invoke` method is used to run synchronization-specific code; synchronous and asynchronous events differ mainly in their implementation of the `invoke` method (we show a concrete implementation for synchronous events below). In the `unapply` method we test whether matching occurs during a dry run. If it does not we dequeue an event invocation.

Synchronous events are implemented as follows:

```
abstract class SyncEvent[R, Arg] extends Event[R, Arg] {
  val waitQ = new Queue[SyncVar[R]]
  def invoke(): R = { val res = new SyncVar[R]
    waitQ += res; owner.matchAndRun(); res.get }
  def reply(res: R) = waitQ.dequeue().set(res) }
```

Synchronous events contain a logical queue of waiting threads, `waitQ`, which is implemented using the implicit wait set of synchronous variables.[10] The `invoke` method is run whenever the event is invoked. It creates a new `SyncVar` and appends it to the `waitQ`. Then, the `owner`'s `matchAndRun` method is invoked to check whether the event invocation triggers a complete join pattern. After that, the current thread waits for the `SyncVar` to become initialized by accessing it. If the owner detects (during `owner.matchAndRun()`) that a join pattern triggers, it will apply the join, thereby re-executing the pattern match (binding variables etc.) and running the join body. Inside the body, synchronous events are replied to by invoking their `reply` method. Replying means dequeuing a `SyncVar` and setting its value to the supplied argument. If none of the join patterns matches, the thread that invoked the synchronous event is blocked (upon calling `res.get`) until another thread triggers a join pattern that contains the same synchronous event.

Thread-safety. Our implementation avoids races when multiple threads try to match a join pattern at the same time; checking whether a join pattern matches (and, if so, running its body) is an atomic operation. Notably, the `isDefinedAt`/`apply` methods of the join set are only called from within the synchronized `matchAndRun` method of the `Joins` class. The `unapply` methods of events, in turn, are only called from within the matching code inside the partial function, and are thus guarded by the same lock. The internal state of individual events is updated consistently: the `apply` method is atomic, and the `reply` method is called only from within join bodies which are guarded by the

[9] To allow the short syntax for declaring events that we have seen before, the owner is passed *implicitly* in the actual implementation. It is defined to be the current object `this` of the pattern-declaring class that inherits from `Joins`. A detailed account of implicit parameters in Scala is out of scope of this paper; the interested reader is referred to the Scala language specification.

[10] A `SyncVar` is an atomically updatable reference cell; it blocks threads trying to access an uninitialized cell.

`owner`'s lock. We don't assume any concurrency properties of the `argQ` and `waitQ` queues.

Optimization. Efficient join implementations represent patterns using bit sets [3,17]. An event with tag n forms part of a pattern iff bit n is set in the corresponding bit set. This representation allows one to efficiently check whether an event invocation triggers a join pattern.

The above implementation cannot use such an optimization as is, since the abstract `PartialFunction` class is the only way to interact with the set of join patterns; for instance the number of patterns is not known *a priori*. However, it is possible to *gradually construct* an efficient bit set representation during the matching process. The idea is to keep track of event invocations while matching a pattern. When a pattern matches, the tags of matched events give rise to a bit set that uniquely represents the pattern. At the point where each pattern has matched at least once, the bit sets are used to efficiently check for a match. If the set of events with queued invocations is represented as a bit set ib, then invoking an event with tag n triggers a pattern represented as pb iff $pb \subseteq ib \cup \{n\}$.

To test the effectiveness of the above optimization, we compared the performance of a bounded buffer implementation using our library without the optimization with a second one using the optimized library. Concurrently reading/writing 10^6 items from/to a bounded buffer of size 100 is about 28% faster using the optimized library. However, this is only a first step towards an efficient implementation. Further optimizations are a worthwhile topic for future work.

4.2 Implementation of Actor-based Joins

Actor-based joins integrate with Scala's pattern matching in essentially the same way as the thread-based joins, making both implementations very similar. We highlight how joins are integrated into the actor library, and how reply destinations are supported.

In the Scala actors library, `receive` is a method that takes a `PartialFunction` as a sole argument, similar to the `join` method defined previously. To make `receive` aware of join patterns, the abstract `JoinActor` class overrides these methods by wrapping the partial function into a specialized partial function that understands join messages. `JoinActor` also overrides `send` to set the reply destination of a join message. When an actor executes `a!msg`, it invokes the `!` method of `a`. This method invokes `a.send`, implicitly passing the reply channel of the sender actor as a second argument.

```
abstract class JoinActor extends Actor {
  override def receive[R](f: PartialFunction[Any, R]): R =
    super.receive(new JoinPatterns(f))
  override def send(msg: Any, replyTo: OutputChannel[Any]) {
    setReplyDest(msg, replyTo)
    super.send(msg, replyTo) }
  def setReplyDest(msg: Any, replyTo: OutputChannel[Any]) {...} }
```

`JoinPatterns` is a special partial function that detects whether its argument message is a join message. If it is, then the argument message is transformed to include out-of-band information that will be passed to the pattern matcher, as is the case for events in

the thread-based joins library. The boolean argument passed to the `asJoinMessage` method indicates to the pattern matcher whether or not join message arguments should be dequeued upon successful pattern matching. If the `msg` argument is not a join message, `asJoinMessage` passes the original message to the pattern matcher unchanged, enabling regular actor messages to be processed as normal.

```
class JoinPatterns[R](f: PartialFunction[Any, R])
  extends PartialFunction[Any, R] {
  def asJoinMessage(msg: Any, isDryRun: Boolean): Any =
    ...
  override def isDefinedAt(msg: Any) =
    f.isDefinedAt(asJoinMessage(msg, true))
  override def apply(msg: Any) =
    f(asJoinMessage(msg, false))
}
```

Recall from the implementation of synchronous events that thread-based joins used constructs such as `SyncVar`s to synchronize the sender of an event with the receiver. Actor-based joins do not use such constructs. In order to synchronize sender and receiver, every join message has a reply destination (which is an `OutputChannel`, set when the message is sent in the actor's `send` method) on which a sender may listen for replies. The `reply` method of a `JoinMessage` simply forwards its argument value to this encapsulated reply destination. This wakes up an actor that performed a synchronous send (`a!?msg`) or that was waiting on a future (`a!!msg`).

5 Discussion and Related Work

Benton et al. [3] note that supporting general guards in join patterns is difficult to implement efficiently as it requires testing all possible combinations of queued messages to find a match. Side effects pose another problem. Benton et al. suggest a restricted language for guards to overcome these issues. However, to the best of our knowledge, there is currently no joins framework that supports a sufficiently restrictive yet expressive guard language to implement efficient guarded joins. Our current implementation handles (side-effect free) guards that only depend on arguments of events that queue at most one invocation at a time.

Cω [3] is a language extension of C# supporting *chords*, linear combinations of methods. In contrast to Scala Joins, Cω allows at most one synchronous method in a chord. The thread invoking this method is the thread that eventually executes the chord's body. The benefits of Cω as a language extension over Scala Joins are that chords can be enforced to be well-formed and that their matching code can be optimized ahead of time. In Scala Joins, the joins are only analyzed at pattern-matching time. The benefit of Scala Joins as a library extension is that it provides more flexibility, such as multiple synchronous events. Russo's C# Joins library [17] exploits the expressiveness of C# 2.0's generics to implement Cω's synchronization constructs. Piggy-backing on an existing variable binding mechanism allows us to avoid problems with C# Joins' delegates where the order in which arguments are passed is merely conventional. Scala Joins extends both

Cω and C# Joins with *nested patterns* that can avoid certain redundancies by generalizing events and patterns.

CCR [4] is a C# library for asynchronous concurrency that supports join patterns without synchronous components. Join bodies are scheduled for execution in a thread pool. Our library integrates with JVM threads using synchronous variables, and supports event-based programming through its integration with Scala Actors. Singh [18] shows how a small set of higher-order combinators based on Haskell's software transactional memory (STM) can encode expressive join patterns. CML [16] allows threads to synchronize on first-class composable events; because all events have a single commit point, certain protocols may not be specified in a modular way (for example when an event occurs in several join patterns). By combining CML's events with all-or-nothing transactions, transactional events [5] overcome this restriction but may have a higher overhead than join patterns.

Synchronization in actor-based languages is a well-studied domain. Salsa [22] is a Java language extension with support for actors. It introduces the notion of a *join continuation*. However, join continuations are not to be mistaken with join patterns: the former only allow an actor to synchronize on multiple replies to previously sent messages. Activation based on message sets [10] is more general than joins since events/channels have a fixed owner, which enables important optimizations. Other actor-based languages allow for a synchronization style similar to that supported by join patterns. For example, *behavior sets* in Act++ [13] or *enabled sets* in Rosette [21] allow an actor to restrict the set of messages which it may process. They do so by partitioning messages into different sets representing different actor states. Joins do not make these states explicit, but rather allow state transitions to be encoded in terms of sending messages. The novelty of Scala Joins for actors is that such synchronization is integrated with the actor's standard message reception operation using extensible pattern matching. Recent work by Sulzmann et al. [19] extends Erlang-style actors with receive patterns consisting of multiple messages, which is very similar to our join-based actors. The two approaches are complementary: their work focuses on providing a formal matching semantics in form of Constraint Handling Rules whereas the emphasis of our work lies on the integration of joins with extensible pattern matching; Scala Joins additionally permits joins for standard (non-actor) threads that do not have a mailbox.

6 Conclusion

We presented a novel implementation of join patterns based on extensible pattern matching constructs of languages such as Scala and F#. The embedding into general pattern matching provides expressive features such as nested patterns and guards. The resulting programs are often as concise as if written in more specialized language extensions. We implemented our approach as a Scala library that supports join patterns with multiple synchronous events and guards and furthermore integrated it with the Scala Actors concurrency framework without changing the syntax and semantics of existing programs.

References

1. Gul A. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, Massachusetts, 1986.
2. Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang, Second Edition*. Prentice-Hall, 1996.
3. Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for C#. *ACM Trans. Program. Lang. Syst*, 26(5):769–804, 2004.
4. Georgio Chrysanthakopoulos and Satnam Singh. An asynchronous messaging library for C#. In *Proc. SCOOL Workshop, OOPSLA*, 2005.
5. Kevin Donnelly and Matthew Fluet. Transactional events. In *Proc. ICFP*, pages 124–135. ACM, 2006.
6. Burak Emir, Martin Odersky, and John Williams. Matching objects with patterns. In *Proc. ECOOP*, volume 4609 of *LNCS*, pages 273–298. Springer, 2007.
7. Cédric Fournet, Fabrice Le Fessant, Luc Maranget, and Alan Schmitt. JoCaml: A language for concurrent distributed and mobile programming. In *Advanced Functional Programming*, volume 2638 of *LNCS*, pages 129–158. Springer, 2002.
8. Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proc. POPL*, pages 372–385. ACM, January 1996.
9. Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A Calculus of Mobile Agents. In *Proc. CONCUR*, pages 406–421. Springer, August 1996.
10. Svend Frølund and Gul Agha. Abstracting interactions based on message sets. In *Object-Based Models and Languages for Concurrent Systems*, volume 924 of *LNCS*, pages 107–124. Springer, 1994.
11. Philipp Haller and Martin Odersky. Actors that unify threads and events. In *Proc. COORDINATION*, volume 4467 of *LNCS*, pages 171–190. Springer, 2007.
12. Robert H. Halstead, Jr. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.
13. D. Kafura, M. Mukherji, and G. Lavender. ACT++: A Class Library for Concurrent Programming in C++ using Actors. *J. of Object-Oriented Programming*, 6(6), 1993.
14. Martin Odersky. Functional Nets. In *Proc. ESOP*, LNCS. Springer, 2000.
15. Chris Okasaki. Views for Standard ML. In *Proc. SIGPLAN Workshop on ML*, 1998.
16. John H. Reppy. CML: A higher-order concurrent language. In *Proc. PLDI*, pages 293–305. ACM, 1991.
17. Claudio V. Russo. The Joins concurrency library. In *Proc. PADL*, pages 260–274, 2007.
18. Satnam Singh. Higher-order combinators for join patterns using STM. In *Proc. TRANSACT Workshop, OOPSLA*, 2006.
19. Martin Sulzmann, Edmund S. L. Lam, and Peter Van Weert. Actors with multi-headed message receive patterns. In *Proc. COORDINATION*, LNCS. Springer, 2008.
20. Don Syme, Gregory Neverov, and James Margetson. Extensible pattern matching via a lightweight language extension. In *Proc. ICFP*, pages 29–40. ACM, 2007.
21. C. Tomlinson and V. Singh. Inheritance and synchronization with enabled-sets. *ACM SIGPLAN Notices*, 24(10):103–112, 1989.
22. Carlos Varela and Gul Agha. Programming dynamically reconfigurable open systems with SALSA. *ACM SIGPLAN Notices*, 36(12):20–34, 2001.
23. G.S. von Itzstein and David Kearney. Join Java: An alternative concurrency semantic for Java. Technical report, University of South Australia, 2001.
24. Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proc. POPL*, pages 307–313, 1987.
25. Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In *Proc. OOPSLA*, pages 258–268, 1986.