Implementing Joins using Extensible Pattern Matching

Philipp Haller[1], Tom Van Cutsem[2⋆]

[1] École Polytechnique Fédérale de Lausanne (EPFL)
1015 Lausanne, Switzerland
firstname.lastname@epfl.ch
+41 21 693 6483, +41 21 693 6660
[2] Programming Technology Lab, 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 dynamic joins, Ada-style rendezvous, and constraints. 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, Actors

1 Introduction

Join patterns [6,7] 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 functional languages such as JoCaml [5] and Funnel [12]. Join patterns have also been implemented as extensions to existing languages [2,18]. Recently, Russo [14] and Singh [15] 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. A library-based approach has several advantages over a language extension, such as support for dynamic joins, cross-language portability, and ease of experimentation.

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 [4,16]. Extensible patterns open up new possibilities of experimenting with declarative approaches to synchronization in libraries rather than languages. In this paper we present a novel implementation of join patterns that exploits Scala's extensible pattern matching. More concretely, we make the following contributions:

1. Our implementation technique overcomes several limitations of previous library-based designs and language extensions. In all library-based implementations that we know of, pattern variables are represented implicitly as parameters of join continuations. Mixing up parameters of the same type inside the join body may lead to

⋆ supported by a Ph.D. fellowship of the Research Foundation Flanders (FWO).

obscure errors that are hard to detect. Our design avoids these errors by using the underlying pattern matcher to bind variables that are explicit in join patterns. The programmer may use a rich pattern syntax to express constraints using nested patterns and guards. However, efficiently supporting general guards in join patterns is currently an open problem, and we do not attempt to solve it. Moreover, the integration with a powerful pattern matcher brings compile-time exhaustivity checking of join patterns closer to library-based implementations than ever (our implementation does not currently support it, but we don't see fundamental problems).

2. We present a complete implementation of our design as a Scala library[3] that supports dynamic joins, Ada-style rendezvous, and constraints. Moreover, we integrate our library into an existing event-based, non-blocking 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. Our integration is unique in the sense that the new library provides a conservative syntax extension. That is, existing programs continue to run without change when compiled or linked against the extended framework.

The rest of this paper is structured as follows. In the following section we review join patterns written using our library. In section 3 we discuss a concrete Scala implementation of our techniques. Section 4 presents an integration of our joins library into an existing event-based concurrency framework. Its implementation is outlined in section 5. Section 6 discusses related work, and section 7 concludes.

2 A Scala Joins Library

Our joins library draws on Scala's extensible pattern matching facility [4]. This has several advantages: first of all, the programmer may use Scala's rich pattern syntax to express constraints using nested patterns and guards. Moreover, reusing the existing variable binding mechanism avoids typical problems of other library-based approaches where the order in which arguments are passed to the function implementing the join body are merely conventional. Instead of (a)synchronous methods, as in Cω, joins in our library are composed of synchronous and asynchronous *events*. Events are strongly typed and are invoked just like normal methods. Join patterns define linear combinations of events which guard the execution of a join body. The join body typically operates on the arguments of the corresponding event invocations and replies to the synchronous events.

Using our library, we can implement the unbounded buffer example as follows:

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

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

First of all, the `Buffer` class inherits the `Joins` trait to enable join patterns. Furthermore, it declares two events. `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). When declaring synchronous events, 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. It contains 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. We allow any number of synchronous events to appear in a join pattern. Arguments of events are usually specified as variable patterns. In the above example, the argument of the `Put` event is specified as the variable pattern `x` that matches any value. This means that on the right-hand side, `x` is bound to the argument of the `Put` event when the join pattern matches.

The right-hand side of a join pattern defines the action (an ordinary block of code) that is executed when the join pattern matches. An important thing to do inside actions is replying to all of the synchronous events that are part of the join pattern on the left-hand side. Synchronous events are replied to by invoking their `reply` method. In the example, we reply to the synchronous `Get` event with the value that was passed as argument to the `Put` event.

3 Implementation

Our implementation technique for joins is unique in the way events interact with an extensible pattern matching mechanism. We explain the technique using a concrete implementation in Scala. However, we expect that implementations based on, e.g., the active patterns of F# [16] would not be much different. In the following we first talk about pattern matching in Scala. After that we dive into the implementation of events which crucially depends on properties of Scala's extensible pattern matching.

In the previous section we used the `join { ... }` construct to declare joins. In Scala, the pattern matching expression inside braces is treated as a first-class object that is passed as an argument to the `join` method (which is inherited from the `Joins` class). The argument is an instance of the `PartialFunction` class, which is a subclass of `Function1`, the class of unary functions. The two classes are defined as follows.

```
abstract class Function1[A, B] {
  def apply(x: A): B }
abstract class PartialFunction[A, B] extends Function1[A, B] {
  def isDefinedAt(x: A): Boolean }
```

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

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 matches, a `MatchError` exception is thrown.

3.1 Extractors

Emir et al. [4] recently introduced *extractors* for Scala that provide representation independence for objects used in patterns. 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 }
```

As mentioned before, 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[4]. 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.

3.2 Events

Events are represented as classes that contain queues to buffer invocations. Thanks to their `apply` methods, they can be invoked just like normal methods. The `Event` class is the super class of all synchronous and asynchronous events:

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

```
trait SyncEventBase[R] extends Event[R] {
  val waitQ = new Queue[SyncVar[R]]
  def test(): R = {
    val res = new SyncVar[R]
    waitQ += res
    owner.test()
    res.get }
  def reply(res: R): Unit =
    waitQ.dequeue().set(res) }
class NullarySyncEvent[R](implicit joins: Joins) extends SyncEventBase[R] {
  val owner = joins
  def apply(): R = synchronized { test() }
  def unapply(isDryRun: Boolean): Boolean = {
    owner.notify(tag, isDryRun)
    !waitQ.isEmpty } }
```

Fig. 1. Synchronous events.

```
abstract class Event[R] {
  val owner: Joins
  val tag = owner.freshTag
  def test(): R }
```

Events have a unique `owner` which is a subclass of the `Joins` trait[5]. This trait provides the `join` method that we used in the buffer example (see section 2) to declare (and activate) a set of join patterns. Note that `join` calls may be nested to dynamically change the set of active join patterns. An event can appear in several join patterns declared by its `owner`. The `tag` field holds an integer constant that is unique for the events declared by `owner`. The `test` method is used to run synchronization-specific code when the event is invoked. Note that `Event` has a type parameter `R` that designates the result type of event invocations. Asynchronous events instantiate `R` with `Unit`.

Figure 1 shows the class hierarchy of nullary synchronous events. The `SyncEventBase` trait contains the scheduling logic essential to synchronous events. Synchronous events contain a logical queue of waiting threads, `waitQ`, which is implemented using the implicit wait set of synchronous variables (objects of type `SyncVar[R]`[6]). The `test` method is run whenever the event is invoked. It creates a new `SyncVar` and appends it to the `waitQ`. Then, the `owner`'s `test` 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.test()`) that a join pattern triggers, it will apply the join, thereby 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. As shown in figure 1,

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

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

replying means dequeuing a `SyncVar` and setting its value to the supplied argument. Note that the `SyncEventBase` trait cannot be instantiated since the `owner` field inherited from `Event` is still abstract.

Figure 1 also shows the concrete `NullarySyncEvent` class that extends the `SyncEventBase` trait. Our design requires that concrete classes are passed a `Joins` instance as a constructor argument. In Scala, constructor arguments are passed in parentheses following the class name (with optional type parameters). The body of a class declaration is the primary constructor[7]. The single `joins` constructor argument carries the `implicit` modifier. This means that the compiler implicitly passes an appropriate value (declared `implicit`) which is in scope at the point where the class is *instantiated*. We use implicit parameters to avoid some initialization boilerplate which has to be done explicitly by the user in other library-based designs [14].

Events have an `apply` method which is run when it is invoked, and an `unapply` method which is run during pattern matching. When a nullary synchronous event is invoked we simply call the `test` method of its super class that we discussed before. The `unapply` method is more interesting. As discussed in the previous section, matching on a nullary object (e.g. `Get()` in the buffer example) invokes its boolean-returning `unapply` method that tells whether the argument value matches or not. Essentially, the `unapply` method of `NullarySyncEvent` should return `true` whenever there is a thread waiting in the `waitQ`.

Since the scrutinee has no influence on the matching outcome, we use it to carry out-of-band information about the ongoing match. Note that the events' `unapply` methods are called in two different contexts, namely when the owner merely *tests* whether a join pattern matches (*dry run*), and when the owner *triggers* the execution of a join body, respectively. In the latter case, the matching is necessary to bind event arguments to variables (if any). In the former case, the owner has to collect the tags of all events participating in a matching join pattern (see below for an explanation of `owner.notify(tag, isDryRun)`).

Events that carry arguments extend the abstract `NonNullaryEvent` class which is shown in figure 2. The class has two type parameters `R` and `Arg` which model the result type and argument type of event invocations respectively. Whenever the event is invoked via its `apply` method, we append the provided argument to the `argQ`. By invoking the inherited `test` method we execute further synchronization code. A synchronous event inherits the `test` method from the `SyncEventBase` trait that we have already discussed. Note that this implementation requires multiple arguments to be wrapped in tuples, that is, the type parameter `Arg` has to be instantiated with a type `Tuple_n[P_1, ..., P_n]` where the `P_i` are the types of the individual arguments. Convenience classes, such as the following one, wrap multiple arguments in tuples.

```
class Async2[A1, A2](implicit joins: Joins) extends AsyncEvent[(A1, A2)] {
  def apply(a1: A1, a2: A2) = super.apply((a1, a2)) }
```

Matching against a non-nullary event works by implementing the `unapply` method as follows. First, `unapply` returns an option type which models success and failure, as well as the variable binding in case of success. For example, assume we are matching against an event `Put(x)` inside a (larger) join pattern. Moreover, assume the first thread that invoked `Put` passed the value `42` as argument. Then, matching against `Put(x)` should

[7] Additional constructors are defined as methods with name `this` and an inferred return type.

```
abstract class NonNullaryEvent[R, Arg] extends Event[R] {
  val argQ = new Queue[Arg]
  def apply(arg: Arg): R = synchronized {
    argQ += arg
    test() }
  def unapply(isDryRun: Boolean): Option[Arg] = {
    ...
    if (isDryRun && !argQ.isEmpty)
      Some(argQ.front)
    else if (!isDryRun && owner.takes(tag)) {
      Some(argQ.dequeue())
    } else None } }
class AsyncEvent[Arg](implicit joins: Joins) extends NonNullaryEvent[Unit, Arg] {
  val owner = joins
  def test(): Unit = owner.test() }
```

Fig. 2. Non-nullary asynchronous events.

result in x being bound to 42 inside the join body in case of a successful match of the join pattern. We implement this behavior by returning the first queued argument value in the case of a previous invocation. If there has been no previous invocation, the join pattern cannot possibly match and we signal failure. As before, we test whether matching occurs during a dry run. If it does not, we ask the owner whether the event belongs to a matching join pattern in which case an event invocation is dequeued.

Using the `NonNullaryEvent` class, it is now easy to define non-nullary asynchronous events. Figure 2 shows the `AsyncEvent` class which is the super class of all non-nullary asynchronous events. We have already explained the implicit constructor argument when we discussed the `NullarySyncEvent` class. It only remains to provide an implementation of the `test` method. Here, we simply invoke the `test` method of `owner` which we are going to discuss shortly.

A class that wants to define joins has to inherit from the `Joins` trait. The essential parts of its implementation are as follows:

```
trait Joins {
  implicit val joinsOwner = this
  var count = 0
  val tags: Set[Int] = new BitSet(8)
  def freshTag = synchronized { count += 1; count }
  var joinSet: PartialFunction[Any, Any] = _
  ...
  def test(): Unit = synchronized {
    ...
    if (joinSet.isDefinedAt(true))
      joinSet(false)
  } }
```

The trait defines an implicit field pointing to `this`. It is implicitly passed to the constructors of all events instantiated in the body of a subclass (see above). The `count` field holds an integer indicating the current number of registered events. Events call the `freshTag` method to obtain a *tag* that is unique for all events of `this` owner. The private `joinSet` field holds the current enabled set of joins. As discussed before, joins are represented as partial functions where each of the cases represents a single join.

We have only shown parts of the join synchronization code when discussing the implementation of events. Now, we are going to fill in the missing parts. Consider the invocation of a (non-nullary) asynchronous event, such as `Put(42)` in the buffer example. First, the argument is put into an event-local queue. After that, the `owner Joins` instance is called to check whether this event invocation triggers a join pattern. The `Joins` object does essentially two things. First, it checks whether any of the registered join patterns match, given the new state of the event invocation buffers. Second, it triggers the execution of a join body in the case of a match.

Collecting tags of events participating in a matching join is not trivial. Consider the following example:

```
join { case Grow(x) & Size(n) if n < MAX => ...
       case Put(x) & Get() => ... }
```

Assume we have observed the following event invocations: $\{$`Grow(10)`, `Put(42)`, `Get()`, `Size(MAX)`$\}$ Then, clearly, `Grow(10)` does not belong to a matching join, although it matches individually. On the other hand, the tags of `Put` and `Get` belong to a matching join. To filter out tags that match but do not belong to a matching join, we have to find out when the matcher starts matching a new case (i.e. a new join). When this happens, the tags collected so far are apparently not part of a matching join (matching stops when the first match has been found). For this, we have to consider the order in which `unapply` methods are called. In join patterns the `unapply` methods of all `&` operators are called first, followed by the `unapply` methods of all events that form part of a join. Exploiting this property, we detect the start of a new case when the `unapply` method of `&` is called after the `unapply` method of some event. During the application of the partial function, each event asks the owner whether it should remove an element from its buffer (`owner.takes(tag)`). Since patterns are linear, it suffices to let each event in the matching set remove an element only once from its buffer. The purpose of the `&` operator is to notify the `Joins` object and to forward out-of-band information to the connected events.

3.3 Dynamic Joins

Similar to Russo's library-based design [14], our implementation supports dynamically-sized join patterns. Apart from a radically different implementation, a notable difference concerns addressing of individual parts of a join pattern. In Russo's design, the dynamically-sized portions of a join pattern are packed into arrays, whereas in our case *sequence matching* allows addressing individual components of a join pattern. For ex-

ample, the `JoinMany[Arg]` class below can be used to create a compound event that is composed of n asynchronous events of argument type `Arg`[8].

```
class JoinMany[Arg](evts: AsyncEvent[Arg]*) {
  def unapplySeq(scrut: Any): Option[Seq[Arg]] = {
    val matched = evts.map(x => x.unapply(scrut))
    if (matched.exists(x => x.isEmpty)) None
    else Some(matched map { case Some(arg) => arg })
  } }
```

A `JoinMany` instance is created by passing a variable number of events to its constructor, indicated by the star following the argument type. Unlike primitive events, a `JoinMany` compound event defines an `unapplySeq` method which allows to match a sequence of arguments in a pattern match. In the following join pattern, the arguments passed to the `Put1` and `Put2` events are accessible *individually* as `a` and `b`, respectively.

```
val Put1 = new AsyncEvent[Int]; val Put2 = new AsyncEvent[Int]
val ManyPuts = new JoinMany(Put1, Put2)
join { case ManyPuts(a, b) & GetSum() =>
         GetSum reply a+b }
```

4 Joins for Actors

In the previous section, we described the Scala Joins library for a traditional multi-threaded concurrency framework. We will now describe an integration of this Joins library with Scala's actor library [9].

4.1 Concurrent Programming with Scala Actors

Scala's actor library is largely inspired by Erlang's model of concurrent processes communicating by message-passing [1]. New actors are defined as classes inheriting the `Actor` trait. The actor's life cycle is described by its `act` method. The following code snippet shows a counter actor that encapsulates an integer value which can be incremented and read by other processes in a thread-safe manner.

```
class Counter extends Actor {
  override def act() { loop(0) }
  def loop(value: Int) {
    receive {
      case Incr() => loop(value + 1)
      case Value() => reply(value); loop(value) }
  } }
```

[8] The `isEmpty` method of the `Option` type returns `true` when invoked on `None`, and `false` otherwise.

The `receive` method allows an actor to selectively wait for certain messages to arrive in its mailbox (messages can be any objects). It is defined by means of a partial function, in the same vein as the `join` construct described previously. The following code illustrates a typical interaction with the above counter actor:

```
val counter = new Counter; counter.start()
counter ! Incr()
println(counter !? Value())
```

Messages may be sent to an actor by means of the `actor ! msg` syntax. Synchronous message sends are also supported by the `!?` syntax, which make the sending process wait for the actor to reply to the message (by means of the `reply(val)` method). Scala actors also offer more advanced synchronization patterns such as futures [10,19]. `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.

Finally, next to the `receive` construct, the Scala actors library also provides the `react` construct. Unlike `receive`, `react` allows an actor to wait for an incoming message *without* suspending its underlying worker thread. In other words: `react` allows the creation of purely event-based actors, which have the advantage of being extremely lightweight, as they require fewer worker threads to execute [8].

4.2 Example: Unbounded Buffer Revisited

In order to illustrate the added expressive power that Joins can bring to actors, let us reconsider the buffer example, but this time in the context of Scala's actor library. In what follows, we describe two implementation techniques to implement a buffer in an actor-oriented way, without Joins. We describe the limitations of these implementation techniques and subsequently introduce an adaptation of the Scala Joins library for actors.

Using Guards The following example shows how to implement a buffer actor by means of an auxiliary list data structure, encapsulated within the actor:

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

In this 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.

Using Nested Message Reception The following example shows how to implement a buffer actor by using the actor's mailbox itself as the buffer.

```
class Buffer extends Actor {
  override def act() { loop() }
  def loop() {
    react { case Put(x) =>
      react { case Get() => reply(x); loop() } }
  } }
```

The above implementation makes use of the fact that when a message is received that does not match any case, it is kept in the actor's mailbox. The actor declares that it initially only accepts `Put` messages. Once it has accepted at least one `Put` message, it specifies–by means of a nested `react` block–that it can now also accept a `Get` message. Unacceptable `Get` or `Put` messages queue up in the actor's mailbox. This approach of synchronizing messages based on a set of acceptable messages is reminiscent of the Act++ language's behavior sets abstraction [11].

The advantage of using nested `react` blocks over the use of guards as shown previously is that the relation between `Put` and `Get` is more explicit. However, this approach of nesting does not scale in the context of multiple joins and especially in the context of joins consisting of more than two messages. In essence, every call to `react` represents a state in a finite state machine that encodes the join pattern's availability. However, the number of states quickly explodes as the number of joins and join patterns increases.

4.3 Applying Actor-based Joins

The following example illustrates the use of Joins in Scala's actors library.

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

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

- The `Buffer` class uses the `JoinActor` trait 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 overrides `act` and awaits incoming messages by means of `react`. Note that it is still possible for the actor to serve regular messages within the `react` block. In fact, regular actor messages can be regarded as unary join patterns.

The following example illustrates how the buffer actor is 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 `!?`)[9].

```
val buffer = new Buffer; buffer.start()
val prod = actor { buffer ! Put(42) }
val cons = actor { (buffer !? Get()) match { case x:Int => /* process x */ } }
```

By applying joins to actors, the synchronization dependencies between `Get` and `Put` can be specified declaratively by the buffer actor. The actor will receive `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 synchronization between sender and receiver can only be derived implicitly from the code. This has one important drawback: if the sender actor sends a join message synchronously and the receiver actor forgets to reply to that message, the sender is blocked indefinitely. Even though we consider this to be a limitation of our current design, this problem could already arise without the introduction of join patterns. Scala actors always have to explicitly reply to a message, so there is always the possibility that the reply is erroneously forgotten. The chords design of Cω avoids these issues by restricting join patterns to include at most one synchronous method invocation. That way, the return value of the body can be used automatically to "reply to" the synchronous invocation [2].

The advantage of making join messages synchronization agnostic is that they can be used in arbitrary synchronization modes, including more advanced synchronization modes such as ABCL's future-type message sending [19] or Salsa's token-passing continuations [17]. Every join message instance has an associated *reply destination*, which is an output channel on which processes may listen for possible 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

[9] Note that the `Get` message has return type `Any`. The type of the argument values is often recovered by pattern matching on the result, as shown in the example.

was sent synchronously, the reply awakes the sender. If it was sent using a future-type message send, the reply resolves the future.

5 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` and `react` are methods that take a `PartialFunction` as a sole argument, similar to the `join` method defined previously. To make `receive` and `react` aware of join patterns, the `JoinActor` trait 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. Message sends such as `a!msg` are interpreted as calls to a's `send` method.

```
trait 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]) {
    enqueueReplyDestination(msg, replyTo)
    super.send(msg, replyTo) }
  def enqueueReplyDestination(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 `checkJoinMessage` 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, `checkJoinMessage` 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(checkJoinMessage(msg, true))
  override def apply(msg: Any) =
    f(checkJoinMessage(msg, false)) }
```

Recall from figure 1 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`).

6 Discussion and Related Work

Benton et al. [2] 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 does not handle general guards, although they are permitted in Scala's pattern syntax [13]. We find that guards often help at the interface to a component that uses private messages to represent values satisfying a guard, as in the following example:

```
join { case Put(x) if (x > 0) => this.PositivePut(x)
       case PositivePut(x) & Get() => Get reply x }
```

Cω [2] 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 dynamic joins and multiple synchronous events. Russo's `Joins` library [14] 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 `Joins`' delegates where the order in which arguments are passed is merely conventional. Scala's implicit (constructor) arguments also help to alleviate some of the initialization boilerplate. CCR [3] 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 [15] shows how a small set of higher-order combinators based on Haskell's software transactional memory (STM) can encode expressive join patterns. Salsa [17] is a language extension of Java supporting actors. In Salsa, actors may synchronize in an event-driven way upon the arrival of multiple messages by means of a *join continuation*. However, join continuations only allow an actor to synchronize on gathering replies to previously sent messages. Using joins, Scala actors may synchronize on any incoming message.

7 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 for free. 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 dynamic joins, Ada-style rendezvous, and constraints. Furthermore, we integrated our library into the Scala Actors event-based concurrency framework without changing the syntax and semantics of existing programs.

References

1. Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang, Second Edition*. Prentice-Hall, 1996.
2. Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for C#. *ACM Trans. Program. Lang. Syst*, 26(5):769–804, 2004.
3. Georgio Chrysanthakopoulos and Satnam Singh. An asynchronous messaging library for C#. In *Proc. SCOOL Workshop, OOPSLA*, 2005.
4. Burak Emir, Martin Odersky, and John Williams. Matching Objects with Patterns. LAMP-Report 2006-006, EPFL, Lausanne, Switzerland, December 2006.
5. 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.
6. Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proc. POPL*, pages 372–385. ACM, January 1996.
7. Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A Calculus of Mobile Agents. In *CONCUR*, pages 406–421. Springer-Verlag, August 1996.
8. Philipp Haller and Martin Odersky. Event-based programming without inversion of control. In *Proc. JMLC 2006*, volume 4228 of *LNCS*, pages 4–22. Springer, 2006.
9. Philipp Haller and Martin Odersky. Actors that Unify Threads and Events. In *International Conference on Coordination Models and Languages*, LNCS, 2007.
10. Robert H. Halstead, Jr. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985.
11. Dennis Kafura. Act++: building a concurrent C++ with actors. *Journal of Object-Oriented Programming*, 3(1):25–37, 1990.
12. Martin Odersky. Functional Nets. In *European Symposium on Programming 2000*, Lecture Notes in Computer Science. Springer Verlag, 2000. Invited paper.
13. Martin Odersky and al. An Overview of the Scala Programming Language. Technical Report IC/2004/64, EPFL, Lausanne, Switzerland, 2004.
14. Claudio V. Russo. The Joins concurrency library. In *PADL*, pages 260–274, 2007.
15. Satnam Singh. Higher-order combinators for join patterns using STM. In *Proc. TRANSACT Workshop, OOPSLA*, 2006.
16. Don Syme, Gregory Neverov, and James Margetson. Extensible Pattern Matching via a Lightweight Language Extension. In *Proc. ICFP*, 2007.
17. Carlos Varela and Gul Agha. Programming dynamically reconfigurable open systems with SALSA. *ACM SIGPLAN Notices*, 36(12):20–34, 2001.
18. G.S. von Itzstein and David Kearney. Join Java: An alternative concurrency semantic for Java. Technical Report ACRC-01-001, University of South Australia, 2001.
19. Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In *Proc. OOPSLA*, pages 258–268, 1986.