JEScala: Modular Coordination with Declarative Events and Joins

Jurgen M. Van Ham[1,2] Guido Salvaneschi[1] Mira Mezini[1] Jacques Noyé[2]

[1] Software Technology Group, Technische Universität Darmstadt, Germany

[2] ASCOLA Team, Mines Nantes & Inria & LINA, Nantes, France

{vanham, salvaneschi, mezini}@informatik.tu-darmstadt.de, Jacques.Noye@mines-nantes.fr

Abstract

Advanced concurrency abstractions overcome the drawbacks of low-level techniques such as locks and monitors, freeing programmers that implement concurrent applications from the burden of concentrating on low-level details. However, with current approaches the coordination logic involved in complex coordination schemas is fragmented into several pieces including join patterns, data emissions triggered in different places of the application, and the application logic that implicitly creates dependencies among communication channels, hence indirectly among join patterns. We present JEScala, a language that captures coordination schemas in a more expressive and modular way by leveraging a seamless integration of an advanced event system with join abstractions. We validate our approach with case studies and provide a first performance assessment.

Categories and Subject Descriptors D.1.3 [*Software*]: Programming Techniques—Concurrent Programming; D.1.5 [*Software*]: Programming Techniques—Object-oriented Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages

Keywords Scala; Event-driven Programming; Concurrency; Join Patterns

1. Introduction

Concurrency is required in a wide class of applications. However, writing correct concurrent software is hard. Conceptually, programmers are interested in the *coordination schemas* that an application must implement. For example, they want a *producer-consumer* model to correctly order data processing or *finite state machines* to regulate the progress of the program. Unfortunately, many synchronization primitives – like semaphores or monitors – are low-level and force programmers to focus on details – leading to error-prone code and reducing maintainability. For this reason, researchers have proposed language constructs that rise the level of abstraction and support high-level reasoning about concurrency. This includes Actors [18], Futures [1] and Join Patterns (Joins for short) [13].

MODULARITY '14, April 22–26, 2014, Lugano, Switzerland.
Copyright © 2014 ACM 978-1-4503-2772-5/14/04…$15.00.
http://dx.doi.org/10.1145/2577080.2577082

The Join Calculus [13] introduced join patterns and disjunctions thereof as key concepts for expressing the interaction among a set of processes that communicate by emitting data over communication channels. Since then these concepts have gained special attention because they combine abstraction and practicality – they are abstract enough to overcome the limitations of low-level constructs but still applicable in a wide variety of scenarios. As a result, several languages have been proposed to directly support joins (referred in the rest as *join languages*), including JoCaml [9], Join-Java [19], Polyphonic C# [2], the Join concurrency Library [33], Scala Joins [17] and JErlang [29].

In this paper, we argue that current join languages fall short with respect to capturing the whole logic of a coordination schema in a modular way due to the way they model channels and emissions. The latter are modeled either by method/function calls [2, 9, 19, 23, 26] or by explicitly triggered events [17]. In either case, the emission points need to be explicitly hardwired in the application code; in the case of method/function channels, the destination is hardwired too. As a result, the coordination logic gets fragmented into several pieces including the join patterns, different places in the application where data emission is triggered, and the application logic that implicitly creates dependencies among emissions, thus indirectly among joins. To infer the logic of a coordination schema, programmers have to *connect the dots* following the flow of the application and harvest how data emissions are related and interact. This hampers program understanding and makes programs harder to maintain and extend; since the coordination logic is not modularized, changes to the coordination schemas cannot benefit from local reasoning.

The design of the JEScala language proposed in this paper approaches this problem by exploring the synergy of Join Calculus concepts with the advanced event system of EScala [16]. The latter supports three kinds of events. In addition to explicitly triggered events, there are implicitly triggered events, referable points in the execution of the program similar to join points in aspect-oriented programming, as well as composite events that are declaratively defined by expressions correlating other events. The use of implicit events and composite events enables the modular definition of data emission sources together with the synchronization logic, capturing complex coordination schemas that would be otherwise scattered in the code base. In summary, the paper makes the following contributions:

- We motivate the need for abstractions to overcome deficiencies of current join languages regarding scattering of coordination logic.
- We present the design of JEScala, a language that exploits a seamless integration of an advanced event system with joins

processes	
$P ::= \mathtt{0}$	empty process
$\quad c(\tilde{d})$	emission of $\tilde{d}$ on c
$\quad P_1 \& P_2$	parallel composition
$\quad \mathtt{def}\ D\ \mathtt{in}\ P$	definition
definitions	
$D ::= J \triangleright P$	reaction
$\quad D_1 \vert D_2$	disjunction
join patterns	
$J ::= c(\tilde{d})$	reception of $\tilde{d}$ on c
$\quad J_1 \& J_2$	synchronization

Figure 1. The basic constituents of the Join Calculus.

to capture coordination schemas in an expressive and modular way.

- We validate our approach with case studies that show the validity of our design and we provide a first performance assessment. The implementation of the language, the examples, and validation code are available in [20].

The paper is organized as follows. Section 2 motivates the work. Section 3 presents the design of JEScala and Section 4 key elements of its implementation. Section 5 validates the approach. Section 6 discusses related work. Section 7 summarizes the paper and outlines areas of future work.

2. Motivation

2.1 Basic Concepts of Join Languages

The Join Calculus The foundation for join languages is laid down by the Join Calculus [13, 21]. In its basic form [21] (see Figure 1), the calculus includes three kinds of constructs: *processes*, *definitions* and *join patterns*. Processes are the primary structuring entities. They communicate through asynchronous emissions of tuples of data across *channels*. In addition to emissions, processes are built from definitions and parallel compositions of other processes. An elementary definition, called *reaction*, associates a new process to a *join pattern*. A join pattern can be elementary or composite. An elementary pattern is simply a reception of a tuple, whereas a composite pattern synchronizes several receptions from different channels. A join pattern is *active* if there is data present on all referenced channels. In its general form, a definition is a *disjunction* of *reactions*, where a reaction associates a new process to a join pattern. A disjunction defines competing reactions. Here, the same channel may appear in different reactions. When several reactions are active, one of the reactions is chosen and the corresponding process spawned.

This basic calculus can be extended with, in particular, sequential composition of instructions and synchronous communications, using an implicit continuation channel to wait for a reply, getting back to the initial calculus of [13]. A fundamental property of the calculus, which makes it a practical foundation for concurrent and distributed programming languages, is that interprocess communication does not require global, distributed synchronization. Also, the join operator `&` turns out to be very expressive, combining atomically interprocess communication and synchronization.

Join Languages Channels can be implemented in various ways. In a functional setting, e.g., in JoCaml [9, 23] or Funnel [26], an elementary channel definition looks like a function definition and an emission like a function call. In an object-oriented setting, e.g., in Polyphonic C# [2] or Join Java [19], functions are replaced by methods. Reactions can then be seen as defining several functions or methods at once with a single shared body (a process in the calculus). In Scala Joins [17], method headers are replaced by *events*, objects of type `Event` that can then be used to build join patterns, associated to a body via a case clause. A case clause plays the same role as a multi-header method in Polyphonic C#. As in Polyphonic C#, an emission looks like a method invocation.

Disjunctions can take various forms. For instance, in JoCaml, Funnel and Scala Joins disjunctions are explicitly defined. In Polyphonic C#, disjunctions are implicit: If several reactions are defined in a class, they form a disjunction.

Concurrency In the following, we will focus on object-oriented languages and consider threads as the underlying support of concurrency, a choice shared by most object-oriented join languages. The concepts of the join calculus have also been combined with actors in JErlang [29], not an OO language, and Scala Joins (in Scala, threads and actors coexist).

In such join languages, writing concurrent programs does not require direct manipulation of threads any longer. Threads are implicitly created through asynchronous data emission or the use of parallel composition.

Depending on the language, all data emissions not returning any result are handled asynchronously, or the emissions have to be explicitly declared as asynchronous. This is in general done on the receiver's side. For instance, in Polyphonic C#, method headers can be qualified with the keyword `async`. In Scala Joins, this is a matter of defining an event as an instance of `AsyncEvent` (a subclass of `Event`) rather than `SyncEvent`. Whereas asynchronicity is useful for creating concurrency, synchronicity is useful when a result has to be returned. It is also useful for synchronization purposes: a thread can be blocked on data emission, waiting for a join pattern to be selected or, in other words, waiting for the complementary data receptions. As a result, join patterns can then be used both to synchronize threads and communicate between threads.

2.2 Limitation of Join Languages by Example

In this section, we discuss deficiencies in the design of applications that use existing join languages by means of a case study. In order to ease comparisons, we use a hypothetical language as a typical join language. We call it Polyphonic Scala as it relies on the syntax and semantics of Scala while implementing joins à la Polyphonic C#.

Case Study Our case study is a Web server that hosts two applications, an online booking application for flight tickets `OB` and a marketplace application `MP`[1] (Figure 2a). In both applications, client requests are managed by `handle` methods (Lines 3 and 10). Since the Web server shares the host with other services, we want to ensure that all services are responsive in the presence of a high number of requests. To do so, when the load is high, we limit the number of concurrently handled requests by controlling the execution frequency of `handle` methods[2]. This requires a coordination schema among the threads executing the components of Figure 2a.

Specifically, under high load, clients need to consume a token to be admitted into the server. When the load is high, client threads should block at the boundary of the `handle` method of each application, waiting for a token to be produced by the Token Generator (Line 16 in Figure 2a).

[1] In Scala, the keyword `object` introduces singleton classes.

[2] As in real Web servers, we assume that client connections not timely routed to applications are dropped after a timeout by the Web container.

```
 1 object CL { // Coordination Logic
 2   def OB_beforeHandle() { mayBlock() }
 3   def MP_beforeHandle() { mayBlock() }
 4   def mayBlock() {
 5     if (systemLoadHigh())
 6       RL.block()
 7   }
 8   def unblock() { RL.unblock() }
 9 }
10
11 object RL { // Rate Limiter as 2-state Free-Busy FSM
12   def block():Unit & free():async {
13     // Stat.toBusy()  // future extension
14     busy()
15   }
16   def unblock():async & busy():async {
17     // Stat.toFree() // future extension
18     free() // busy to free
19   }
20   def unblock():async & free():async {
21     // Stat.toFree() //future extension
22     free() // absorbed unblock
23   }
24   free() // initial state in constructor
25 }
```

(a)

```
 1 object CL { // Coordination Logic
 2   evt block = (OB.beforeSync(handle) || MP.beforeSync(handle)) &&
 3               systemLoadHigh()
 4   evt unblock = TG.beforeAsync(createToken)
 5 }
 6
 7 object RL { // Rate Limiter as 2-state Free-Busy FSM
 8   imperative async evt free[Unit], busy[Unit]
 9   evt (toBusy, freed, absorbed) =
10     ( CL.block  & free
11     | CL.unblock & busy
12     | CL.unblock & free )
13   evt toFree = freed || absorbed
14   toBusy += ((arg:Any)=>busy())
15   toFree += ((arg:Any)=>free())
16   free() // initial state
17 }
```

(b)

Figure 4. Coordination logic with Polyphonic Scala (a) and JEScala (b).

```
 1 // OnlineBooking App
 2 object OB { //
 3   def handle(...) = {
 4     ...
 5   }
 6 }
 7
 8 // MarketPlace App
 9 object MP {
10   def handle(...) = {
11     ...
12    }
13 }
14
15 // Token Generator
16 object TG extends Thread {
17   def createToken() = {
18     ...
19   }
20   override def run = {
21     while(true) {
22       sleep(1000)
23       createToken()
24     }
25   }
26 }
```

(a)

```
 1 // OnlineBooking App
 2 object OB {
 3   def handle(...) = {
 4     CL.OB_beforeHandle()
 5     ...
 6   }
 7 }
 8 // MarketPlace App
 9 object MP {
10   def handle(...) = {
11     CL.MP_beforeHandle()
12     ...
13   }
14 }
15 // Token Generator
16 object TG extends Thread{
17   def createToken() = {
18     CL.unblock()
19     ...
20   }
21  ...
22 }
```

(b)

Figure 2. Web server: basic components (a) and instrumented components (b).

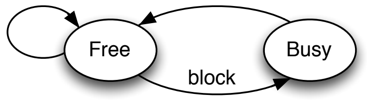


Figure 3. Rate limitation as a state machine

Instrumentation In order to be controlled by the coordination logic (implemented by the object CL in Figure 4a), the basic components of Figure 2a are instrumented as shown in Figure 2b.

The beforeHandle methods are called (Lines 4, 11) in the bodies of the handle methods to notify the coordination logic of the arrival of a new request to *one of* the applications. By calling a dedicated method of the coordination logic for each application, the coordination logic can be changed without requiring further changes to the web applications.

Similarly, the availability of new tokens is notified to the coordination logic by calling the unblock method each time a new token is about to be generated (Line 18).

Coordination Logic The top level of the coordination logic is implemented by the object CL (Lines 1–9 of Figure 4a), which turns the notifications from the basic components into calls to the object RL implementing rate limitation (Lines 11–25). Essentially, the calls from the web applications return immediately when the load is low, allowing the requests to be handled. They are turned into calls to the block method of the rate limiter otherwise. The calls to unblock are just forwarded to the rate limiter.

The role of the rate limiter is to delay returning from invocations to block until a token is available, that is, until an invocation to unblock occurs. We can represent the interaction between block and unblock calls as a state machine (Figure 3). When the rate limiter is in state Free (there is no application waiting for handling a request), calls to unblock are simply ignored, while calls to block switch the state to Busy (there is an application waiting). Then, a call to unblock brings back the rate limiter to state Free.

A common technique in join languages (see, for instance, [2, 17, 23]) represents states as pending data receptions. We apply it, with synchronous and asynchronous method calls, to the aforementioned state machine (see Figure 4a). A simple invariant underlies the technique: in the object implementing the state machine, there is always one single pending asynchronous method call. This pending call represents the current state. We refer to the corresponding methods (free and busy in our example) as *state methods*. The actions responsible for state transitions can be implemented by either synchronous or asynchronous method calls. We refer to these methods (`block` and `unblock`) as *action methods*. Each transition is implemented as a binary pattern between a state method and an action method (Lines 12, 16 and 20). The body executed when matching this pattern calls a state method. This maintains the invariant. The invariant is established when initializing the object by calling the state method for the initial state (`free`, Line 24). Note that a single pending call of a state method implies that the machine can handle only a single action at a time, the next action can only take place after the body has called a new state method.

Whereas the `unblock` method headers (Lines 16 and 20) are also declared as asynchronous in order not to block the token generator, calls to `block` are synchronous. The join pattern on Line 12 blocks the underlying thread until the state is free.

Discussion Even if the coordination schema is simple, its realization in Figure 4a has limitations.

First, the components to be coordinated are intrusively modified to add the notifications (Lines 4, 11, and 18 of Figure 2b) necessary to make observable the points in the execution that are relevant to the coordination schema. Further modifications may have to be considered, both in the application components and the coordination logic for further extensions. For instance, we will consider in Section 3.4 adding a component `Stat`, which will require the modifications commented out on Lines 13, 17 and 21.

Second, several indirections are needed: one to deal with requests to *any of* the applications (Lines 2 and 3 of Figure 4a) and another to take the load condition into account (Line 5). In the first case, the indirection implements a “union” semantics, which is not explicit in the code. Alternatively, the indirection can be suppressed by directly calling `mayBlock` within the applications. Yet, the cure is worse than the disease. In Figure 2b, the applications call distinct `beforeHandle` methods and the union of these calls is properly modularized within the coordination schema. Calling `mayBlock` directly from the applications would move part of the coordination logic (a call from *either* `OB` *or* `MP`) away from the coordination schema, and hardcode it into the coordinated components. As a result, it would be, for instance, impossible to implement a balancing strategy between `OB` and `MP` just by modifying the coordination code.

In summary, with only the abstractions from the join calculus, the logic of the coordination schema is hardly captured by the abstractions of the language. Instead, it must be harvested from the calls inserted within the coordinated components in order to capture the events of interest and the logic of corresponding (possibly multi-header) bodies in the coordination code. In addition, the application is not extensible and requires invasive changes to introduce new components.

3. A Rich Event System to the Rescue

3.1 On Events, Implicit Invocation and Joins

The requirements for both being able to capture and combine program execution points is, of course, highly reminiscent of the join points and pointcuts of Aspect-Oriented Programming (AOP). AOP would indeed make it possible to capture in a pointcut, in a non-intrusive way, the fact that a request is about to take place. It also makes it possible to combine such pointcuts in order to deal with several kinds of requests under a specific condition, namely the fact that the load is high. Unsurprisingly, assuming the availability of AOP facilities within Polyphonic Scala would improve the implementation of our case study.

Still, some discrepancy would remain between the composition of join points via pointcuts using logical operators and the composition of channel endpoints via the join operator. Our previous work on ECaesarJ [25] and EScala [16], which did not provide any support for concurrency, suggests a way to eliminate this discrepancy. The main idea behind this previous work is that the join points of AOP and the explicit triggering of events encountered in event-driven programming are of the same nature. A join point can be seen as an occurrence of an *implicit event* whereas an *explicit event* is explicitly triggered. Composite events can then be created by composing events through logical operators. The basic idea to solve our discrepancy issue is therefore twofold:

- Let us consider a data emission as triggering an explicit event.
- Let us consider the join operator as an additional composition operator.

Of course, this is not enough to get (implicit) concurrency: we also need a way to choose between synchronous and asynchronous event triggering, i.e. between sequentially proceeding with event handling or forking a new thread to deal with it.

All these ideas have been injected into EScala, leading to JEScala, described in detail in the next section. Using JEScala, our case study can be rewritten as shown in Figure 4b without the limitations previously discussed.

But, before presenting JEScala, let us first clarify what we mean by *event*. Indeed, this term can be confusing as, beyond the general idea that an event refers to a noteworthy state change, the semantics of event constructs varies a lot. In particular, the events of Scala Joins and the explicit events of (J)EScala share some common characteristics: they are instance members and triggered using method-call syntax. In both cases, events make it possible to split the traditional way methods are defined: the *event* itself corresponds to a method header and its *event handler* to a method body. In Scala Joins, these two parts have to be defined in the same class in order to be bound. This is quite different in (J)EScala where they can be defined in different classes and bound to each other in yet another class. There may also be different bindings, hence different handlers attached to the same event, resulting in some form of implicit invocation [15]. Hence, unlike languages that model channels with functions/methods, data is not sent to a single destination but to multiple destinations, i.e., triggering an event corresponds to an emission on multiple channels.

Implicit invocation is central to our event model. It exchanges the rigid connection between a sender and its single receiver for a flexible broadcast mechanism. Without modifying the sender, the group of receivers can be changed. Implicit invocation makes no assumption on the number of receivers. Therefore, it does not require to create dependencies that are not actually needed (e.g. a tracer can be detached). Receivers can be added and removed at runtime.

3.2 Basic Concepts of JEScala

JEScala combines asynchronous channels of join languages with implicit invocation of event systems. In addition to explicitly triggered events, JEScala supports EScala’s implicit and composite events. However, unlike EScala’s events, which are triggered only synchronously, in JEScala events are subject to both synchronous or asynchronous execution.

3.2.1 Asynchronous Events with Implicit Invocation

In the following, we present the event system of JEScala. For reference, the syntax of JEScala that is relevant for the discussion is given in Figure 5. JEScala inherits the event system of EScala, while extending it with asynchronous execution semantics. Yet, the extension is designed to ensure backward compatibility with EScala.

Imperative events In JEScala, the declaration of each explicitly triggered event begins with the keyword `imperative`, followed by an optional `sync` or `async` modifier. Hence, the programmer specifies whether an imperative event is handled synchronously or asynchronously at the declaration point of the event. Synchronicity is part of the interface of the entity declaring the event, documenting the fact that handlers may block or not the execution of the code that triggers the event. This does not, however, prevent a handler attached to a synchronous event to spawn a new computation via, for instance, the use of a Scala future or by triggering an asynchronous event.

For compatibility with EScala, if the synchronicity of events is not explicitly specified, they are assumed to be synchronous. As a result, each EScala program is a valid JEScala program with the same semantics.

Implicit events Implicit events make method executions observable a.k.a. join points in AOP parlance. EScala provides the implicit events `before(`*method-name*`)` and `after(`*method-name*`)` that are executed when a method enters – respectively, finishes – its execution[3]. We extended EScala's implicit events to account for synchronicity. In JEScala four implicit events are available: `beforeSync`, `beforeAsync`, `afterSync` and `afterAsync`. They all take a method name as argument. In the spirit of remaining compatible with EScala, the events `before(`*method-name*`)` and `after(`*method-name*`)` are still valid and mapped to the corresponding synchronous events.

Declarative events Like EScala, JEScala also supports declarative events defined in terms of event expressions. They are defined by composing other events through operators, like in e_1`||`e_2 (occurrence of one among e_1 or e_2), e_1`&&`p (e_1 occurs and the predicate p is satisfied[4]), e_1`.map(`f`)` (the event obtained by applying f to e_1). Declarative events have no synchronicity by themselves. Instead, they inherit the synchronicity of the event that triggers them. For example, the event e_1`&&`p is executed synchronously – respectively asynchronously – if e_1 is synchronous – respectively asynchronous. Note that there is no ambiguity, since the `||` composite event is triggered by one event only and inherits the synchronicity of that event.

This design choice requires more explanation. Providing a `sync`/`async` modifier for declarative events would make it possible to express event combinations like:

```
imperative async evt e1
sync evt e2 = e1 && (predicate)
```

whose semantics is unclear. The imperative event `e1` is asynchronous, so one expects that all the handlers bound (even indirectly) to the event are executed asynchronously. However `e2`, which depends on `e1`, is declared synchronous. This should imply that handlers attached to `e2` are executed immediately, which contradicts the modifier of `e1`. For this reason we decided to avoid explicit synchronicity for declarative events and let them inherit the synchronicity of the event that triggers them.

It still makes sense to force a declarative event to be asynchronous. To this aim, the prefix `!!` operator converts a possibly synchronous event expression into an asynchronous one.

Abstract events In EScala, it is possible to declare *abstract* events that are defined in concrete subclasses. In JEScala, the synchronicity of abstract events is not specified.

This design decision is motivated by the fact that an abstract event can be overridden by either a primitive or a composite event. Allowing synchronicity modifiers in abstract events would, for instance, allow one to define an abstract `sync` event and override it in a subclass with a declarative event – running into the trouble previously described.

[3] Unlike most AOP languages, only methods declared in the interface of a class as such are observable outside the class.

[4] This filter operator is overloaded with the logical and operator. This is questionable and may change in future versions.

```
event-decl ::= prim-event | decl-event
prim-event ::= imperative [sync-modifier] evt event-name
decl-event ::= evt event-name = event-express
     | evt (event-name { , event-name}+) =
        (join-express { | join-express}+)
event-express ::= [ obj-ref . ] event-name
     | event-prefix-operator event-express
     | event-express event-infix-operator event-express
     | event-express fun-operator fun
     | implicit-event
implicit-event ::= [ obj-ref . ] implicit-selector ( method-name )
implicit-selector ::= beforeSync | afterSync
     | beforeAsync | afterAsync
     | before | after
sync-modifier ::= sync | async

event-prefix-operator ::= !!
event-infix-operator ::= || | & | ...
fun-operator ::= map | && | ...
join-express ::= ( event-name { & event-name }+)
```

Figure 5. The syntax of JEScala.

Actually, the synchronicity of an abstract event cannot be known until it has been defined. Defined as a primitive event, its synchronicity is known statically. Defined as a composite event, its synchronicity may not be known until runtime, depending on its definition and its evaluation.

3.2.2 Joins on Implicit and Declarative Events

As already mentioned, the key feature of JEScala is the combination of the rich event system described above with join concepts. This combination enables a succinct and well-localized definition of synchronization logic. For illustration, Figure 4b shows the implementation of the Web server example by using joins with implicit and declarative events.

The execution of `handle` in `OB` and `MP` is captured by the implicit events `OB.beforeSync(handle)` and `MP.beforeSync(handle)`, which are composed so that the declarative event `block` (Line 2) only fires when the load is high. The implicit event `TG.beforeAsync(createToken)` that captures the creation of a token is aliased to `unblock` (Line 4).

Finally, the state machine from Figure 3 is implemented as described in Section 2.2, except that state and action methods are replaced by *state* and *action events*: `free` and `busy` (declared Line 8), and `block` and `unblock` (defined in `CL` Lines 2 and 4). The implicit disjunction of Figure 4a is replaced by an explicit one (Line 9). Each alternative combines a state and an action event and triggers one of the events `toBusy`, `freed` or `absorbed` when it matches. These new events are necessary to attach a handler to each alternative and have the advantage that sharing can be made explicit, here by defining the event `toFree` that signals a transition to the `Free` state. Triggering the `free` event (Line 16) sets the initial state of the machine.

The implementation in Figure 4b has several design advantages compared to that in Figure 4a. The coordination logic is captured in one place (lines 2 – 17). There is no footprint of the coordination logic in the components to coordinate; all the execution points relevant to coordination are captured by implicit events. As a result, the coordination logic is properly modularized. and expressed declaratively, improving clarity and extensibility, e.g., the balancing strategy is clearly captured at Line 2 thanks to event expressions. Moreover, introducing an additional application in the coordination schema is e.g., as straightforward as adding a new im-

```
1  object CL {
2    evt blockOB = OB.beforeSync(handle)
3    evt blockMP = MP.beforeSync(handle)
4    evt unblock = TG.beforeAsync(createToken) map (()=>currentTime)
5  }
6  object RL {
7    imperative sync evt requestUnblockOB[Unit], requestUnblockMP[Unit]
8    imperative sync evt innerBlockOB[Unit], innerBlockMP[Unit]
9
10   CL.blockOB += ()=>{ requestUnblockOB(); innerBlockOB() }
11   CL.blockMP += ()=>{ requestUnblockMP(); innerBlockMP() }
12
13   evt _ = innerBlockOB & grantUnblockOB
14   evt _ = innerBlockMP & grantUnblockMP
15
16   evt (mayUnblockOB, mayUnblockMP) =
17      (requestUnblockOB & CL.unblock) |
18      (requestUnblockMP & CL.unblock)
19
20   evt grantUnblockOB = mayUnblockOB   && ((ts)=> !isExpired(ts))
21   evt expiredUnblockOB = mayUnblockOB && ((ts)=> isExpired(ts))
22   expiredUnblockOB += ()=>requestUnblockOB()
23
24   evt grantUnblockMP = mayUnblockMP   && ((ts)=> !isExpired(ts))
25   evt expiredUnblockMP = mayUnblockMP && ((ts)=> isExpired(ts))
26   expiredUnblockMP += ()=>requestUnblockMP()
27 }
```

Figure 6. Distributing load among Web applications in JEScala.

plicit event in Line 2. Given that events are values, the coordination schema can be a separate reusable component parameterized by events to coordinate. For the sake of simplicity, we have used singleton classes in our example. A more realistic implementation of the rate limiter would use a class with a primary constructor taking a `block` and an `unblock` event as parameters.

3.3 Advanced Use of Disjunctions

To introduce more abstractions of JEScala we extend the Web server example. So far, `OB` and `MP` clients have been served indistinctly. As a result, a high request rate in one application can significantly slow down the other. To address this issue, we shall introduce load distribution between `OB` and `MP`. Furthermore, we shall improve token generation to avoid that tokens accumulate when they are generated at a higher frequency than client arrivals – in the new version of the Web server, tokens simply expire after some time. The implementation of the extended Web server is shown in Figure 6. Instead of immediately discarding surplus `unblock` events by a state machine (Figure 3), this extension accumulates `unblock` events, which are consumed without effect after they expire. Since the example is not trivial, we start with a high-level description of the coordination schema. On arrival (exposed by events in Lines 2–3), each client is blocked until unblocking is granted (Lines 13–14). If two requests from different applications are performed, only one is chosen non-deterministically (Lines 16–18). The authorization to proceed is given only by not expired tokens. Token expiration is managed in Lines 20–26.

For a detailed description of the coordination schema of Figure 6, consider the flow of the events associated to `OB` (the event flow for `MP` is similar). When a client request for `OB` arrives, `blockOB` is synchronously triggered (Line 2). Its handler (Line 10) triggers `requestUnblockOB` and is blocked in the disjunction Line 17, waiting for an `unblock` event. The `unblock` events are generated by attaching a timestamp to events produced by the token generator (Line 4). When the disjunction pattern is selected, triggering a `mayUnblockOB` event, the handler proceeds but blocks at once on triggering the `innerBlockOB` event, involved in the disjunction Line 16. Concurrently, depending on whether the token is expired or not, either a `grantUnblockOB` event is triggered (Line 20) releasing the `blockOB` handler waiting at the disjunction Line 13 or a `requestUnblockOB` event is regenerated (Line 22) and the handler remains blocked at the disjunction Line 17. In the first case, the `blockOB` handler returns at once and the application can proceed.

The example in Figure 6 demonstrates several aspects of the design of JEScala.

Disjunctions consume and fire events Like in other join languages, disjunctions can be used to compose multiple conceptually-related join patterns; multiple join patterns in a disjunction can share an event (see e.g., Lines 13–14), consumed by the first matching pattern. If multiple patterns match, the one that fires is chosen non-deterministically. In JEScala, *disjunctions fire events*; it is possible to distinguish the join that fires by associating a specific event to each join in the disjunction (see e.g., Line 16). In other join languages when a join matches, a handler is executed. The model of JEScala is homogeneous: events generated by a disjunction can be freely composed with other events. For example, a union operator `||` can be used to merge the events from the same disjunction, if we do not need to distinguish among them. The same effect can be achieved in other join languages only by triggering the same emission from each handler registered to a join, which unnecessarily resorts to imperative code and bloats the coordination logic.

Multiple disjunctions inside the same class The example in Figure 6 shows another feature of JEScala. Many join languages group all join patterns defined in an object into a single *implicit* disjunction associated to the object. In JEScala, each disjunction *explicitly* defines a set of joins that are checked for a possible match. Therefore, JEScala allows one to define *multiple disjunctions inside the same class*. This fosters modularity. In case of complex coordination schemas, like in Figure 6, several disjunctions are required, with some of them (Lines 13-14) reduced to a single alternative.

In languages with implicit disjunctions, we would need both to create an artificial disjunction with two alternatives composing the two disjunctions with a single alternative (if we want to keep them in the same class) and split the coordination patterns into at least two separate classes in order to also implement the second disjunction (Line 16). In JEScala, disjunctions that logically belong to the same schema can be properly modularized inside the same class.

Supporting multiple disjunctions also affects the interaction between join abstractions and object-oriented inheritance. Consider the case in which the superclass defines a join pattern among events a, b, c, and d. If a developer adds a join on events a and b in a subclass, and the patterns of a class are implicitly correlated by a disjunction the join in the superclass may never trigger. The interaction between inheritance and join operators is subtle and can easily lead to deadlocks, e.g., in case c and d are synchronous events (the interested reader is referred to [2] for a detailed discussion). To tackle these issues, object-oriented join languages impose limitations on inheritance to prevent adding new joins in subclasses. In JoinJava, only final classes can define a join; in Polyphonic C# it is possible to override an inherited disjunction by replacing the associated handler, but it is not possible to add a join. Since in JEScala joins defined within a class are not implicitly correlated into a disjunction, classes can be freely extended regardless of the presence of disjunctions. Yet, advised by the lesson of existing join languages, we forbid breaking existing disjunctions by extending them with new joins. In our design, subclasses can only entirely override them.

3.4 Dynamic Registration of Handlers

While, in existing join languages, handlers are statically bound to join patterns, handlers in JEScala can be dynamically (un)registered. Figure 7 shows an extension of the Web server from Figure 4b that computes statistics about the queuing time of the

```
 1 object Stat {  // Statistics
 2   var sTime:timeStamp=null
 3   def hdlToBusy:Unit={ // handler
 4     sTime=currentTime  }
 5   def hdlToFree:Unit=if (sTime!=null) {
 6     log_busy(currentTime−sTime)
 7     sTime=null  }
 8
 9   var isEnabled: Boolean = false
10   imperative sync evt enable[Unit]  // trigger to enable
11   imperative sync evt disable[Unit] // trigger to disable
12   evt doEnable = enable && ()=>!isEnabled // enable only once
13   evt doDisable = disable && ()=>isEnabled
14   // register an anonymous handler with doEnable event
15   doEnable  += ()=>{
16     isEnable=true
17     RL.toBusy += hdlToBusy _ // register with RL.toBusy event
18     RL.toFree += hdlToFree _  }
19   doDisable += ()=>{
20     RL.toBusy −= hdlToBusy _ // unregister
21     RL.toFree −= hdlToFree _
22     isEnabled = false  }
23 }
```

Figure 7. Dynamic handler registration in JEScala.

applications inside the server. Logging the time spent by the rate limiter (RL) in the busy state requires the observation of entering the states Free and Busy in the rate limiter. We prepared the code in Figure 4a by inserting explicit method calls into the RL component (Lines 13, 17 and 21). In JEScala, we just need to register additional handlers (Line 17) with the exposed events toFree and toBusy of RL without modifying the code of RL (Figure 4b). By using the declarative event toFree we can register each of our handlers with a single event with a descriptive name.

Triggering the enable event registers the handlers toBusy and toFree. Since we are not all the time interested in these statistics, triggering the disable event removes the handlers. The internal declarative events doEnable and doDisable prevent incorrect double registrations. The anonymous handler of doEnable registers the handlers from Stat with the exposed events from RL. Its counterpart for doDisable unregisters them. It also sets the isEnabled flag that is used by the filters defining doEnable and doDisable. An implementation in Polyphonic Scala, without declarative events, would need additional conditions in the methods toBusy and toFree to enable and disable statistics at runtime.

In other join languages, dynamic binding between join patterns and handlers can be obtained only by adding a layer of indirection with an intermediate handler that is responsible for notifying the right handler in case a certain condition is met. This approach has the drawback of moving the event logic from high-level operations among events to handlers. Further, it introduces a performance penalty, because the intermediate handler is *always* notified regardless of whether a reaction is needed. More importantly, this solution does not account for situations where the binding depends on the execution. JEScala solves these issues thanks to the uniform representation of join pattern outputs as events and dynamic event handler registration.

4. Implementation

The implementation of JEScala[5] required us to modify the EScala event system to support joins and asynchronous events. Before describing how these were implemented, we briefly summarize the mechanism behind the EScala event system [16].

[5] In its current version, this implementation is provided as a library, to be completed with compiler support, as was done for EScala, when a stable version of the new Scala compiler (2.11) is available.

EScala Event Propagation System in a Nutshell Internally, EScala events are organized in a graph. The graph is incrementally updated every time an event definition is executed by introducing a node for the newly defined event. Handlers are directly attached to each event node. Edges in the graph model dependencies between events. For example, the event e3 = e1 || e2 creates a new node for e3 that depends on the nodes previously created for e1 and e2. Imperative events and implicit events (i.e. primitive events) are leaves in the graph since they are triggered directly. When a leaf event is triggered, the handlers to execute are collected. This process consists of a depth-first tree traversal of all nodes that are transitively reachable from the firing node, collecting the handlers associated to each visited node. Also, the process takes into account dynamic conditions introduced by event expressions. For example, a filter node can stop the evaluation along a branch if the condition is not satisfied. Finally, the collected handlers are executed in sequence. As an optimization, nodes are only *deployed* (i.e. take part to event triggering) if they have outgoing edges or handlers. Once deployed, they can be undeployed if the condition becomes false. This applies, for instance, to the Stat component of Figure 7.

Adding Asynchronous Event Handling In JEScala, collecting handlers applies to both synchronous and asynchronous primitive events. Differently from EScala, when the triggered event is asynchronous, a new thread is used to execute the collected set of handlers. As a result, the continuation of the code that triggers the event, and the handlers, can run concurrently. This also means that several collections may be active concurrently on the same graph, which requires us to make the process thread-safe. In particular, each new thread has access to its own buffer to collect handlers. Dynamic handler registration also requires that the handlers associated to an event are protected against concurrent accesses.

Handling disjunctions Disjunctions are implemented as sets of queues $q_1 ... q_n$, a queue q_i for each event e_i that appears in a pattern of the disjunction. When an event e_i is fired, it is stored with its arguments in the queue q_i and we check if a pattern can be completed with the new event. If none of the patterns can be completed, e_i remains in q_i. If a single pattern matches, the associated events are removed from the queue and the resulting event is fired. If multiple patterns match, one is chosen non deterministically and the associated events are removed.

Synchronous events that appear in a disjunction require a few additional steps. If the event is fired and none of the patterns applies, not only do we store the arguments, but we also block the thread and store it in the queue. Afterwards, when the pattern matches, one of the stored threads is chosen to execute the handlers of the resulting event (to be deterministic we select the thread of the first synchronous event in the pattern). To achieve a form of fairness, we randomly select among the patterns that can match when an event arrives.

The queues are new nodes in the event graph. By default, when a queue node is encountered during the collection step, queuing is not handled at once but postponed to handler execution by creating a new handler responsible for this queuing task. This is necessary so that synchronous events do not block the collection step.

Optimizations We applied a number of optimizations to increase the efficiency of asynchronous events. *Thread pool* (1) Asynchronous events need a thread to execute the handlers concurrently with the thread that triggers the event. Instead of creating a new thread every time, we recycle the thread from a thread pool. In case a number of handlers of asynchronous events are executed in parallel, no more threads are available, and some handlers may be delayed. Note, however, that this does not violate the semantics of asynchronous events, since the handler is still executed concurrently to the continuation of the caller. *Disj Only* (2) When an asynchronous event is fired and only propagated to disjunctions, enqueuing does not require a dedicated handler and can be part of the collection step. This optimization avoids the use of a separate thread for enqueuing. *Counters* (3) Event arguments are stored in the associated queue of a disjunction. However, if the event has no arguments, keeping a counter of the event occurrences is sufficient. Since synchronous events need to store the blocked threads anyway, a queue is still required, so the optimization is only applicable to asynchronous events. Note that in JEScala the synchronicity of an event cannot be determined statically. We apply the optimization when the event is trivially known to be asynchronous (it is a synchronous primitive event or the result of an event expression prefixed by the `!!` operator). Static analysis could be used to broaden the applicability of the optimization.

5. Evaluation

This section demonstrates the design advantages of JEScala in several small case studies and provides a preliminary performance evaluation. The code used for the evaluation is available online [20].

5.1 Qualitative Evaluation

We use several small case studies instead of a single larger one for two reasons. First, with several *synthetic* examples we can challenge JEScala with intentionally complex coordination schemas, while a *real* application would be probably less compelling from a coordination standpoint. Second, a larger example would dilute the coordination schema with the application logic. On the contrary, our studies distillate the essence of a coordination schema and sharpen the effects that we want to observe.

The case studies (Figure 8 Col. 1) include classic concurrency patterns (e.g., critical section, producer-consumer, actors), simulations that require coordination across several components (e.g., cellular automaton, binary adder, virus spreading over a complex network), and the Web server running example. Case studies also include client code that stresses the implemented features, e.g., threads accessing a critical section. The 2nd and 3rd columns in the figure report the number of threads and the number of components (classes and Scala objects) for each case study. We implemented each case study in JEScala and a subset JL (for Join Language) of JEScala excluding its specific features. JL programs are direct encodings of Polyphonic Scala programs (see Section 2) in JEScala.

For each implementation we measured the following metrics: lines of code ignoring comments and white spaces measured by CLOC[6], number of events, number of handlers and number of imperatively triggered events. To test the effect of different concurrency solutions, some case studies are implemented in both a single-threaded and a multithreaded version – marked with ST, respectively MT in the table. The need for coordination in a single-threaded context is not a contradiction, since a single thread must be "scheduled" to accomplish several tasks in a coordinated way. The variability between the ST and the MT versions does not affect our results significantly.

Based on the numbers reported in Figure 8, we make the following observations. Petri Nets and Parallel Graph Exploration use handlers to represent transitions, therefore we do not expect many differences between both implementations. JEScala captures coordination schemas in a more compact way; JEScala implementations have fewer lines of code (Columns LOC). The proportion of event declarations required by JEScala and by JL depends on the case study (Columns Events). JEScala implementations define fewer handlers (Columns Handlers). Furthermore, the number of statements in the code, where events are imperatively triggered are considerably reduced in the JEScala versions (Columns Imp. Evts). The reduction of handlers and imperative events indicate that the coordination logic is moved from handlers and imperatively triggered events to declarative event expressions. As a consequence, developers do not risk forgetting firing events and coordination patterns are more composable, easier to extend, and express the intention of the programmer in a more declarative way.

[6] `httpc://cloc.sourceforge.net`

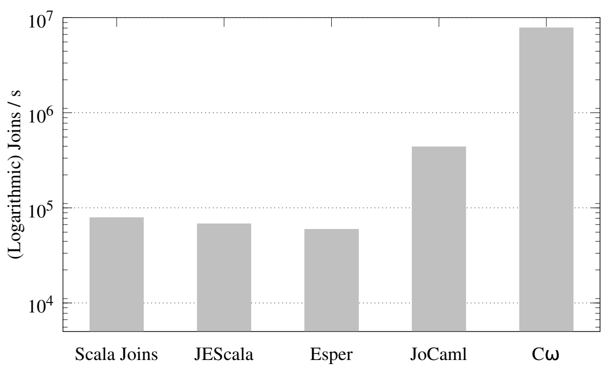


Figure 9. Performance of join languages.

5.2 Quantitative Evaluation

Our focus so far has been on the design of JEScala. To gain an idea about its performance, we implemented a number of benchmarks[7].

Comparison with other languages We initially compare JEScala with other languages that support joins. The benchmark consists of an automaton with n states. A transition fires when the join of the event associated to the current state and the event associated to the transition to another state fires. We measured the throughput (i.e., joined events per second) for Scala Joins, JEScala, the Esper complex event processing engine [11], JoCaml and Cω (Figure 9). Varying n from 1 to 5 (not shown) does not change the results significantly. Languages based on a dedicated compiler like JoCaml and Cω have the best performance. The results also show that performance degrades by increasing expressivity. This result is not surprising: for example Cω intentionally limits the constructs available to the programmer to achieve better performance [2]. At the opposite, Esper supports an extremely expressive language for event combination. JEScala exhibits a performance that minimally outperforms Scala Joins. However, JEScala is more expressive since it supports implicit invocation, event combination and real asynchronous events.

Effect of pattern complexity The complexity of the matching patterns has an impact on performance, which we measured with a dedicated benchmark. The size of the matching patterns in the benchmark increases; the cases $n = 3$ and $n = 4$ are shown in Figure 11a and Figure 11b. We measured the performance of each language for $n \in [2..6]$. The results for each language, normalized to the case $n = 2$, are in Figure 10. The benchmark shows performance degradation when the size of the pattern increases. Cω and JoCaml outperform JEScala, which however does better than Scala Joins.

Effect of optimizations To measure the effect of the optimizations for asynchronous events described in Section 4, we implemented a simple version of the rock-paper-scissors game. Two players running in different threads trigger an event that correspond to

[7] All measurements were performed on a MacBookPro6.2 with CPU I7 (2 cores, 2.66Ghz) with 8Gb ram, running OSX 10.6.8, Java 6 and Scala 2.10.

			LOC		Events		Handlers		Imp. Evts	
Case Study	**Th.**	**Comp.**	**JL**	**JE**	**JL**	**JE**	**JL**	**JE**	**JL**	**JE**
Critical Section (CS)	3	5	67	60	3	5	1	0	3	1
Alternating CS	3	4	49	42	7	8	5	1	8	3
Condition Variable	3	3	56	56	6	8	3	3	5	2
Monitor	6	7	86	80	9	10	3	1	7	2
Concurrent Barrier	2	3	46	37	4	4	1	0	3	0
Readers-Writer Lock	6	7	81	71	12	8	8	3	12	3
Threadsafe Counter	5	5	47	44	6	6	3	1	6	4
Hoare Cond. Crit. Region	4	7	90	71	7	9	4	1	7	2
Rendezvous	2	3	68	64	3	3	1	1	2	0
Concurrent Futures	2	3	58	48	7	7	3	3	5	2
Producer-Consumer (PC)	2	3	79	72	8	8	0	0	4	0
PC (Bounded Buffer)	4	4	72	68	7	7	2	2	5	3
Finite State Machine ST	1	2	76	66	14	11	11	5	12	6
Finite State Machine MT	4	4	74	64	14	11	11	5	12	6
Petri Net ST	1	2	46	44	9	12	13	14	9	8
Petri Net MT	3	2	56	54	11	14	12	11	9	8
Semaphore Petri Net	2	4	56	51	5	6	2	1	4	1
Tennis Players Petri Net	3	2	80	74	13	16	18	9	15	6
Agents (3 Ping-pong) ST	1	4	67	64	7	7	4	4	7	6
Agents (3 Ping-pong) MT	4	3	60	57	3	5	1	1	5	2
Agents (Token Ring) ST	1	4	54	54	7	6	4	4	5	4
Agents (Token Ring) MT	4	3	53	54	3	3	1	1	3	2
Elem. Cellular Automaton	1	3	87	84	8	11	4	1	9	2
Game Of Life	1	1	95	74	25	17	27	2	27	2
Shift Register	1	2	36	30	6	6	3	1	9	8
4 Bit Binary Adder	1	4	82	69	20	19	8	3	12	3
Logic Ports Circuit	3	4	70	64	10	7	7	1	7	1
Random Walks	7	8	185	179	16	16	12	3	17	4
Parall. Graph Explor.	21	20	184	181	9	10	5	6	10	7
Epidemic Model ST	1	11	184	172	18	16	16	4	18	6
Epidemic Model MT	11	11	190	178	18	16	16	4	18	6
Web Server	2	4	44	41	4	4	2	1	4	0
Web Server (Extended)	3	5	75	68	7	6	4	0	7	0
Web Server (Section 2.3)	4	5	111	97	18	18	8	4	16	8

Figure 8. Main metrics for the case studies.

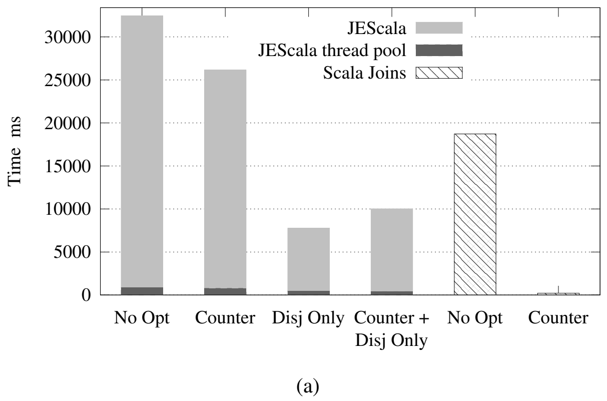
rock, *paper* or *scissors*. A game component matches those events in a disjunction. Each pattern in the disjunction captures a possible combination. Depending on the matching pattern, the first or the second player wins. We measure the time required to run $5{\cdot}10^4$ games.

The results are in Figure 12a. Column *No Opt.* shows the non optimized version of JEScala. Subsequent columns show the effect of the *Counters* optimization, of the *Disj. Only* optimization, and of both optimizations in action. To give an intuition of what the values mean in absolute terms, the last two columns show the performance of Scala Joins in the same benchmark for events with and without parameters. The former is obtained by adding a dummy parameter to the event, which forces Scala Joins to switch off the *Counters* optimization, the latter shows the case in which the counters optimization is applied.

The dark bars in Figure 12a show the performance of JEScala when adding the *Thread Pool* optimization. Figure 12b focuses on this case. The *Thread Pool* optimization is by far the most important to improve the performance of JEScala, and it is sufficient to make JEScala faster than Scala Joins in the case of an event with a parameter (Figure 12a). However, other optimizations are also significant and further double the performance of JEScala.

6. Related work

Join languages Key design aspects of join languages are summarized in Figure 13. Most join languages are based on existing idioms (*Language* column). The *Channels* column shows how chan-



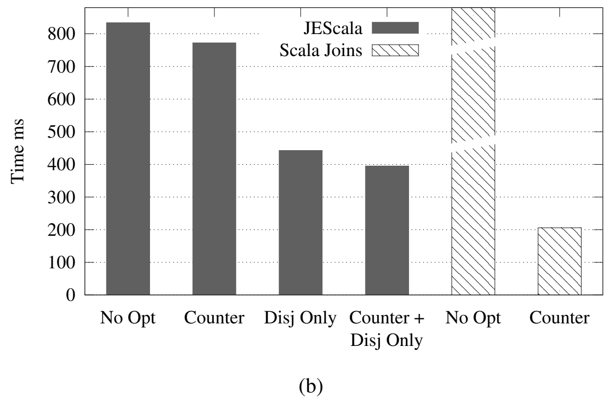


Figure 12. Optimization of asynchronous events in JEScala (a) zoom on the *Thread Pool* optimization (b).

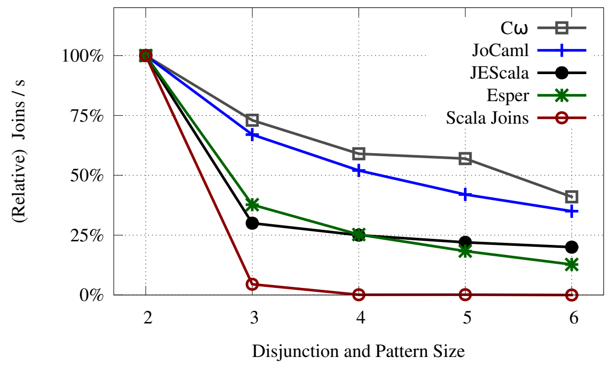


Figure 10. Effect of pattern complexity on performance.

```
var cnt:Long=0
//...evt decl
evt (toD,toC,toB,toA)=
    ( a & b )
  | ( c & a )
  | ( b & c )

toA += ()=>{ a(); cnt+=1 }
toB += ()=>{ b(); cnt+=1 }
toC += ()=>{ c(); cnt+=1 }
```

(a)

```
var cnt:Long=0
// ...evt decl
evt (toD,toC,toB,toA)=
    ( a & b & c)
  | ( d & a & b)
  | ( c & d & a)
  | ( b & c & d)
toA += ()=>{ a(); cnt+=1 }
 ...
toD += ()=>{ d(); cnt+=1 }
```

(b)

Figure 11. Benchmark: increasing complexity of matching patterns with n=3 (a) and n=4 (b).

nels are implemented. The *Sync* column indicates whether channels are synchronous or asynchronous. The *Disj* column shows how disjunctions are defined: through a specific explicit construct (*Explicit*) or, implicitly, via existing language abstractions. The *Matching* column shows how to select a reaction among active ones.

What sets JEScala truly apart from other join languages is its advanced event system that supports event composition and implicit events, in addition to imperatively triggered events. To the best of our knowledge, JEScala is the only language that explores the synergy between such an event system and join operators. The effect of using imperative events rather than method or function calls to implement channels was discussed in Section 2; the advantages of the synergy of implicit and declarative events with joins were discussed in Section 3.

The closest cousin of JEScala is the Scala library Scala Joins [17], since it also implements channels as events. Scala Joins demonstrate the use of extensible pattern matching to express joins and provides guards for join matching. Similarly to JEScala, disjunctions are explicit. Subclasses can redefine the disjunction in the superclass, but cannot modify it. However, unlike JEScala, the event model is simple: events must be declared together with the patterns they are involved in, including their synchronicity, and cannot be composed. When considering Scala Joins as a building block for implementing JEScala, the necessity of predefining the synchronicity of the events provided by the library turned out to be problematic for implementing JEScala declarative events, whose synchronicity may vary from one occurrence to the other.

Polyphonic C# [2] extends C# with advanced concurrency abstractions for asynchronous programming that are compiler-checked and optimized. Unlike JEScala it supports only one synchronous method per pattern and subclasses can change the body of a reaction but cannot extend disjunctions with new reactions. For our experiments we used the research language Cω [8], which offers the same extensions to C#. The Join Concurrency Library [33] provides a type-safe implementation of Polyphonic C# features by using C# 2.0 generics. The advantages are portability across the .NET platform and easier extensibility at the cost of fewer optimization opportunities. Further work [36] shows that impressive performance improvements can be achieved by fine-grained locking. Concurrent Basic [34] integrates the Join Concurrency Library into Visual Basic by including explicit channels and methods with multiple headers to define join patterns. Unlike JEScala, Concurrent Basic allows subclasses to add new reactions to existing disjunctions, as proposed in the Objective Join Calculus [14]. This however incurs the issues discussed in [2]. The latter is also true of JoinJava [19].

Funnel [26] is, as JoCaml, a language explicitly using the Join Calculus as its foundations (with the variation that only one synchronous channel is allowed in a join pattern). Unlike JoCaml, the language supports object-oriented programming with classes and inheritance on top of its functional basis.

In JErlang [29], channels are messages exchanged by actors. Erlang patterns are extended to express matching of multiple subsequent messages. Patterns are matched in their declaration order. JCThorn [27] extends the scripting language Thorn [3]. *Components* are actor-like containers for objects that share the same mailbox. JEScala implements a finer-grained event system and, unlike in JCThorn, concurrency abstractions and events are at the object granularity level. MogeMoge [24] is a prototype-based scripting language for game programming. Interactions are defined by asyn-

	Language	Channels	Sync	Disj.	Matching
JoCaml	Caml	Function	Async	Explicit	Non deterministic
Funnel	Funnel	Function/Method	Both	Explicit	Non deterministic
Polyphonic C#	C#	Method	Both	Object	Non deterministic
JoinJava	Java	Method	Both	Object	Both
Scala Joins	Scala	Imperative Event	Both	Explicit	Non deterministic
JErlang	Erlang	Message	Async	Actor	Deterministic
JCThorn	Thorn	Message	Async	Component	Deterministic
MogeMoge	MogeMoge	Join token	Async	Global	Deterministic
Join Conc. Lib.	.NET	Channel	Both	Explicit	Non deterministic
Concurrent Basic	Visual Basic	Channel	Both	Implicit	Non deterministic
JEScala	EScala	Advanced Events	Both	Explicit	Non deterministic

Figure 13. Languages implementing join abstractions.

chronous events called *join tokens* sharing a single global disjunction, the *token pool*.

Calculi JEScala has been designed in a pragmatic way. Proper theoretical foundations may bring a better understanding of its properties, in particular with respect to concurrency, with useful inspiration to be drawn from existing join-based calculi. We have already mentioned the objective join calculus [14], which deals with objects and inheritance. The aspect join calculus [35] may also be an interesting source of inspiration with respect to implicit invocation.

Other languages Pāñini [22] is a programming language that aims to coordinate concurrent components in a program by using explicit typed events, à la Ptolemy [32], except that these events are asynchronous, with a different meaning than the one we have used so far: events are fired synchronously with respect to their source but theirs handlers are executed asynchronously. However, no declarative ways of combining events are provided. Current versions of Pāñini aim at implicit concurrency [30, 31] with a programming style close to sequential programming. Using *capsules* to group objects into single threaded entities, which are combined into concurrent *systems*, results in coarse-grained concurrency compararable to actors [18]. Communication between capsules looks like method calls in a sequential program instead of asynchronous messages between actors.

Implicit invocation with traits [28] is another proposal based on explicit event types. There are no means of composition. The synchronicity of an event simply depends on whether it is associated to a block of code or not. In the first case, two synchronous, before and after, events are defined, otherwise a single asynchronous event is defined.

Join point interfaces [5] provide an interesting alternative to the event model of JEScala by using event types and still providing both explicit and implicit events, including declarative events. Also, this proposal is closer to Aspect-Oriented Programming than JEScala: its events (join points) can return a value and its handlers (advices) can be composed with `proceed`. However, it does not include any specific support for concurrency.

Other approaches Complex event processing (CEP) is about correlating time-changing streams of data. The available operators include joins with rich (and often subtle) semantic alternatives, typically applied to time windows. The interested reader is referred to [10] for an overview. Esper [11] is a CEP system implemented in Java. It is an enterprise-level product used in real applications with an expressive language and great emphasis on performance. As such, it is often used for comparison in CEP research, and was our choice as an indicator of the performance of CEP engines. Unlike join languages, most CEP solutions are not integrated into a programming language and applications interface with the CEP engine via SQL-like queries. A noticeable exception is EventJava [12].

Sequential and Parallel Object Monitors (SOM and POM) [6, 7] aim to to separate fine-grained synchronization concerns from the application logic. An *object monitor* is a programmable threadless scheduler that applies to reified calls to methods belonging to the object (or set of objects) the monitor is attached to. The framework is very expressive and makes it possible to implement, in Java, monitors able to deal with, among others, join patterns and disjunctions, with the possibility of providing different semantics, for instance in terms of determinacy. JEScala also deals with separation of concerns and concurrency but not at the same level and with a different purpose. Our concurrency abstractions are fixed and our interest is in seamlessly integrating them, at the language level, in order to improve modularity. Sometimes, but not always, both approaches almost completely overlap. For instance, in Java, it would make sense to write the rate limiter of Figure 4 as an object monitor (it would still require more programming). The mapping is less clear as soon as the rate limiter becomes more complex. On the one hand, the issue is not to systematically extract concurrency and, on the other hand, an issue is also to modularize the synchronization concern itself.

Finally, [4] introduces a rich programming model combining a form of implicit event types and aspects. Two salient features of the proposal are the possibility to define, within event declarations, side-effect-free code collecting data to be carried by the event as well as fine-grained means to control composition. Concurrency issues are not addressed.

7. Summary and Future Work

In this paper we have presented the design of JEScala, a language that combines the advanced event system of EScala with concurrency abstractions from the Join Calculus. We have shown that this solution captures coordination patterns in a way that is more compact and more declarative than existing join languages while preserving the OO style of modular reasoning (events are object members). Still, we have found that concurrency issues related to non-determinism and the mixing of synchronous and asynchronous events (a source of deadlocks) is challenging. We feel that JEScala provides an interesting practical language to study these issues.

Future work includes improving performance, providing compiler support as well as exploring the theroretical underpinning of JEScala. Finally, as already discussed in Section 6, CEP engines offer a richer semantics for event correlation than event-based languages – most noticeably, by including *time* in the form of various types of windows over the event streams. We plan to explore the semantic alternatives that joins offer in the context of event correlation over time windows. This field has been partially explored in CEP, but language integration of a flexible semantics for correlating events is still a research challenge.

Acknowledgments

This work has been partially supported by the German Federal Ministry of Education and Research (BMBF) under grant No. 16BY1206E ACCEPT and by the European Research Council, grant No. 321217.

References

[1] H. C. Baker, Jr. and C. Hewitt. The incremental garbage collection of processes. In *Symposium on Artificial intelligence and programming languages*, pages 55–59. ACM, 1977.

[2] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C#. *ACM TOPLAS*, 26(5):769–804, Sept. 2004.

[3] B. Bloom, J. Field, N. Nystrom, J. Östlund, G. Richards, R. Strniša, J. Vitek, and T. Wrigstad. Thorn: robust, concurrent, extensible scripting on the JVM. In *OOPSLA '09*, pages 117–136. ACM, 2009.

[4] C. Bockisch, S. Malakuti, M. Akşit, and S. Katz. Making aspects natural: events and composition. In *AOSD '11*, pages 285–300. ACM, 2011.

[5] E. Bodden, E. Tanter, and M. Inostroza. Joint point interfaces for safe and flexible decoupling of aspects. *ACM TOSEM*, 2014. To appear.

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

[7] D. Caromel, L. Mateu, and É. Tanter. Sequential object monitors. In *ECOOP '04*, volume 3086 of *LNCS*, pages 316–340. Springer, 2004.

[8] Cω. Language Website. http://research.microsoft.com/en-us/um/cambridge/projects/comega/.

[9] S. Conchon and F. Le Fessant. JoCaml: mobile agents for Objective-Caml. In *ASAMA '99*, pages 22–29. IEEE Computer Society, 1999.

[10] G. Cugola and A. Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, June 2012.

[11] EsperTech. Company Website. http://www.espertech.com.

[12] P. Eugster and K. R. Jayaram. EventJava: An extension of Java for event correlation. In *ECOOP '09*, volume 5653 of *LNCS*, pages 570–594. Springer, 2009.

[13] C. Fournet and G. Gonthier. The reflexive CHAM and the join-calculus. In *POPL '96*, pages 372–385. ACM, 1996.

[14] C. Fournet, C. Laneve, L. Maranget, and D. Rémy. Inheritance in the join calculus. In *Proceedings of the 20th Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 1974 of *LNCS*, pages 397–408. Springer, 2000.

[15] D. Garlan and D. Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *VDM '91*, volume 551 of *LNCS*, pages 31–44. Springer, 1991.

[16] V. Gasiūnas, L. Satabin, M. Mezini, A. Núñez, and J. Noyé. EScala: modular event-driven object interactions in Scala. In *AOSD '11*, pages 227–240. ACM, 2011.

[17] P. Haller and T. Van Cutsem. Implementing joins using extensible pattern matching. In *COORDINATION '08*, volume 5052 of *LNCS*, pages 135–152. Springer, 2008.

[18] C. Hewitt, P. Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In *IJCAI '73*, pages 235–245. Morgan Kaufmann, 1973.

[19] G. S. Itzstein and M. Jasiunas. On implementing high level concurrency in Java. In *Advances in Computer Systems Architecture*, volume 2823 of *LNCS*, pages 151–165. Springer, 2003.

[20] The JEScala site. http://www.stg.tu-darmstadt.de/research, 2014.

[21] J.-J. Lévy. Some results in the join-calculus. In *TACS '97*, volume 1281 of *LNCS*, pages 233–249. Springer, 1997.

[22] Y. Long, S. L. Mooney, T. Sondag, and H. Rajan. Implicit invocation meets safe, implicit concurrency. In *GPCE '10*, pages 63–72. ACM, 2010.

[23] L. Mandel and L. Maranget. *The JoCaml language - Documentation and user's manual*. Inria, Aug. 2012. Release 4.00.

[24] T. Nishimori and Y. Kuno. Join token-based event handling: A comprehensive framework for game programming. In *SLE '11*, volume 6940 of *LNCS*, pages 119–138. Springer, 2011.

[25] A. Núñez, J. Noyé, V. Gasiūnas, and M. Mezini. *Aspect-Oriented, Model-Driven Software Product Lines - The AMPLE Way*, chapter Product Line Implementation with ECaesarJ. Cambridge University Press, 2011.

[26] M. Odersky. An introduction to functional nets. In *Applied Semantics*, volume 2395 of *LNCS*, pages 333–377. Springer, 2002.

[27] I. S. Paula. JCThorn: Extending Thorn with joins and chords. Master's thesis, Department of Computing, Imperial College London, 2010.

[28] T. Pawlitzki and F. Steimann. Implicit invocation of traits. In *SAC '10*, pages 2085–2089. ACM, 2010.

[29] H. Plociniczak and S. Eisenbach. JErlang: Erlang with joins. In *COORDINATION '10*, volume 6116 of *LNCS*, pages 61–75. Springer, 2010.

[30] H. Rajan, S. M. Kautz, E. Line, S. Kabala, G. Upadhyaya, Y. Long, R. Fernando, and L. Szakács. Capsule-oriented programming. Technical Report 13-01, Iowa State U., Computer Sc., 2013.

[31] H. Rajan, S. M. Kautz, and W. Rowcliffe. Concurrency by modularity: design patterns, a case in point. In *OOPSLA '10*, pages 790–805. ACM, 2010.

[32] H. Rajan and G. T. Leavens. Ptolemy: A language with quantified, typed events. In *ECOOP '08*, volume 5142 of *LNCS*, pages 155–179. Springer, 2008.

[33] C. Russo. The joins concurrency library. In *PADL '07*, volume 4354 of *LNCS*, pages 260–274. Springer, 2007.

[34] C. V. Russo. Join patterns for Visual Basic. In *OOPSLA '08*, pages 53–72. ACM, 2008.

[35] N. Tabareau. A theory of distributed aspects. In *AOSD '10*, pages 133–144. ACM, 2010.

[36] A. J. Turon and C. V. Russo. Scalable join patterns. In *OOPSLA '11*, pages 575–594. ACM, 2011.