

# Industry-strength join calculus

## Declarative concurrent programming with Chymyst

Anonymous Author(s)

### Abstract

Join calculus (JC) is a declarative message-passing concurrency formalism that has been ignored by the software engineering community, despite its significant promise as a means of solving the problems of concurrent programming. I introduce Chymyst, a new open-source framework that aims to bring industry-strength JC programming to Scala practitioners. Taking advantage of its embedding into the Scala language, Chymyst enhances JC with features such as arbitrary non-linear join patterns with guard conditions, synchronous rendezvous, time-outs, and incremental construction of join definitions. The current implementation also performs static analysis of user code, early error detection, and automatic performance optimizations. To ease the learning curve for engineers unfamiliar with the concepts of JC, I develop a pedagogical presentation of JC as an evolution of the well-known Actor model whereby actors are made type-safe, immutable, and are automatically managed. By comparison with the popular Akka library, I identify a comprehensive set of additional features necessary to make JC an industry-ready concurrency paradigm. These features include ordered channels, APIs for unit testing, thread pool control, and fault tolerance. I show how Chymyst embeds such features in the JC paradigm.

**CCS Concepts** •Computing methodologies → Concurrent programming languages; •Software and its engineering → General programming languages;

**Keywords** join calculus, concurrency, Scala

### ACM Reference format:

Anonymous Author(s). 2017. Industry-strength join calculus. In *Proceedings of 8th ACM SIGPLAN Symposium on Scala, Vancouver, Canada, October 22-23, 2017 (Scala’17)*, 11 pages. DOI: 10.1145/nnnnnnn.nnnnnnn

## 1 Introduction and summary

Advanced programming models developed by the theoretical computer science community are often ignored by software practitioners. One such case is join calculus (Fournet and Gonthier 1996), which can be seen as a domain-specific language (DSL) for declarative, functional concurrent programming. Given the high importance of concurrent programming and a growing adoption of functional languages, one would expect that software practitioners would take

*Scala’17, Vancouver, Canada*

2017. 978-x-xxxx-xxxx-x/YY/MM...\$15.00

DOI: 10.1145/nnnnnnn.nnnnnnn

advantage of this high-level and type-safe concurrency paradigm. The message-passing idiom is particularly suitable for implementing distributed algorithms (see, e.g., (Baragatti et al. 2007)). Nevertheless, there appears to be no practical adoption of join calculus (JC) by the software industry.<sup>1</sup> Perhaps not coincidentally, there are very few open-source implementations of JC available for download and use. To date, the only fully maintained implementation of JC is the JoCaml language (Fournet et al. 2003).

Another significant barrier for software practitioners is the lack of suitable documentation and example code. The existing documentation and tutorials for JC, such as the JoCaml user’s manual<sup>2</sup>, the original authors’ introduction to JC (Fournet and Gonthier 2000), and the lecture notes (Fournet et al. 2003) were intended for CS graduate students and are largely incomprehensible to software engineers. Effective JC programming requires a certain paradigm shift and facility with JC-specific design patterns, which is not readily achieved without working through numerous examples.

Industry-strength frameworks must provide a number of important facilities such as fault tolerance and process supervision, performance tuning and metrics, or unit testing and debugging, to name just a few. Existing presentations of JC do not consider these features and do not describe how they are to be integrated into the JC paradigm. Neither do any of the existing JC implementations provide such features. This may be the biggest obstacle for industrial use of JC.

In this paper, I present a new open-source implementation of JC as a library called Chymyst,<sup>3</sup> consisting of an embedded Scala DSL and a light-weight runtime engine. The main design focus of Chymyst is to enable high-level, declarative concurrency in idiomatic Scala, using the JC paradigm. An equally important goal is to provide industry-strength features such as performance tuning, fault tolerance, and unit testing APIs. Finally, the Chymyst project offers tutorial documentation adapted to the software developer audience, presenting numerous examples that illustrate the patterns of programming in join calculus. In these ways, I hope to enable industry adoption of this powerful concurrency paradigm.

### 1.1 Contributions of this paper

I describe the main design decisions made in the Chymyst project while implementing join calculus as an embedded

<sup>1</sup> A Google search yields a number of academic projects but no mentions of industrial JC use.

<sup>2</sup> See [jocaml.inria.fr/doc/index.html](http://jocaml.inria.fr/doc/index.html).

<sup>3</sup> The name is borrowed from the early treatise (Boyle 1661) by Robert Boyle, who was one of the founders of the science of chemistry.

DSL in Scala. Chymyst lifts several restrictions that are present in other JC projects, and offers some additional features:

- separate definition of channel names and processes
- arbitrary non-linear patterns and guards in process definitions
- syntax that supports synchronous rendezvous
- incremental construction of join definitions from first-class process definitions
- static code analysis and early error detection

I argue that certain facilities need to be added to a JC implementation in order to make it viable for industry adoption, including:

- thread pool management for performance tuning
- time-outs for synchronous channels
- interoperability with Future's and other async APIs
- APIs for unit testing and debugging
- per-process fault tolerance options
- ordered channels, message pipelining

I show how these facilities can be embedded in the JC paradigm and describe their current implementation in Chymyst.

I point out specific deficiencies of the academic terminology of JC (message / channel / process / join definition) that make it unhelpful for explaining the concepts of JC to software developers. Instead, I show how to describe JC as an evolution of the Actor model, which is ideal for developers already familiar with the Akka library. When introducing JC from scratch, the “chemical machine” metaphor and the corresponding terminology (molecule / emitter / reaction / reaction site) prove to be more visual and intuitive. Synchronous channels are presented as a shorthand for carrying out a continuation-passing code transformation to code that uses only asynchronous, non-blocking channels. The continuation-passing syntax used by Chymyst for synchronous channels (instead of the traditional “reply to” syntax) makes this code transformation more transparent.

## 1.2 Previous work

Since its invention more than 20 years ago, join calculus has been implemented by a number of researchers, typically by creating an entirely new JC-based programming language or by modifying an existing language compiler. It is hard to assess the practical use of these implementations, since most of them appear to be unmaintained proof-of-concept projects.

Here I will not attempt to survey the theoretical advances made by those researchers. Since the main goal of the Chymyst project is to enable industry acceptance of JC, I will focus on the practical availability and usability of the existing JC implementations.

JoCaml was one of the first implementations of JC (Fournet et al. 2003), and remains today the best-supported one. This

implementation is a patch for the OCaml compiler, which is, however, fully compatible with the OCaml library ecosystem.

M. Odersky created a new language called “Funnel”, based on the JC paradigm (Odersky 2000). The Funnel project appears to be abandoned, since M. Odersky went on to develop Scala, which does not include any concepts or features of JC — either in the language itself or in its standard libraries.

G. S. von Itzstein implemented JC as a patch for the Java compiler (von Itzstein 2004). The project is abandoned.

The first appearance of JC in Scala was a “Join in Scala” compiler patch by V. Cremet (2003).<sup>4</sup> The project is unmaintained, and the Scala language has changed radically since 2003, rendering the project unusable.

T. Rompf implemented an experimental (unnamed) language based on JC and illustrated its use for important application design patterns, such as “fork/join” synchronization and asynchronous continuations (Rompf 2007). The project appears to be abandoned, as T. Rompf moved on to research on multi-stage compilation (Rompf 2012).

“Joinads” is a set of compiler patches for F# and Haskell, developed by T. Petricek (Petricek and Syme 2011). The project is unmaintained.

Creating a *new* programming language, either from scratch or via compiler patches, has been a common pattern in JC implementations. The reason seems to be the difficulty of accommodating join definitions within the syntax of existing languages. Short-lived projects such as Polyphonic C# (Benton et al. 2002), *Cω* (Russo 2007), Join Diesel (Osera 2005), JErLang (Plociniczak and Eisenbach 2010), and Hume (Hammond and Michaelson 2004) also follow that pattern. All these new languages have since been abandoned by their creators. It appears that maintaining and supporting a completely new research language is hardly possible, even for a corporation such as Microsoft. Therefore, we turn our attention to implementations of JC as an embedded DSL in a well-established programming language.

C. Russo created the “Scalable Joins” library for the .NET platform (Russo 2007). The library is unsupported.<sup>5</sup>

Y. Liu implemented the basic JC primitives in 2007-2009 as part of the C++ Boost library.<sup>6</sup>

S. Winitzki implemented JC for Objective-C on iOS and for Java on Android<sup>7</sup>; these projects are unmaintained. Chymyst reuses some design decisions made in these earlier works.

In 2014, S. Yallop implemented “Join Language” as a DSL embedded in Haskell.<sup>8</sup> The implementation uses advanced features of Haskell's type system to provide a concise syntax for join definitions.

<sup>4</sup> See [lampwww.epfl.ch/~cremet/misc/join\\_in\\_scala/](http://lampwww.epfl.ch/~cremet/misc/join_in_scala/).

<sup>5</sup> See [github.com/JoinPatterns/ScalableJoins](https://github.com/JoinPatterns/ScalableJoins).

<sup>6</sup> See [channel.sourceforge.net](http://channel.sourceforge.net).

<sup>7</sup> See [github.com/winitzki](https://github.com/winitzki).

<sup>8</sup> See [github.com/syallop/Join-Language](https://github.com/syallop/Join-Language).

The first embedding of JC as a Scala DSL was P. Haller's "Scala Joins" library (Haller and Van Cutsem 2008). Thereafter, J. He improved upon "Scala Joins" by streamlining the syntax, removing restrictions on pattern matching, and implementing remote processes (He 2014). Chymyst is a further development of P. Haller and J. He's syntax for embedding JC into Scala.

C. Russo's library (Russo 2007) allowed synchronous rendezvous in join patterns as well as incremental construction of processes and join definitions, while T. Rompf's language (Rompf 2007) used a continuation-passing syntax for synchronous channels, instead of the traditional "reply to" syntax. Chymyst also adopts these design choices.

## 2 Programming in Chymyst

Neither of the words "join" and "calculus" are particularly explanatory or visually suggestive. In my experience, the absolute majority of software developers are unfamiliar with join calculus or its terminology (channel / message / process). However, most Scala concurrency practitioners know the Actor model through the Akka library. Accordingly, I would argue that introducing JC concepts to the Scala developer audience should build upon the existing Actor model knowledge. However, reasoning about JC programs is most direct and convenient when using the visual metaphors and the terminology of the "chemical machine" (molecule / reaction). The next subsection is a brief overview of JC as seen through the chemical machine metaphor (Fournet and Gonthier 1996). This introduction is suitable also for readers not already familiar with either JC or the Actor model.

### 2.1 The chemical metaphor for concurrency

In the chemical metaphor, concurrent processing of data is visualized as chemical reactions between molecules that carry the data. To begin, one imagines a **reaction site**, i.e. a virtual place where many molecules are floating around and possibly reacting with each other. Each molecule has a chemical designation (such as *a*, *b*, *c*) and also *carries a value* of a fixed ground type (such as `Unit` or `List[Int]`). Since the "chemistry" here is completely imaginary, the programmer is free to declare any number of chemical designations and to choose the corresponding value types. In Chymyst, these declarations have the form

```
val c = m[List[Int]]
val t = m[Unit]
```

and create new **molecule emitters** as local values *c* and *t*. Emitters can be seen as functions returning `Unit` that have a concurrent side-effect of emitting the corresponding molecules into the reaction site:

```
c(List(1,2,3))
t() //unit value implicit
```

Since the emitters are statically typed, the newly emitted molecules will carry specified values of the correct types.

Further, the programmer defines the "chemical laws" describing the permitted reactions between molecules. For reactions, Chymyst uses the syntax of Scala partial functions with a single case clause, wrapped into an auxiliary method called `go()`, for example:

```
go { case t(_) + c(x :: xs) => c(xs) }
```

This reaction consumes two input molecules, `t()` and `c()`, and evaluates the reaction body (the Scala code in the body of the case clause). In this example, the reaction body simply emits one molecule, `c()`, carrying a computed new value *xs*. In effect, this reaction computes the tail part of a list. Due to the pattern-matching condition, this reaction will start only when the molecule `c()` carries a non-empty list value.

In Chymyst, reaction definitions may use all features of Scala partial functions, including arbitrary guard conditions and pattern matching constructs.

Reactions are first-class values:

```
val r1 = go { case t(_) => println("done") }
```

Creating the reaction value *r1* does not actually run a computation; it merely defines the available computation declaratively. In order to make the chemical machine run reactions, the programmer needs to create a reaction site using the `site()` call, e.g. `site(r1, r2)`, which activates the reactions listed as the arguments. A reaction site typically includes several reactions declared inline for brevity:

```
site(
  go { case t(_) + c(x :: xs) => c(xs) },
  go { case c(Nil) => done() }
)
```

Once a reaction site with these reactions is created, the molecules `t()` and `c()` can be emitted into it. The chemical machine will interpret the declared "chemical laws" and start reactions whenever appropriate input molecules are available. Typically, reactions will emit new molecules, and the chemical machine will continue monitoring all declared reaction sites, looking for further reactions to run. Any number of different reactions may start concurrently if their input molecules are available.

Thus, JC application code defines a number of molecule emitters and reaction sites with reactions, and then emits a number of initial molecules. Emitting a molecule is a non-blocking operation: reactions will start concurrently with the emitting process. In this way, JC models arbitrary asynchronous and concurrent computations.

The Chymyst project embraces the chemical metaphor and its visually suggestive terminology. In the academic literature on JC, molecule emitters are called "channels", emitting a molecule with a value is called "sending a message on a channel", blocking molecules are "synchronous messages",

reactions are “processes”, and reaction sites are “join definitions.” To make the reading of the present paper easier for academic researchers, I use the academic terminology in what follows, except when describing pedagogical approaches to JC.

### 2.1.1 Synchronous channels as shorthand for continuations

In a fully asynchronous, non-blocking programming style, a continuation can be used in order to simulate waiting until some concurrently running computation is finished. However, using asynchronous channels with continuations is a bit cumbersome: The continuation function must be explicitly created as a closure and passed as a message value. Manually creating a continuation results in “stack ripping” (Adya et al. 2002), which disrupts the normal code flow.

The JoCaml implementation of JC uses synchronous channels to mitigate this problem. For example, a JoCaml process

```
def a(x) & f(y) = reply x+y to f
```

defines  $f()$  as a synchronous channel. Chymyst also implements synchronous channels as a first-class language feature. Chymyst’s chosen syntax for synchronous channels resembles continuation-passing style and looks like this:

```
go { case a(x) + f(y, reply)  $\Rightarrow$  reply(x+y) }
```

The “reply” expression can be seen to resume the continuation in the process that sent the  $f()$  message. The syntax of Chymyst makes this semantics more explicit.

Synchronous channels are defined using the syntax  $\text{val } f = b[T, R]$ , where  $T$  is the type of the message value and  $R$  is the type of the reply value. Here is a more detailed example of a Chymyst process with a synchronous channel:

```
val f = b[Int, Int]
site( go { case f(x, reply)  $\Rightarrow$ 
  val y = ... //compute some value
  reply(y) //resume continuation
} )
val z = f(123) //wait for reply
```

Note that the message  $f(123)$  is sent without specifying the continuation argument  $\text{reply}$ , which is nevertheless available in the reaction body. The semantics of the blocking call  $f(123)$  is equivalent to capturing the current continuation as  $\text{reply}$  and sending the tuple  $(123, \text{reply})$  as a message on the channel  $f$ .

As I will show below in Sec. 2.2.3, this syntax yields additional flexibility in defining synchronous processes.

## 2.2 The Chymyst flavor of JC

In this section, I compare the implementation of JC in Chymyst to that of JoCaml and motivate the relevant design choices and enhancements.

### 2.2.1 “Chemical” syntax

The syntax “ $a(x) + b(y) \Rightarrow \dots$ ” is reminiscent of the notation for chemical reactions. It is somewhat easier to read than the traditional JC syntax “ $a(x) \& b(y)$ ”.

Sending a message in Chymyst is implemented as a function call such as “ $a(1)$ ” and does not require a special keyword such as JoCaml’s “spawn”.

### 2.2.2 Explicit channel definitions

JoCaml defines new channels implicitly, as soon as a new join definition is written. However, implicit declaration of new channels is not possible in Chymyst because Scala macros cannot insert a new top-level symbol declaration into the code. So, channel declarations and their types need to be explicit (“ $\text{val } a = m[\text{Unit}]$ ”) and written separately from process definitions. This is a common design in embedded DSL implementations of JC.

The separation of channel definitions from process definitions brings several advantages. One is the ability to create a number of new channels at run time and to define processes for them incrementally (see Sec. 2.2.4 below), which removes the requirement to specify all processes in a join definition at compile time. Another is an increased code clarity due to the explicit labeling of blocking vs. non-blocking channel types, which remains implicit in JoCaml.

A drawback of this separation is that programmers might define a new channel but forget to define any processes waiting on that channel. Sending messages to such “unbound” channels is an error that can only be detected at run time because channels are first-class values and, for instance, can be passed as parameters to functions that send messages on those channels.

Chymyst throws an exception if the application code attempts to send a message to an unbound channel. An exception is also thrown when a new process is being defined that uses a channel already bound to another join definition. (In that case, JoCaml silently redefines the channel name, creating a potential for bugs.) These exceptions are generated at “early” run time, immediately after creating the join definition and before running any processes.

### 2.2.3 Non-linear join patterns

Another enhancement is the lifting of the linearity restriction for join patterns. In Chymyst, a process may wait on any number of repeated channels:

```
go { case a(x) + a(y) + a(z)  $\Rightarrow$  a(x+y+z) }
```

Some computations are most directly expressed in this form. A strictly linear-pattern JC implementation, such as JoCaml, would require cumbersome auxiliary definitions to implement equivalent behavior.

Processes may also wait on repeated *synchronous* channels. This last feature, together with the continuation-passing syntax for those channels, enables a declarative implementation



of synchronous rendezvous (Milner 1999) where two distinct users of the same channel can exchange values:

```
go { case f(p1, reply1) + f(p2, reply2) =>
  reply2(p1); reply1(p2)
}
```

The traditional JC reply syntax (`reply x to f`) does not allow the code to select the specific copy of `f()` to which a reply is sent. JoCaml can express equivalent behavior only at the cost of using two additional channels and an additional process that also performs a synchronous reply.

## 2.2.4 First-class process definitions

Since process definitions in Chymyst are first-class values, join definitions can be constructed incrementally at run time by aggregating a dynamically defined number of process definitions. For example, the well-known “dining philosophers” problem has a simple declarative solution in join calculus (see e.g. (Varela and Agha 2013), Sec. 5.4.3) where, however, the number of philosophers needs to be defined statically. In Chymyst, this solution can be easily extended to  $n$  philosophers by creating  $n$  philosopher channels at run time, defining an array of  $n$  processes for these channels, and aggregating the  $n$  processes into a join definition.

A run-time dependent number of channels and processes can be defined like this,

```
val cs = (1 to n).map(_ => m[Int])
val rs = (1 to n).map(i => go { ... })
site(rs: _*) //join definition
```

Despite this, processes and channels remain immutable. Once a join definition has been created, is impossible to modify its constituent processes or to add more processes waiting on the same channels.

## 2.2.5 Static code analysis

Scala’s macros are used extensively in Chymyst for user code analysis. The `go()` method is a macro that gathers detailed compile-time information about input and output channels of each defined process. For example, the process definition

```
go { case t(-) + c(x :: xs) => c(xs) }
```

internally produces a rich information structure indicating that the process waits on the channels `t` and `c`, that the channel `t` may have arbitrary message values while `c` requires a nontrivial pattern match with two pattern variables, and that the process will finally send a message on channel `c` (but not on `t`).

Using this information, Chymyst detects certain errors in user code, such as unavoidable livelock and non-determinism. An example of unavoidable livelock is the process

```
go { case c(x) => c(x+1) }
```

Since the process accepts messages `c(x)` with any value  $x$ , the programmer has no means of stopping the infinite loop that follows once a message is sent on channel `c`. This Chymyst code generates a *compile-time* error, with a message indicating unavoidable livelock.

Unavoidable nondeterminism occurs when one process waits on a subset of messages that another process is also waiting on, for instance

```
site( go { case c(x) + i(-) => c(x+1) },
      go { case c(x) => done() } )
```

If both `c()` and `i()` messages are present, the runtime engine has a choice of whether to run the first or the second process. The programmer has no control over this choice, since there are no conditions on the value  $x$  of the `c(x)` message. It is unlikely that the resulting non-determinism would be useful in any practical application. Chymyst assumes that this is a programmer’s error and throws an exception. The exception is thrown at “early” run time.

An additional benefit of static analysis is a performance optimization for processes that use pattern matching or guard conditions. For process definitions that impose no conditions on input message values, a quicker scheduling algorithm is used. Additionally, complicated cross-molecule guard conditions such as

```
go { case c(x) + d(y) if x>0 && y<1 => ... }
```

are converted into the conjunctive normal form and factorized if possible. For instance, the example above is converted to (pseudo-code)

```
c(x if x>0) + d(y if y<1) => ...
```

In many cases, this transformation allows the runtime engine to find new runnable processes faster.

Static analysis is also used to enforce the linear type discipline on synchronous channels. For instance, this Chymyst code will generate a compile-time error:

```
go { case f(x, reply) => if (x>0) reply(x)
      else false }
```

Correct usage of synchronous channels requires that a reply is sent in all clauses of an `if / else` or `match / case` expression.

## 2.3 Details of the current implementation

Chymyst implements each join definition as a separate instance of class `ReactionSite`. A reaction site is a virtual place where messages arrive and wait to be consumed by auto-created JC processes. A `ReactionSite` instance contains the list of available processes and their properties, a JVM thread pool for running the processes, a message multiset holding the message values currently present, and a separate JVM thread dedicated to process scheduling and bookkeeping operations. The decision to make scheduling

single-threaded was motivated by the desire to avoid thread contention and to make process scheduling faster.

Since join definitions are local *values* (although they are not directly available to the application code), new join definitions can be created dynamically at run time. Nevertheless, join definitions are immutable since the user cannot access a reaction site.

Channels are local values available to the application code. Each channel is an instance of class *M* for asynchronous channels or *B* for blocking (synchronous) channels. A channel holds a reference to the join definition where its messages are consumed; according to the JC semantics, each channel must be bound to a unique join definition.

When a message is sent on a channel, the message's value is appended to the message multiset at the respective join definition. At the same time, a "process search" job is queued to be run on the scheduler thread of that join definition. The process search will examine the messages currently present in the multiset and determine which processes (if any) will be scheduled to run. These processes will then be queued on the reaction site's thread pool, after the consumed messages are atomically removed from the message multiset.

Due to using separate threads, sending a message as well as running a process are asynchronous, concurrent operations, as they should be in JC. The number of available concurrent execution threads can be specified per join definition (see Sec. 2.4.1).

Reactions that impose complicated cross-molecule conditions may take a comparatively long time to schedule because the process search may need to enumerate many possibilities before it finds appropriate molecule values. Since the process search is performed asynchronously on a dedicated scheduler thread, sending a message is always a quick operation. The downside is a somewhat slower scheduling of all processes due to additional thread switching.

## 2.4 What it means to be industry-strength

To determine the feature set required for "industry strength," I have reviewed the widely used Akka actor framework as described in (Roestenburg et al. 2016) and other similar books intended for software engineering audiences. These books suggest that the industry expects a concurrency framework to implement features such as:

- logging facilities for execution tracing and for performance metrics
- unit testing with assertions about asynchronous behavior of messages
- fault tolerance, various scenarios of recovery from errors, per-process supervision
- timeouts for blocking calls
- graceful stop and graceful restart, possibly separately for each join definition
- a standard library of join definitions that establishes the relevant design patterns

- thread pool management, thread priority, configurable performance tuning
- ordered message queues for asynchronous pipelines
- interoperability with other asynchronous APIs such as *Future*'s, *Actors*, *Tasks*, or a library of adapters to other async frameworks
- deployment-time configuration facilities for distributed and remote execution (e.g. cluster deployment)
- consistent distributed data management (CRDT) support

No currently available descriptions or implementation of join calculus include *any* of these features. JoCaml provides a remote messaging facility<sup>9</sup>, but the burden of implementing a robust distributed application and configuring its deployment rests fully on the JoCaml application code. There does not appear to be any JoCaml-based industrial implementations of distributed applications.

I will now describe the features that Chymyst currently supports.

### 2.4.1 Thread pools

To enable users to fine-tune the performance of their concurrent applications, Chymyst offers control over the JVM thread pools used by join definitions. There is a default thread pool available, but users may create other thread pools and assign them to join definitions. Application code can create two types of thread pools: a *FixedPool* holds a fixed specified number of JVM threads, while a *BlockingPool* is aware of blocking operations (such as sending synchronous messages) and will increase and decrease the number of threads dynamically at run time in order to maintain a given parallelism. Optionally, application code can set a specific thread priority for the thread pool's processes.

A typical use case for separate thread pools is an application with two join definitions, one having a large number of slow processes and another with a small number of fast processes. For instance, the "fast" join definition might be reporting the progress achieved by the "slow" processes. Since it is desirable to obtain timely progress reports, the "fast" join definition is typically required to have low latency as compared with the "slow" join definition. The application code achieves this by using separate thread pools for the two join definitions:

```
val tpFast = FixedPool(1)
val tpSlow = FixedPool(cpuCores)
site(tpFast)(...) //first join definition
site(tpSlow)(...) //second join definition
```

A thread pool can be shut down, which will cause the join definition(s) using it to stop processing messages.

<sup>9</sup> See [jocaml.inria.fr/doc/distributed.html](http://jocaml.inria.fr/doc/distributed.html).

### 2.4.2 Timeouts

The call that sends a message on a synchronous channel, e.g.  $f(x)$ , will be blocked until a reply is received. In Chymyst, it is possible to specify a timeout for this blocking call:

```
f.timeout(2 seconds)(x)
```

The replying process can also check whether the waiting process received the reply or timed out. This is implemented as a Boolean return value from a reply function  $r$ :

```
go { case f(_,r) => if (r("done")) ... }
```

The reply-sending call  $r()$  returns `true` only if the waiting process did not time out. This timeout-checking functionality may be sometimes required to avoid race conditions when using timeouts on synchronous channels.

### 2.4.3 Ordered channels

In JC, the message consumption order is not specified and may be chosen arbitrarily by the implementation. However, certain applications—notably, asynchronous streaming pipelines—require messages to be processed in the order sent. Because of the practical importance of asynchronous pipelines, it appears that an implementation of JC should have direct support for ordered channels. It would be too inefficient to implement ordered message processing on top of unordered JC by, say, attaching an extra timestamp to each message and enforcing message order via process guards.

Not all JC programs are compatible with ordered channel semantics. Consider the case of several processes waiting on the same channel with different conditions on the message value,

```
site( go { case c(0) + a(x) => ... },
      go { case c(y) if y>0 => ... } )
```

If the channel  $c$  is implemented as an ordered message queue, a message  $c(0)$  will block the queue and prevent any processes from running until a message  $a(x)$  becomes available so that  $c(0)$  may be consumed. This behavior is certainly undesirable. Programs like this one can achieve satisfactory performance only if processes are able to consume messages out of order.

Given a JC program, Chymyst automatically assigns ordered queues to *all* channels that could admit ordered semantics without adverse effects. Such channels are called “pipelined” in the Chymyst source code.

For a channel  $c$  to be pipelined, the processes waiting on  $c$  must be such that, at any time and for any message  $c(x)$  on the channel, one of the three statements is true:

1. the message  $c(x)$  can be consumed immediately by some process
2. it can be determined that  $c(x)$  will never be consumed by any process
3. at this time, no messages on channel  $c$  can be consumed at all

Equivalently, the condition for  $c(x)$  to be potentially consumable should be independent of the presence of other messages on this or other channels. If this holds for a channel  $c$ , a JC implementation can achieve the correct JC semantics by only inspecting a single message on the channel  $c$  when choosing new processes to run.

To formulate this condition more rigorously, consider that each of the defined processes  $P_1, \dots, P_n$  introduces a predicate  $p_i(x, a, b, \dots)$  that determines whether the process  $P_i$  can start and consume  $c(x)$ . These predicates depend on the presence of input messages and on their values. Here we denote by  $x$  the value of the message  $c(x)$  on channel  $c$ , while  $a, b, \dots$  are some variables that describe other channels on which the  $i$ -th process is waiting. The condition for *some* process consuming  $c(x)$  to start is

$$\bigvee_{i=1}^n p_i(x, a, b, \dots).$$

Chymyst uses static analysis to try factorizing this Boolean expression as

$$\bigvee_{i=1}^n p_i(x, a, b, \dots) = p_c(x) \wedge q(a, b, \dots)$$

for some predicates  $p_c$  and  $q$ . The factorized Boolean formula  $p_c(x) \wedge q(\dots)$  indicates that the condition  $p_c(x)$  for being able to consume  $c(x)$  is logically independent of any other messages that may be currently present. If this Boolean factorization exists, Chymyst marks  $c$  as a pipelined channel with the predicate  $p_c$ .

Another optimization for pipelined channels is that Chymyst will discard all messages  $c(x)$  for which  $p_c(x)$  does not hold: these messages will never be consumed by any processes.

### 2.4.4 Logging and unit testing

In the author’s experience, errors in JC programs are often due to messages that should have been sent but were not, and to erroneous process definitions that start processes where they should not have. A straightforward approach to debugging these situations is to examine the execution trace of a join definition. Such a trace would show the decisions made by the a join definition’s scheduler after each new process search, detailing whether a new process was scheduled, and what messages were present at that time on any channels.

Chymyst includes an event reporting facility that allows join definitions to write an execution trace to console or to memory. A design issue is how to indicate in the application code that a certain join definition needs to log its trace. In Chymyst, as in JoCaml, user code has no direct access either to join definitions themselves or to the message multisets held by join definitions. This is a useful encapsulation and safety feature. However, user code can assign a thread pool to each join definition, and thus it is convenient to attach an

event reporting interface to thread pool objects:

```
val tp = FixedPool(2).withReporter(reporter)
site(tp)( ... )
```

Frequently used event reporters are supplied by Chymyst, and application code can define custom event reporters. This facility is suitable for reporting errors and for gathering runtime performance metrics.

However, logging is inadequate as a means of verifying the desired functionality of a JC program. For this reason, Chymyst implements a special Future-based API that can be used for unit testing.

For any chosen channel, the unit testing API can be used to obtain a Scala Future that completes when:

- a message was sent on this channel
- process search has concluded and involved a message sent on this channel
- a message present on this channel was consumed by a new process

The process search event may have a positive outcome (a new process was scheduled, consuming this message) or a negative outcome (no new processes could be scheduled), in which case the Future will fail.

The unit testing API can be used to verify correctness of asynchronous process logic embodied in a JC program, without changing the program's code. For example, one could write a test that, sends a message on a certain channel, waits until that message is consumed, then waits until a new message is sent on another channel, and finally asserts a condition on the value of the new message.

The unit testing API also allows users to write tests that are robust against varying execution speed, since there is no need to wait for some arbitrarily chosen "reasonable" time before checking for conditions to hold.

The API just described applies to asynchronous channels. Synchronous channels block and thus generally do not require a special API for testing. However, a useful convenience is to be able to convert a blocking message call into a Future, which makes sending the message a non-blocking call. This facility is available as the `futureReply()` method on blocking channels.

#### 2.4.5 Process supervision

Since processes run on separate JVM threads, the application will not automatically receive a notification when a process throws an exception and fails. Chymyst supervises all processes and catches their run-time exceptions, reporting them to the user via the logging mechanism. Additionally, Chymyst will warn the user if a process fails to reply to a synchronous channel.

If it is expected that an exception will sometimes occur but the failure is transient, the user may specify an option

to retry running the process:

```
go { case ... }.withRetry
```

A more general supervision mechanism is to attach a second process that could recover from expected failures:

```
go { case ... }.onError(e => go { case ... })
```

The recovery process receives the same input messages as the failed process and also may use the exception value `e`. The recovery code will run on the same thread and can be visualized as a direct replacement for the failed process.

## 2.5 Pedagogical considerations

The choice of terminology and notation is important if we aim to explain an unfamiliar paradigm clearly and comprehensibly to newcomers. Here we again encounter difficulties when it comes to learning about join calculus.

The Wikipedia page on JC describes it as "*an asynchronous  $\pi$ -calculus with several strong restrictions: 1) Scope restriction, reception, and replicated reception are syntactically merged into a single construct, the definition; 2) Communication occurs only on defined names; 3) For every defined name there is exactly one replicated reception.*"<sup>10</sup>

Explanations using technical jargon such as "replicated reception" or "communication on defined names" are impenetrable for anyone not already well-versed in the concurrency research literature. Since Wikipedia is the most popular go-to resource for learning new concepts, it is quite understandable that software practitioners today remain unaware of join calculus.

Another obstacle for comprehending JC is that academic literature typically uses the terms "channel", "message", and "process", which are inherited from  $\pi$ -calculus but are not especially helpful for understanding how JC works and how to write concurrent programs in it.

Indeed, a channel in JC holds an *unordered* collection of messages, rather than an ordered queue or mailbox, as the word "channel" suggests. Another meaning of "channel" is a persistent path for exchanging messages between fixed locations, but this is far from the semantics of JC channels.

The phrase "sending a message" usually implies that a well-known recipient will consume messages one by one. But this is very different from what happens in JC, where a process may wait for several messages at once, different processes may contend on messages they wait for, and several copies of a process may start concurrently, consuming their input messages in non-deterministic order.

The word "process" suggests a fixed, persistent thread of computation with which we may communicate. However, JC does not have persistent threads; instead, processes are spawned on demand as input messages become available.

<sup>10</sup> See [en.wikipedia.org/wiki/Join-calculus](http://en.wikipedia.org/wiki/Join-calculus), as of June 2017.



While JoCaml is currently the sole well-maintained standard implementation of JC, its developer documentation<sup>11</sup> is especially confusing as regards the semantics of channels, messages, processes, and “spawning.” It is ill-suited as a pedagogical introduction either to using JoCaml or to join calculus. For example, the JoCaml manual does not clearly distinguish the notion of “spawning” a new process from the usage of the `spawn` keyword, which has a quite different semantics in JoCaml (sending ad-hoc messages on channels).

Instead of using academic JC terminology, I follow the chemical machine metaphor and terminology when giving tutorial presentations about Chymyst programming. With this approach, I have had success in conveying effectively both the basic concepts and the subtleties of JC semantics to developers who were previously unfamiliar with it.

## 2.6 From actors to reactions

Many Scala developers interested in concurrent programming are already familiar with the Actor model. In this subsection, I outline how the JC in its “chemical machine” presentation can be introduced to those developers.

In the Actor model, an actor receives messages and reacts to them by running a computation. An actor-based program declares several actors, defines the computations for them, stores references to the actors, and starts sending messages to the available actors. Messages are sent either synchronously or asynchronously, enabling communication between actors.

The chemical machine paradigm is in certain ways similar to the Actor model. A chemical program also consists of concurrent processes, or “chemical actors”, that communicate by sending messages. The chemical machine paradigm departs from the Actor model in two major ways:

1. Chemical actors are automatically started and stopped; the user’s code only sends messages and does not manipulate actor references.
2. Chemical actors may wait for a set of different messages to be received atomically at once.

If we examine these requirements and determine what should logically follow from them, we will arrive at the chemical machine paradigm.

The first requirement means that chemical actors are not created explicitly by the user’s program. Instead, the chemical machine runtime will automatically instantiate and run a chemical actor whenever some process sends a relevant input message. A chemical actor will be automatically stopped and deleted when its computation is finished. Therefore, the user’s code now does not create an instance of an actor but merely *defines the computation* that an auto-created actor will perform after consuming a message. As a consequence, chemical actors must be *stateless*, and their computations must be functions of the input message values.

Implementing this functionality will allow us to write pseudo-code like this,

```
val c1 = go { x: Int => ... }
c1 ! 123
```

The computation under the `go` keyword receives a message with an `Int` value and performs some processing on it. The computation will be instantiated and run concurrently, whenever a message is sent. In this way, we made the first step towards the chemical machine paradigm.

What could happen if we quickly send many messages?

```
val c1 = go { x: Int => ... }
(1 to 100).foreach(c1 ! _)
```

Since our computations are stateless, it is safe to run several instances of the computation `{ x: Int => ... }` concurrently. The runtime engine may automatically adjust the degree of parallelism depending on CPU load.

Note that `c1` is not a reference to a particular *instance* of the computation. Rather, the computation `{ x: Int => ... }` is merely a declarative description of what needs to be done with any message sent via `c1`. We could say that the value `c1` plays the role of a *label* attached to the value `123`. This label implies that the value `123` should be used as the input parameter `x` in a particular computation.

To express this semantics more clearly, let us change our pseudo-code notation to

```
go { x: Int from c1 => ... }
c1 ! 123
```

Different chemical actors are now distinguished only by their input message labels, for example:

```
go { x: Int from c1 => ... }
go { x: Int from d1 => ... }
c1 ! 123
d1 ! 456
```

Actor references have disappeared from the code. Instead, input message labels such as `c1`, `d1` select the computation that will be started.

The second requirement means that a chemical actor should be able to wait for, say, two messages at once, allowing us to write pseudo-code like this,

```
go { x: Int from c1, y: String from c2 => ... }
c1 ! 123
c2 ! "abc"
```

The two messages carry data of different types and are labeled by `c1` and `c2` respectively. The computation starts only after *both* messages have been sent, and consumes both messages atomically.

<sup>11</sup> See [jocaml.inria.fr/doc/index.html](http://jocaml.inria.fr/doc/index.html).

It follows that messages cannot be sent to a linearly ordered queue, as they will be consumed in an unknown order. Instead, messages must be kept in an unordered bag.

It also follows from the atomicity requirement that several computations could *jointly contend* on input messages:

```
go { x: Int from c1, y: String from c2 => ... }
go { x: Int from c1, z: Unit from e1 => ... }
```

Messages that carry data are now completely decoupled from computations that consume the data. All computations start concurrently whenever their input messages become available. The runtime engine needs to resolve message contention by making a non-deterministic choice of the messages that will be actually consumed. Among the several contending computations, only one will be actually started.

This concludes the second and final step towards the chemical machine paradigm, where “chemical actors” are called reactions and “messages” are called molecules.

It remains to use the actual Scala syntax instead of pseudo-code. In Scala, we need to declare message types explicitly and to register reactions with the runtime engine as a separate step. The syntax used by Chymyst looks like this:

```
val c1 = m[Int]
val c2 = m[String]
site(go { case c1(x) + c2(y) => ... })
c1(123); c2("abc")
```

It is useful to summarize the ways in which the chemical machine paradigm departs from the Actor model.

Whenever there are sufficiently many input messages available for processing, the chemical machine will automatically instantiate several concurrent copies of the same reaction that will consume the input messages concurrently. The application code does not need to specify how many parallel processes to run at any given time. This helps the programmer focus on the application logic at a higher level, because it is easier to reason about data than about processes.

Since reactions are stateless and instantiated automatically on demand, the application code does not manipulate explicit actor references, which is error-prone.<sup>12</sup> The application code also does not need to implement actor lifecycle management, actor hierarchies, backup and recovery of actors’ internal state, or deal with the special “dead letter” actor. This removes a significant amount of complexity from the architecture of concurrent applications.

<sup>12</sup> For example, books on Akka routinely warn against calling `sender()` in a `Future`, which may yield an incorrect actor reference when the `Future` is resolved.

### 3 Status and future roadmap

Chymyst is an Apache-licensed open-source project published to the Maven repository, currently at version 0.2.1. Extensive tutorial and developer documentation is already available and is being continually expanded. Tutorial examples include `map/reduce`, `merge-sort`, and Conway’s “Game of Life”, as well as a number of standard concurrency problems such as the “dining philosophers” and the “readers/writers.” The Chymyst development workflow follows industry-standard practices such as test-driven development and continuous integration. Unit tests exercise all features of the DSL (with 100% code coverage). Timings of the process scheduling code indicate that the overhead for starting a new process after sending a message can be as short as 10  $\mu$ s on a 2GHz laptop.

Features on the Chymyst development roadmap include:

- a standard library of JC definitions, to implement various concurrency patterns
- reporting of JVM thread performance metrics via JMX
- bindings for GUI toolkits such as JavaFX/ScalaFX
- possible support for Scala.js (with no multithreading)
- compile-time code transformations for performance optimization
- use of consensus algorithms to support distributed and remote execution

Implementing these features will create a robust, full-featured, and well-tested Scala implementation of join calculus that is viable for industry adoption.

### References

- Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. 2002. Cooperative Task Management Without Manual Stack Management. *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference* (2002).
- Alberto Baragatti, Roberto Bruni, Hernán Melgratti, Ugo Montanari, and Giorgio Spagnolo. 2007. Prototype Platforms for Distributed Agreements. *Electronic Notes in Theoretical Computer Science* 180, 2 (2007), 21 – 40. Proceedings of the Third International Workshop on Foundations of Coordination Languages and Software Architectures (FOCLASA 2004).
- Nick Benton, Luca Cardelli, and Cédric Fournet. 2002. Modern Concurrency Abstractions for C#. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP '02)*. Springer-Verlag, London, UK, 415–440.
- Robert Boyle. 1661. *The Sceptical Chymist*. J. Cadwell, London.
- Cédric Fournet and Georges Gonthier. 1996. The reflexive CHAM and the join-calculus. In *In Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*. ACM Press, 372–385.
- Cédric Fournet and Georges Gonthier. 2000. *The Join Calculus: a Language for Distributed Mobile Programming*. Vol. Applied Semantics. International summer school, APPSEM 2000. Microsoft Research. 268–332 pages.
- Cédric Fournet, Fabrice Le Fessant, Luc Maranget, and Alan Schmitt. 2003. JoCaml: A Language for Concurrent Distributed and Mobile Programming. In *Advanced Functional Programming: 4th International School, AFP 2002, Oxford, UK, August 19-24, 2002. Revised Lectures*. Springer Verlag, Berlin, Heidelberg, 129–158.
- Philipp Haller and Tom Van Cutsem. 2008. Implementing Joins Using Extensible Pattern Matching. In *Proceedings of the 10th International*

- Conference on Coordination Models and Languages, COORDINATION'08 (LNCS), Vol. 5052. Springer-Verlag, Berlin, Heidelberg, 135–152.
- Kevin Hammond and Greg Michaelson. 2004. *The Design of Hume: A High-Level Language for the Real-Time Embedded Systems Domain*. Springer Verlag, Berlin, Heidelberg, 127–142.
- Jiansen He. 2014. *Type-parameterized actors and their supervision*. Master's thesis. University of Edinburgh.
- Robin Milner. 1999. *Communicating and mobile systems: the Pi-calculus*. Cambridge University Press.
- Martin Odersky. 2000. Functional Nets. In *Proc. European Symposium on Programming (LNCS)*. Springer Verlag, 1–25.
- Peter-Michael Osera. 2005. *Join Diesel: Concurrency primitives for Diesel*. undergraduate research thesis. University of Washington.
- Tomas Petricek and Don Syme. 2011. *Joinads: A Retargetable Control-Flow Construct for Reactive, Parallel and Concurrent Programming*. Springer Verlag, Berlin, Heidelberg, 205–219.
- Hubert Plociniczak and Susan Eisenbach. 2010. JErLang: Erlang with Joins. In *Coordination Models and Languages: 12th International Conference, COORDINATION 2010, Amsterdam, The Netherlands, June 7-9, 2010. Proceedings*. Springer Verlag, Berlin, Heidelberg, 61–75.
- Raymond Roostenburg, Rob Bakker, , and Rob Williams. 2016. *Akka in Action*. Manning.
- Tiark Rompf. 2007. *Design and Implementation of a Programming Language for Concurrent Interactive Systems*. Master's thesis. University of Lübeck.
- Tiark Rompf. 2012. *Lightweight Modular Staging and Embedded Compilers: Abstraction Without Regret for High-Level High-Performance Programming*. Ph.D. Dissertation. LAMP, EPFL.
- Claudio Russo. 2007. *The Joins Concurrency Library*. Springer Verlag, Berlin, Heidelberg, 260–274.
- C.A. Varela and G. Agha. 2013. *Programming Distributed Computing Systems: A Foundational Approach*. MIT Press.
- G. Stewart von Itzstein. 2004. *Introduction of high level concurrency semantics in object oriented languages*. Ph.D. Dissertation. University of South Australia.