# Industry-strength join calculus: Declarative concurrent programming with `Chymyst`

Sergei Winitzki

June 23, 2017

**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. After a comparison of `Chymyst` 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 APIs for unit testing, performance monitoring, and fault tolerance, and are next steps on the `Chymyst` project's development roadmap.

## 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 (JC) [4], which can be seen as a DSL (domain-specific language) for high-level, 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 advantage of this high-level and type-safe concurrency paradigm. The message-passing idiom is particularly suitable for implementing distributed algorithms (see, e.g., [1]). Nevertheless, there appears to be no practical adoption of JC by the software industry.[1] Perhaps not coincidentally, there are very few open-source imple-

[1] A Google search yields several academic projects but no mentions of industrial JC use.

mentations of JC available for download and use. The only fully maintained implementation of JC is the JoCaml language [6].

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[2], the original authors' introduction to JC [5], and the lecture notes [6] were intended for graduate students in computer science 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-ready frameworks must provide a number of important facilities such as integration with legacy asynchronous interfaces, fault tolerance and process supervision, performance tuning and performance metrics, or features for unit testing, to name just a few. Unfortunately, academic presentations of JC usually 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 industry acceptance of JC.

In this paper, I present a new open-source implementation of JC as a library called `Chymyst`,[3] providing an embedded Scala DSL and a light-weight runtime. 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, or unit testing APIs. Finally, the `Chymyst` project offers tutorial documentation adapted to the software developer audience. In these ways, I hope to enable industry adoption of this promising 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 DSL in Scala.

`Chymyst` lifts several restrictions that are present in most other JC projects, and offers some additional features:

- separate definition of channel names and processes
- arbitrary non-linear patterns and guards in process definitions
- synchronous rendezvous
- first-class processes with arbitrary process bodies
- incremental construction of join definitions from processes
- automatic performance optimizations

[2] See jocaml.inria.fr/doc/index.html.

[3] The name is borrowed from the early treatise [3] by Robert Boyle, who was one of the founders of the science of chemistry.

- static code analysis and early error detection

I argue that certain new facilities need to be added to a JC implementation in order to make it viable for industry adoption. These facilities include:

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

I outline the ways these facilities can be embedded in the JC paradigm and describe their current implementation in `Chymyst`.

To better explain the concepts of JC to new software developers, I avoid the traditional academic terminology (message / channel / process / join definition). 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, I rely on the "abstract chemical machine" metaphor and use the corresponding terminology (molecule / emitter / reaction / reaction site), which is more visual and intuitive. Synchronous channels ("blocking emitters") are most easily understood by carrying out a continuation-passing code transformation from blocking code to code that uses only asynchronous, non-blocking channels. The syntax used by `Chymyst` for blocking channels 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 academic researchers, typically by creating an entirely new JC-based programming language or by patching an existing language. It is hard to assess the scope and practical use of these implementations, since most of them appear to be proof-of-concept projects developed to accompany academic publications.

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 [6], and remains today the best-supported one. This implementation is a patch for the OCaml compiler, which is fully compatible with the OCaml library ecosystem.

M. Odersky created a new language called "Funnel", based on the JC paradigm [10]. 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 in the language itself or in its standard libraries.

G. S. von Itzstein implemented JC as a patch for the early Java compiler [19]. The "Join Java" project appears to be abandoned.

The first appearance of JC in Scala was a "Join in Scala" compiler patch by V. Cremet (2003).[4] The syntax of Scala 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 or asynchronous continuations [15]. The project appears to be abandoned, as T. Rompf moved on to research on multi-stage compilation [16].

"Joinads" is a set of compiler patches for F# and Haskell, developed by T. Petricek [12]. The project is not maintained.

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# [2], C$\omega$ [17], Join Diesel [11], JErlang [13], and Hume [8] 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 for JC 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 [17]. The library appears to be unsupported.[5]

Y. Liu implemented the basic JC primitives in 2007-2009 as part of the C++ `Boost` library.[6]

In 2013, the present author created experimental JC prototypes for Objective-C on iOS and for Java on Android[7]; these projects are unmaintained. However, the present `Chymyst` implementation reuses some design decisions made in these earlier projects.

In 2014, S. Yallop implemented "Join Language" as a DSL embedded in Haskell.[8] The implementation uses advanced features of Haskell's type system to provide a concise syntax for join definitions.

The first embedding of JC as a Scala DSL was P. Haller's "Scala Joins" library [7]. Thereafter, J. He improved upon "Scala Joins" by streamlining the syntax, removing restrictions on pattern matching, and implementing remote processes [9]. `Chymyst` is a further development of P. Haller and J. He's syntax for embedding JC into Scala.

C. Russo's library [17] allowed synchronous rendezvous in join patterns as well as incremental construction of processes and join definitions, while

[4] See lampwww.epfl.ch/∼cremet/misc/join_in_scala/.
[5] See github.com/JoinPatterns/ScalableJoins.
[6] See channel.sourceforge.net.
[7] See github.com/winitzki.
[8] See github.com/syallop/Join-Language.

T. Rompf's language [15] 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`

In my experience, the absolute majority of software developers are unfamiliar with join calculus or its "channel" / "message" / "process" terminology, but the majority of Scala concurrency practitioners know about the Actor model. 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 Abstract Chemical Machine ("emitter" / "molecule" / "reaction").

The next subsection is a brief introduction to programming in join calculus using `Chymyst`. This introduction is suitable for readers not already familiar with either join calculus or the Actor model.

## 2.1 The chemical metaphor for concurrency

Neither of the words "join" and "calculus" are particularly explanatory or visually suggestive. The "chemical machine", on the other hand, is a visually concrete execution model that can be directly used for designing and reasoning about a JC program. This subsection is a brief overview of JC as seen through the chemical machine metaphor, in order to establish terminology.

To begin, one imagines a **reaction site**, i.e. a virtual place where many molecules are floating around and reacting with each other. Each molecule has a chemical designation (such as `a`, `b`, `c`) and also *carries a value* of a fixed type. 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 result in creating new **molecule emitters** `c` and `t`. Emitters can be seen as functions that are called in order to **emit** the corresponding molecules into the reaction site:

```
c(List(1,2,3))
t()
```

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

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()`:

```
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 output molecule, `c`, with a computed new value `xs`. The effect of this reaction is to compute the tail of a list. Due to the pattern-matching requirement, this reaction will start only when the molecule `c` carries a non-empty list value.

In `Chymyst`, reaction definitions can use all features of Scala partial functions, including arbitrary guard conditions and pattern matching on molecule values, and include arbitrary Scala code within the function body.

Reactions are first-class values:

```
val r1 = go { case t(_) ⇒ ???  }
```

Creating the reaction value `r1` does not actually schedule 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, such as `site(r1, r2)`, which activates the reactions listed in its arguments. A reaction site typically includes several reactions that may be declared inline for brevity:

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

Once a reaction site is created, the code can emit any number of molecules, such as `t` or `c`, or molecules bound to other reaction sites. The chemical machine will interpret the declared "chemical laws" and start reactions whenever appropriate input molecules are available at each reaction site. According to the operational semantics of JC, any number of different reactions may start concurrently if sufficiently many input molecules are available.

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 completely asynchronous, non-blocking programming style, a continuation can be used in order to simulate waiting until some concurrent computations are 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 "scope ripping," which disrupts the normal code flow.

The JoCaml implementation of JC uses synchronous channels to mitigate this problem. `Chymyst` also implements synchronous channels as a language feature. `Chymyst's` chosen syntax for synchronous channels resembles continuation-passing style and looks like this:

```
go { case a(x) + f(y, cont) ⇒ cont(x+y) }
```

The same process is defined in JoCaml as

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

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.

Here is a more extended example of a `Chymyst` process with a synchronous channel:

```
val f = b[Int, Int]
site( go { case f(x, cont) ⇒
 val y = ...  // compute some value
 cont(y) // reply, i.e. invoke continuation
} )
val z = f(123) // wait for reply
```

Note that the message `f(123)` is sent without specifying the continuation argument `cont`, which is nevertheless available in the reaction body. The semantics of the blocking call `f(123)` is essentially to construct the current continuation as `cont` and to send it to the process as a message on the channel `f`, together with the ordinary payload `123`.

As I will show below in Sec. 2.2.3, the `Chymyst` 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) ⇒ ...`" 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. Unfortunately, implicit declaration of new channels is not possible in `Chymyst` because Scala macros do not allow us to insert a new top-level symbol declaration into the code. So, channel declarations need to be explicit ("`val a = m[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 user-specified number of new channels and to define processes for them incrementally (see Sec. 2.2.4 below), removing 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 may create 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.

`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. 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) ⇒ a(x+y+z) }
```

A linear-pattern JC implementation, such as JoCaml, would require cumbersome auxiliary definitions to implement this functionality.

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** where two distinct users of the same channel will 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 users to specify the copy of `f()` to which the reply is sent. JoCaml can express this behavior only at the cost of using two auxiliary channels and an additional process with 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. [18], 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 and fork 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()` macro 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, for instance, that the process waits on the channels `t` and `c`, that the channel `t` may have arbitrary message values while `c` requires a pattern match, and that the process will finally send a message on channel `c` but not on `t`.

Using this information, `Chymyst` can then detect many cases of unavoidable livelock, deadlock, and non-determinism in user code. 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 single message is sent on channel `c`. This `Chymyst` code generates a *compile-time* error, with a message indicating unavoidable livelock.

Deadlocks can only happen when using synchronous channels and are harder to detect reliably. A deadlock warning is given when the process sends a synchronous message followed by another message that is consumed together with

the synchronous one:

```
go { case a(_) + c(x) + f(_, r) ⇒ c(f() + 1); r(x) }
```

This code is suspicious because the process waits for a reply to `f()` and *then* sends `c()`, while a reply to `f()` happens only *after* both `f()` and `c()` are sent.

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

```
site( go { case i(_) + c(x) ⇒ c(x + 1) },
      go { case i(_) ⇒ 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 values of the `i()` 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. Scala macros are used to determine whether process definitions impose any guard conditions on input message values. If not, a quicker scheduling algorithm can be used. Additionally, complicated guard conditions such as

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

are converted into the conjuctive normal form and split between molecules if possible. For instance, the example above is converted to (pseudo-code)

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

In many cases, this transformation allows the runtime engine to select message values faster, without enumerating all combinations of message values in a search for suitable inputs for a process.

## 2.3 Implementation details

`Chymyst` implements each join definition as a separate instance of class `ReactionSite`. A reaction site is visualized as 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 *single* 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 see the join definition object and cannot manipulate the contents of the 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 channel's join definition. At the same time, a process search action is queued to be run on the scheduler thread of that join definition. A process search action 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, while 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. `Chymyst` guarantees that processes declared in separate join definitions will be scheduled concurrently, on independent threads.

## 2.4 What it means to be "industry ready"***

Rather than try to guess what feature set constitutes "industry readiness", I have reviewed the widely used Akka actor framework as described in [14] and other similar books intended for sofware engineering audiences. These books suggest that the industry may expect a concurrency framework to implement features such as:

- logging facilities for execution tracing and for performance metrics
- support for unit testing with assertions about asynchronous behavior of messages
- fault tolerance, various scenarios of recovery from errors, per-process supervision
- timers and timeouts for blocking messages
- 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

- message pipelining (ordered vs. unordered mailboxes) for improved performance
- interoperability with other asynchronous APIs such as `Future's`, Actors, Tasks, and a standard 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[9], however the burden of implementing a robust distributed deployment environment rests on the application code. There does not appear to be any JoCaml-based industrial implementations of any distributed applications.

***

I will now describe the features that `Chymyst` currently implements.

### 2.4.1 Thread pools

To enable users to fine-tune the performance of their concurrent applications, `Chymyst` offers control over the thread pools used by various join definitions. There is a default thread pool available, but users may create other thread pools and assign them to join definitions. There are 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.

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
```

### 2.4.2 Time-outs

When a synchronous message is sent, such as `f()`, the emitting process is blocked until a reply is received. In `Chymyst`, it is possible to specify a time-out for this

[9] See jocaml.inria.fr/doc/distributed.html.

blocking call:

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

A process that sends the reply can also check whether the waiting process received the reply value or timed out. This is implemented as a `Boolean` return value from the reply emitter:

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

The timeout-checking functionality may be sometimes required to avoid race conditions when using timeouts on synchronous channels.

## 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[10] 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.*"

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 (deservedly or not) is a popular go-to resource for learning new concepts, it is quite understandable that software practitioners today remain unaware of join calculus even 20+ years after its invention.

Another obstacle for comprehending JC is that academic literature typically uses terms such as "channel", "message", and "process", which are inherited from $\pi$-calculus but are not 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 what a JC "channel" actually does.

The phrase "sending a message" usually implies that a fixed recipient will consume the sent 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 several "messages" they wait for, and several copies of a "process" may start concurrently, consuming their input "messages" in random order.

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

[10] See en.wikipedia.org/wiki/Join-calculus, as of December 2016.

While JoCaml remains today the only well-maintained standard implementation of JC, its developer documentation[11] 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 mixes the `spawn` keyword used for sending messages with the notion of "spawning" a new process, which has a quite different semantics in JC.

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 chemical machine paradigm 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 some of the actors. Messages are sent either synchronously or asynchronously, enabling communication between different concurrent 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.

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.

[11] See jocaml.inria.fr/doc/index.html.

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

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

The computation labeled as `c1` 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 full chemical machine paradigm.

What should 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 `c1` 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 a computation. Rather, the computation `{ x: Int ⇒ ... }` is being defined *declaratively*, as a 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`. The 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 are 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.

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

It also follows from the atomicity requirement that we may define several computations that *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. It remains to use the Scala syntax instead of pseudo-code.

In Scala, we need to declare message types explicitly and to register chemical computations 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")
```

As we have just seen, the chemical machine paradigm is a radical departure from the Actor model:

- Whenever there are sufficiently many input messages available for processing, the runtime engine may automatically instantiate several concurrent copies of the same computation that will consume the input messages concurrently. This is the main method for achieving parallelism in the chemical paradigm. The runtime engine is in the best position to optimize the CPU load using low-level OS threads. The application code does not need to decide how many concurrent actors to instantiate at any given time.

- Since chemical actors are stateless and instantiated automatically on demand, users do not need to implement actor lifecycle management, actor supervision hierarchies, backup and recovery of actors' internal state, or a special "dead letter" actor. This removes a significant amount of complexity from the architecture of concurrent applications.

- Input message contention is used in the chemical machine paradigm as a general mechanism for synchronization and mutual exclusion. (In the Actor model, these features are implemented by creating a fixed number

of actor *instances* that alone can consume certain messages.) Since the runtime engine will arbitrarily decide which actor to run, input contention will result in nondeterminism. This is quite similar to the nondeterminism in the usual models of concurrent programming. For example, mutual exclusion allows the programmer to implement safe exclusive access to a resource for any number of concurrent processes, but the order of access among the contending processes remains unspecified.

In the chemical machine paradigm, "chemical actor" computations are called **reactions**, their input messages are **input molecules**, messages sent by a chemical computation are **output molecules** of the reaction, while input message labels are **molecule emitters**.

In the academic literature, chemical computations are called "processes" and input message labels are "channels" or "channel names".

# 3 Future roadmap

# References


[1] Baragatti, A., Bruni, R., Melgratti, H., Montanari, U., and Spagnolo, G. 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).

[2] Benton, N., Cardelli, L., and Fournet, C. Modern concurrency abstractions for c#. In *Proceedings of the 16th European Conference on Object-Oriented Programming* (London, UK, UK, 2002), ECOOP '02, Springer-Verlag, pp. 415–440.

[3] Boyle, R. *The Sceptical Chymyst.* J. Cadwell, London, 1661.

[4] Fournet, C., and Gonthier, G. The reflexive cham and the join-calculus. In *IN PROCEEDINGS OF THE 23RD ACM SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES* (1996), ACM Press, pp. 372–385.

[5] Fournet, C., and Gonthier, G. *The Join Calculus: a Language for Distributed Mobile Programming*, vol. Applied Semantics. International summer school, APPSEM 2000. Microsoft Research, 2000.

[6] Fournet, C., Le Fessant, F., Maranget, L., and Schmitt, A. *JoCaml: A Language for Concurrent Distributed and Mobile Programming.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2003, pp. 129–158.

[7] Haller, P., and Van Cutsem, T. Implementing joins using extensible pattern matching. In *Proceedings of the 10th International Conference on*

*Coordination Models and Languages* (Berlin, Heidelberg, 2008), vol. Lecture Notes in Computer Science 5052 of *COORDINATION'08*, Springer-Verlag, pp. 135–152.

[8] Hammond, K., and Michaelson, G. *The Design of Hume: A High-Level Language for the Real-Time Embedded Systems Domain.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 127–142.

[9] He, J. Type-parameterized actors and their supervision. Master's thesis, University of Edinburgh, 2014.

[10] Odersky, M. Functional nets. In *Proc. European Symposium on Programming* (2000), no. 1782 in LNCS, Springer Verlag, pp. 1–25.

[11] Osera, P.-M. Join diesel: Concurrency primitives for diesel. undergraduate research thesis, University of Washington, 2005.

[12] Petricek, T., and Syme, D. *Joinads: A Retargetable Control-Flow Construct for Reactive, Parallel and Concurrent Programming.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 205–219.

[13] Plociniczak, H., and Eisenbach, S. *JErlang: Erlang with Joins.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 61–75.

[14] Roestenburg, R., Bakker, R., , and Williams, R. *Akka in Action.* Manning, 2016.

[15] Rompf, T. Design and implementation of a programming language for concurrent interactive systems. Master's thesis, University of Lübeck, 2007.

[16] Rompf, T. *Lightweight Modular Staging and Embedded Compilers: Abstraction Without Regret for High-Level High-Performance Programming.* PhD thesis, LAMP, EPFL, 2012.

[17] Russo, C. *The Joins Concurrency Library.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 260–274.

[18] Varela, C., and Agha, G. *Programming Distributed Computing Systems: A Foundational Approach.* MIT Press, 2013.

[19] von Itzstein, G. S. *INTRODUCTION OF HIGH LEVEL CONCURRENCY SEMANTICS IN OBJECT ORIENTED LANGUAGES.* PhD thesis, University of South Australia, 2004.