Chorex: Restartable, Language-Integrated Choreographies

Ashton Wiersdorf[a] and Ben Greenman[a]

a University of Utah, Salt Lake City, UT, USA

Abstract We built Chorex, a language that brings choreographic programming to Elixir as a path toward robust distributed applications. Chorex is unique among choreographic languages because it tolerates failure among actors: when an actor crashes, Chorex spawns a new process, restores state using a checkpoint, and updates the network configuration for all actors. Chorex also proves that full-featured choreographies can be implemented via metaprogramming, and that doing so achieves tight integration with the host language. For example, mismatches between choreography requirements and an actor implementation are reported statically and in terms of source code rather than macro-expanded code. This paper illustrates Chorex on several examples, ranging from a higher-order bookseller to a secure remote password protocol, details its implementation, and measures the overhead of checkpointing. We conjecture that Chorex's projection strategy, which outputs sets of stateless functions, is a viable approach for other languages to support restartable actors.

ACM CCS 2012

- **Computing methodologies** → **Concurrent programming languages**;
- **Software and its engineering** → **Extensible languages**; *Compilers*;

Keywords choreographies, concurrency, metaprogramming, macros

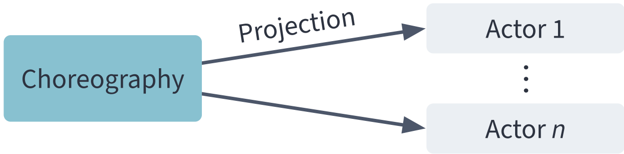
The Art, Science, and Engineering of Programming

Perspective The Engineering of Programming

Area of Submission Choreographic programming, Language implementation, Distributed systems

1 Introduction

Choreographic programming adds a layer of organization to concurrent or distributed systems. A choreographic language introduces a domain-specific notation for *choreographies*—programs that describe the interactions among actors in a system—and it *projects* each choreography to a set of local programs, one for each actor [8, 40, 41].



A choreography makes a global view of the system explicit as code, and is deeply connected to the actual behavior of actors. By contrast, in traditional distributed systems, the global view is merely a design document or a sketch on a whiteboard, and it is up to programmers to ensure that individual actors work together to realize the global protocol design. Actors can easily fall out of sync, as only end-to-end testing holds them together.

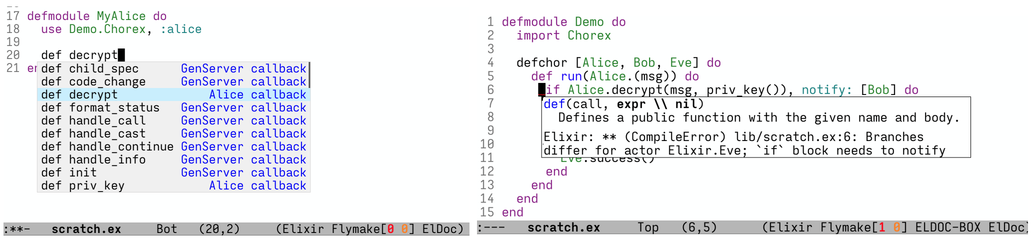
With choreographies, classes of communication errors become unrepresentable. Sends and receives cannot be mismatched because they are paired by design in a combined form: `send` ⇝ `recv`. Deadlocks cannot occur because lexical scope rules them out. For example, our language Chorex reports a compile-time error for the would-be deadlock below because the variable `A.val` is used before it is bound:

```
...
defchor [A, B, C] do # choreography for 3 actors
  def run() do
    A.val ~> B.val
    B.val ~> C.val
    C.val ~> A.val
end
```

```
ERROR: undefined variable "val"
|
|     A.val ~> B.val
|       ^^^
|
+- deadlock.exs:6 A.run/1
```

Languages that support choreographic programming are on the rise and quickly growing to support full-featured programs. Choral [29] brings choreographies to Java and recently added interoperability with legacy code to enable an Internet Relay Chat (IRC) implementation [39]. MultiChor, for Haskell, introduces a dynamic approach to projection that has been ported to Rust and TypeScript [4]. These and other implementations (surveyed in Section 6) have taken great strides toward practical choreographic programming. However, choreographic languages fail to address all of the seminal *eight fallacies of distributed computing* [55, 57]: (1) the network is reliable, (2) latency is zero, (3) bandwidth is infinite, (4) the network is secure, (5) topology doesn't change, (6) there is one administrator, (7) transport cost is zero, and (8) the network is homogeneous. Languages meant to support distributed programming must be robust against all of these critical issues.

In this paper, we present a language, Chorex, that addresses fallacy (5), *topology doesn't change*, through a novel projection strategy and runtime monitoring. Chorex



(a) Autocomplete in an actor lists functions required by the choreography.

(b) Missing knowledge-of-choice annotations lead to a static error during projection.

Figure 1 Examples of language integration in Chorex.

introduces try/rescue blocks to specify recovery behavior. During projection, every try block introduces code to checkpoint only the necessary state and to prevent actors from advancing to an un-recoverable position in the choreography. At runtime, a supervisor restarts actors as needed, restores checkpoint state, and shares the address of the new participant via out-of-band messages to other actors (who are, by design, prepared to receive such messages).

In the following minimal example, Alice crashes due to division by zero. After restarting, a new Alice is able to exchange messages with Bob:

```
try do
  Alice.f(1 / 0) ~> Bob.y
rescue
  Alice.f(1) ~> Bob.y
end
Alice.(2 + 2) ~> Bob.sum
Bob.(sum + sum) ~> Alice.result
Alice.result
```

Chorex brings choreographic programming to Elixir. The implementation is notable because it follows the *languages as libraries* [53] design method to achieve a high level of integration with the standard Elixir toolchain. Figure 1 presents two benefits of language integration:

- Figure 1a shows that functions required by a choreography appear as suggestions when a programmer edits an actor module. The pictured suggestion appears in the context of an actor module named `MyAlice` that fills the role named `:alice` from a choreography named `Demo.Chorex` (not pictured). There are two required functions: `decrypt` and `priv_key`.
- Figure 1b shows a tooltip box with a compile-time error due to missing annotations. This sort of error illustrated in Figure 1b is not detectable in choreographies implemented as runtime libraries because it requires two passes over the input [4, 47]. Either a bespoke compiler or Chorex-style metaprogramming is needed.

Both affordances are IDE-agnostic because they leverage the Elixir language server. Our IDE of choice happens to be Emacs, but Neovim or VSCode users would see similar tooltips.

Concretely, the Chorex compiler is an Elixir macro that analyzes source code, projects the code to actor implementations, and propagates source locations so that errors in the output point to useful source locations. Additional features of Chorex include first-class functions, out-of-order message receives, message transport over TCP, and shared state across different instantiations of the same choreography. Section 4 covers the implementation in detail.

Outline This paper begins with an example-driven tour of the Chorex language, including a TCP socket server and Secure Remote Password protocol (Section 2). These examples serve a dual purpose as an evaluation of Chorex, illustrating its expressiveness on realistic use-cases. Next, the paper presents the design of Chorex (Section 3), with emphasis on its novel support for restartable actors, and follows with a close look at how Chorex achieves beneficial language integration through metaprogramming (Section 4), and a performance evaluation of its `try`/`rescue` recovery mechanism (Section 5). The paper concludes with a landscape of the rapidly-evolving area of choreographic programming (Section 6) and a discussion of next steps (Section 7).

Data Availability Statement Chorex is open source and available on GitHub and the Elixir/Erlang package manager Hex. Links omitted for double-blind review. We plan to submit an artifact that contains the latest release of Chorex and code that substantiates all examples in this paper.

Notation For readability and to save space, code listings in this paper make two abuses of Elixir notation. First, they often omit `end` delimiters, which are required to close blocks opened by `def ... do` and other forms. Second, they omit parentheses around located variables, writing `A.val` rather than the preferred `A.(val)`, which cooperates better with the Elixir autoformatter. Refer to the artifact for runnable Elixir code.

2 Chorex By Example

Chorex is a domain-specific language for choreographic programming in Elixir. This section illustrates the basics of Chorex through several motivating examples.

Elixir [49, 51] is the chosen target language for several reasons. First, it compiles to the Beam VM (the Erlang virtual machine), and thus has access to primitives that support low-latency, distributed, fault-tolerant systems. These primitives have enabled fast prototyping of choreographic features. Second, Elixir has a large userbase to engage with in future work. Third, Elixir comes with a hygienic macro system. Thanks to macros, Chorex is implemented as a library and integrates smoothly with the Elixir build system, Mix [22], and package manager, Hex [21].

2.1 Socket Server

Our first example is a minimal socket server inspired by Thousand Island [52], a full-fledged socket server written in Elixir. There are two choreographies involved. A first choreography, between a Listener actor and an Acceptor actor, initializes a server that clients can connect to. A second choreography specifies interactions between a

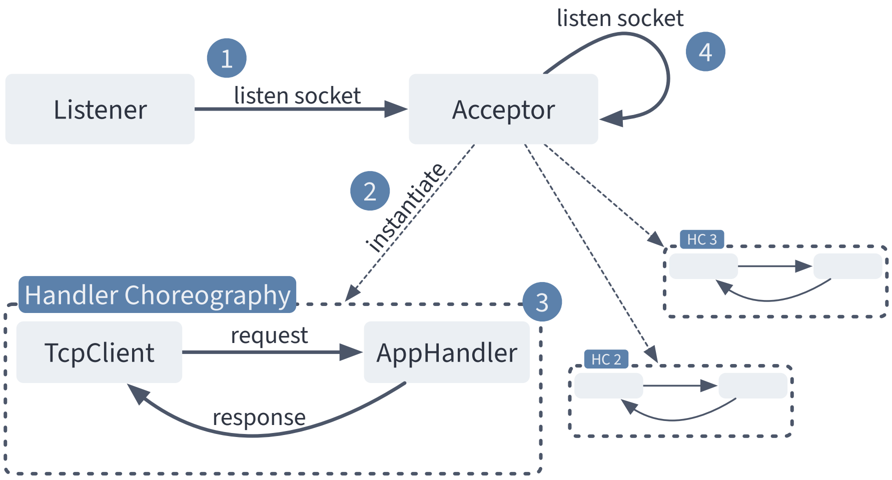


Figure 2 Socket server architecture: (1) Listener sends a socket to Acceptor; (2) Acceptor waits for a TCPClient, then spawns a Handler Choreography; (3) TCPClient and AppHandler exchange messages; (4) Acceptor listens for a next client.

Client actor and an AppHandler actor. Figure 2 maps out the overall design. Crucially, the top-level choreography is able to start multiple instances of the inner Handler Choreography. Every client that connects to the server gets forwarded to a unique choreography with an AppHandler actor.

Located Expressions. First, a note on syntax. A choreography cannot store state because it does not have a first-class representation at runtime. All state, such as program variables, must be stored on actors. The dot notation `Actor.x` reads from a variable `x` whose value is stored at the actor named `Actor`. A located expression such as `Actor.(2 + 2)` is to run on the named actor. Arbitrary Elixir expressions can appear after a dot, such as function calls.

Choreography Structure. The Chorex form `defchor` introduces a choreography. Each choreography must have a `run` function, which serves as an entry point. To run a choreography, callers invoke the helper function `Chorex.start`, which manages configuration and calls the matching `run` function. Choreographies may include additional functions, introduced by the `def` keyword.

Top-Level, Listener Choreography The listener choreography for our TCP server expects configuration data as input and immediately calls a helper function (located on the Listener actor) to acquire a socket connection. The Listener sends this socket to an Acceptor actor. At this point, the Listener's work is done. The choreography then enters a loop in which the Acceptor awaits and manages incoming connections.

```
defmodule Tcp.ListenerChor do
  import Chorex

  defchor [Listener, Acceptor] do
    def run(Listener.config) do
      Listener.get_listener_socket(config) ~> Acceptor.{:ok, socket}
      loop(Acceptor.socket)
```

```
    def loop(Acceptor.listen_socket) do
      Acceptor.accept_and_handle_connection(listen_socket)
      loop(Acceptor.listen_socket)
end
```

The following module is an implementation of the Acceptor actor. It imports the choreography above and, on the same line, specifies the actor role that it plans to implement (`:acceptor`). Inside the helper function, this Acceptor calls `Chorex.start` to invoke a choreography named `Tcp.HandlerChor`. The other arguments to `Chorex.start` are a map from actor roles to module names and a list of arguments to the HandlerChor's `run` function. In this case, the only argument is the socket:

```
defmodule Tcp.AcceptorImpl do
  use Tcp.ListenerChor.Chorex, :acceptor

  @impl true
  def accept_and_handle_connection(listen_socket) do
    {:ok, socket} = :gen_tcp.accept(listen_socket)
    Chorex.start(
      Tcp.HandlerChor.Chorex,
      %{Handler => Tcp.HandlerImpl, TcpClient => Tcp.ClientImpl},
      [socket]
    )
end
```

If anything goes wrong at runtime, the Acceptor will exit cleanly and bring the Listener down as well. This exit behavior comes out of the box with Chorex.

Inner, Handler Choreography The second choreography describes an interactive loop. First, the AppHandler actor initializes a dictionary to track the total bytes sent by the client. Inside the loop, the Client sends a message to the AppHandler and the AppHandler updates its state, decides whether to continue, and sends a reply:

```
defmodule Tcp.HandlerChor do
  import Chorex

  defchor [AppHandler, TcpClient] do
    def run(TcpClient.sock) do
      loop(AppHandler.(%{byte_count: 0}), TcpClient.sock)

    def loop(AppHandler.state, TcpClient.sock) do
      TcpClient.read(sock) ~> AppHandler.msg
      with AppHandler.{resp, st2} <- AppHandler.run(msg, state) do
        AppHandler.fmt_reply(resp) ~> TcpClient.resp
        TcpClient.send_over_socket(sock, resp)
        if AppHandler.continue?(resp, st2) do
          loop(AppHandler.st2, TcpClient.sock)
        else
          TcpClient.shutdown(sock)
          AppHandler.ack_shutdown()
end
```

This choreography uses a Chorex `if` expression. In it, one actor (AppHandler) makes a choice that determines the future of the conversation. The other actor thus needs *knowledge of choice* to decide which branch to take in its own code. By default, Chorex sends a message to every other actor in a choreography whenever an `if` branch is taken. To restrict to a subset of actors, programmers can insert a `notify:` annotation. Chorex statically rejects notify annotations that list too few participants (Section 3.1).

2.2 Secure Remote Password

Secure Remote Password (SRP) [58] is an authentication method based on zero-knowledge-proofs [30]. Our Chorex implementation drives a simple command-line application that lets one user register a password and then serves login requests:

```
iex(1)> ZkpLogin.register_srp()
  [New User SRP] username: alice
  [New User SRP] password: bob
  Server responds {:registered, "alice"}
  Client responds :registered
  :ok

iex(2)> ZkpLogin.login_srp()
  [Login] username: alice
  [Login] password: notbob
  Server responds {:fail, :reject_client_digest}
  Client responds {:fail, :server_rejected_digest}
```

The choreography has two `run` functions corresponding to the registration and login phases. Chorex distinguishes these functions by their arity. The registration function expects a username and password located at the client, and a `:register` token located at the server. The login function expects no input at either actor. The code below shows the registration `run` function:

```
defmodule Zkp.SrpChor do
  import Chorex

  defchor [SrpServer, SrpClient] do
    def run(SrpClient.{uname, pwd}, SrpServer.(:register)) do # register
      SrpServer.get_params() ~> SrpClient.{salt, g, n}
      with SrpClient.v <- SrpClient.gen_token(uname,pwd,salt,g,n) do
        SrpClient.{uname, salt, v} ~> SrpServer.{uname, salt, v}
        if SrpServer.register(uname, salt, v) do
          SrpServer.{:registered, uname}
          SrpClient.(:registered)
        else
          SrpServer.({:error, :no_registration, uname})
          SrpClient.({:error, :no_registration})

    def run() do ... # login
end
```

An important property of SRP is that secret information, such as the password on the client or the the value b on the server, never get transmitted to the other. This property is straightforward to verify by inspecting the choreography and confirming that secret located values, e.g. `SrpClient.pwd`, are not sent to another actor. In a traditional distributed SRP implementation, such as srp-elixir [46], this sort of property is harder to establish because communication logic is split across multiple files.

1. Client sends user_id to server.
2. Server finds user's salt s and auth key v, uses constants g, N to compute $k = hash(g, N)$, generates random secret b, computes $B = kv + g^b$, and sends g, N, s, B to client.
3. Client generates random secret a and computes several values, including a session key:
$$\begin{aligned} A &= g^a \\ k &= hash(g, N) \\ u &= hash(A, B) \\ K &= (B - kg^x)^{a+ux} \\ M_1 &= hash(A, B, K) \end{aligned}$$
Client sends A, M_1 to server.
4. Server computes $K = (Av^u)^b$ and checks that $hash(A, B, K) = M_1$. Server sends $M_2 = hash(A, M_1, K)$ to confirm session key.

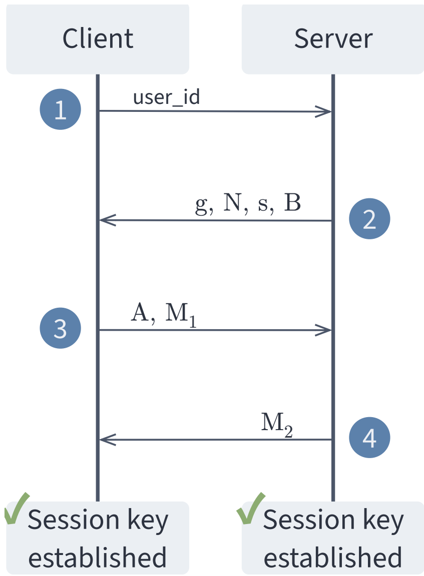


Figure 3 Secure Remote Password login protocol to find key K.

During the login phase of SRP, there are several rounds of communication that take place between server and client to establish a session key K. The choreographic function that implements these exchanges is roughly twice as long as the registration function yet uses similar language features (with, ~>, if), so we defer it to the artifact. It is enough to say that comparing the choreography against a high-level algorithm description, shown in Figure 3, is straightforward. For example, the login choreography (in artifact) has exactly four communication terms (~>), matching the figure.

2.3 Discrete Logarithm

As a third example, we have implemented a zero-knowledge proof protocol to convince a verifier that the prover knows the logarithm of a number in a finite field. The choreography is approximately 70 lines long and is included in the artifact.

The choreography uses two closely-knit helper functions: one that loops through several rounds of challenge and verification, and another that handles the logic of a single round. Compiling these functions to a core language that supports restarts was a key milestone in the development of Chorex.

2.4 More Examples: Higher-Order, Out-of-Order, Persistent State

Chorex has several other features inspired by prior work on choreographies. For one, it supports *first-class choreographic functions*. The program in Figure 4, adapted from the Pirouette paper [32], has a run function that executes either a one-buyer or two-buyer bookseller scenario by passing one function to another function. Both scenarios are inspired by the session types literature [7, 33].

```
defchor [Buyer, Contributor, Seller] do
  def run(Buyer.include_contributions?) do
    if Buyer.include_contributions? do
      bookseller(@two_party/1)
    else
      bookseller(@one_party/1)

  def bookseller(f) do
    Buyer.get_book_title() ~> Seller.the_book
    with Buyer.decision <- f.(Seller.get_price("book:" <> the_book)) do
      if Buyer.decision do
        Buyer.get_address() ~> Seller.the_address
        Seller.get_delivery_date(the_book, the_address) ~> Buyer.d_date
        Buyer.d_date
      else
        Buyer.nil

  def one_party(Seller.the_price) do
    Seller.the_price ~> Buyer.p
    Buyer.(p < get_budget())

  def two_party(Seller.the_price) do
    Seller.the_price ~> Buyer.p
    Seller.the_price ~> Contributor.p
    Contributor.compute_contrib(p) ~> Buyer.contrib
    Buyer.(p - contrib < get_budget())
end
```

Figure 4 Higher-order choreographic function for two classic bookseller scenarios.

References to functions, namely `@two_party/1` and `@one_party/1`, are prefixed with an `@` sign so that Chorex can increment the arity to account for an implicit argument representing the choreography state (Section 4.2), during compilation. This prefix is a slight twist on Elixir's standard `&`-sign prefix for function references. The suffix `/1` is standard for Elixir; it describes the arity of the function.

A second important feature is *out-of-order message receives*. In the following example, the two messages sent to `MainServer` arrive when they are ready. The first send does not block and the second, being data-independent, can run immediately:

```
defchor [KeyServer, MainServer, ContentServer, Client] do
  def run() do
    ContentServer.getText() ~> MainServer.txt # may arrive 2nd
    KeyServer.getKey() ~> MainServer.key      # may arrive 1st
    ...
```

This feature is inspired by the Ozone language [42]. However, unlike the calculus that Ozone is based on (O_3), Chorex will not reorder two sends from the same actor, nor will it move expressions in or out of an `if` branch.

A third feature is *persistent state* across instances of a choreography. For example, one Seller can manage a bookstore that several buyers interact with concurrently. Similar to prior work on objects with multiple owners [15, 29], the Chorex approach is to decouple the state from the choreography itself, similar to using a database.

3 Elements of Chorex

Chorex is designed to advance research on choreographic programming by integrating with the rich Elixir/Erlang ecosystem for distributed systems. Key principles of the language design include the following:

- Maintain a smooth Elixir workflow by implementing the choreography language and projection through metaprogramming.
- Manage actors using a standard Elixir supervision tree.
- Provide custom mailboxes and control stacks for actors to handle out-of-order messages, messages from the supervisor, and recovery.

Furthermore, Chorex draws inspiration from Elixir syntax and programming idioms, which has shaped the "look and feel" of the language.

This section explains user-facing aspects of the language design, including how to write a choreography (Section 3.1), how to implement an actor (Section 3.2), and the framework for monitoring and supervision (Section 3.3).

3.1 The Choreography Language

To write a Chorex choreography, define a module, import Chorex, and use `defchor`:

```
defmodule SampleChor do
  import Chorex
  defchor [Alice, Bob, Carol] do
    ...
end
```

The `defchor` macro expects two input forms: a list of actor names (CamelCase, to match the Elixir convention for module names) and a block of code. It contains a sequence of function definitions (`def`). One function named `run` must be included; this function is the entry point to the choreography. No other definition forms are allowed. Each `def` projects to several variant functions, one for each actor. A `defchor` outputs a standard Elixir module named `Chorex` that provides code and an API for actors. Expanded code thus has the following shape:

```
defmodule SampleChor do
  import Chorex
  defmodule Chorex do
    ...
end
```

Other modules must refer to this macro-defined module (`SampleChor.Chorex`), either to implement an actor or to start the choreography. The module `Tcp.AcceptorImpl` from Section 2.1 illustrates both uses.

Variables in a choreography must be *located* at an actor, using the syntax `Actor.var` (used in this paper) or `Actor.(var)` (preferred in Chorex, because it cooperates with Elixir's `mix format` tool). This rule includes arguments to standard functions, as in the following header, which expects a field named `arg` at the actor `Alice` and another field, also named `arg`, at the actor `Bob`:

```
def some_function(Alice.arg, Bob.arg) do ...
```

There is one exception to the rule that every variable must be located. Function arguments to a higher-order function are not located (example in Section 2.4).

Elixir encourages the use of overloaded functions that each perform a specific job rather than functions that do one of several jobs after inspecting their input (via match, if, etc.). Chorex supports overloaded functions as well. The two run functions in the SRP choreography (Section 2.2) are one example. Cases in an overloaded function must have distinct signatures, either through the number of arguments or the pattern-matching shape of those arguments. Crucially, the case-distinction must hold after a choreographic function is projected to actor code. This is why the SRP registration function expects a token :register located on the Server:

```
defchor [SrpClient, SrpServer] do
  def run(SrpClient.{username, password}, SrpServer.(:register)) do ... # register
  def run() do ... # login
```

Without this extra placeholder argument, the projected code for SrpServer would contain two run functions that both expect zero arguments. Currently, this results in a Chorex runtime error. Adding static detection is future work.

Sending and Receiving Messages Arrow notation (send ⇝ recv) sends a value from one actor to another. In Chorex, the sender can prepare any located expression. Variables, function calls, arithmetic, and other expression forms are valid on the left side of a send. The receiver can use Elixir pattern matching to bind variables:

```
Alice.{:answer, 42} ~> Bob.{:answer, the_answer}
```

Conditionals and Knowledge of Choice Chorex repurposes if expressions from Elixir for choreographic conditionals (with one change: Chorex requires an else branch; multi-way conditionals are future work). In the following example, Alice is the *deciding actor* for this conditional, as the branch hinges on the result computed at Alice. The projection for Bob inserts a receive to wait for a *knowledge of choice* message to know which branch to take:

```
if Alice.make_decision() do
  Alice.yes_branch() ~> Bob.d1
  Bob.report(d1)
else
  Alice.no_branch() ~> Bob.d2
  Bob.report(d2)
end
```

Unlike prior work (e.g. [32]), an if in Chorex need not appear in tail position.

Chorex does not (yet) infer the actors in a conditional, and thus by default shares knowledge of choice with *every other actor* in the choreography. To limit the notified actors, a programmer can add a notify: annotation:

```
if Alice.make_decision(),
  notify: [Bob, Carol] do ...
```

When a notify: fails to include all necessary actors, Chorex raises a compile-time error as it projects code for each actor. Below, Carol is missing a notify:

```
defchor [Alice, Bob, Carol] do
  def run(Alice.msg) do
    if Alice.decrypt(msg, priv_key()), notify: [Bob] do
      Bob.notify_success()
      Carol.foiled()
    else
      Bob.notify_failure()
      Carol.success()
```

The error output explains the problem with an accurate line number:

```
== Compilation error in file bad_branch.ex ==
** (CompileError) bad_branch.ex:3: Branches differ for actor Elixir.Carol; `if' block needs to notify
```

Error Rescue and Restarts Chorex adapts Elixir's `try`/`rescue` blocks to handle errors that may arise in actor code.(Unlike in Elixir, `rescue` declares a block and not a match clause.) If `Alice` or `Bob` were to fail in the following `try` block, both actors would execute the `rescue` block with the failing actor restored as a new actor instance with recovered state:

```
try do
  Alice.dangerous_operation() ~> Bob.x
  Bob.success(x)
rescue
  Alice.safe_operation() ~> Bob.x
  Bob.fallback(x)
end
```

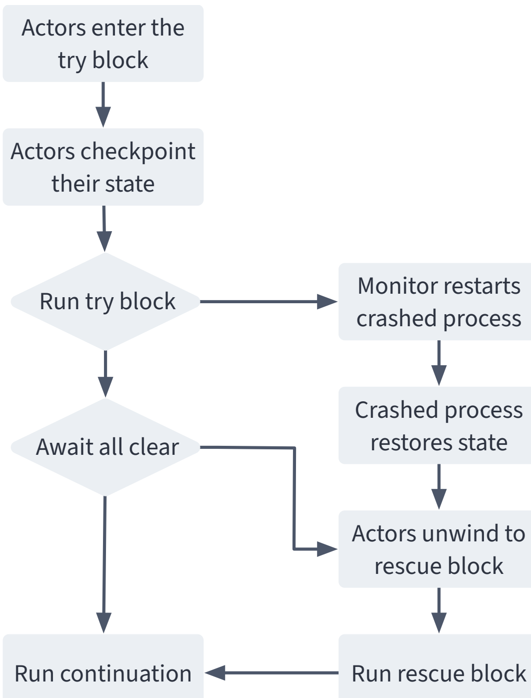


Figure 5 Chorex `try`/`rescue` logic.

Figure 5 presents a flowchart description of `try`/`rescue` semantics. At the start of the `try` block, every actor checkpoints its state. This checkpoint is used to enter the `rescue` block, if needed. During execution of the `try` block, a runtime monitor watches for crashed processes. If all goes well for an actor, it pauses at the `end` of the `try`/`rescue` in case another actor crashes. If a failure occurs, the monitor restarts the failed actor, sends updates to all other actors, and unwinds to the checkpointed state.

Actor-Local Variables An actor can bind variables using Elixir's `with` notation. For example, here `Alice` creates a located variable `x`:

```
with Alice.x <- compute_value() do ...
```

Run Results When actors finish executing the main `run` function, they each send a value to the mailbox of the calling process. This value defaults to `nil`.

3.2 Actor Interface

In order to start a Chorex choreography, driver code must provide modules that implement each of the actors. Section 2.1 presented an example driver; module `Tcp.AcceptorImpl` called `Chorex.start` with a mapping from actor names to module names. The `AcceptorImpl` module also happens to be an implementation module, as it implements the role `Acceptor` from the top-level TCP choreography:

```
defmodule Tcp.AcceptorImpl do
  use Tcp.ListenerChor.Chorex, :acceptor
  ...
```

Whereas the actor modules generated by Chorex handle all the communication for that actor in a choreography, Actor implementation modules must contain all the local functions listed in the choreography. For example, the choreography code below, the first line calls a function `get_money()` located on the Alice actor, the second line asks Bob to `fetch_apples`, and the third line asks Alice to `fetch_sugar` and `bake_pie`:

```
Alice.get_money() ~> Bob.payment
Bob.fetch_apples(payment) ~> Alice.apples
Alice.bake_pie(apples, fetch_sugar())
```

An implementation for `Alice` must provide each function in the wishlist, similar to the outline below.

```
defmodule AliceImpl do
  use DemoChor.Chorex, :alice

  def get_money(), do: ...
  def fetch_sugar(), do: ...
  def bake_pie(apples, sugar), do: ...
end
```

To guide actor implementation, Chorex gathers the set of local functions for each actor during projection and creates an interface specification—called a *behaviour* in Elixir parlance—that an implementing module must contain. Elixir issues compile-time errors if an actor implementation does not satisfy the behaviour.

Language Server Integration. Chorex provides guidance to implementation modules in the form of language server tooltips, illustrated in Figure 1a. These are enabled through metaprogramming and cooperation with Elixir's behaviour mechanism. In particular, the line `use DemoChor.Chorex, :alice` expands at compile time to code that glues this implementation module to the choreography's projected module for Alice.

3.3 Supervision Protocol

Chorex leverages Elixir/Erlang process monitoring to supervise choreographic actors. In Elixir, when a process `A` monitors another process `B` and `B` exits, process `A` receives a message that describes how `B` terminated (e.g., normal exit vs. crash). Chorex creates monitoring links to build a supervision tree in the style of Figure 6 every time a program starts a choreography.

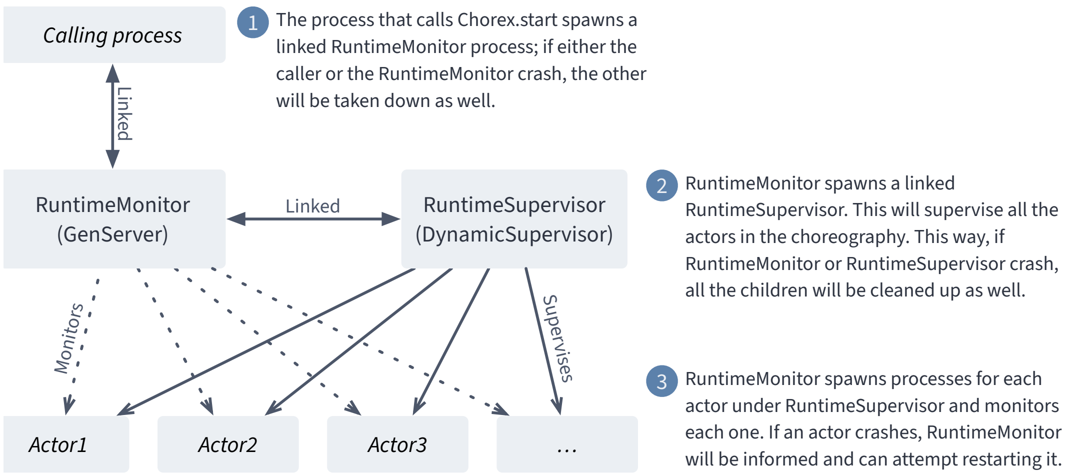


Figure 6 Chorex creates one monitor and one supervisor for each choreography.

The supervision tree adds two processes to a choreography, in addition to the calling process and actor processes: a `RuntimeMonitor` and a `RuntimeSupervisor`. The Monitor, Supervisor, and calling process are linked together such that if any one process crashes, the entire choreography terminates. The Supervisor's job is to perform this cleanup on the actors. The Monitor's job is to watch for actors that crash, restart them, and send updates to reconfigure the network.

Technically, the Supervisor implements an Elixir behaviour called DynamicSupervisor [12]. The Monitor implements the so-called GenServer behaviour [13]. Each actor is a GenServer as well; Section 4 explains the significance of GenServers.

Restarting a Crashed Process When an actor enters a `try`/`rescue` block, it creates a checkpoint that includes the control stack, message inbox, and bindings for local variables. The actor sends the control stack and variable bindings to the Monitor process. If a crash occurs, the Monitor spawns a new process, which has no state initially, and restores the missing pieces. After reinstating an actor process, the Monitor alerts all other actors that a crash has occurred.

Storing checkpoints in the Monitor is a convenient choice. This data could easily move to the filesystem, a database, or even another process. The main benefit of the Monitor is that it is linked to the actors and to the calling process to support clean exits.

Handling Out-of-Band Recovery Messages Chorex manages its own call stacks and inboxes instead of relying on Elixir's mechanisms to enable out-of-band message receives. When an actor crashes, the Monitor broadcasts a message that the most recent `try` block failed along with a token indicating which stack frame the actors must unwind to. Actors handle this message by unwinding their control stacks and resetting their environments to the frame that transfers control to the `rescue` block.

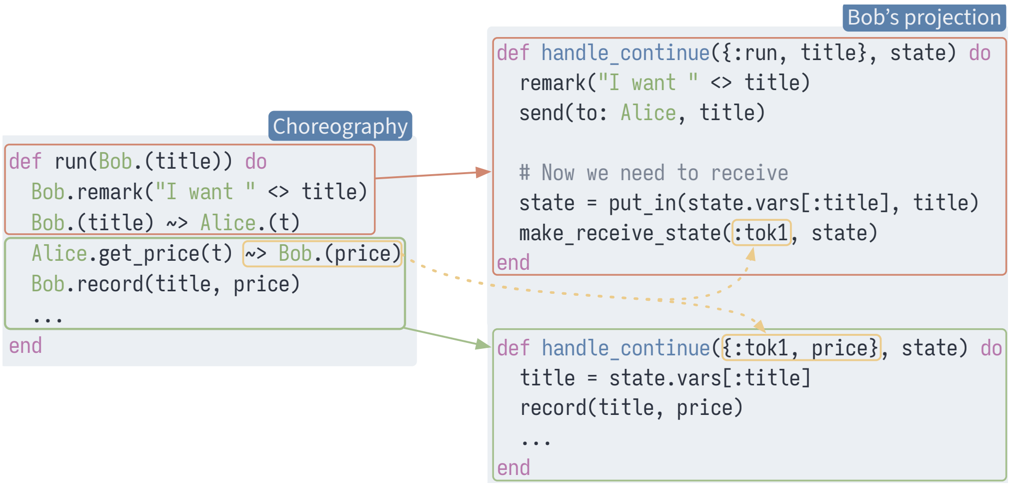


Figure 7 Projection splits actor code into a set of callbacks: one per receive.

Can a Restarted Process Miss Messages? Any messages that arrive while an actor is executing a `try` block go to its inbox, but not to its checkpointed state, and will be lost if the actor crashes. This is not, in fact, a danger. The messages that arrive in a `try` block must have originated from other actors *within the same `try` block* because sends and receives are matched up and and all actors must complete the `try` before any one can continue execution. Forcing actors to wait at the end of a `try` also ensures that they are available to unwind if a `rescue` is needed; without synchronization, an actor might terminate early.

4 Implementation Highlights

Although Chorex builds on Elixir, its high-level strategy for enabling restarts is applicable to other choreographic languages. This section discusses key aspects of the implementation to facilitate adaptation.

Chorex translates choreographies to sets *of sets* of message-passing functions (Figure 7). Each function in the choreography projects into several variants, one for each actor, and each variant is in turn split into several functions (distinguished by generated tokens) corresponding to receive-separated chunks of the choreography. Getting these functions to cooperate required a number of significant components: a tailored message format (Section 4.1), a method of organizing component functions (Section 4.2), and an implementation of the compiler as a metaprogram to maximize host-language integration (Section 4.3). We conclude this section with lessons learned (Section 4.4).

4.1 Message Format

There are two categories of messages that a Chorex actor needs to handle: choreography messages and control messages. *Choreography messages* go between actors, and originate from the sends and receives in a choreography. *Control messages* come from the Chorex runtime; these messages propagate knowledge of choice, notify actors of crashes and network updates, and synchronize actor execution of `try`/`rescue` blocks.

Most messages have a 3-tuple format: `MSG = {message_type, civ_token, payload}`

The `message_type` field may have one of five possible values, which determines both the payload and the tuple shape of the remaining message:

- `:chorex`, for a choreography message. The corresponding payload is a message (any Elixir value) created by one actor and intended for another actor.
- `:choice`, for knowledge-of-choice. Payload is a Boolean choice value.
- `:revive`, for error recovery. This type of message is a 2-tuple; it does not include a CIV token. Payload is the new actor state to install.
- `:recover`, for error recovery. This type of message is sent to actors that did not crash, and asks them to unwind to the nearest `rescue` point and reset their environments. Payload represents the new network configuration.
- `:barrier`, for synchronization at `try` blocks. Indicates that all actors successfully completed the `try` block and may proceed into the continuation.

The `civ_token`, inspired by Ozone [42], preserves communication integrity in the presence of out-of-order messages. Each CIV token is a 4-tuple:

```
CIV = {session_token, metadata, sender, receiver}
```

The `session_token` is a UUID generated when the choreography is instantiated (during `Chorex.start`). Every instance of a choreography has its own session token. The `metadata` field describes the source-code location within the choreography where this message originated (i.e., the original send $\rightsquigarrow$ recv). Crucially, both parties involved in the message receive equal `metadata` values, and different `send` sites lead to different `metadata` values. The `sender` and `receiver` components are the names of the actors involved. All together, a CIV token ensures that a message goes only to the intended destination.

4.2 Actors as Server Processes

Chorex projects actors to Elixir GenServer behaviors [3, 13] rather than straight-line processes. This choice solves the following problems with default Elixir processes:

1. no way to prioritize messages from the Monitor process (Figure 6),
2. no way to supervise for crashes.

GenServers—in contrast to straight-line processes—can be supervised and monitored. Additionally, they can handle messages that arrive in any order and can give priority to messages from the Monitor process. GenServers have their own drawbacks as well, e.g., complicated variable scope, but Chorex works around these problems.

GenServer Primer GenServer (short for "Generic Server") is an Erlang library that makes it easy to build processes that manage state and can respond to ad-hoc messages. Elixir inherits GenServers from Erlang. To behave as a GenServer, a module must define an `init` function that returns a value representing the server state, and callbacks to

```
defmodule Counter do
  use GenServer

  def init(start_count), do: {:ok, start_count} # final element is server state
  def handle_cast(:increment, count), do: {:noreply, count + 1}
  def handle_call(:get_count, sender, count), do: {:reply, count, count} # 2nd element goes to caller
```

Figure 8 Example GenServer that implements a counter.

handle incoming messages. There are three kinds of callbacks that deal with messages: `handle_call`, `handle_cast`, and `handle_info`. These callbacks can expect at least a message and state value as input, and must return a new state value. A `handle_call` receives a process ID as well, and must include a reply to this process in its return value.

As an example, the GenServer in Figure 8 implements a counter. The initial state is the number `start_count`. One callback, `handle_cast`, awaits messages with the tag `:increment` and a numeric value; it adds the input value to the counter state. Another callback, `handle_call`, awaits `:get_count` messages. It logs information about the request and returns a 3-tuple with two copies of the state. One copy is a reply to the `sender` process and the other copy is the new state (same as before the call). To start an instance of this module with an initial count of zero, call `GenServer.start(Counter, 0)`.

Multiple processes can interact with a GenServer at the same time. The GenServer handles each incoming message, one at a time. This behavior enforces linearity, so that concurrent processes can interact with shared state in a coherent way.

Challenges The key challenge of GenServers as a target for projection is that every message that an actor can receive must be anticipated with a callback. An actor cannot be implemented with a single function; it must be split across several functions as shown in Figure 7. Consequently, variables defined earlier in a choreography must become part of the GenServer state in order to reach later parts of the choreography. A second challenge is that actors can no longer use the Elixir call stack to handle function calls. Chorex must manage its own actor-specific control stacks.

Correct Scope via Live Variable Analysis Below, two actors send a message to `Bob`:

```
Alice.one() ~> Bob.x
Carol.two() ~> Bob.y
Bob.(x + y)
```

Projection for the `Bob` actor introduces two callbacks, one for each receive. A direct but incorrect projection would use the variables `x` and `y` directly. This is wrong because `x` is not in scope for the second callback, which needs to return a sum:

```
# WRONG projection for Bob
def handle_info({Alice, x}, state), do: ...
def handle_info({Carol, y}, state), do: x + y # WRONG
```

Chorex builds a correct projection by tracking the set of live variables through a choreography. At runtime, Chorex stores a map from variable to values in GenServer state, and reads from the map as needed:

```
# CORRECT projection for Bob
def handle_info({Alice, x}, state) do
  state = put_in(state.vars[:x], x)
  ...

def handle_info({Carol, y}, state) do
  x = state.vars[:x]
  x + y
```

Determining free variables in Elixir has some subtleties. For example, the match expression `[x, ^y, x] = make_list()` contains three variable names, `x`, `y`, and `x`, but binds only the first one (`x`). The `y` is *pinned* with the ^ prefix, meaning it is a variable reference, and the second `x` is a reference to whatever value the first occurrence of `x` receives in the match. Chorex reuses Elixir's tree-walking API to facilitate analysis, but it implements a custom set of rules to find free variables.

Receives and Function Calls The second challenge of projection to GenServers is that function calls in a choreography no longer map cleanly to function calls in the Elixir output. An actor might call a function that receives several messages; each receive will introduce a new callback. In the example below, the function `test_system` receives one message for each of the actors `Mike` and `Joe`:

```
def run() do
  Joe.(:begin) ~> Mike.start_message
  with Joe.response <- test_system() do
    Joe.(String.length(response))

def test_system() do
  Joe.("Hello Mike") ~> Mike.("Hello " <> my_name)
  Mike.("Hello Joe, you said #{my_name}") ~> Joe.reply
  Joe.("Received " <> reply)
```

To recover typical call/return behavior, each actor tracks a stack of call frames in the GenServer state. When a callback finishes, the GenServer inspects the top stack frame to decide where to go next. It then invokes a next callback, and this jump corresponds to a return in the source language.

When projecting the `run` function above, Chorex does not know whether the call to `test_system` will perform any receives. (Higher-order functions make it impractical to statically track which functions do receive.) At each function call, Chorex thus creates a unique token to identify the call site and its continuation. This token is used in two places: first, it goes onto the control stack in the GenServer state; second, it appears in the argument specification of the callback that holds the continuation code.

Actor State Each Chorex actor keeps the following state, which gets passed between every message handler in the GenServer implementation:

- a queue of choreography messages to be processed,
- a stack of control frames (to recover receives and function calls),
- a map of live variables,
- the `session_token`,

- a map representing the network configuration, and
- a reference to the implementing module.

Chorex actors also share some functionality via a runtime module. The runtime can: push new messages on a queue as they arrive, inspect the control stack to determine which message in the inbox is needed next, and unwind execution stacks.

4.3 Projection via macro expansion

Chorex uses Elixir's macro system to embed a choreography language. Macros expand during compilation, which allows Chorex to perform static checks such as sufficient knowledge-of-choice propagation. Macros are also part of the standard Elixir toolchain, which makes for a seamless workflow. *No extra build steps are needed.*

With its macro implementation, Chorex reuses many affordances of the host language. Local expressions get lowered as-is, making the entirety of Elixir available. Macro hygiene means Chorex users do not have to worry about macro implementation details leaking out, and Chorex itself is, in principle, macro extensible.

Defchor Internals The `defchor` form is a macro that takes a list of actor names and a block of choreography code. It projects the choreography body for each of the actors. Projection takes an actor name and a sequence of expressions and returns three values: (1) a sequence of expressions, representing the actor's view of the expressions; (2) a list of function clauses, which `defchor` will splice into the GenServer for the actor; and (3) a list of function specifications, which will be required of actor implementations. With these pieces, the `defchor` macro generates a module for each actor that contains code to realize that actor's communications as well as a behaviour spec which actor implementations (Section 3.2) must satisfy.

Elixir AST nodes include metadata about source code, including line and column numbers. Chorex uses this metadata whenever possible in expanded code to ensure that error messages get reported in terms of the source language. For example, for the following faulty code: `Alice.one(bad_variable_name) ~> Bob.x` macro output causes the Elixir compiler to report a readable error:

```
error: undefined variable "bad_variable_name"
  Alice.one(bad_variable_name) ~> Bob.x
```

4.4 Reflections

GenServers as a compilation target enabled flexible, out-of-order and out-of-band message receives. This critical ability was well worth the pain of having to implement custom mailboxes, live variable analysis, and execution stacks.

One major issue in Elixir's macro system is its lack of support for pattern-matching on quoted syntax. The base language has excellent pattern matching for values [23]. A tool similar to Racket's `syntax-parse` [18] would make macros easier to write.

In Choral [29], developers are expected to modify projected code, for example, to interoperate with legacy interfaces [39]. With two codebases at play, the choreography

Table 1 Overhead in two realistic programs and three microbenchmark configurations compared to counterparts with no try/rescue.

	try	try + rescue		try	try + rescue
State Machine	1.01 x	1.05 x	Flat-10k	1.04 x	4.37 x
Mini BlockChain	12.15 x	4.74 x	Nest-1k	1.59 x	1.91 x
			Nest-10k	15.82 x	1.93 x

and its projection, a story for *round-trip development* is important so that that changes in the projected code can be propagated to the choreography. Chorex (and HasChor etc. [4]) makes it impractical to edit projected code. This appears to be a step forward, but more experience applying Chorex to existing projects is needed.

Chorex projects choreographies into one module and requires actor implementations to provide application-specific details in a separate module. This design allows the reuse of one protocol across several implementations, and gives Chorex a natural way to reuse Elixir language tooltips.

5 Performance Overhead

Since actors checkpoint their state upon entering a try block and unwind their execution stack to enter a rescue block, it is important to benchmark the run-time overhead. We have tested with two realistic case studies of try/rescue, inspired by programs from Section 2, and several microbenchmark variants. Table 1 lists representative results. State Machine is based on the TCP choreography. Mini Blockchain computes hashes in a loop, similar to zero-knowledge challenges. Flat-10k is a microbenchmark of 10,000 iterations through a recursive function; in each iteration, two actors do some work in a single try block. Nest-1k and Nest-10k run 1,000 and 10,000 iterations of a similar recursive function, but with a recursive call in the try block as well.

All experiments ran on a single-user Apple M1 Pro with 32 GB RAM and 10 available cores, using Chorex 0.8.14, Elixir 1.18.0, and Erlang 27.2. Performance overhead is minimal for flat loops, but is excessive with nesting. This appears to be due to how recursively descending into try/rescue blocks forces actors to accumulate recovery states. Potential ways to reduce overhead are to store diffs of actor states as checkpoints and, for highly-distributed choreographies, save diffs on each actor's local node.

Compile times scale linearly with the number of actors in the choreography. A choreography with 100 actors (already bigger than any practical example we were able to find) took approximately 11 seconds, and 1000 actors took approximately 2 minutes to compile. Details are in the artifact.

6 Related Work

The design and implementation of choreographic languages has become a lively research area. Java, Haskell, Racket, Rust, and (now) Elixir all have third-party

Table 2 Recent advances in choreographic programming. Functions-as-values has seen wide adoption (Functional column). Other recently-proposed features have yet to permeate the landscape. Chorex adds error-restarts to the feature space.

	Functional	Restartable	Full OoO	Census Poly	Agreement-τ
Choret [6]	✓				
HasChor [47]	✓				
⋆ Chorex	✓	✓			
Choral [14, 29, 42]	✓		✓		
Klor [38]	✓				✓
MultiChor,ChoRus, ChoreoTS [4]	✓			✓	

support for choreographies today, and most of these implementations appeared within the past year [4, 6, 10, 29, 38]. Choral is a notable exception with nearly a decade of engineering under its belt [28]. A first workshop on choreographies [1] and an introductory zine [2] appeared last year as well.

As Table 2 outlines, the overall expressiveness of choreographies as a protocol language is expanding. Functional (or higher-order [15, 32]) choreographies, which can use functions as first-class values, are standard. Restarts and network changes are unique to Chorex. Fully out-of-order execution (Full OoO) lets a choreography reorder code in any way that respects data dependencies. An efficient realization of (partial) reordering is in the Ozone API [42], which compiles to Choral. Census polymorphism allows abstraction over the number of participants in a choreography, analogous to variable-arity functions [20]. MultiChor, ChoRus, and ChoreoTS (which were introduced simultaneously [4]) are the first languages to support census polymorphism. Agreement types in Klor track the participants involved in a subroutine and thus provide a compositional way to infer knowledge-of-choice annotations. Chorex provides a modicum of reordering to avoid performance pathologies; messages sent to an actor arrive as soon as they are ready, instead of queuing. We have no plans at this time to implement full reordering. The other features in Table 2 are on the agenda for future work improving Chorex.

An orthogonal dimension is whether to implement choreographies as a standalone language or as a library. Libraries are simpler to implement and use, but limited in power. For example, HasChor [47] broadcasts knowledge-of-choice to all participants—turning every conditional into a choreography-wide sync point—because it cannot perform a two-pass static analysis (as in [32]). MultiChor and its relatives reduce the actors in each broadcast through a *conclave* mechanism. A thesis of Chorex, and of Choret [6], is that the best implementation of all comes through a meta-programmable host language such as Racket [19, 24]. Similarly, recent work improves HasChor with static projection via a Haskell compiler plugin [36].

Theoretical foundations of choreographies have a long history [8, 16, 17, 32, 42, 43]. The Pirouette calculus was the blueprint we followed to start Chorex [32].

Elixir is soon to acquire a full-featured gradual type system based on set-theoretic types [9]. Release v1.18 [54] infers types from match patterns and function bodies to report certain high-confidence warnings. However, users cannot write types and the type checker assumes the permissive dynamic type by default, limiting the guarantees that types provide. Extending this type system to gradually check choreographies is an exciting future direction.

Lastly, we mention areas that have close ties to choreographies. Multiparty session types [33, 34] and secure multiparty computation (SMC) [44, 50] both coordinate distributed actors via a global protocol. Session types merely specify required behavior, giving developers the freedom to build a conforming implementation. One notable realization of session types is ElixirST, a type checker for a language of Elixir processes [26]. Writing programs that conform to session types can be challenging; techniques for API generation address the implementation burden [11], similar to how Chorex choreographies generate requirements for actor modules. SMC languages avoid the conformance question by generating code from a protocol. They are effectively choreographic languages, but designed and constrained by security considerations. Restarts have been formally modeled and implemented in the session-types language Links [25]. Links compiles to custom effect handlers, whereas Chorex cooperates with Elixir's built-in exception mechanism. Dezyne brings formal verification to concurrent industrial processes via a domain-specific language, simulator, and language server integration [5]. Verification in Chorex is an important next step. There is a long history of related work on verification for MPI programs to draw from as well [37, 48, 56].

Recent work on hybrid session types shows how to compose a global protocol from local, application specific protocols [27]. While Chorex allows choreographies to start other choreographies, and thus supports some composition, it cannot make static guarantees. Hybrid choreographies may be the way forward. Another closely-related work is the Corps calculus for hierarchical choreographies [31].

7 Conclusion

The essence of Chorex is a compiler from multiparty programs to stateless, message-passing processes. This compiler breaks new ground for choreographic programming with *restartable actors*, which are enabled through a checkpointing protocol and cooperative supervisor process, and with its *tight integration* to the host language, enabled by metaprogramming. It was not at all clear at the start of this project that an expressive choreographic language could be implemented as a meta-program, as opposed to a standalone compiler. The fact that Chorex works for Elixir indicates that other metaprogrammable languages, from Ruby [35] to Racket [19] to Rust [45], can follow suit by building a library for supervised processes (Section 3.3) and a pure-functional server runtime (Section 4.2). Extending the Chorex compiler protocol to support multiple languages in the same toolchain, with projection to actors written in different languages, would be a worthy challenge for the future.

Acknowledgements We thank Lee Barney and Dan Plyukhin for discussions that helped to frame this work, Ashton Snelgrove for introducing us to Deutsch's fallacies, and Andrew Hirsch for teaching us Pirouette.

References

[1] *1st International Workshop on Choreographic Programming*. https://pldi24.sigplan.org/home/cp-2024. Accessed 2025-02-28. 2024.

[2] Ali Ali and Lindsey Kuper. *Communicating Chorrectly with a Choreography*. https://decomposition.al/zines/#communicating-chorrectly-with-a-choreography. Accessed 2025-02-28. 2024.

[3] Erlang documentation authors. *Documentation for the gen_server behaviour*. https://www.erlang.org/doc/apps/stdlib/gen_server.html. Accessed 2025-02-18.

[4] Mako Bates, Shun Kashiwa, Syed Jafri, Gan Shen, Lindsey Kuper, and Joseph P. Near. "Efficient, Portable, Census-Polymorphic Choreographic Programming". In: *PACMPL* 9.PLDI (2025). DOI: 10.1145/3729296.

[5] Rutger van Beusekom, Bert de Jonge, Paul F. Hoogendijk, and Jan Nieuwenhuizen. "Dezyne: Paving the Way to Practical Formal Software Engineering". In: *F-IDE*. 2021, pages 19–30. DOI: 10.4204/EPTCS.338.4.

[6] Alexander Bohosian and Andrew K. Hirsch. *Choret: Functional Choreographic Programming for Racket*. https://github.com/Foundations-of-Decentralization-Group/choret. Accessed 2025-02-26. 2025.

[7] Marco Carbone, Kohei Honda, and Nobuko Yoshida. "A Calculus of Global Interaction based on Session Types". In: *DCM@ICALP*. 3. Elsevier, 2006, pages 127–151. DOI: 10.1016/J.ENTCS.2006.12.041.

[8] Marco Carbone and Fabrizio Montesi. "Deadlock-Freedom-by-Design: Multiparty Asynchronous Global Programming". In: *POPL*. ACM, 2013, pages 263–274. DOI: 10.1145/2429069.2429101.

[9] Giuseppe Castagna, Guillaume Duboc, and José Valim. "The Design Principles of the Elixir Type System". In: *Programming* 8.2 (2024). DOI: 10.22152/PROGRAMMING-JOURNAL.ORG/2024/8/4.

[10] Kaushik Chakraborty. *UniChorn*. https://share.unison-lang.org/@kaychaks/unichorn/. Accessed 2025-03-02. 2024.

[11] Guillermina Cledou, Luc Edixhoven, Sung-Shik Jongmans, and José Proença. "API Generation for Multiparty Session Types, Revisited and Revised Using Scala 3". In: *ECOOP*. Schloss Dagstuhl, 2022, 27:1–27:28. DOI: 10.4230/LIPICS.ECOOP.2022.27.

[12] Elixir Contributors. *Dynamic Supervisor behaviour*. https://hexdocs.pm/elixir/1.13/DynamicSupervisor.html. Accessed 2025-03-03. 2021.

[13] Elixir Contributors. *GenServer behaviour*. https://hexdocs.pm/elixir/1.12/GenServer.html. Accessed 2024-12-16. 2021.

[14] Luís Cruz-Filipe, Eva Graversen, Lovro Lugovic, Fabrizio Montesi, and Marco Peressotti. "Functional Choreographic Programming". In: *ICTAC*. Springer, 2022, pages 212–237. DOI: 10.1007/978-3-031-17715-6_15.

[15] Luís Cruz-Filipe, Eva Graversen, Lovro Lugović, Fabrizio Montesi, and Marco Peressotti. “Modular Compilation for Higher-Order Functional Choreographies”. In: *ECOOP*. Schloss Dagstuhl, 2023, 7:1–7:37. DOI: 10.4230/LIPICS.ECOOP.2023.7.

[16] Luís Cruz-Filipe and Fabrizio Montesi. “A Core Model for Choreographic Programming”. In: *Theoretical Computer Science* 802 (2020), pages 38–66. DOI: 10.1016/J.TCS.2019.07.005.

[17] Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. “A Formal Theory of Choreographic Programming”. In: *Journal of Automated Reasoning* 67.2 (2023), page 21. DOI: 10.1007/S10817-023-09665-3.

[18] Ryan Culpepper. “Fortifying macros”. In: *Journal of Functional Programming* 22.4-5 (2012), pages 439–476. DOI: 10.1017/S0956796812000275.

[19] Ryan Culpepper, Matthias Felleisen, Matthew Flatt, and Shriram Krishnamurthi. “From Macros to DSLs: The Evolution of Racket”. In: *SNAPL*. Volume 136. Schloss Dagstuhl, 2019, 5:1–5:19. DOI: 10.4230/LIPICS.SNAPL.2019.5.

[20] R. Kent Dybvig and Robert Hieb. “A New Approach to Procedures with Variable Arity”. In: *LISP and Symbolic Computation* 3.3 (1990), pages 229–244.

[21] Elixir Contributors. *hex*. https://hex.pm/. Accessed 2025-03-04. 2025.

[22] Elixir Contributors. *Mix*. https://hexdocs.pm/mix/1.18.2/Mix.html. Accessed 2025-03-04. 2024.

[23] Elixir Contributors. *Patterns and Guards*. 2024.

[24] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay A. McCarthy, and Sam Tobin-Hochstadt. “The Racket Manifesto”. In: *SNAPL*. Schloss Dagstuhl, 2015, pages 113–128. DOI: 10.4230/LIPICS.SNAPL.2015.113.

[25] Simon Fowler, Sam Lindley, J. Garrett Morris, and Sára Decova. “Exceptional Asynchronous Session Types: Session Types Without Tiers”. In: *PACMPL* 3.POPL (2019), 28:1–28:29. DOI: 10.1145/3290341.

[26] Adrian Francalanza and Gerard Tabone. “ElixirST: A Session-Based Type System for Elixir Modules”. In: *Journal of Logical and Algebraic Methods in Programming* 135.100891 (2023). DOI: 10.1016/J.JLAMP.2023.100891.

[27] Lorenzo Gheri and Nobuko Yoshida. “Hybrid Multiparty Session Types: Compositionality for Protocol Specification through Endpoint Projection”. In: *PACMPL* 7.OOPSLA1 (2023), pages 112–142. DOI: 10.1145/3586031.

[28] Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. *Choral: Object-Oriented Choreographic Programming*. 2023. arXiv: 2005.09520 [cs.PL]. URL: https://arxiv.org/abs/2005.09520.

[29] Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. “Choral: Object-oriented Choreographic Programming”. In: *Transactions on Programming Languages and Systems* 46.1 (2024), 1:1–1:59. DOI: 10.1145/3632398.

[30] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. “The Knowledge Complexity of Interactive Proof Systems”. In: *SIAM Journal on Computing* 18.1 (1989), pages 186–208. DOI: 10.1137/0218012.

[31] Andrew K. Hirsch. “Corps: A Core Calculus of Hierarchical Choreographic Programming”. In: *CoRR* abs/2406.01456 (2024). DOI: 10.48550/ARXIV.2406.01456. arXiv: 2406.01456.

[32] Andrew K. Hirsch and Deepak Garg. “Pirouette: Higher-Order Typed Functional Choreographies”. In: *PACMPL* 6.POPL (2022), pages 1–27. DOI: 10.1145/3498684.

[33] Kohei Honda, Nobuko Yoshida, and Marco Carbone. “Multiparty Asynchronous Session Types”. In: *POPL*. ACM, 2008, pages 273–284. DOI: 10.1145/1328438.1328472.

[34] Kohei Honda, Nobuko Yoshida, and Marco Carbone. “Multiparty Asynchronous Session Types”. In: *Journal of the ACM* 63.1 (2016), 9:1–9:67. DOI: 10.1145/2827695.

[35] Milod Kazerounian, Sankha Narayan Guria, Niki Vazou, Jeffrey S. Foster, and David Van Horn. “Type-Level Computations for Ruby Libraries”. In: *PLDI*. ACM, 2019, pages 966–979. DOI: 10.1145/3314221.3314630.

[36] Robert Krook and Samuel Hammersberg. “Welcome to the Parti(tioning) (Functional Pearl): Using Rewrite Rules and Specialisation to Partition Haskell Programs”. In: *Haskell*. ACM, 2024, pages 27–40. DOI: 10.1145/3677999.3678276.

[37] Mathieu Laurent, Emmanuelle Saillard, and Martin Quinson. “The MPI Bugs Initiative: a Framework for MPI Verification Tools Evaluation”. In: *Correctness@SC*. IEEE, 2021, pages 1–9. DOI: 10.1109/CORRECTNESS54621.2021.00008.

[38] Lovro Lugović and Sung-Shik Jongmans. *Klor: Choreographies in Clojure*. https://github.com/lovrosdu/klor. Accessed 2024-12-03. 2024.

[39] Lovro Lugović and Fabrizio Montesi. “Real-World Choreographic Programming: Full-Duplex Asynchrony and Interoperability”. In: *Programming* 8.2 (2024). DOI: programming-journal.org/2024/8/8.

[40] Fabrizio Montesi. “Choreographic Programming”. PhD thesis. IT University of Copenhagen, 2013. ISBN: 978-87-7949-299-8. URL: https://www.fabriziomontesi.com/files/choreographic-programming.pdf.

[41] Fabrizio Montesi. *Introduction to Choreographies*. Cambridge University Press, 2023. DOI: 10.1017/9781108981491.

[42] Dan Plyukhin, Marco Peressotti, and Fabrizio Montesi. “Ozone: Fully Out-of-Order Choreographies”. In: *ECOOP*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, 31:1–31:28. DOI: 10.4230/LIPICS.ECOOP.2024.31.

[43] Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. “Dynamic Choreographies: Theory And Implementation”. In: *Logical Methods in Computer Science* 13.2 (2017). DOI: 10.23638/LMCS-13(2:1)2017.

[44] Aseem Rastogi, Matthew A. Hammer, and Michael Hicks. “Wysteria: A Programming Language for Generic, Mixed-Mode Multiparty Computations”. In: *SP*. IEEE Computer Society, 2014, pages 655–670. DOI: 10.1109/SP.2014.48.

[45] Rust Contributors. *Rust Macros*. https://doc.rust-lang.org/reference/procedural-macros.html. Accessed 2024-01-17. 2024.

[46] Tiago Santos. *SRP*. https://github.com/thiamsantos/srp-elixir. Accessed 2025-03-03. 2019.

[47] Gan Shen, Shun Kashiwa, and Lindsey Kuper. “HasChor: Functional Choreographic Programming for All (Functional Pearl)”. In: *PACMPL* 7.ICFP (2023), pages 541–565. URL: https://doi.org/10.1145/3607849.

[48] Stephen F. Siegel and George S. Avrunin. “Verification of MPI-Based Software for Scientific Computation”. In: *SPIN Workshop*. Springer, 2004, pages 286–303. DOI: 10.1007/978-3-540-24732-6_20.

[49] Simon St. Laurent and J. David Eisenberg. *Introducing Elixir: Getting Started in Functional Programming*. 2nd. O’Reilly Media, 2017. ISBN: 978-1491956779.

[50] Ian Sweet, David Darais, David Heath, William Harris, Ryan Estes, and Michael Hicks. “Symphony: Expressive Secure Multiparty Computation with Coordination”. In: *The Art, Science, and Engineering of Programming* 7.3 (2023). DOI: 10.22152/PROGRAMMING-JOURNAL.ORG/2023/7/14.

[51] Dave Thomas. *Programming Elixir*. 1st. Pragmatic Bookshelf, 2018. ISBN: 978-1680502992.

[52] Thousand Island Contributors. *Thousand Island*. https://github.com/mtrudel/thousand_island. Accessed 2025-01-30. 2025.

[53] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. “Languages as Libraries”. In: *PLDI*. 2011, pages 132–141. DOI: 10.1145/1993498.1993514.

[54] Jose Valim. *Elixir v1.18 released*. https://elixir-lang.org/blog/2024/12/19/elixir-v1-18-0-released/. Accessed 2025-03-05. 2024.

[55] Ingrid Van Den Hoogen. *Deutsch’s Fallacies, 10 Years After*. https://web.archive.org/web/20070811082651/http://java.sys-con.com/read/38665.htm. Accessed 2025-02-28. 2007.

[56] Anh Vo, Sarvani S. Vakkalanka, Michael Delisi, Ganesh Gopalakrishnan, Robert M. Kirby, and Rajeev Thakur. “Formal Verification of practical MPI programs”. In: *PPoPP*. ACM, 2009, pages 261–270. DOI: 10.1145/1504176.1504214.

[57] Wikipedia. *Fallacies of distributed computing*. https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing. Accessed 2025-02-28. 2024.

[58] Thomas Wu. *The Secure Remote Password Protocol*. http://srp.stanford.edu/ndss.html. Accessed 2025-03-04. 1997.

About the authors

Ashton Wiersdorf (research@wiersdorfmail.net) is a PhD student at the University of Utah.
https://orcid.org/0000-0001-5524-7930

Ben Greenman (benjamin.l.greenman@gmail.com) is an assistant professor at the University of Utah.
https://orcid.org/0000-0001-7078-9287