Final report: Extract Choreography Automata for Program Understanding

Student: Gabriele Genovese
Supervisor: Cinzia Di Giusto

March 24, 2025

Abstract

The recent development of concurrent and distributed applications has raised new interest in programming paradigms incorporating threads and message passing into their logic. The use of choreographies can ensure some typical properties of concurrent systems (such as liveness, lock freedom and deadlock freedom). ChorEr is a preliminary static analysis tool for a fragment of Erlang programs that generates choreography automata. We plan to build on the existing tool to implement new functionalities and cover more primitives, thus extending the expressiveness of the language under consideration.

Contents

1 Introduction

The rise of service computing and microservices led to a shift from monolithic apps to distributed components using message-passing. In this field, *choreographic* coordination [19] lets designers focus on *application-level protocols*, i.e., a description of *communication interactions*[1] among system components (called *participants*). This focus is typically expressed by two distinct, yet related views of distributed computations: the so-called *global* and *local* views. The former view abstracts away from the actual communication infrastructure in order to give a blueprint of the communication protocol. The latter view provides the description of the communication behavior of participants in isolation and can guide their implementation. Choreographies can be expressed in modeling languages or formalism (like WS-CDL [19], BPMN diagrams [27]), multiparty session types [16], message-sequence charts, multiparty contracts, and many others (see also the survey in [17]).

Besides offering a suitable development for message-passing systems, global views yield a high-level description of application-level protocols.[2] It is therefore crucial that global views faithfully capture all the interactions in the system. While this is relatively simple to guarantee when participants' implementations are driven by the global view (e.g., in the top-down approach), the correspondence can be easily spoiled when software evolves or dynamic composition takes place (as advocated in microservices architectures). The classical top-down approach is then of little help: one needs to write a global description of the desired behavior and then use type checking or monitoring to find possible discrepancies.

Instead, we aimed to explore the concept of a *bottom-up approach*. Bottom-up methods (such as [26, 20, 21, 7, 5]) focus on "extracting" global views from existing code. This is particularly useful when no pre-existing global description is available or when the code has been modified without keeping the choreography up to date for legacy reasons.

Moreover, automatic derivation of choreographies can aid developers in understanding the system's behavior. The extracted choreography should *at least capture all the correct behaviors* of the system while also *highlighting potential misbehavior*. Indeed, our primary motivation for extracting a choreographic description is to support *debugging* and *program comprehension*. Therefore, the extracted choreography should explicitly flag communication issues such as deadlocks, orphan messages, and unspecified receptions.

This approach naturally applies to languages with a well-defined concurrency model and dedicated message-passing primitives, such as Erlang, Go, and Scala.

In this context, I contributed to enhancing the development of an existing tool called Chorer, a static analyzer that extracts choreography automata from Erlang source code. This type of analysis presents two main challenges. First, obtaining a perfect global specification from code is generally impossible, as it

[1] An interaction is a full message-passing event, including both sending and receiving.

[2] According to the so-called top-down approach, a global view (formalized, e.g., as a multiparty session type) can be algorithmically projected on a local view preserving relevant properties.

would require solving undecidable problems such as program termination. Thus, our goal is to extract an *approximation* of the system. An *over-approximation* ensures that all correct behaviors are included, while also capturing possible misbehavior.

The second challenge is the potential generation of *huge choreographic descriptions*. Large automata are difficult to interpret, making it necessary to explore strategies for mitigating this issue.

My contributions to this work include formalizing parts of the tool, improving the codebase through new features and bug fixes, and creating a benchmark suite to evaluate the effectiveness of the approach.

2 Chorer

The project centers around ChorEr, a static analyzer developed as part of my Bachelor's thesis at the University of Bologna [13]. This tool is implemented in Erlang and is openly available under the GPLv3 license on GitHub [12]. ChorEr generates multiple DOT files (a commonly employed graph description language) representing Choreography Automata of a given Erlang program's local and global views.

2.1 Preliminaries

Choreographies are a formal model used to represent systems of communicating processes, enabling semantic proofs regarding the presence or absence of the mentioned properties. Choreographies are **global views** of the behavior of a system, giving a comprehensive perspective of the communication exchanges among actors (also called participants). From the global view, via a simple projection, one can obtain the **local view**, i.e., the individual behavior of each participant in the communication process. Notice that, the local view is limited and actors are unaware of the behavior of the rest of the system.

A *visual* way to formalize choreographies is through **Choreographic Automata** [2], which use finite-state automata to describe communication systems. This representation effectively illustrates program flow, showing loops and branching, while leveraging existing results [28].

This project aims to enhance a tool that extracts choreographic specifications from Erlang programs as Choreographic Automata. The process involves deriving local views for each actor and composing them into a global choreography, an inverse operation w.r.t. projection, which may not always be feasible. The following example illustrates this extraction. This example differs from the one shown in the previous Description of Work, and it's taken from the benchmark suite.

Example 2.1 (Async example)**.** In this example, two actors, `dummy1` and `dummy2`, exchange a message asynchronously, meaning that we don't know which message will arrive first. Because the *send* operation is non-blocking, and we don't know which *receive* will be performed first, the expected graph should

show two possible lines of execution: one where the exchange of `ciao` occurs first and one where the exchange of `bello` occurs first.

```
-module(async).
-export([main/0, dummy1/0, dummy2/0]).

dummy1() ->
    d2 ! bello,
    receive
        ciao -> done
    end.

dummy2() ->
    d1 ! ciao,
    receive
        bello -> done
    end.

main() ->
    A = spawn(?MODULE, dummy1, []),
    register(d1, A),
    B = spawn(?MODULE, dummy2, []),
    register(d2, B).
```

Listing 1: Two processes exchanging messages asynchronously

Listing 1 shows the Erlang code representing the described scenario. The `register` function is a built-in feature that enables the global registration of a process identifier. This allows the use of a defined atom to communicate with the corresponding process. In this example, `register` is used to simplify communication, eliminating the need to exchange process identifiers between processes. However, this approach is included solely for demonstration purposes, as it is generally not considered good practice in Erlang programs. Figures 1, 2 and 3 depict the local views of the actors.

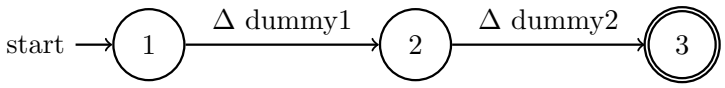

Figure 1: Local view of the `main` actor

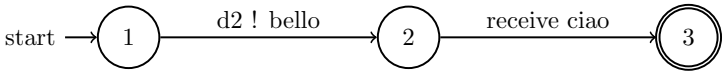

Figure 2: Local view of the `dummy1` actor

After associating the local views with the actors, the algorithm generates the global view shown in Figure 4. The automaton expresses two possible global executions of the program: from State 1 to State 3, the actors `dummy1` and `dummy2` are spawned. Then, since the *send* operation is non-blocking, either

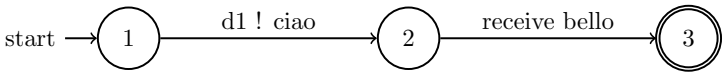

Figure 3: Local view of the `dummy2` actor

`dummy1` sends `ciao` first (leading to State 6) or `dummy2` sends `bello` first (leading to State 4). Consequently, the final state is reached through two possible paths, depending on the order in which messages are received.

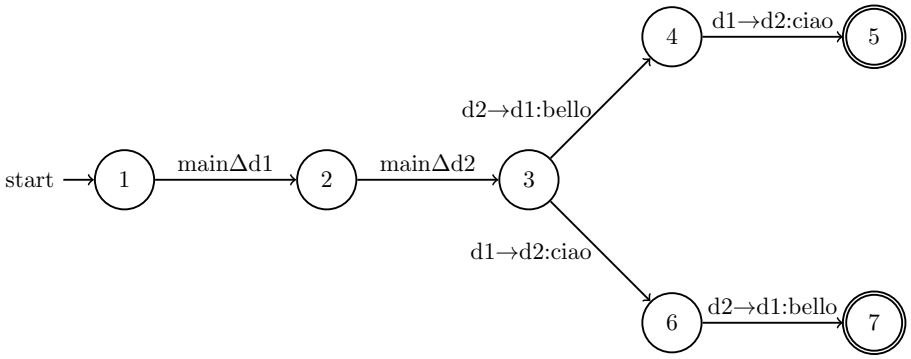

Figure 4: Global view of Listing 1

2.2 Description of the tool

How to use the tool The easiest way to use the tool is from the command line interface (CLI) using the help of `rebar3`, a standard build tool that provides various features such as package management for community-created libraries, compilation, and automated project testing. By cloning the project from the public GitHub repository and running the `rebar3 escriptize` command in the main directory, `rebar3` will, in this order, check for and download dependencies if needed, compile the project. After that, an executable can be used in `./_build/default/bin/chorer`.

```
Usage:
  chorer <input> <entrypoint> <output> <ming> <gstate> <minl>

Extract a choreography automata of an Erlang program.

Arguments:
  input      Erlang source file (string)
  entrypoint Entrypoint of the program (atom)
  output     Output directory for the generated dot files (string),
     default: ./
  ming       Minimize the globalviews , default: false
```

```
11  gstate      Global state are formed with previous messages ,
      default: true
12  minl        Minimize the localviews , default: true
```

Listing 2: Usage message

The mandatory arguments are:

- **Input**: The relative path string of the input Erlang program, from which the tool will generate local and global views.
- **Entrypoint**: The atom representing the function where the execution of the input program begins. This parameter is essential because Erlang does not have a conventional entry point function (i.e. the main in C, but in Erlang can be every exported function). It will be passed to the function that creates the global view to start the simulation.

The optional arguments are:

- **Output**: The relative path string of the output folder where the local and global view files will be saved. Local views will be named `[function name with arity]_local_view.dot`, while the global view file will be named `[Entrypoint]_global_view.dot`.
- **Options**: Additional arguments define specific operations to perform on either the local or global view. For instance, they can be used to generate an optimized or minimized version of a graph as output. These options allow for greater flexibility in refining and analyzing the extracted views, enabling different meaningful representations.

2.2.1 From Erlang to Choreographies

Local view The code of a `receive` operation corresponds to the Figure 5: each branch will be evaluated recursively, continuing the construction from the branches. Finally, all branches will merge into a common node with ϵ transitions, from which the evaluation of the local view will resume. The send operation (`Pid ! message`) operation corresponds to Figure 6. The `spawn` function call corresponds to Figure 7.

Recursive calls In the case of *recursive* calls, a transition ϵ is created from the last node generated to the first node, as shown in Figure 8. For now, further evaluation of the function is blocked.

Function call When a call to an unknown function is encountered, the algorithm will create the local view of the called function and "connect" the beginning of the graph of the called function with the last state created in the local view of the calling function, linking them with an ϵ transition. The local view will continue from the last vertex of the call graph.

In graph 9, state 1_a is the initial state of the calling function, and state 1_b represents the first state of the local view of the called function.

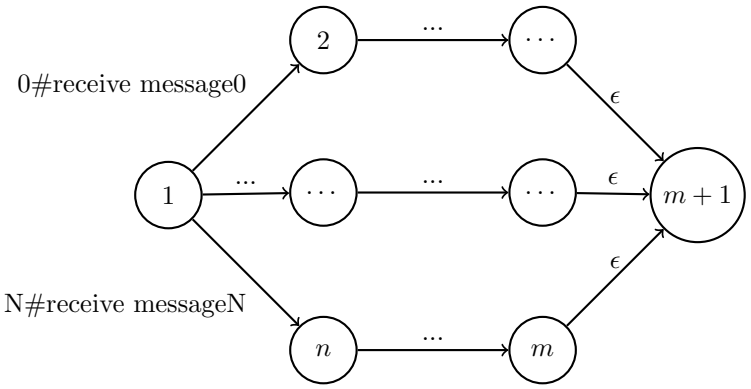

Figure 5: Localview graph for the `receive` keyword

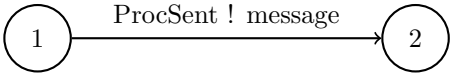

Figure 6: Localview graph for `!` keyword

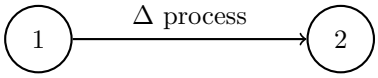

Figure 7: Localview graph for the `spawn` function

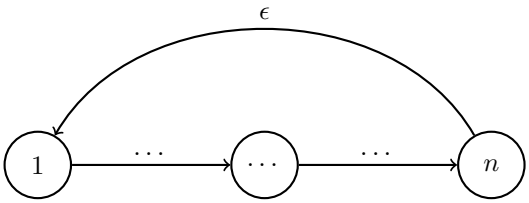

Figure 8: Localview graph for a recursive call

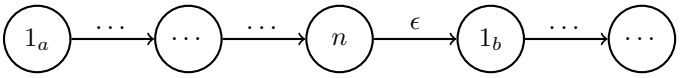

Figure 9: Localview graph of a function call

Global view For global views, in `spawn`, the actor performing the operation is specified to the left of the symbol, as shown in Figure 10. A spawn will be directly inserted into the graph. Each process will also be numbered in case multiple actors are created for the same function.

For message sending and receiving, during the simulation of actors, if two

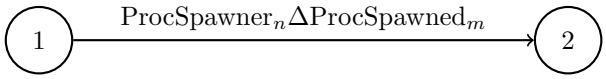

Figure 10: Global view graph for the `spawn` function

actors are found executing a compatible *send* and *receive*, a state will be added to the global view as shown in Figure 11. To be compatible, the receiving process must match the data recipient, and the pattern matching of the *receive* must correspond to the sent data. Transitions for *send* and *receive* that are "empty" (i.e., messages that are sent but not processed by a *receive* in any process) will not be shown in the global automaton.

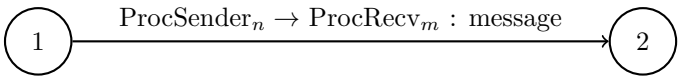

Figure 11: Global view graph for `receive` and `!` keywords

2.2.2 Main algorithms

The execution is divided into four main phases:

1. Initialization of the `db_manager`, data structures, and extraction of preliminary information (handled by the `md.erl` module). Possible actors are extracted from the `export` attribute, the number of `spawn` executions is counted, and the ASTs of all functions are saved in the `db_manager`. Meanwhile, an initial evaluation of the program flow is performed by initializing the names of actors and saving the argument passing to the `spawn` functions.

2. Creation of local views for all possible actors, i.e., all functions that appear in the `export` at the beginning of a program (obtained from the first phase). This phase essentially acts as a compiler, translating Erlang programs into local choreographies.

3. Creation of the global view starting from the program's starting point and combining the local views created in phase two.

4. Post-processing of the local and global view, inferring information on the graph and extracting data useful for the benchmarks

While creating a local view follows the function code line by line, the global view must compose local views while considering the created actors. An approximate execution is simulated from the startup function using the actor philosophy. Each actor has a manager process that tracks available transitions and the process state (i.e., local variables). This function provides the main process

of the global view with actor-related information. Messages may travel within the same virtual machine or across a network, making message exchange asynchronous. If two sends target the same process, their order is unpredictable, altering the program flow.

The global view creation algorithm follows an overapproximation approach, combining local views while dynamically creating communicating actors and exchanging messages. It explores all possible executions by handling forks in local views and evaluating message reception. A branch duplicates and takes another path in three cases: encountering a fork in a local view, multiple processes ready to receive, and evaluating message reception within a process. This ensures a thorough exploration of execution paths.

The strategy for composing local views is to have all actors continue until any *receive* is encountered. While searching for the first *receive*, *spawn* and *send* operations are also checked. If a *spawn* occurs, the node is added to the graph, and the actor is created. If a *send* occurs, the message is enqueued in the target process' message queue. After blocking on a *receive*, possible messages are processed, creating a new execution path for each, adding new states to the global graph.

The algorithm runs recursively until a relevant state change occurs. When an iteration fails to modify any actor's state, the execution stops. The final graph represents the asynchronous communication among actors for messages exchanged.

3 Extended examples

In order to better illustrate the features of the tool, we present in this section two examples (using Erlang-like, actor-based pseudocode), each one composed by a pseudocode and the corresponding Choreography Automata [2] of the global view of the communicating system. Here, a *global view* is an abstract description of all the possible behaviors of the full system. We consider a language where receive statements are blocking operations. A state where each participant has completed its task and terminated is called *final*. As such, a state that is not final and has no outgoing transitions is a *deadlock*.

The first example is a concise reproduction of the dining philosophers problem, which highlights a possible deadlock. The second example shows a possible mutual exclusion error when operating a simple bank account. Both examples are available online [10]; the corresponding automata have been obtained with the tool.

In the examples, we exploit two operations for sending and receiving messages, respectively. More precisely, `send msg to proc` sends message `msg` to process `proc`, while `receive pat1 from proc1 -> e1;...;patn from procn -> en` represents a branching point where the process receives the first message that matches a pattern `pati` and, then, continues with the execution of `ei`. As in Erlang, pattern matching is tried from top to bottom. When a receive has only one clause, we abbreviate it as `receive pat from proc` (and continues with the execution of the

next sentence).

3.1 Dinning philosophers

Dining Philosophers Example This example is taken from the `dining` case study from the benchmark suite of the tool. Let us consider a program with two participants playing the role of a dining philosopher who shares two forks with the other participant.

```
philosopher(Fork1, Fork2) ->
  send req to Fork1,
  receive ack from Fork1,
  send req to Fork2,
  receive ack from Fork2,
  eat(),
  send release to Fork1,
  send release to Fork2,
  philosopher(Fork1, Fork2).

fork() ->
  receive req from Phil,
  send ack to Phil,
  receive release from Phil,
  fork().
```

The behavior of the philosophers is given by the pseudocode on the right while pseudocode for the behavior of the forks is discussed below. Each philosopher first acquires the forks (starting with the one on its right, that is the one with the same index), then eats (with the function call `eat()` representing some terminating local computation), and finally releases the forks before recurring. Parameters `Fork1` and `Fork2` are references to processes executing the behavior of forks described by the pseudocode on the left that repeatedly waits for the request from process `Phil`, acks the request, and waits for the `release` message from `Phil`.

There are two possible behaviors of the system. The first (good) one where the philosophers alternate eating infinitely and a second (bad) behavior where both philosophers manage to take only one fork each resulting in a deadlock. Figure 12 depicts the global Choreography Automaton representing the program above. We have two recursive executions where both philosophers eat, represented as loops, and two executions which end in the same deadlock state. The deadlock is visible since State 5 is not final, and it has no outgoing transitions. The automaton is simplified as it does not display the `ack` messages, and the `release` messages are merged since their order is irrelevant. The focus here is solely on the order of the `req` messages. One can imagine to first extract the full choreography automaton, and then merge equivalent behaviors and abstract away from uninteresting transitions.

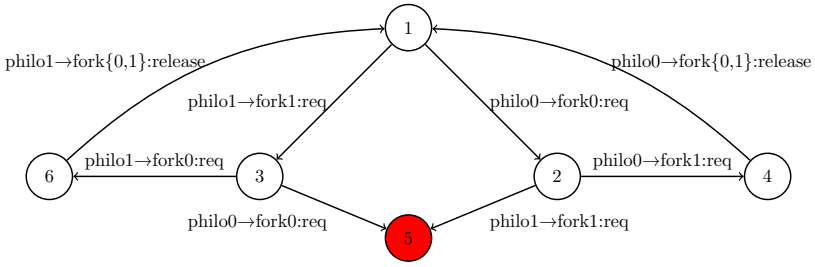

Figure 12: Global view of the dining philosophers example.

3.2 Bank account

This case study is taken from the `account` example from the benchmark suite of the tool. We now consider a system where a bank account is accessed by two clients, dubbed C1 and C2.

```
account(Value) ->
  receive
    read from Client ->
      send Value to Client,
      account(Value);
    NewValue from Client ->
      account(NewValue).

client() ->
  send read to Acc,
  receive Value from Acc,
  % operations on Value
  send NewValue to Acc.
```

The pseudocode yields the behavior of the bank account, where Value represents its current balance. This process waits for requests from a client. A request can either be a read access to know the current balance or an update request of such value to a NewValue.

Symmetrically, client processes C1 and C2 behave according to the pseudocode on the left: the process reads the current balance from the account, performs some internal operations based on such value, and updates the balance. The global view of the communicating system is depicted in Figure 13.

We can observe two correct executions where the operations are performed in a read-update-read-update order (taking the path via states 1-2-4-7-11 or the one via states 1-3-6-10-14).

However, there is also a read-read-update-update order on the highlighted paths. Although the choreography is not inherently incorrect, these highlighted paths could represent a violation of mutual exclusion which may be undesirable for developers in certain contexts. The choreography automaton in Figure 13 helps in spotting this issue.

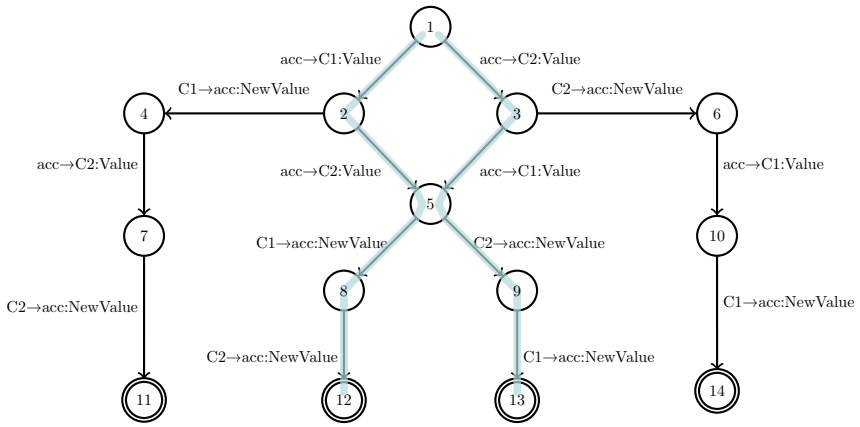

Figure 13: Global view of the bank account example

4 Contributions

While the previous section discussed the basics of the tool, most of its content was already available at the beginning of this project. In contrast, this section focuses on the new contributions made to enhance the tool. We detail the improvements, additional features, and optimizations introduced, highlighting their impact on usability, performance, and functionality.

4.1 Attributed grammar

At the beginning of this period, it was challenging to clearly define the role of the parser. There was no formal specification outlining how the tool should process a program during its initial analysis. This lack of clarity led to difficulties in understanding the expected behavior and designing a structured approach for parsing. To address this, we first examined the tool's existing implementation and identified key areas requiring specification. Establishing a well-defined set of rules became essential for ensuring consistent behavior and improving the accuracy of the parsing process. Through iterative refinement, we aimed to create a structured framework that guides the tool in extracting meaningful information from the program. Therefore, to better understand how the parsing of a file and the creation of a localview should work, we created an attributed grammar for a subset of the Erlang language. An attributed grammar extends a context-free grammar by associating attributes with its symbols and defining semantic rules. Attributes can be *synthesized* (computed from child nodes) or *inherited* (passed from parent nodes).

The attribute part can be seen as a set of operations that the compiler performs during its analysis. This serves two key purposes: it provides a clear understanding of what the tool can parse from the programming language, and it defines how information propagates through the parse tree. This propagation

helps identify the core data structures that underpin the tool's functionality.

The following attributed grammar will be presented alongside comments to provide insights into the tool's behavior. First, we introduce the grammar fragment, followed by the corresponding operations applied to the local view data structure. These operations illustrate how the tool processes and interprets parsed elements, highlighting its internal mechanisms.

The `link` function is used to establish a logical connection between two nodes. Its third argument defines the label of the transition. If this argument is not provided, the transition is considered an ϵ-transition, representing an unlabeled connection.

Program For simplicity, we consider an Erlang program as a set of one or more functions, that can be called by the user. All the needed function are within the program.

$prog \rightarrow (function)^+$

A complete program does not have a local view, so there is no need to define attributes.

Function A function can have its own local view representation. In Erlang, pattern matching allows multiple definitions, so we must distinguish between the base case and the pattern matching case. The base case is straightforward, so we only present the pattern matching case.

Base case: $function \rightarrow fun$.
With multiple definitions: $function \rightarrow fun_1; ...; fun_N$.

```
function.nodes = new U fun1.nodes U ... U funN.nodes
function.edges = link(new,fun1.first) U ... U
                 link(new,funN.first) U
                 fun1.edges U ... U funN.edges
function.first = new
```

Each definition has its own local view representation, so we must link each one to a new state. This state serves as the starting point of the entire local view. The nodes consist of all other nodes, and the same applies to the edges.

Function body The body of a function consists of a name (which is an atom), a set of arguments (which may be empty), and a list of expressions.

$fun \rightarrow Atom(X_1, ..., X_n) \rightarrow Exprs$

```
fun.nodes = Exprs.nodes
fun.edges = Exprs.edges
fun.first = Exprs.first
```

```
fun.last = Exprs.last
fun.context = [ X1 -> Param[1], ..., Xn -> Param[n] ]
fun.ret_var = Exprs.ret_var
```

When encountering a function, we simply add its arguments to the context. The `Param` is taken as input, with a default value of `ANYDATA`. Additionally, we can perform semantic checks to verify the function's existence.

Expressions A list of expressions can have one or more elements. The single-expression case is straightforward, so its attributes are omitted.

Base case: $Exprs \rightarrow expr$
With multiple expression: $Exprs \rightarrow expr, Exprs'$

```
Exprs.nodes = expr.nodes U Exprs'.nodes
Exprs.edges = expr.edges
              U Exprs'.edges
              U link(expr.last, Exprs'.first)
Exprs.first = expr.first
Exprs.last = Exprs'.last
Exprs.context = expr.context U Exprs'.context
Exprs.ret_var = Exprs'.ret_var
```

Expressions represent the lines of code in an Erlang program. We cover only the essential ones (e.g., assignments, function calls) and those related to communication.

Send As mentioned earlier, a send should add a transition to the local graph.

$expr \rightarrow expr' \ ! \ expr''$

```
expr.nodes = expr'.nodes U expr''.nodes U new1 U new2
expr.edges = expr'.edges U expr''.edges
             U link(expr'.last, expr''.first)
             U link(expr''.last, new1)
             U link(new1, new2, expr'.ret_var
                  + " ! "
                  + expr''.ret_var)
expr.first = expr'.first
expr.last = new2
expr.context = expr'.context U expr''.context
expr.ret_var = expr''.ret_var
```

In Erlang, both the left-hand side and right-hand side of an operation can be expressions. First, the left-hand side expression is evaluated, followed by the right-hand side. Finally, a send edge is inserted with the corresponding variable as the label. The left-hand side must contain a process identifier.

Receive Like for a send, a receive operation should add a transition to the local graph for each matching pattern.

$expr \rightarrow receivepatter_1 \rightarrow Exprs_1; ...; pattern_n \rightarrow Exprs_n end$

```
expr.first = new1
expr.last = new2
expr.nodes = Exprs1.nodes U ... U Exprsn.nodes U new1 U new2
expr.edges = Exprs1.edges U ... U Exprsn.edges
             U link(new1, Exprs1.first)
             U ...
             U link(new1, Exprsn.first)
             U link(Exprs1.last, new2)
             U ...
             U link(Exprsn.last, new2, espilon)
```

A `receive` operation can contain an entire block of code that must be evaluated. Its return data cannot be determined statically, as it varies depending on the matching pattern during the global view phase. Additionally, since the context is modified during global view simulation, we leave these two attributes unchanged.

Spawn call The same reasoning used before applies to the `spawn` function.

$expr \rightarrow spawn(Atom, Params)$

```
expr.nodes = new1 U new2
expr.edges = link(new1, new2, "spawn Atom")
expr.first = new1
expr.last = new2
expr.ret_var = newVar(type: pid, value: random)
```

The `spawn` function returns a new variable as output. Its type should be `pid`, with no assigned name. An additional logical identifier is included to differentiate it during global view construction. The context is left untouched as it's not modified.

Recursive call Recursive calls must be considered, as they are fundamental in functional languages like Erlang.

$expr \rightarrow FunName(Params)$

```
expr.edges = expr.edges U link(FunName.first, expr.last)
expr.ret_var = null
```

For recursive calls, the last added node must be linked to the first node of the local view being created. Using the function name, we can retrieve its previously created first node. Since a single analysis cannot determine the function's return type, we set the return variable to `null`. After evaluating the recursive call, the analysis stops, allowing only tail-recursive functions.

Generic function call To complete the previous analysis, we define attributes for standard function calls.

$expr \rightarrow Atom(Params)$

```
G = get_localview(Atom, Param)
expr.nodes = G.nodes
expr.edges = G.edges
expr.first = G.first
expr.last = G.last
expr.ret_var = G.ret_var
```

With `get_localview`, we retrieve the necessary information to complete the local view for a generic function call.

Assignment Finally, assignments must be handled to support basic programs with variables. Assignments in Erlang are peculiar as it has immutable variables: they behave as pattern matching when a variable is already defined. Thus, semantic checks can be performed.

$expr \rightarrow pattern = expr'$

```
expr.nodes = expr'.nodes
expr.edges = expr'.edges
expr.first = expr'.first
expr.last = expr'.last
expr.context = [pattern -> expr'.ret_var]
```

The pattern should add the new variable to the context with its assigned name.

Conclusion We defined the attributed grammar for the `localview` module of our tool. This process highlighted the need for a `localview` with the following attributes: `nodes` and `edges`, with labels to represent the automaton; a `first` and `last` node to track entry and exit points; and a `context` along with `ret_var` for basic variable management. Some of these attributes are already present in the current codebase, but their handling differs, and additional arguments must be incorporated. Thus, a refactoring process is required to align the implementation with this attributed grammar.

This technique is particularly powerful in our case. For instance, a known issue with mutual recursive functions can be effectively addressed by improving context management inspired by the attributed grammar. Using this method extensively could be key to efficiently representing `localviews`. However, the refactoring process is extensive and time-consuming. Therefore, it has been identified as a high-priority task for future work.

4.2 Improvements on the tool

One key goal of the tool is to generate correct graph outputs, ensuring an accurate representation of program behavior. While refining abstract aspects of the project, several features, improvements, and bug fixes have been added to enhance the precision and consistency of the global view specification. The iterative process of refining the tool aims to minimize errors, and provide a more reliable emulation of actor-based execution. Future work will continue to focus on refining these aspects to offer greater flexibility while maintaining correctness of the analyzed program.

Features Two key features have been added to the tool. The first one improves the local view module. The tool can now correctly parse programs where values are passed as arguments to functions, allowing for a more precise representation of data flow between processes. Previously, only functions without value passing were supported. This was a significant limitation and, in fact, one of the highest-priority features. It was the final major feature of a programming language to be added, and it is now included in the parsing module, enabling a broader range of program inputs.

The second feature added to the tool enhances over-approximation in the global view module. During computation, the value of some data can become everything because we cannot know the outcome of an operation performed on the data (i.e. we don't support mathematical operation). The tool now substitutes this unknown value with the `ANY` keyword to indicate that it can represent any kind of data. If a message is exchanged with `ANY` as its content, it will match every possible receive branch, ensuring a more over-approximated analysis of the program.

General improvements Some general improvements have been made on the tool. The tool is now more resilient to unexpected errors, reducing the likelihood of crashes and ensuring that it provides meaningful output in most cases, even when encountering invalid input. This was achieved through the implementation of more error checks and recovery patterns within the codebase, where there were none before.

Also, a new library has been integrated for command-line argument management, making the CLI more robust and user-friendly, with better parsing of parameters and improved error messages.

Some effort has been put to improve the testing environment and benchmark generation. The tool now produces useful benchmarking information, which is processed by a dedicated Python script. This script collects and analyzes the data to generate meaningful performance benchmarks. Additionally, if the `correct_gv.dot` file is present (that is a dot file containing the expected global view) the script performs an automatic correctness check. This feature lays the groundwork for more expressive and rigorous testing in the future, enabling better validation of the tool's accuracy and performance across different scenarios and use cases.

Bug fixes Some notable bug fix improvements were made within the global view module that are needed to be mentioned. It has been resolved an issue where the tool incorrectly printed warnings about failing to find the correct process during the sending of a message. This bug caused the useless show of some false positive warnings.

It also has been addressed a problem that prevented the exploration of certain branches during the global view emulation, ensuring a more comprehensive traversal of all possible execution paths. This bug prevented a full branch exploration in certain programs.

It has been fixed a bug that was causing the creation of excessive duplicated branches. This patch corrected an issue where unnecessary duplicated branches were created after receiving a message and defining a new variable. This fix improves the accuracy of the global view by reducing redundant computations.

4.3 Benchmarks

Significant effort has been dedicated to developing a generic and automated script (written in Python) to test the tool and extract relevant information about the tool and its output. Table 1 presents empirical data on the evaluation of various examples processed by the tool.

The examples are primarily designed ad hoc to test specific aspects of the tool. They are not sourced from well-known Erlang suites, as the tool is still in an early stage and cannot yet parse most of them. The columns of Table 1 provide the following information:

- **Example**: Name of the test case.
- **Lines**: Number of lines of code in the example.
- **Tot LV**: Total number of local views generated by the example.
- **GV Nodes**: Number of nodes in the global view graph.
- **GV Edges**: Number of edges in the global view graph.
- **Warns**: Number of warnings raised.
- **Errors**: Number of errors detected (i.e. deadlocks).
- **Time**: Execution time in seconds.

The entries are ordered by the number of lines of code. The generated graphs included in the table are presented without minimization. Applying minimization techniques to the global view graph can lead to incorrect reductions in some cases, cause of a known bug. Therefore, we opted to maintain the full graph representation to ensure correctness.

Analysis of global view empirical data From the Table 1, we can observe the overall behavior of the tool, focusing on the number of warnings and errors in the output. As previously mentioned, a high number of message exchanges directly correlates with an increased number of nodes and edges in the global view. Warnings usually occur when the tool encounters unknown elements in the source code, such as a new keyword or an external function. They also appear when the tool loses track of a process identifier, preventing it from mapping message exchanges correctly. This occurs in the `if-cases` example, which generates 185 warnings due to the tool's failure to instantiate a variable with the correct process identifier, making it unable to deliver the corresponding message. This issue likely stems from a bug in the actor emulation module.

Errors typically arise when an unexpected situation occurs or when a deadlock state is detected. Most examples do not automatically terminate a process once it completes its task (e.g., waiting for requests), which contributes to this issue. However, one particularly notable case is `foo8`, which has 561 nodes, 560 edges, and 191 errors. The reason lies in its heavy use of case constructs—some information is lost between the local and global views, leading the tool to explore every possible execution path.

This results in a high number of states, edges, and undefined deadlock conditions. Using a minimized version of the local graph during actor emulation can significantly reduce these values. However, this conflicts with the principle of using an over-approximated approach. To address this, minimization could be added as a program argument, allowing users to trade some generality for a more precise global view graph.

About complexity A significant portion of the execution time is obviously consumed by the global view composition algorithm. This algorithm attempts to explore all possible program executions, considering every potential message order to meet the overapproximation requirement. In the worst-case scenario, without any restrictions, it must evaluate all permutations of messages, resulting in a factorial time complexity of $O(n!)$. However, in most cases, more efficient strategies can be applied to mitigate this complexity.

The execution times show that our algorithm performs efficiently even on large examples (an example is considered large when there are many nodes and edges because this corresponds to a high number of messages exchanged, leading to a high number of possible combination of these messages), with the most complex cases (e.g., `foo8`) taking only a few seconds.

Correctness of Global View One of the main contributions was enabling automatic verification of whether a global view matches its correct version. Since knowing the complete specification in advance is not trivial, we conducted tests on a few selected examples. The correct versions of the graph can be seen as what we expect to be the final global view.

Table 2 briefly shows the correctness of the global view. The **Check** column indicates whether the global view correctly represents the expected behavior.

Example	Lines	Tot LV	GV Nodes	GV Edges	Warns	Errors	Time
unknown	13	2	2	2	0	0	0.175s
foo9	14	4	4	3	1	3	0.187s
foo9c	15	3	10	15	0	0	0.182s
pass	16	3	3	2	0	0	0.174s
foo9d	16	3	3	2	0	0	0.186s
funcall	17	3	4	3	1	2	0.180s
forloop	18	3	7	6	0	0	0.188s
foo1	18	3	8	7	0	0	0.175s
foo5	18	3	79	165	1	0	0.316s
async	20	3	7	6	0	0	0.194s
foo4	20	4	16	19	0	2	0.182s
hof	21	4	15	17	0	3	0.189s
foo9b	21	4	4	4	14	1	0.183s
spawn	22	3	9	8	0	0	0.179s
foo3	22	3	13	16	0	0	0.178s
account	23	3	28	39	0	2	0.211s
airline	23	3	15	26	1	0	0.232s
foo2	23	4	4	3	1	1	0.180s
foo9h	23	4	24	35	0	5	0.196s
hello	24	3	5	6	2	0	0.190s
trick	24	4	9	9	0	0	0.184s
foo6	24	5	9	9	15	2	0.190s
foo9e	24	5	14	14	0	5	0.186s
foo9f	25	5	7	6	0	4	0.183s
foo9g	25	5	44	83	0	7	0.226s
conditional	26	2	25	24	1	16	0.197s
foo8	29	5	561	560	0	191	3.590s
producer	30	4	11	10	0	1	0.183s
dining	31	3	45	72	0	2	0.232s
ticktackloop	32	4	6	6	2	0	0.182s
airline	33	3	35	68	1	0	0.232s
ping	36	3	6	5	1	0	0.188s
meViolation	40	4	63	82	2	4	0.266s
serverclient	41	5	9	8	8	3	0.187s
foo7	41	3	149	229	0	6	0.513s
ticktackstop	46	5	19	27	7	0	0.214s
purchase	47	5	49	66	6	0	0.257s
customer	54	5	17	22	1	0	0.205s
if-cases	57	4	148	210	185	30	0.525s

Table 1: Global view empirical data

Some examples, such as `async` and `ticktackloop`, are correctly generated because they are very simple example, while others fail. This may result from a

missing feature in local or global view generation or from the expected version not aligning with actual results. A key future goal of the project is to generate as many correct global view versions as possible and to systematically check and refine each one.

Example	Check
unknown	False
async	True
ticktackloop	True
ticktackstop	False
customer	False

Table 2: Global view correctness data

5 Conclusion

The project began with the goal of creating a proof of concept, a prototype, for an analyzer that could extract choreography automata directly from an existing codebase, which is quite unique in its field, having a bottom-up over-approximation approach leaning to mainstream programming languages. A significant amount of effort and resources have been invested, primarily in the tool's codebase and its design around examples and use cases. Additionally, extensive discussions have taken place regarding the high-level and general aspects of the tool.

The results of this prototype are promising, demonstrating that such a tool is feasible and worth further exploration. We discussed on how the analysis should work, defining motivations, requirements, and challenges surrounding the tool. At the same time, we have introduced multiple improvements to the existing codebase to refine its output.

Finally, efforts have been made to advance a formalization process, highlighting what has been done well and what still needs improvement. This has provided valuable insights into the tool's design and future directions.

The formalization process has just begun and will require further refinement in the future. A new testing method for the tool has been introduced, but additional effort is needed to generate correct global views. This is crucial for properly assessing the tool's precision and could lead to refinements in its existing components.

The formalization process has also highlighted the need to refactor the local view component. A possible improvement is the addition of a parsing module capable of reading local views from files (e.g., DOT format). This enhancement would make the tool more general, removing its strict dependency on Erlang programs. With such a feature, developers could create a separate tool to translate their preferred programming language into a local view, outputting it as a DOT file. This file could then be provided as input to our tool to

compute the corresponding global specification, making it a more versatile and widely applicable solution.

Another area for improvement is the minimization and post-processing module. It can be expanded with better heuristics to generate clearer, more structured, and easily interpretable graphs. Enhancing these optimizations will improve readability and usability, making the tool's output more insightful and practical.

5.1 Related works

In the industrial context, Erlang's actor model has also inspired other programming languages or frameworks, such as Elixir, a programming language based on Erlang's virtual machine. Other programming languages also implement the actor model, such as Go with its GoRoutines (comparable to Erlang's spawns) and channels (similar to Erlang's send and receive). Software industry is increasingly devoting attention to choreographic approaches [27, 3, 11, 1] because they naturally support modularization and decoupling. In fact, distributed components coordinate according to a global description without the need of an explicit coordinator. In the academic context, research in the field of *choreography* focuses on two main topics: *choreography specification* and *choreographic programming*.

- *Choreography specifications*: this area includes formal methods, such as multiparty asynchronous session types [15], which have been established to describe the interactive structure of a fixed number of actors from a global perspective. These methods enable the syntactic verification of actors' correctness by projecting the global specification onto individual participants. Choreography specifications are also studied as contracts, which provide abstract descriptions of program behavior, known as *multiparty contracts* [17].

- *Choreographic programming*: this programming paradigm has been explored both theoretically, as in [6], and industrially, as in [18]. Several choreographic programming languages have been designed and studied to support this paradigm [24, 25, 14, 9].

In this context, global specifications are crucial for guaranteeing correctness (since they are blueprints of complex distributed systems and feature model-driven development) as well as for program comprehension. Most formal works on communication protocol specifications emphasize the projection of a global specification onto local specifications. However, the process of choreography extraction remains challenging and has been explored in [7], with a general framework for extracting choreographies presented in [8]. To the best of our knowledge, the first attempts to extract global specifications for message-passing systems goes back to [26, 20, 21]. These approaches aim to identify "meaningful" global specifications according to general properties such as deadlock-freedom or absence of orphan messages [4]. A limitation of these approaches is that they

do not start from components written in a full-fledged programming language. Rather, distributed components are specified in [20, 21] as abstract models (respectively, π-calculus processes [29, 22, 23] and communicating finite-state machines [4]).

References

[1] Marco Autili, Paola Inverardi, and Massimo Tivoli. Automated synthesis of service choreographies. *IEEE Softw.*, 32(1):50–57, 2015.

[2] Franco Barbanera, Ivan Lanese, and Emilio Tuosto. Choreography automata. In Simon Bliudze and Laura Bocchi, editors, *Coordination Models and Languages - 22nd IFIP WG 6.1 International Conference, COORDINATION 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15-19, 2020, Proceedings*, volume 12134 of *Lecture Notes in Computer Science*, pages 86–106. Springer, 2020.

[3] Jonas Bonér. *Reactive Microsystems - The Evolution Of Microservices At Scale*. O'Reilly, 2018.

[4] Daniel Brand and Pitro Zafiropulo. On Communicating Finite-State Machines. *Journal of the ACM*, 30(2):323–342, 1983.

[5] Marco Carbone, Fabrizio Montesi, and Carsten Schürmann. Choreographies, logically. *Distri. Comput.*, (31):51–67, 2018.

[6] World Wide Web Consortium. Web Services Choreography Description Language Version 1.0. `https://www.w3.org/TR/ws-cdl-10/`. [Online; accessed 06-June-2023].

[7] Luís Cruz-Filipe, Kim S. Larsen, and Fabrizio Montesi. The paths to choreography extraction. In *Proceedings of the 20th International Conference on Foundations of Software Science and Computation Structures - Volume 10203*, page 424–440, Berlin, Heidelberg, 2017. Springer.

[8] Luís Cruz-Filipe, Kim S Larsen, Fabrizio Montesi, and Larisa Safina. Implementing choreography extraction. *arXiv preprint arXiv:2205.02636*, 2022.

[9] Mila Dalla Preda, Saverio Giallorenzo, Ivan Lanese, Jacopo Mauro, and Maurizio Gabbrielli. Aiocj: A choreographic framework for safe adaptive distributed applications. In *Software Language Engineering: 7th International Conference, SLE 2014, Västerås, Sweden, September 15-16, 2014. Proceedings 7*, pages 161–170. Springer, 2014.

[10] Two simple examples. Dining example and Account example, `https://github.com/gabrielegenovese/chorer/tree/master/examples/dining` `https://github.com/gabrielegenovese/chorer/tree/master/examples/account`.

[11] Leonardo Frittelli, Facundo Maldonado, Hernán C. Melgratti, and Emilio Tuosto. A choreography-driven approach to apis: The opendxl case study. In Simon Bliudze and Laura Bocchi, editors, *Coordination Models and Languages - 22nd IFIP WG 6.1 International Conference, COORDINATION*

2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15-19, 2020, Proceedings, volume 12134 of *Lecture Notes in Computer Science*, pages 107–124. Springer, 2020.

[12] Gabriele Genovese. ChorEr. `https://github.com/gabrielegenovese/chorer`. [Online; accessed 05-July-2023].

[13] Gabriele Genovese. Chorer: un analizzatore statico per generare automi coreografici da codice sorgente erlang. Bachelor's thesis, University of Bologna, 2023.

[14] Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. Object-oriented choreographic programming. *arXiv preprint arXiv:2005.09520*, 2020.

[15] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 273–284, 2008.

[16] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9:1–9:67, 2016.

[17] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, 2016.

[18] IBM. Creating BPMN choreography diagrams. `https://www.ibm.com/docs/en/rational-soft-arch/9.7.0?topic=diagrams-creating-bpmn-choreography`. [Online; accessed 18-June-2023].

[19] Nickolas Kavantzas, David Burdett, Gregory Ritzinger, Tony Fletcher, Yves Lafon, and Charlton Barreto. Web services choreography description language version 1.0. Technical report, W3C, 2005. `http://www.w3.org/TR/ws-cdl-10/`.

[20] Julien Lange and Emilio Tuosto. Synthesising choreographies from local session types. In *CONCUR 2012*, volume 7454 of *LNCS*. Springer, 2012.

[21] Julien Lange, Emilio Tuosto, and Nobuko Yoshida. From Communicating Machines to Graphical Choreographies. In *POPL15*, pages 221–232, 2015.

[22] Robin Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press, 1999.

[23] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes, I and II. *Inf. and Comp.*, 100(1), 1992.

[24] Fabrizio Montesi. Jolie: a service-oriented programming language. Master's thesis, University of Bologna, 2010.

[25] Fabrizio Montesi. *Choreographic programming.* IT-Universitetet i København, 2014.

[26] Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. Global principal typing in partially commutative asynchronous sessions. In Giuseppe Castagna, editor, *Programming Languages and Systems*, pages 316–332, Berlin, Heidelberg, 2009. Springer.

[27] OMG. *Business Process Model and Notation (BPMN), Version 2.0*, January 2011. `https://www.omg.org/spec/BPMN`.

[28] Simone Orlando, Vairo Di Pasquale, Franco Barbanera, Ivan Lanese, and Emilio Tuosto. Corinne, a tool for choreography automata. In *Formal Aspects of Component Software: 17th International Conference, FACS 2021, Virtual Event, October 28–29, 2021, Proceedings 17*, pages 82–92. Springer, 2021.

[29] Davide Sangiorgi and David Walker. *The π-Calculus: a Theory of Mobile Processes*. Cambridge University Press, 2002.