Final report: Extract Choreography Automata for Program Understanding

Student: Gabriele Genovese
Supervisor: Cinzia Di Giusto

February 6, 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 of the 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 **3**
- 1.1 Motivations . . . 3
- 1.2 Aim . . . 4
- 1.3 Requirements . . . 4
- 1.4 Challenges . . . 5
- 1.5 State of the Art . . . 6

2 Chorer **7**
- 2.1 Basics of the theory . . . 7
- 2.2 Description of the tool . . . 8
  - 2.2.1 From Erlang to Choreographies . . . 10
  - 2.2.2 Main algorithms . . . 12
- 2.3 What's changed . . . 16
  - 2.3.1 Attributed grammar . . . 16
  - 2.3.2 New examples . . . 16
  - 2.3.3 Feature . . . 19
  - 2.3.4 Bug Fix . . . 19
  - 2.3.5 Benchmarks . . . 19

3 Conclusion **19**
- 3.1 Related works . . . 19

1 Introduction

The impetus of service-oriented computing and, more recently, of microservices has determined a radical shift from monolithic applications to distributed components cooperating through message-passing. In this domain, *choreographic* coordination [25] allows designers to focus on the so-called *application-level protocols*, i.e., a description of the *communication interactions*[1] among the system's components (hereafter 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 formalisms (like WS-CDL [25], BPMN diagrams [34]), multiparty session types [21], message-sequence charts, multiparty contracts, and many others (see also the survey in [23]).

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.

1.1 Motivations

The primary motivation behind this research project is to explore the choreography and choreography automata framework, making it more aligned with the needs of developers, through tools that enable choreography extraction. The tool not only facilitates the visualization and understanding of concurrent systems but also serves as a practical implementation that demonstrates how the abstract concepts of choreographies can be applied to existing technologies. This approach can provide several advantages:

- Debugging: developers can use the tool to gain insights into the global and local behaviors of their concurrent programs;
- Verification of concurrency models: it offers a mechanism to verify that the program's implementation aligns with the intended choreography, ensuring

[1] With interaction, we refer to a full message-passing communication, comprised of both a send of a message and the corresponding receive.

[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.

correctness and reducing potential synchronization errors.

1.2 Aim

Explore and evaluate various models and paradigms that facilitate the development of robust and scalable concurrent applications is one of the primary aim of this project. We explore the bottom-up approaches that "extract" global views from code. Note that bottom-up approaches exist [33, 26, 27, 9, 6], but they have several limitations. Firstly, they produce global views out of abstract models of local views (such as communicating-finite state machines [5] or some kind of behavioral types [23]) and not from actual code written in a mainstream programming language (such as Erlang). Secondly, the extracted global views do faithfully capture the behavior of a local view only under some "well-formedness" conditions. Crucially, this would hide buggy or unexpected behavior. This is exactly the case where a global view would be most useful: the program is buggy and, in order to fix the bug, we need to understand the application-level protocol via a global abstract description. For this project, we focus particularly on enhancing an existing tool that tries to extract choreographies with a bottom-up, over-approximated approach. In the next section, we'll define what are the requirements and the challenges to address this problem in the best way.

1.3 Requirements

Below, we define the requirements that an automatic choreographic bottom-up approach should satisfy to enable its use for program understanding:

- *Bottom-up approach:* one should be able to automatically derive a choreography from code, so that it can be used to help understand the code.
- *Push-button technique:* the extraction of the choreography should be fully automatic, to be applicable to existing code without the need to add special annotations or any other input from the programmer.
- *Always return a choreography:* even if the system is not well-behaved, hence its behavior can not be described by a choreography in the classical sense (since classical choreographies ensure by construction properties such as race and deadlock freedom). The extracted choreography should contain at least all the good behaviors, and possibly information on the not-well behaved parts.
- *Highlight misbehavior:* debugging is our key reason to extract a choreographic description from code; therefore, extracted choreographies should explicitly flag misbehavior due to communications such as deadlocks, orphan messages, unspecified receptions, etc.

- *Applicable to mainstream languages:* one should be able to extract the choreography from a real program written in a mainstream language. Natural targets are languages with a clean concurrency model and dedicated primitives for message-passing such as Erlang, Go, and Scala.
- *Support creation and termination of participants:* in real message-passing systems new processes can be spawned, and some processes may terminate. Hence, a choreographic description should allow for a dynamic number of participants.
- *Support races:* races are disallowed by many choreographic approaches, yet are common in real programs. As such, they should be described (and possibly highlighted as potentially wrong in line with the idea of highlighting misbehavior), but not forbidden.
- *Accessible yet precise notation:* choreographies should be represented with an intuitive, possibly graphical, formalism to improve readability. Instead of the usual algebraic formalisms one should appeal to graph-like notations such as labeled transition systems or finite state automata that pair a graphical representation with a well-defined mathematical definition.

1.4 Challenges

We have given above a number of requirements that the approach we envisage should satisfy, but the reader familiar with the topic may have already found a number of potential difficulties. Indeed, we are aware of a few problems that need to be solved in order to make such an approach feasible. We describe them below, together with possible mitigation measures:

Undecidability: extracting a precise description of all the behaviors of a system is in general impossible, since it would require, e.g., deciding termination. Hence, one can focus on extracting a precise choreography in simple cases and giving approximations of the behavior otherwise. An over-approximation may exhibit spurious behavior w.r.t. the actual behavior of the system. Thus, one can understand the actual behavior, including bugs; however, it is necessary to verify if the reported bugs are false positives. Therefore, care should be taken to limit the number of false positives, since a too high number would make the approach not viable. Over-approximations can be too coarse; e.g., if it is not possible to statically determine which is the expected recipient of a message, an over-approximation may yield a huge number of spurious communications by adding an interaction for each participant in the system. When, as in the case discussed above, over-approximations are not suitable, an under-approximation may be more useful, possibly paired with warnings highlighting issues. The problem in this case is to make sure that false negatives are avoided, i.e., cutting off misbehavior from extracted choreographies.

Huge descriptions: choreographies of real programs may be huge, thus hindering their usefulness for program understanding. We believe this issue should be tackled by providing tools to abstract, explore, or better visualize the choreography. For instance, one may decide that in order to understand a particular behavior interaction, some participants are not of interest, hence should be removed (e.g., like ϵ-transitions in automata based approaches). Another option to reduce the size of the description could be to collapse behaviors which are equal up to swap of concurrent actions (as in partial order reduction techniques within model checking [18]), or collapse the behaviors of multiple processes executing the same code.

1.5 State of the Art

In the industrial context, Erlang's actor model has also inspired other programming languages or frameworks, such as Elixir [37], a programming language based on Erlang's virtual machine. Other programming languages also implement the actor model, such as Go [19] with its GoRoutines (comparable to Erlang's spawns) and channels (similar to Erlang's send and receive).

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 [20], 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* [22].

- *Choreographic programming*: this programming paradigm has been explored both theoretically, as in [8], and industrially, as in [24]. Several choreographic programming languages have been designed and studied to support this paradigm [31, 32, 17, 12].

A way to formalize choreographies is through Choreographic Automata [2], which describe a communication system using a finite-state automaton. By using this model, it is possible to use the results of automaton studies to demonstrate the mentioned properties [35]. 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 [10], with a general framework for extracting choreographies presented in [11]. In our work, we aim at extracting choreographies from an existing and widely-used programming language like Erlang, despite it not being originally designed with choreography in mind.

2 Chorer

The project centers around ChorEr, a Proof of Concept for a static analyzer developed as part of a Bachelor's thesis at the University of Bologna [16]. This tool is implemented in Erlang and is openly available under the GPLv3 license on GitHub [15]. 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 Basics of the theory

The use of the actor model to conceptualize and create concurrent programs has led the scientific community to focus on solving long-known but challenging behavioral problems and on verifying semantic properties (e.g. liveness, lock and deadlock freedom). One of the mathematical models developed in recent years is **choreographies**: 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 describe a communication system using a set of finite-state automata. Representing choreographies using FSA is particularly well-suited for illustrating the program flow, as it clearly shows loops and branching, while using previously described results and notions. [35].

The goal of this project is to build a tool that starting from an existing Erlang program extracts the choreographic specification in the form of Choreographic Automata. This amounts to first extract the local views corresponding to each actor and then combine the local views into the global choreography. This step is a sort of inverse operation with respect to the projection and will not be always possible. To illustrate what it means to extract a choreography from an Erlang program, consider the following example.

Example 2.1 (Extraction example)**.** Take two actors: a `main` and a `dummy` process. These two actors subsequently exchange a message. However, one of the actors will execute *send* first and then *receive*, while the other actor will perform the operations in reverse order. Therefore, the expected graph should show a single line of execution, where the exchange of `ciao` occurs first, followed by the exchange of `bello`.

```
1 -module(simple).
2 -export([main/0, dummy/0]).
3
```

```
4  main() ->
5      D = spawn(simple, dummy, [self()]),
6      D ! ciao,
7      receive
8          bello -> done
9      end.
10
11 dummy(M) ->
12     receive
13         ciao -> done
14     end,
15     M ! bello.
```

Listing 1: Two processes exchanging messages synchronously

Listing 1 shows to the described scenario in Erlang. The `dummy` process sends the atom `ciao` to the `main` process. This action unlocks the sending of the atom `bello` to `dummy`. Figures 1 and 2 depict the local views of the actors.

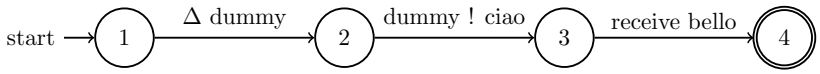

Figure 1: Local view of `main`

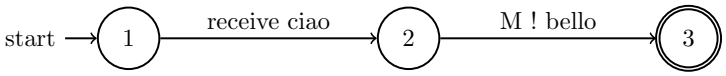

Figure 2: Local view of `dummy`

After associating the local views with the actors, the algorithm generates the global view shown in Figure 3. As expected, the automaton in Figure 3 expresses only one possible global execution of the program: from State 1 to State 2 the actor with the unique identifier `main` spawns the actor with the identifier `dummy`, and then from State 2 to State 3 `main` sends successfully the message `ciao` to `dummy`. Finally, from State 3 to State 4, `dummy` will response to `main` with `bello`.

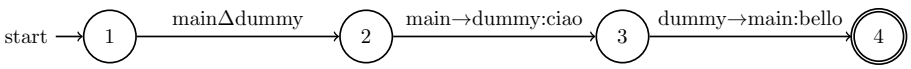

Figure 3: Global view of Code 1

2.2 Description of the tool

How to use the tool The tool can be used from the command line interface (CLI). It can be used by compiling each module from the Erlang Shell (Eshell)

or, more conveniently, by using `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, and run tests if present. 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 soure file (string)
  entrypoint Entrypoint of the program (atom)
  output     Output directory for the generated dot files (string),
     default: ./
  ming       Minimize the globalviews , default: false
  gstate     Global state are formed with previous messages ,
    default: true
  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**: Currently, this is a tuple of two booleans that allow customization of local views. The first boolean determines whether to identify final states (by default, it is set to true). The second adds more information to the local view by including transitions related to other language constructs in addition to communication constructs (by default, it is set to false). This parameter may be expanded in the future with additional booleans.

```
shell> ./_build/default/bin/chorer ./path/to/file.erl entrypoint/0
```

Listing 3: Use example of the tool

Struttura del tool Il progetto si divide in due cartelle principali. Sotto la cartella `examples` ci sono vari esempi di programmi in Erlang su cui provare il tool. Il codice del tool si trova invece nella cartella `src`.

Il file principale è `chorer_app.erl`, dove si trova la funzione `start`. I file `rebar.config`, `rebar.lock` e `chorer.app.src` servono a far funzionare il tool `rebar3`. Nella cartella `choreography` si trovano i moduli adibiti all'analisi statica del programma e alla creazione delle viste locali e globali. In particolare, il modulo `db_manager.erl` si occupa della gestione di dati in comune, utile per non passare tanti dati attraverso i parametri delle funzioni. I moduli `metadata.erl`, `local_view.erl` e `global_view.erl` si occupano rispettivamente di gestire l'estrazione dei dati preliminari, la creazione delle viste locali e la creazione della vista globale. Invece, nella cartella `common` si trovano le funzioni "in comune" per tutti i moduli, quindi la gestione e minimizzazione degli automi (in `fsa.erl`), la conversione di un grafo in formato DOT (in `digraph_to_dot.erl`) e il salvataggio su file (in `common_fun.erl`). Il file `common_data.hrl` contiene le strutture dati utilizzate nel progetto.

2.2.1 From Erlang to Choreographies

In questa sezione, verranno definite le corrispondenze tra codice in Erlang e automa coreografico, rispettivamente per viste locali e globali.

Viste locali Il codice di un'operazione `receive` (come nel codice **??**) corrisponde al grafo 4; ogni ramo verrà valutato ricorsivamente, continuando quindi la costruzione dai rami; infine, tutti i rami si ricongiungeranno su un nodo comune con delle transizioni ϵ, dal quale ripartirà la valutazione della vista locale. L'operazione `Pid ! message` corrisponde al grafo 5. La chiamata alla funzione `spawn` corrisponde al grafo 6.

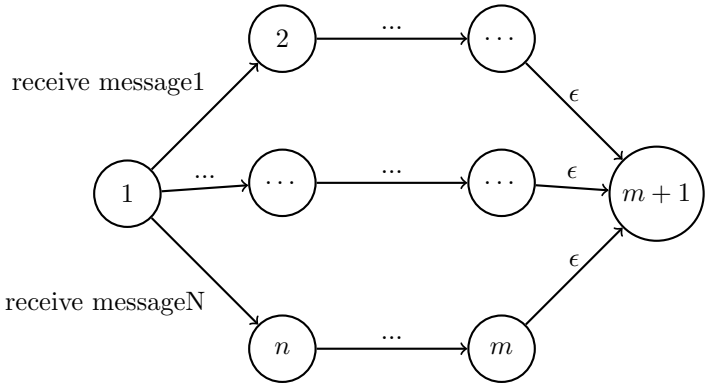

Figure 4: Grafo locale per il costrutto `receive`

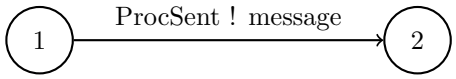

Figure 5: Grafo locale per il costrutto `!`

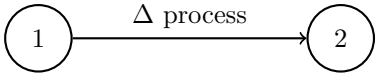

Figure 6: Grafo locale per il costrutto `spawn`

Chiamate di funzioni Essendo un linguaggio di programmazione funzionale, vengono eseguite numerose chiamate di funzione. Le funzioni possono essere presenti nello stesso file, in un modulo differente, oppure possono essere funzioni *built-in*. Di seguito vengono mostrate le chiamate di funzioni che modificano il comportamento del grafo.

Chiamate ricorsive Nel caso delle chiamate *ricorsive*, si crea una transizione ϵ dall'ultimo nodo creato fino al primo nodo, come mostrato in figura 7.

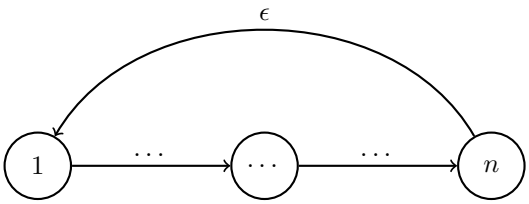

Figure 7: Grafo della chiamata ricorsiva di una funzione

Chiamata a una funzione generica Quando si incontra in chiamata ad una funzione non conosciuta, l'algoritmo creerà la vista locale della funzione chiamata e "collegherà" l'inizio del grafo della funzione chiamata con l'ultimo stato creato nella vista locale della funzione chiamante, collegandolo con una transizione ϵ. La vista locale continuerà dall'ultimo vertice del grafo della chiamata.

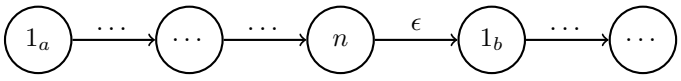

Figure 8: Grafo di una chiamata di funzione

Nel grafo 8, lo stato 1_a è lo stato iniziale della funzione chiamante e lo stato 1_b indica il primo stato della vista locale della funzione chiamata.

Viste globali Per le viste globali, nelle `spawn` si specifica l'attore che esegue l'operazione a sinistra del simbolo come mostrato nella figura 9. Una spawn verrà direttamente inserita nel grafo. Ogni processo verrà anche numerato, nel caso vengano creati molteplici attori per la stessa funzione.

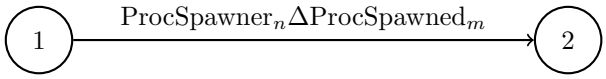

Figure 9: Grafo globale per il costrutto `spawn`

Per l'invio e la ricezione di messaggi, durante la simulazione degli attori, se vengono trovati due attori che eseguono una send e una receive compatibile, allora verrà aggiunta alla vista globale uno stato come in figura 10. Per essere compatibili, il processo ricevente deve corrispondere al destinatario dei dati e il pattern matching della receive deve combaciare con i dati inviati. Le transizioni *send* e *receive* fatte a "vuoto" (cioè i messaggi che vengono inviati, ma non vengono processati da una *receive* di un qualsiasi processo) non verranno mostrate nell'automa globale.

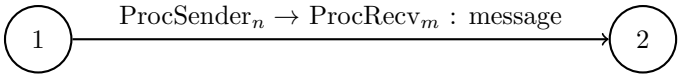

Figure 10: Grafo globale per `receive` e `!`

2.2.2 Main algorithms

Execution L'esecuzione è divisa in 3 fasi principali:

1. inizializzazione del `db_manager`, delle strutture dati e estrazione delle informazioni preliminari (se ne occupa il modulo `metadata.erl`): vengono estratti i possibili attori dall'attributo `export`, vengono contate quante `spawn` vengono eseguite e vengono salvati gli AST di tutte le funzioni nel `db_manager`. Nel mentre, viene effettuata una prima valutazione del flusso del programma, inizializzando i nomi degli attori e salvando i passaggi degli argomenti alle `spawn`;

2. creazione delle viste locali di tutti i possibili attori, cioè di tutte le funzioni che compaiono nell'`export` all'inizio di un programma (ottenuti dalla prima fase);

3. creazione della vista globale a partire dal punto di inizio del programma e componendo le viste locali create nella fase due.

Modulo `local_view.erl` Nel modulo che crea viste locali, la funzione principale è `eval_codeline`. La funzione valuta la singola linea di codice e i suoi argomenti. Inoltre, aggiunge nodi al grafo della vista locale nel caso dei costrutti di comunicazione. La funzione crea un legame tra una variabile e il suo contenuto, se valutabili. Inoltre, crea biforcazioni con archi ϵ nel caso di `if` e `case`. In seguito, le transizioni ϵ verranno eliminate tramite la minimizzazione del grafo. Sono anche presenti alcune valutazioni di funzioni *built-in* utili per la comunicazione, come `self()` che restituisce l'id del proprio processo e `register` che registra l'id di un processo in un *atom*.

Di seguito, ne mostriamo lo pseudo codice di alcuni rami interessanti.

```
eval_codeline(CodeLine, FunctionName, UsefulData) ->
    case CodeLine of
        %%% Eval recursive call
        {call, _, {atom, _, FunctionName}, ArgList} ->
            manage_recursive_call();
        %%% Evaluate the spawn function
        {call, _, {atom, _, spawn}, ArgList} ->
            add_spawn_to_local_view();
        %%% Evaluate a call to a generic function
        {call, _, {atom, _, Name}, ArgList} ->
            manage_call();
        %%% Eval Var = something
        {match, _, RightContent, LeftContent} ->
            manage_var();
        %%% Evaluate case with pattern matching
        {'case', _, Data, PMList} ->
            manage_case();
        %%% Evaluate if like case
        {'if', _, PMList} ->
            manage_if();
        %%% Evaluate receive with pattern matching
        {'receive', _, PMList} ->
            add_receive_to_local_view();
        %%% Evaluate send
        {op, _, '!', ProcSent, DataSentAst} ->
            add_send_to_local_view();
        %%% Evaluate data types
        {atom, _, Value} -> return_var();
        {integer, _, Value} -> return_var();
        {string, _, Value} -> return_var();
        ...
        _ -> nomatch
    end.
```

Listing 4: Codice di `eval_codeline`

N.B.: in Erlang, difficilmente si specifica da quale processo si vuole ricevere un determinato messaggio perché non si sa a priori quali attori ci sono.

Di conseguenza, la vista globale userà l'etichetta `receive msg` per esprimere il ramo dove si riceve uno specifico messaggio. Verranno anche specificati i pattern matching, valutando dove è possibile, per facilitare la creazione della vista globale.

Nella sezione 2.2.1 sarà esplicitata la corrispondenza tra codice e grafo per i costrutti che modificano la vista locale. Invece, i rami che corrispondono ai tipi dei dati o alle funzioni *built-in* (come `self()`) ritornano una struttura dati che rappresenta il dato, che eventualmente sarà legata ad una variabile nel ramo `match` oppure sarà passata in una funzione come argomento.

Modulo `global_view.erl` Se per creare una vista locale basta seguire il codice della funzione linea per linea, nella vista globale bisogna comporre le viste locali tenendo conto dei vari attori creati. Di conseguenza verrà simulata una esecuzione approssimata a partire dalla funzione di avvio. Ad ogni attore verrà associato un processo della funzione `proc_loop`, che mantiene le informazioni sui rami disponibili e in quale stato si trova questo processo. Questa funzione si occupa di fornire al processo principale alcune informazioni relative all'attore.

```
1  proc_loop(LocalView, CurrentState, MarkedEdges) ->
2      receive
3          {use_transition, Edge} ->
4              % per evitare loop infiniti
5              NewState = verify_not_marked();
6              proc_loop(LocalView, NewState, MarkedEdges);
7          {P, get_info} ->
8              P ! {someinfo},
9              proc_loop(LocalView, CurrentState, MarkedEdges);
10         stop -> terminated
11     end.
```

Listing 5: Funzione `proc_loop` che simula un attore

Inoltre, i messaggi potrebbero viaggiare sulla stessa macchina virtuale o attraverso la rete, quindi lo scambio di messaggi avviene in modo totalmente *asincrono*, cioè se vengono effettuate due send A e B verso lo stesso processo, potrebbe arrivare prima A o B e cambiare completamente l'esecuzione del programma.

Nel seguente codice 6, è presente la funzione principale del modulo `global_view.erl`, che crea la vista globale combinando gli attori e le loro viste locali. L'idea di base è quella di creare una *vista in profondità* (in inglese Depth-first search, DFS) degli automi degli attori, fermandosi in modo opportuno secondo le regole di comunicazione.

```
1  progress_branches(BranchList) ->
2      NewBranchList = lists:foreach(
3          fun(Item) -> progress_single_branch(Item) end,
4          BranchList
5      ),
6      progress_branches(NewBranchList).
7  progress_single_branch(Data) ->
8      SendList = lists:foreach(
9          fun(Name) -> eval_proc_until_send(Name) end,
10         Data.proc_list
11     ),
12     lists:foreach(
13         fun(SendData) ->
14             manage_send(duplicate_branch(SendData))
15         end,
16         SendList
17     ).
```

Listing 6: Codice principale per costruire una vista globale

La strategia di composizione delle viste locali che viene adottata è quella di far proseguire tutti gli attori fino a una qualsiasi *send* (riga 3 del codice 6). Mentre viene cercata e trovata la prima *send*, vengono anche cercate *spawn* e *receive*. Se si incontra una spawn, viene subito aggiunto il corrispettivo nodo nel grafo e creato l'attore. Invece, se viene incontrata una *receive*, si controlla la coda dei messaggi del processo. Se ne viene trovata una compatibile, allora si crea la transizione sul grafo e i due attori proseguiranno l'esecuzione.

I processi, dopo essere stati bloccati su una *send* o *receive*, varranno duplicati su rami diversi per esecuzioni diverse (riga 14 del codice 6). Ogni "ramo" dell'esecuzione differirà in base a quale *send* valutare prima. Contemporaneamente, viene controllato se è presente un processo che può ricevere quel messaggio. In caso affermativo, viene creata la transizione sul grafo globale e vengono fatti proseguire gli attori. Altrimenti, la *send* "vacante" viene inserita in una struttura dati opportuna.

L'algoritmo 6 viene eseguito ricorsivamente finché una struttura dati rilevante viene modificata (come il grafo). Alla prima iterazione che non modifica nessuna struttura dati, l'algoritmo si fermerà. Il grafo finale rappresenterà la comunicazione avvenuta in modo asincrono per i messaggi che partono da diversi attori. Vengono fatti esempi di "diramazioni" tra *send* nella sezione degli esempi.

Ogni modulo, dopo aver creato i rispettivi automi, si occuperà di convertirli in linguaggio DOT e salvarli su file. Per visualizzare i grafi basterà copiare il contenuto del file su un applicativo che interpreta il formato DOT.

Old state The tool can accurately parse the main constructs and features of the Erlang language. In Table **??**, we outline the current status: what is supported, what is not supported, and what are the next objectives. It was

decided not to support certain constructs because they are not useful at this stage of the project. For example, supporting error handling with try-catch is unnecessary when variable passing is not yet implemented. Additionally, some constructs were excluded because they are orthogonal to others, such as complex data structures. The tool generates a local view that closely aligns with the behavior of the original actor, and the algorithm for combining local views into a global view performs well on simple examples. However, at present, it does not provide automatic error detection for certain properties in the global choreography.

2.3 What's changed

2.3.1 Attributed grammar

2.3.2 New examples

In order to better illustrate our point, we present in this section two examples (using Erlang-like, actor-based pseudocode), each one composed by a pseudocode and the corresponding Choreography Automaton [3] (a graphical way of representing a finite-state machine) 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 [13]; the corresponding automata have been obtained with the help of our prototype tool Chorer [7].

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).

Dining Philosophers Example 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,
```

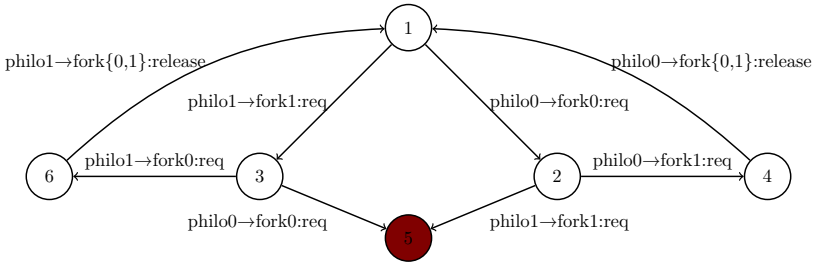

Figure 11: Global view of the dining philosophers example.

```
  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 11 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.

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 12. We can observe two correct executions where the operations are performed in

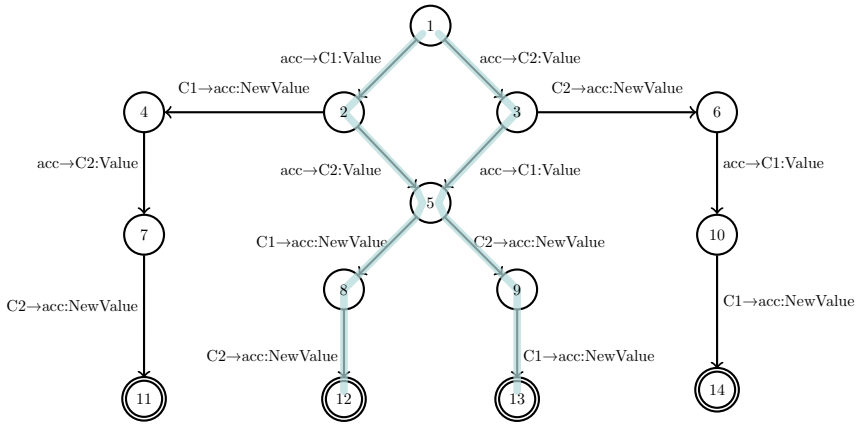

Figure 12: Global view of the bank account example

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 12 helps in spotting this issue.

2.3.3 Feature

2.3.4 Bug Fix

2.3.5 Benchmarks

3 Conclusion

3.1 Related works

Software industry is increasingly devoting attention to choreographic approaches [34, 4, 14, 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 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. To the best of our knowledge, the first attempts to distil global specifications for message-passing systems goes back to [33, 26, 28]. These approaches aim to identify "meaningful" global specifications according to general properties such as deadlock-freedom or absence of orphan messages [5]. 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 [26, 28] as abstract models (respectively, π-calculus processes [36, 29, 30] and communicating finite-state machines [5]).

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 *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 22*, pages 86–106. Springer, 2020.

[3] 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.

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

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

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

[7] Chorer github repository. `https://github.com/gabrielegenovese/chorer`.

[8] 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].

[9] 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.

[10] Luís Cruz-Filipe, Kim S Larsen, and Fabrizio Montesi. The paths to choreography extraction. In *International Conference on Foundations of Software Science and Computation Structures*, pages 424–440. Springer, 2017.

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

[12] 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.

[13] 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`.

[14] 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.

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

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

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

[18] Patrice Godefroid. Model checking for programming languages using verisoft. In *POPL*, pages 174–186, 1997.

[19] Google. GoLang. `https://go.dev/`, 2009. [Online; accessed 26-May-2023].

[20] 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.

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

[22] Hans Hüttel, Ivan Lanese, Vasco T Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, et al. Foundations of session types and behavioural contracts. *ACM Computing Surveys (CSUR)*, 49(1):1–36, 2016.

[23] 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.

[24] 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].

[25] 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/`.

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

[27] Julien Lange, Emilio Tuosto, and Nobuko Yoshida. From communicating machines to graphical choreographies. In Sriram K. Rajamani and David Walker, editors, *POPL*, pages 221–232. ACM, 2015.

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

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

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

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

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

[33] 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.

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

[35] 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.

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

[37] The Elixir Team. Elixir. `https://elixir-lang.org/`, 2012. [Online; accessed 26-May-2023].