Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica

ChorEr: un analizzatore statico per generare Automi Coreografici da codice sorgente Erlang

Relatore:
Prof. Ivan Lanese

Presentata da:
Gabriele Genovese

Sessione I
Anno Accademico 2022/2023

"Try to leave this world a little better than you found it..."

Robert Baden-Powell

Sommario

Il recente sviluppo di applicazioni concorrenti e distribuite ha dato origine a un nuovo interesse verso il linguaggio Erlang (ed Elixir che si basa su una Erlang Virtual Machine BEAM) e verso altri linguaggi e paradigmi di programmazione che implementano nella loro logica l'uso di thread e di message passing. Il modello ad attori e le coreografie rivestono un ruolo importante nel futuro dei sistemi mobili e distribuiti: il modello ad attori è un modello concorrente sicuro e intuitivo; invece, le coreografie permettono di evitare sintatticamente alcuni degli errori tipici dei sistemi concorrenti (liveness, lock freedom e deadlocks freedom). In questa tesi viene presentato ChorEr, un tool di analisi statica per programmi in Erlang che genera degli Automi Coreografici, cioè degli automi a stati finiti che permettono una verifica semplificata delle proprietà citate.

Parole chiave: coreografie, analisi statica, concorrenza, Erlang.

Indice

Elenco delle figure

Elenco delle tabelle

Listings

Capitolo 1

Introduzione

Recentemente, il mondo dell'informatica ha visto numerosi cambiamenti. Con il tempo, internet ha raggiunto, in modo capillare, ogni posto della terra e quindi può capitare di trovarsi da un giorno all'altro con migliaia o anche milioni di utenti che usano la propria applicazione o servizio lanciata sul mercato. Lo sviluppo di applicazioni concorrenti distribuite, scalabili e robuste è diventata una necessità e la comunità scientifica ha risposto a questa sfida, formalizzando nuovi modelli e paradigmi per assicurare determinate proprietà.

Uno dei modelli più usati al giorno d'oggi per la programmazione concorrente è il *modello ad attori*. Gli "attori", che tipicamente sono processi concorrenti e leggeri, non condividono memoria tra di loro, quindi comunicano tramite l'invio e la ricezione di messaggi (*message passing*). La comunicazione avviene attraverso delle primitive definite dal linguaggio chiamate *send* (invio di un messaggio) e *receive* (ricezione di un messaggio da un qualsiasi o da uno specifico attore). Questo tipo di modello è implementato in modo semplice ed efficace dal linguaggio di programmazione Erlang [7], che implementa primitive per creare piccoli processi isolati molto leggeri (tramite la funzione `spawn`) e per inviare e ricevere messaggi (tramite i costrutti `!` e `receive`).

Erlang è un linguaggio di programmazione funzionale nato nel contesto delle telecomunicazioni, quando c'era bisogno di un linguaggio che permettesse la costruzione di applicazioni distribuite, con alta tolleranza ai guasti e tempo di funzionamento (*uptime*) molto alto. Anche oggi, Erlang viene usato principalmente per la costruzione di applicazioni distribuite, scalabili e robuste o per creare servizi che gestiscono migliaia o milioni di connessioni e messaggi ogni giorno. Queste caratteristiche hanno anche ispirato altri linguaggi di programmazione o framework, come Elixir [22], un linguaggio di programmazione basato sulla macchina virtuale di Erlang; oppure Akka [15], un toolkit scritto in Scala che implementa il modello ad attori per la Java Virtual Machine. Anche altri linguaggi di programmazione implementano il modello ad attori, come Go [10] con le GoRoutine (paragonabili alle spawn di Erlang) e i canali (simili alle send e receive di Erlang) o come il framework Actix [21] che implementa il modello ad attori in Rust [23].

Questo nuovo modo di vedere e creare i programmi concorrenti ha fatto concentrare la comunità scientifica sul risolvere problemi comportamentali noti da tempo, ma di difficile risoluzione, e sulla verifica di proprietà semantiche. Di seguito, ne vengono informalmente descritte alcune:

- Liveness: stabilisce il "progresso" del sistema di comunicazione interessato, cioè il continuo evolvere del percorso, evitando di bloccarsi;
- Lock Freedom: assicura che eventualmente l'automa giungerà in uno stato finale e che quindi determinate comunicazioni vengano eseguite;
- Deadlock Freedom: assicura che non ci siano situazioni di stallo, cioè situazioni dove il programma si ferma e non prosegue.

Uno dei modelli matematici nati negli ultimi anni sono le coreografie: un modello formale per rappresentare sistemi di processi comunicanti attraverso le quali è possibile dimostrare semanticamente la presenza o meno delle proprietà appena descritte.

Le coreografie sono studiate in ambito scientifico da molti anni e in vari modi. Nella teoria dei tipi, si sono consolidati i multiparty session types che descrivono la struttura interattiva di un numero fisso di attori da un punto di vista globale e che verifica sintatticamente la correttezza degli attori attraverso la proiezione della vista globale sui partecipanti [2].

Vengono formalizzate anche nei contratti (che sono la descrizione astratta del comportamento di un programma, in inglese *multyparty contracts*) [12] oppure vengono anche formalizzate come paradigma di programmazione in modo teorico in [3], ma anche a livello industriale in [13]. Sono stati creati e studiati anche molti linguaggi di programmazione coreografici [17, 9, 4].

Un modo per formalizzare le coreografie sono gli Automi Coreografici [1], che descrivono un sistema di comunicazione tramite un automa a stati finiti. Grazie all'uso di questo modello è possibile utilizzare i risultati degli studi sugli automi per dimostrare le proprietà citate prima [19]. In questa tesi, verrà presentato un tool per creare l'Automa Coreografico di un programma in Erlang.

Capitolo 2

Nozioni preliminari

In questo capitolo verranno introdotti i riferimenti formali definiti dalla comunità scientifica e utili alla comprensione di questa tesi: gli automi a stati finiti. In seguito verranno definiti formalmente i sistemi di comunicazione e le coreografie.

2.1 Automi a Stati Finiti

Prima di parlare degli Automi Coreografici, è necessario introdurre alcune nozioni alla base della teoria dei linguaggi e della computazione: gli Automi a Stati Finiti, abbreviati da qua in avanti con l'acronimo FSA (dall'inglese Finite State Automata). Un FSA è un modello matematico di calcolo che permette di astrarre e di formalizzare il comportamento di molti sistemi. Alcuni oggetti o sistemi che sfruttiamo tutti i giorni sono degli FSA implementati in modo elettronico o digitale. Alcuni esempi sono i distributori automatici, i semafori, gli ascensori e i tornelli della metropolitana. In Informatica, gli FSA sono spesso usati come analizzatori lessicali per riconoscere linguaggi ed espressioni regolari. A seconda di come sono definiti, possono esistere diversi tipi di FSA. I più comuni e studiati sono quelli deterministici (DFA) e non deterministici (NFA). In questa tesi verranno usati gli Automi Coreografici, che sono un tipo particolare di FSA.

Gli FSA non sono gli unici modelli matematici di calcolo. Per esempio, esistono le Macchine di Turing che sono in grado di implementare qualsiasi algoritmo, ma in questo contesto si è preferito sfruttare la semplicità degli FSA. Inoltre, usando un modello ampiamente studiato come gli FSA, è possibile riutilizzare tutti i teoremi dimostrati e gli algoritmi relativi alla teoria degli automi e dei linguaggi formali. In questa sezione, riportiamo solo le definizioni e gli algoritmi usati per questa tesi.

2.1.1 Definizioni

Di seguito, verranno riportate alcune delle definizioni prese da [16] e tradotte da [1].

Definizione 2.1.1 (Sistema a Transizioni Etichettato)**.** Un Sistema a Transizioni Etichettato (*Labelled Transition System*, abbreviato da qua in poi *LTS*) è una tupla $A = \{\mathbb{S}, s_0, \mathcal{L}, \rightarrow\}$ dove:

- $\mathbb{S}$ è un insieme di stati (chiamati s, q, ...) e $s_0 \in \mathbb{S}$ è lo *stato iniziale*;
- $\mathcal{L}$ è un insieme finito di etichette;
- $\rightarrow \subseteq \mathbb{S} \times (\mathcal{L} \cup \{\epsilon\}) \times \mathbb{S}$ è un insieme di transizioni dove $\epsilon \notin \mathcal{L}$ è un'etichetta distinta.

N.B.: l'insieme di stati $\mathbb{S}$ può essere infinito. Verrà usata la notazione $s_n \xrightarrow{\lambda} s_m$ per denotare una qualsiasi transizione $(s_n, \lambda, s_m) \in \rightarrow$, per un qualunque $\lambda \in \mathcal{L}$ e $s_n, s_m \in \mathbb{S}$. Il simbolo ϵ rappresenta la mossa nulla, cioè senza etichetta.

Definizione 2.1.2 (Run e traccia)**.** Un **run** di un LTS $A = \langle \mathbb{S}, s_0, \mathcal{L}, \rightarrow \rangle$ è una (possibilmente vuota) finita o infinita sequenza di transizioni consecutive che iniziano da s_0. La *traccia* (o parola) w di una run di A è una concatenazione delle etichette della run.

Definizione 2.1.3 (Automa a Stati Finiti non deterministico)**.** Un automa a stati finiti non deterministico (non deterministic finite state automaton, per brevità: NFA) è un caso particolare di LTS, dove:

- $\mathbb{S}$ è un insieme di stati finito;
- $F \subseteq \mathbb{S}$ è l'insieme di stati finali (o "*accettanti*");
- l'insieme $\rightarrow$ diventa un funzione con tipo

$$\delta : \mathbb{S} \times (\mathcal{L} \cup \{\epsilon\}) \rightarrow \mathcal{P}(\mathbb{S})$$

  che dunque associa un insieme di stati ad ogni coppia (s, l) formata da uno stato e da un simbolo di input.

$\mathcal{P}$ è la funzione che computa l'insieme delle parti.

Informalmente, possiamo vedere un NFA come un LTS, ma con alcune modifiche. Un NFA avrà un insieme di stati finiti, un sottoinsieme di stati finali e, al posto di un insieme di transizioni possibili, userà una *funzione di transizione* per determinare in quale stato finisce la transizione.

Esempio 2.1.1. Sia $\Sigma = \{a, b\}$, $Q = \{q_0, q_1, q_2\}$ con q_0 stato iniziale e q_2 stato finale. Possiamo rappresentare la funzione di transizione con una tabella (chiamata *tabella delle transizioni*). La tabella 2.1 rappresenta le transizioni di questo esempio.

δ	a	b	ϵ
q_0	$\{q_0\}$	$\{q_0, q_1\}$	$\emptyset$
q_1	$\{q_2\}$	$\emptyset$	$\emptyset$
q_2	$\emptyset$	$\emptyset$	$\emptyset$

Tabella 2.1: Tabella delle transizioni dell'esempio 2.1.1

È possibile rappresentare un NFA in modo intuitivo ed efficace attraverso un *diagramma di transizione*. Nel diagramma, i nodi rappresentano gli stati dell'automa; lo stato iniziale è indicato con una freccia entrante; gli stati finali con un doppio cerchio e le transizioni con archi che partono dallo uno stato di partenza e giungono in uno stato di arrivo, i quali sono etichettati con un simbolo di $\mathcal{L}$ oppure il simbolo ϵ. Nel caso degli NFA, possono partire più archi da uno stesso stato con le stesse etichette, il non determinismo sta quindi nella scelta di quale transizione usare. Il diagramma 2.1 rappresenta l'NFA dell'esempio 2.1.1.

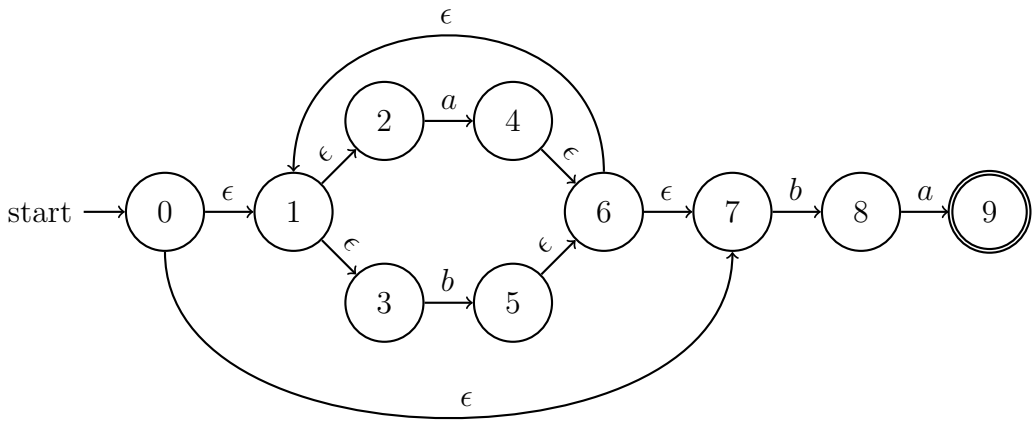

Figura 2.1: Digramma delle transizioni dell'esempio 2.1.1

Possiamo dire, informalmente, che l'NFA definito nell'esempio 2.1.1 *riconosce* l'espressione regolare $(a|b)^*ba$.

NB: Questo è *uno* dei possibili NFA in grado di riconoscere questa espressione regolare.

Di seguito, definiamo un FSA che è un caso particolare di NFA: l'automa a stati finiti deterministico.

Definizione 2.1.4 (Automa a Stati Finiti Deterministico)**.** Un automa a stati finiti deterministico (deterministic finite state automaton, per brevità: DFA) è una quintupla $(\mathcal{L}, \mathbb{S}, \delta, s_0, F)$ dove $\mathcal{L}$, $\mathbb{S}$, q_0 e F sono come in un NFA, mentre la funzione di transizione ha tipo $\delta : \mathbb{S} \times \mathcal{L} \to \mathbb{S}$.

In modo più informale, possiamo dire che un DFA è un NFA senza transizioni ϵ e l'insieme delle mosse possibili dato un simbolo di input è al massimo un singoletto (un insieme di un elemento), cioè $|\delta(s, l)| \leq 1, \forall s \in \mathbb{S}, \forall l \in \mathcal{L}$.

I due tipi di automi rappresentano la stessa classe di linguaggi e sono ugualmente espressivi. Inoltre, è sempre possibile passare da un NFA ad un DFA, attraverso un algoritmo. L'algoritmo in questione usa le nozioni di ϵ-closure e della funzione *mossa*, definite di seguito.

Definizione 2.1.5 (ϵ-closure)**.** Sia N un NFA e q uno stato qualsiasi di N. La ϵ-closure di q è l'insieme degli stati raggiungibili da q solo con mosse ϵ (incluso lo stato q). Se X è un insieme di stati, definiamo l'ϵ-closure di X come l'unione delle ϵ-closure di tutti gli stati in X.

Definizione 2.1.6 (Funzione Mossa)**.** Sia N un NFA, $a \in \Sigma$ un qualsiasi simbolo di input del NFA e $X \subseteq Q$ un sotto-insieme degli stati del NFA. Definiamo la funzione mossa come $\mathcal{P}(Q) \times \Sigma \to \mathcal{P}(Q)$ tale che $\text{mossa}(X, a) = \bigcup_{x \in X} \delta(x, a)$, ovvero l'insieme degli stati raggiungibili da un dato insieme di stati di partenza, leggendo il simbolo a in input.

2.1.2 Algoritmi usati

Convertire un NFA in un DFA

Di seguito, viene definito lo pseudo-codice dell'algoritmo per l'ϵ-closure usato nel progetto.

Algorithm 1 ϵ-closure

Require: q stato di un NFA

$\mathcal{E} \leftarrow$ prendo la lista delle transizioni uscenti dallo stato q
$\mathcal{L} \leftarrow \{q\}$
for all $e \in \mathcal{E}$ **do**
  if e non è marcato e Label $= \epsilon$ **then**
    marco(e) ▷ per evitare loop infiniti
    $\mathcal{L} \cup \epsilon$-clos(stato dove arriva e)
  end if
end for
return L

Avendo definito tutto quello che serve, di seguito verrà presentato lo pseudo-codice dell'algoritmo per ricavare un DFA da un qualsiasi NFA. L'idea di base è quella di considerare uno stato del DFA come un sottoinsieme degli stati del NFA, da questo "costruzione per sottoinsiemi".

Algorithm 2 Costruzione per sottoinsiemi

$x \leftarrow \epsilon$-clos(s_0) ▷ x sarà il primo stato del DFA
$\mathcal{T} \leftarrow \{x\}$
while $\exists t \in \mathcal{T}$ non marcato **do**
  marco(t)
  for all $\alpha \in \mathcal{L}$ **do** ▷ Per ogni simbolo nell'insieme dei simboli
    $r \leftarrow \epsilon$-clos(mossa(t, α)) ▷ Creo un nuovo stato per il DFA
    if $r \notin \mathcal{T}$ **then** ▷ Aggiungo r a $\mathcal{T}$ se non presente
      $\mathcal{T} \leftarrow \mathcal{T} \cup \{r\}$
    end if
    $\delta(t, \alpha) = r$ ▷ Creo la transizione per il DFA
  end for
end while

L'algoritmo 2 produce in output $x, \mathcal{T}$ e δ che sono rispettivamente lo stato iniziale, l'insieme degli stati e le transizioni del DFA appena creato.

Applicando l'algoritmo al NFA 2.1, si ottiene la seguente tabella 2.2, dove si osserva come uno stato del DFA sia un sottoinsieme di stati dell'NFA. Si può rappresentare il DFA con il diagramma 2.2.

Stato DFA	Stati del NFA
A	$\{0, 1, 2, 3, 7\}$
B	$\{1, 2, 3, 4, 6, 7\}$
C	$\{1, 2, 3, 5, 6, 7, 8\}$
D	$\{1, 2, 3, 4, 6, 7, 9\}$

Tabella 2.2: Sottoinsiemi degli stati del NFA in figura 2.1

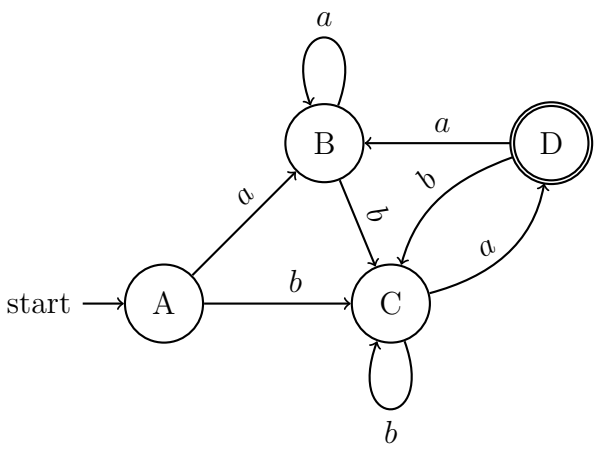

Figura 2.2: DFA dell'esempio 2.1.1

N.B.: ogni stato ha una sola transizione per tutti i simboli.

Minimizzare un DFA

L'algoritmo 2, nel caso peggiore, potrebbe costruire un DFA con 2^n stati partendo da un NFA con n stati. Quindi, sono stati pensati diversi algoritmi per rendere "minimo" un DFA, cioè per ridurre il numero di stati usati mantenendo però il linguaggio accettato.

In genere, si effettuano 3 tipi di operazioni diverse:

- rimozione degli stati non raggiungibili;
- rimozione degli stati "*morti*" (stati dai quali non sono raggiungibili stati finali) e delle transizioni a loro collegati;
- "*fusione*" degli stati non distinguibili.

Nel progetto presentato insieme a questa tesi, sono stati implementati gli algoritmi per la rimozione degli stati non raggiungibili e non distinguibili. Non è stato però implementato un algoritmo per la rimozione degli stati morti perché avere degli stati morti può essere rilevante ai fini della coreografia. L'algoritmo per la rimozione degli stati non raggiungibili è di banale implementazione, al contrario di quello per la fusione degli stati non distinguibili.

Di seguito, viene definito l'algoritmo di Riempimento della Tabella a Scala, un algoritmo che trova gli stati non distinguibili. Si usa una tabella a scala perché mettendo le coppie degli stati in una tabella ed eliminando le coppie con stati uguali e le coppie ripetute si ottiene una tabella a scala.

Lo pseudo-codice dell'algoritmo 3 marca tutti gli stati distinti e, quando termina, gli stati non marcati si possono considerare indistinguibili l'uno dall'altro. All'inizio si marcano le coppie di stati finali e non finali, perché intrinsecamente diverse. Per le iterazioni successive vengono usate le transizioni, aumentando di uno il salto ogni volta che si fa una nuova iterazione. Quando si arriva a non marcare niente per una iterazione l'algoritmo termina.

Algorithm 3 Riempimento della Tabella a Scala

Inizializza la tabella a scala con le coppie (p, q)
Marca con x_0 le coppie (m, n) con $m \in F$ e $n \notin F$
while $\exists$ almeno un marchio x_i all'iterazione i **do**
  if $\exists \alpha \in \mathcal{L}, \exists p, q \in \mathbb{S}$ tale che $\delta(p, \alpha), \delta(q, \alpha))$ è marcata **then**
    Marca (p, q) con marca x_i
  end if
  Considera all'iterazione seguente solo gli stati non marcati
end while

Esempio 2.1.2. Eseguendo l'algoritmo 3 sul DFA 2.2, otterremo la seguente tabella 2.3:

B		//	//
C	x_1	x_1	//
D	x_0	x_0	x_0
	A	B	C

Tabella 2.3: Tabella a scala costruita usando l'algoritmo 3

Da questo esempio, si può notare come tutte le coppie di stati sono marcate, tranne A e B. Di conseguenza, A e B sono equivalenti.

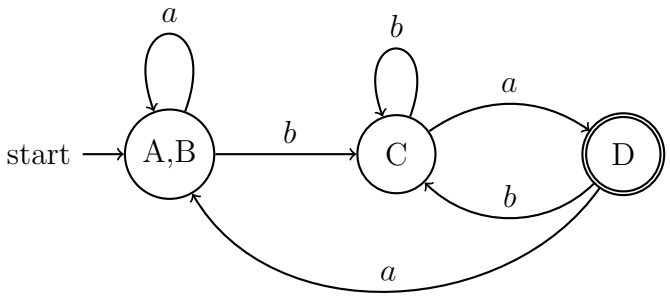

Figura 2.3: Diagramma delle transizioni del DFA 2.2 minimizzato

2.2 Coreografie

Definite tutte le nozioni e algoritmi che ci interessano, possiamo ora parlare delle coreografie e degli Automi Coreografici.

Le coreografie sono un modello logico che descrive il comportamento tra due o più attori. Invece, gli Automi Coreografici sono una tipologia di rappresentazione di coreografie, tramite Automi a Stati Finiti [1]. La rappresentazione delle coreografie tramite FSA si adatta molto bene a rappresentare il flusso del programma, potendo mostrare con chiarezza loop e diramazioni, oltre a poter usare risultati e nozioni descritti prima.

2.2.1 Viste locali e globali

Le coreografie possono essere intese da due "punti di vista". Il punto di vista del singolo attore (anche detto *local view*) mostra il comportamento di un partecipante alla coreografia. Chiaramente, questo tipo di vista risulta essere limitato poiché l'attore non è a conoscenza degli altri partecipanti alla coreografia. Invece, il punto di vista d'insieme o globale (anche detto *global view*) unisce il comportamento di tutti i partecipanti alla comunicazione e quindi restituisce una visione completa del sistema di comunicazione.

2.2.2 Automi Coreografici

Di seguito, riportiamo alcune nozioni, tradotte dallo studio [1], per capire appieno la definizione di Automi Coreografici.

Definizione 2.2.1 (Communicating Finite State Machine)**.** Una Macchina a Stati Finiti di Comunicazione (*Communicating Finite State Machine*, abbreviato da qua in poi *CFSM*) è un FSA C con un insieme finito di etichette

$$\mathcal{L}_{act} = \{A\,B\,!\,m,\ A\,B\,?\,m \mid A,\ B \in \mathcal{P}, m \in \mathcal{M}\}$$

di azioni; dove $\mathcal{P}$ e $\mathcal{M}$ sono:

- $\mathcal{P}$ è l'insieme dei partecipanti (per esempio A, B, etc...);
- $\mathcal{M}$ è l'insieme dei messaggi che possono essere scambiati (m, n, etc...).

Il *soggetto* di un'azione di *output* per $A\,B\,!\,m$ è A. Al contrario, il *soggetto* di un'azione di *input* per $A\,B\,?\,m$ è B. Un CFSM è *locale ad* A (A-*local*) se tutte le sue transizioni hanno come soggetto A.

Definizione 2.2.2 (Sistema di Comunicazione)**.** Un Sistema di Comunicazione è una mappa $S = (M_A)_{A \in \mathcal{P}}$, che quindi assegna un *A-local CFSM* M_A ad ogni partecipante in $\mathcal{P}$, tale che ogni partecipante che appare in M_A è anch'esso in $\mathcal{P}$.

Alcune condizioni dipendono dall'infrastruttura di comunicazione, quindi la semantica può essere sincrona o asincrona. La semantica sincrona di un sistema di comunicazione è un LTS dove le etichette sono *interazioni*:

$$\mathcal{L}_{int} = \{A \to B : m \mid A \neq B \in \mathcal{P}, m \in \mathcal{M}\}$$

con $\mathcal{P}$ e $\mathcal{M}$ definiti come nella definizione 2.2.1.

Infine, di seguito ecco la definizione di Automi Coreografici:

Definizione 2.2.3 (Automi Coreografici)**.** Un Choreography automaton (*c-automaton*, in italiano Automi Coreografici) è un FSA sull'alfabeto $\mathcal{L}^*_{int}$. Gli elementi di $\mathcal{L}_{int}$ sono le parole coreografiche; i sottoinsiemi di $\mathcal{L}^*_{int}$ sono dei linguaggi coreografici.

Quindi, possiamo vedere gli Automi Coreografici come degli FSA senza transizioni ϵ e con un insieme di etichette definite come su $\mathcal{L}_{int}$. Bisogna specificare però che, anche se non sono ammesse ϵ-transizioni, l'automa può rimanere non deterministico.

2.2.3 Esempio di Automa Coreografico

Di seguito, ecco due viste locali di due partecipanti alla comunicazione. Da notare come le due viste locali corrispondano alla definizione di CFSM.

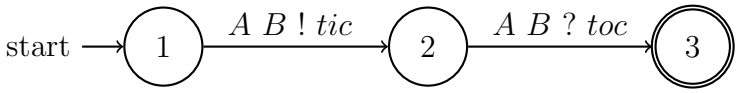

Figura 2.4: Vista locale del partecipante A

In questo caso il partecipante A manda al partecipante B il messaggio `tic` e poi si aspetta come risposta da B `toc`.

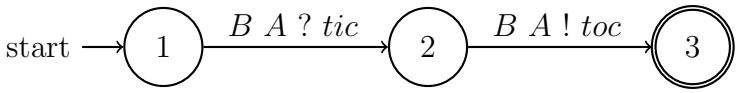

Figura 2.5: Vista locale del partecipante B

In modo del tutto speculare, B si aspetta da A `toc` e risponde ad A con il messaggio `toc`.

Di seguito, ecco la global view del sistema di comunicazione:

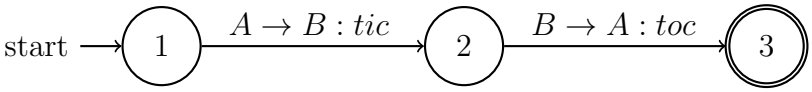

Figura 2.6: Global view di esempio

Attraverso una composizione delle viste locali, viene prodotto l'automa coreografico della vista globale che descrive il passaggio di due messaggi. L'operazione di composizione non verrà definita in modo formale perché nel progetto viene implementata in modo diverso. Ulteriori esempi di coreografie si possono trovare nel capitolo 4.

Capitolo 3

Tecnologie

Questo capitolo introduce le tecnologie usate nel progetto. In particolare, verrà analizzato il linguaggio di programmazione Erlang e le caratteristiche che lo rendono adatto allo studio delle coreografie. Descrivere i costrutti di Erlang non solo può essere utile per capire interamente la struttura del tool ChorEr e degli automi prodotti, ma può indicare quali saranno i lavori futuri del tool. Verrà spiegato brevemente il linguaggio di descrizione di grafi DOT e l'AST (albero sintattico astratto).

3.1 Erlang

Erlang è un linguaggio di programmazione nato nel 1986, creato della multinazionale svedese Ericsson che opera nel settore informatico e delle telecomunicazioni. La creazione di Erlang è stata influenzata dai linguaggi Lisp, Prolog, Smalltalk e nasce dal bisogno di costruire e implementare applicazioni di telefonia che abbiano un alto tempo di funzionamento e che quindi abbiano un'alta tolleranza ai guasti. In seguito è stato adattato per sistemi distribuiti. Nel 1998, viene rilasciato con licenza open source. Ad oggi, Erlang viene usato nell'ambito delle reti wireless da Ericsson, Nortel e T-Mobile, ma anche per software di messaggistica come Whatsapp o nei microservizi di famose grandi aziende come Nintendo e Samsung. Al momento

della stesura di questa tesi, l'ultima versione è Erlang/OTP 26, rilasciata il 16 maggio 2023.

3.1.1 Caratteristiche

Erlang è un linguaggio *general-purpose funzionale*, *fortemente* e *dinamicamente* tipato. Ha un sistema di *garbage-collection* a run-time. Il codice di un programma scritto in Erlang viene compilato in *BEAM bytecode* che successivamente viene interpretato dalla BEAM virtual machine, una macchina virtuale scritta in C. Una delle caratteristiche principali del linguaggio è il suo modello di concorrenza che permette di avere applicazioni real-time distribuite, scalabili e fault-tolerant.

Adotta la filosofia "*let it crash*", cioè un approccio alla gestione degli errori che cerca di preservare l'affidabilità e l'integrità di un sistema lasciando intenzionalmente non gestiti determinati casi d'errore. Si dice che viene "lasciato crashare" perché ci sarà un *processo supervisore* che farà in modo di riavviarlo con uno stato non corrotto attraverso determinate strategie. È possibile fare *hot-swapping* del codice, cioè cambiare il codice di un sistema durante la sua esecuzione.

Non esistono variabili globali; le variabili locali delle funzioni sono immutabili, quindi ogni volta che viene cambiato un parametro in una struttura dati, essa viene duplicata; non esistono dei costrutti per creare cicli come `for` e `while`, ma viene usata la ricorsione per implementarli. Usa il pattern matching (con eventuali guardie) per decidere quale funzione chiamare o quale ramo di codice eseguire nei costrutti `case`, `if` e `receive`. Usa l'estensione `.erl` per i file ed esiste una shell (chiamata Eshell) dove poter provare istruzioni o compilare e lanciare programmi.

Il codice 3.1 implementa l'algoritmo per calcolare il fattoriale di un numero in Erlang.

```
1 -module(test).
2 -export([fac/1]).
3
4 fac(0) -> 1;
5 fac(N) when N > 0, is_integer(N) -> N * fac(N - 1).
```

Listing 3.1: Algoritmo del Fattoriale in Erlang

Il costrutto `module` (a riga 1 del codice 3.1) definisce il nome del modulo (o libreria) e deve essere uguale al nome del file (in questo caso quindi sarà `test.erl`). Il costrutto `export` (a riga 2 del codice 3.1) definisce quali funzioni e con quale arità potranno essere chiamate dall'esterno (da altri moduli o dalla Eshell), bisogna anche inserire le funzioni su cui è possibile fare la `spawn`. Ogni istruzione deve finire con un punto (come a riga 5 del codice 3.1). Le funzioni devono avere la prima lettera minuscola e, quando vengono chiamate, viene eseguito il pattern matching per capire quale ramo della funzione eseguire (con la parola chiave `when` si definiscono delle guardie, cioè delle condizioni da soddisfare). La prima lettera delle variabili deve essere maiuscola. Nelle funzioni viene usata la virgola per eseguire delle istruzioni in sequenza e, dove viene usato il pattern matching, viene usato il punto e virgola per definire diversi rami di esecuzione (a riga 4 del codice 3.1).

Atom

Una caratteristica di Erlang sono gli atom, un tipo di dato definiti come dei letterali, delle costanti con nome. Sono come gli enumeratori, ma possono essere definiti ovunque nel codice. Vengono allocati staticamente e non vengono mai de-allocati dal *garbage collector*, durante l'esecuzione di un programma. Quindi è consigliato di non crearli dinamicamente.

3.1.2 Modello di concorrenza

Il modello di concorrenza è una delle funzionalità principali del linguaggio Erlang e, insieme ad altre caratteristiche, lo rende un sistema concorrente

distribuito, scalabile, leggero e molto sicuro.

Funzioni e processi

Su Erlang, le funzioni vengono lanciate come un *lightweight process* tramite la funzione `spawn` con argomenti `Modulo, Funzione, Lista di argomenti`. Questi processi potrebbero essere comparati ai *thread* ma, al contrario di essi, non condividono informazioni o memoria.

Grazie all'immutabilità delle variabili e all'isolamento dei processi non ci saranno side effect durante l'esecuzione della funzione: se uno dei processi ha un errore, solo quel processo terminerà. Ogni processo inoltre viene identificato con un *process id*, detto `pid`, il quale identifica il processo nei sistemi run-time.

Message passing

I processi comunicano tra di loro tramite il message passing con le seguenti funzioni *built-in*. Per inviare messaggi si usa il comando `Pid ! messaggio`, dove `Pid` rappresenta l'id di un processo (detto process id). Ogni processo ha una coda di messaggi in entrata, dalla quale l'operazione `receive` prende un messaggio uno alla volta. Il messaggio può essere qualsiasi tipo di dato. Per ricevere dei dati si usa:

```
1 receive
2     PatternMatching1 -> code;
3     PatternMatching2 when condizione -> code;
4     ...
5     PatternMatchingN -> code
6 end.
```

Listing 3.2: Esempio di `receive`

Se il messaggio non corrisponde a nessun ramo del pattern matching, il programma darà un messaggio di errore.

N.B.: l'invio di messaggi è un operazione *non bloccante* dove bisogna specificare a chi inviare il messaggio; invece, al contrario, la ricezione di messaggi è *bloccante* e non è possibile scegliere da quale processo ricevere il messaggio.

Con questi costrutti, è già possibile avere un modello ad attori completo e parlare di coreografie. Di seguito verranno esposte brevemente altre caratteristiche di Erlang che sfruttano il message passing.

Distribuzione e scalabilità

È possibile lanciare la EVM esposta sulla rete, attaccandosi quindi ad una rete distribuita di macchine virtuali di Erlang, i quali formano un cluster. Ogni sistema di run-time è un nodo e, tramite il message passing, i nodi possono comunicare ed eseguire codice in modo distribuito. Da notare come poco importa se i processi o i nodi siano in macchine diverse esposte su una rete o sono sulla stessa macchina, ma su core diversi. Grazie a questa caratteristica un sistema può scalare facilmente ogni programma.

Supervisori

In Erlang, un processo può essere un *supervisore* (o supervisor) di altri processi, cioè definisce dei comportamenti o delle strategie a seconda di come si comportano i processi a lui affidati. Un processo può supervisionare anche processi che a loro volta sono supervisor, creando quindi una gerarchia di processi. Un esempio di strategia da adottare quando un processo crasha può essere che si riavvia solo il processo crashato oppure tutti i processi dello stesso livello.

Esempio di un programma che usa i costrutti di comunicazione

Nel seguente programma, verranno usati i costrutti di comunicazione presentati prima, quindi le `spawn`, `receive` e `!`, con a seguito una descrizione del flusso d'esecuzione.

```
-module(simple).
-export([start/0, tac_loop/0, tic_loop/0]).

start() ->
  spawn_process(),
  ticl ! tac. % inizia il loop

spawn_process() ->
  TacPid = spawn(?MODULE, tac_loop, []),
  register(tacl, TacPid), % registro il pid nell'atom tacl
  TicPid = spawn(?MODULE, tic_loop, []),
  register(ticl, TicPid).

tac_loop() ->
  receive
    tic ->
      ticl ! tac,
      tac_loop();
    stop ->
      ticl ! stop % termine del processo
  end.

tic_loop() ->
  receive
    tac ->
      tacl ! tic,
      tic_loop();
    stop ->
      tacl ! stop % termine del processo
  end.
```

Listing 3.3: Esempio programma tic e tac

Per eseguire il codice 3.3 bisogna prima compilarlo, digitando nella Eshell `c(test).`, e poi per avviarlo si usa `test:start()`. Il programma quindi chiamerà `spawn_process` che creerà dei processi isolati che eseguono le funzioni `tac_loop` e `tic_loop`. La funzione `register(testatom, VarP)` ha lo scopo di "salvare" l'id del processo nella variabile `VarP` nell'atom `testatom`. Quindi, per esempio, quando nel codice verrà usato l'atom `tacl` a sinistra di un `!`, verrà sostituito con il pid del processo di `tac_loop`. Dopo aver spawnato i due processi, la funzione `start` invierà a `tic_loop` il messaggio `tac`. La funzione `tic_loop`, ricevuto `tac`, invierà a `tac_loop` il messaggio `tic` e chiamerà ricorsivamente se stesso, tornando in attesa di messaggi. Specularmente, la funzione `tac_loop`, ricevuto `tic`, invierà a `tic_loop` il messaggio `tac`, poi fa una chiamata ricorsiva. Di conseguenza ci sarà un loop infinito di messaggi scambiati tra i due processi. I processi si fermeranno una volta che uno dei due riceverà uno `stop`, ma in questo caso non c'è nessun modo di fermarli.

3.2 Digraph e DOT

Il modulo `digraph` è la libreria di Erlang per creare e manipolare grafi diretti. DOT è un linguaggio descrittivo per grafi diretti o non, utilizzato specialmente nella suite di programmi e librerie open-source di Graphviz [20]. Un Automata Coreografico (che è un FSA) può essere rappresentato tramite un grafo diretto. Di seguito, ecco un esempio di grafo in DOT:

```
digraph graph1 {
    # grafo orientato da destra a sinistra
  rankdir="LR";

    # definizione dei nodi
  n_0 [label="start", shape="plaintext"];
  n_1 [id="1", shape=circle, label="1"];
  n_2 [id="2", shape=circle, label="2"];
  n_3 [id="4", shape=doublecircle, label="3"];

    # definizione degli archi
  n_0 -> n_1;
  n_1 -> n_2 [label="A -> B : tic"];
  n_2 -> n_3 [label="B -> A : toc"];
}
```

Listing 3.4: Descrizione in DOT del grafo 2.6

Nei primi stadi del progetto abbiamo ricercato delle librerie che potessero convertire dei grafi di Erlang in linguaggio DOT e abbiamo trovato una libreria [14] su Github creata da un utente. Abbiamo preso solo il codice nel file `digraph_export_dot.erl`, implementando funzionalità specifiche per il progetto, come:

- la possibilità di orientare i grafi da sinistra a destra,
- creare la distinzione tra stato iniziale e stato finale,
- poter formattare l'output della funzione `convert` usando sia atom che stringhe (grazie al quale si è limitato l'uso degli atom dinamici).

3.3 Analisi statica e AST

In informatica, l'analisi statica dei programmi consiste nell'analizzare programmi senza eseguirli. Ci sono diversi tipi di tecniche di analisi. Nel caso di ChorEr, il progetto prende in input un programma in Erlang e utilizzando la libreria `epp_dodger` e la funzione `quick_parse_file`, passando come stringa il percorso relativo del file, genera l'*albero sintattico astratto* (in inglese Abstract Syntax Tree, chiamato d'ora in poi AST). L'AST di un programma viene generalmente creato nelle prime fasi della compilazione di un programma da un *parser* per verificarne la correttezza sintattica e per facilitare le fasi successive. In questo progetto, viene usato per controllare se sono presenti costrutti di comunicazione e andare a generare un Automa Coreografico. L'AST di Erlang è essenzialmente una lista di strutture dati: ogni costrutto ha il suo `record` formato tipicamente da {`numero linea di codice, tipo, valori`}, ma cambia molto dal tipo di costrutto.

Di seguito, ecco un esempio di codice in Erlang e il suo corrispettivo AST.

```
1 -module(example).
2 -export([greet/1]).
3
4 greet(Who) -> io:fwrite("Hello ~p!~n", [Who]).
```

Listing 3.5: Programma Hello World in Erlang

```
1> epp_dodger:quick_parse_file("example.erl").
{ok,[
  {attribute,1,module,example},
  {attribute,2,export,[{greet,1}]},
  {function,4,greet,1,[
    {clause,4,[{var,4,'Who'}],[],[
      {call,5,{remote,5,{atom,5,io},{atom,5,fwrite}},
      [{string,5,"Hello ~p!~n"},{cons,5,{var,5,'Who'},{nil,5
   }}]}]}]}]}
2>
```

Listing 3.6: AST del codice 3.5

Capitolo 4

ChorEr

ChorEr è il tool presentato insieme a questa tesi, è scritto in Erlang ed è rilasciato con licenza GPLv3 su Github [8]. Permette la creazione di file DOT della descrizione degli automi di viste locali e globali di un programma Erlang in input. In questo capitolo verrà descritto lo stato attuale del progetto; come usare il tool e come visualizzare gli output; quale struttura è stata data al tool e quali decisioni implementative sono state fatte per adattare le coreografie ai costrutti di Erlang. Alla fine del capitolo verranno presentati alcuni esempi d'esecuzione.

4.1 Modalità d'uso

Il tool viene invocato da linea di comando attraverso la shell di Erlang, non è presente una GUI perché non necessaria ai fini del tool. Può essere usato compilando ogni modulo dalla Eshell oppure, più comodamente, si può usare `rebar3`, uno strumento standard di compilazione che implementa molte funzionalità come la gestione di pacchetti creati dalla community, la compilazione e il testing del progetto in modo automatico. Clonando il progetto dalla repository pubblica di Github e usando il comando `rebar3 shell` nella cartella principale, `rebar3` si occuperà, in questo ordine, di verificare ed eventualmente scaricare le dipendenze, compilare il progetto e verificare i

test se presenti; in seguito, si aprirà una Eshell, con la quale poter avviare il tool.

È possibile utilizzare il tool con diversi comandi:

- `chorer_app:generate(Input, Entry)`
- `chorer_app:generate(Input, Entry, Output)`
- `chorer_app:generate(Input, Entry, Output, Options)`

I parametri obbligatori sono:

- **Input**: la stringa del percorso relativo del programma Erlang in input, dal quale il tool creerà le viste locali e globali;
- **Entry**: l'atom che rappresenta la funzione dell'inizio dell'esecuzione del programma in input, questo parametro è essenziale perché in Erlang non esiste una funzione convenzionale come punto di partenza (come, per esempio, il `main` in C). Questo parametro verrà passato alla funzione che crea la vista globale per poter iniziare la simulazione.

I parametri opzionali sono:

- **Output**: la stringa del percorso relativo della cartella di output dove si vogliono salvare i file delle viste locali e della vista globale. Le viste locali avranno `[nome funzione con arità]_local_view.dot` come nome del file. Invece, la vista globale avrà come nome del file `[Entry]_global_view.dot`;
- **Options**: al momento, è una tupla di 2 booleani attraverso la quale è possibile personalizzare le viste locali. Il primo booleano stabilisce se individuare o meno gli stati finali (di *default* è impostato a `true`); invece, il secondo inserisce più informazioni nella vista locale con transizioni relative agli altri costrutti del linguaggio, in aggiunta a quelli di comunicazione (di *default* è impostato a `false`). Questo parametro potrà espandersi nel futuro con altri booleani.

```
1> chorer_app:generate_chor_automata("./example/tictac/
   tictacstop.erl", start, "example/tictac").
finished
2>
```

Listing 4.1: Esempio d'uso del tool

4.2 Struttura del tool

La seguente sezione si dividerà in due sottosezioni: una per la struttura dei file e una per il flusso dell'esecuzione del tool, dove verranno anche descritte le funzioni principali del progetto.

4.2.1 Struttura dei file

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`.

```
chorer
├── examples
│   └── ...
├── src
│   ├── choreography
│   │   ├── db_manager.erl
│   │   ├── global_view.erl
│   │   ├── local_view.erl
│   │   └── metadata.erl
│   ├── common
│   │   ├── common_data.hrl
│   │   ├── common_fun.erl
│   │   ├── digraph_to_dot.erl
│   │   └── fsa.erl
│   ├── chorer_app.erl
│   └── chorer.app.src
├── rebar.config
└── rebar.lock
```

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.

4.2.2 Flusso dell'esecuzione

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} ->
```

```
20              manage_if();
21          %%% Evaluate receive with pattern matching
22          {'receive', _, PMList} ->
23              add_receive_to_local_view();
24          %%% Evaluate send
25          {op, _, '!', ProcSent, DataSentAst} ->
26              add_send_to_local_view();
27          %%% Evaluate data types
28          {atom, _, Value} -> return_var();
29          {integer, _, Value} -> return_var();
30          {string, _, Value} -> return_var();
31          ...
32          _ -> nomatch
33      end.
```

Listing 4.2: 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 4.3 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.

```
proc_loop(LocalView, CurrentState, MarkedEdges) ->
    receive
        {use_transition, Edge} ->
            % per evitare loop infiniti
            NewState = verify_not_marked();
            proc_loop(LocalView, NewState, MarkedEdges);
        {P, get_info} ->
            P ! {someinfo},
            proc_loop(LocalView, CurrentState, MarkedEdges);
        stop -> terminated
    end.
```

Listing 4.3: 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 4.4, 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.

```
progress_branches(BranchList) ->
    NewBranchList = lists:foreach(
        fun(Item) -> progress_single_branch(Item) end,
        BranchList
    ),
    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 4.4: 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 4.4). 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 4.4). 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 4.4 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.

4.3 Conversione da Erlang a Coreografia

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

4.3.1 Costrutti di comunicazione

Viste locali

Il codice di un'operazione `receive` (come nel codice 3.2) corrisponde al grafo 4.1; 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 4.2. La chiamata alla funzione `spawn` corrisponde al grafo 4.3.

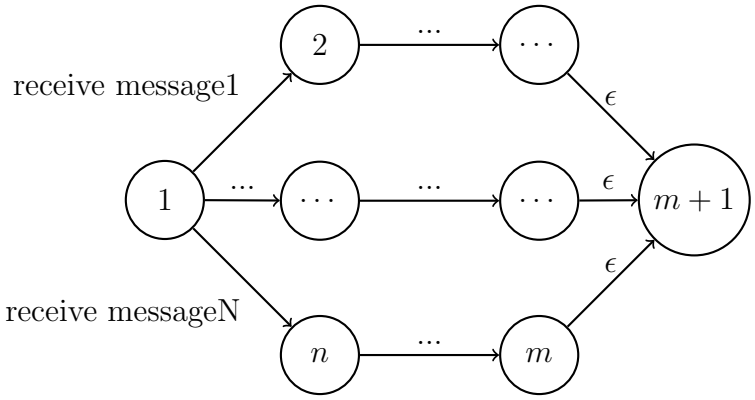

Figura 4.1: Grafo locale per il costrutto `receive`

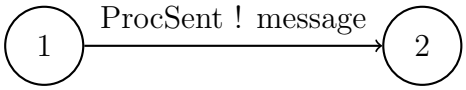

Figura 4.2: Grafo locale per il costrutto `!`

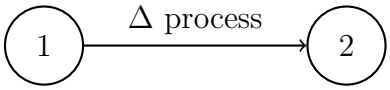

Figura 4.3: Grafo locale per il costrutto `spawn`

Viste globali

Per le viste globali, nelle `spawn` si specifica l'attore che esegue l'operazione a sinistra del simbolo come mostrato nella figura 4.4. Una spawn verrà direttamente inserita nel grafo. Ogni processo verrà anche numerato, nel caso vengano creati molteplici attori per la stessa funzione.

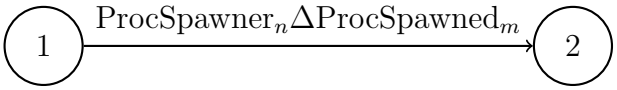

Figura 4.4: Grafo globale per il costrutto `spawn`

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 4.5. 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.

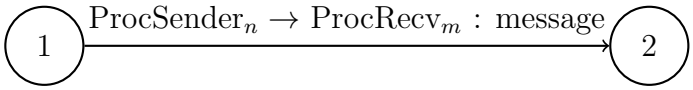

Figura 4.5: Grafo globale per `receive` e `!`

4.3.2 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 4.6.

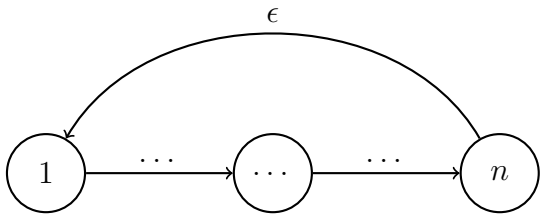

Figura 4.6: Grafo della chiamata ricorsiva di una funzione

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. Un esempio si può trovare nella vista locale 4.14.

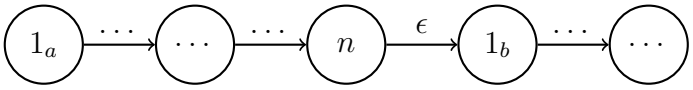

Figura 4.7: Grafo di una chiamata di funzione

Nel grafo 4.7, 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.

4.3.3 Altri costrutti

Per i costrutti `if` e `case`, si usa lo stesso grafo della `receive` (grafo 4.1), ma nelle etichette verranno messe delle transizioni ϵ. Se viene messa a `true` l'opzione dell'espansione delle viste locali, le etichette saranno rispettivamente `if cond` e `match pattern` per i due costrutti.

4.3.4 Costrutti supportati

La seguente tabella 4.1 riassume le parole chiavi supportate dal tool.

Parola chiave	**Supporto**
atom	sì
integer	sì
float	sì
boolean	no
tuple	sì
list	sì
record	no
map	no
binary	no
if	sì
case	sì
receive	sì

Parola chiave	**Supporto**
!	sì
match	sì
function	in parte
when	no
self	sì
spawn	sì
rand:uniform	sì
try catch	no
after	no
math operation	no
fun	no
attribute	in parte

Tabella 4.1: Tabella dei costrutti supportati

4.4 Esempi

In questa sezione, verranno spiegati alcuni esempi d'esecuzione del tool.

4.4.1 Esempio senza diramazione

In questo esempio, c'è una funzione main che spawna due processi `dummy1` e `dummy2`. Questi due attori in seguito, si scambieranno un semplice messaggio. Uno dei due attori, però, eseguirà prima la *send* e poi la *receive*; invece, l'altro attore eseguirà le operazioni al contrario. Ci si aspetta quindi che il grafo risulti in una linea unica di esecuzione, dove prima appare lo scambio di `ciao` e poi quello di `bello`.

```
1  -module(simple).
2  -export([main/0, dummy1/0, dummy2/0]).
3
4  dummy1() ->
5      receive
6          ciao -> done
7      end,
8      d2 ! bello.
9
10 dummy2() ->
11     d1 ! ciao,
12     receive
13         bello -> done
14     end.
15
16 main() ->
17     A = spawn(?MODULE, dummy1, []),
18     register(d1, A),
19     B = spawn(?MODULE, dummy2, []),
20     register(d2, B).
```

Listing 4.5: Due processi con scambio di messaggi in modo sincrono

Il codice 4.5 corrisponde alla descrizione data. Il *main* registra rispettivamente i due processi, in modo che i due possano comunicare. In seguito, `dummy2` invia l'atom `ciao`, che verrà ricevuto da `dummy1`. Quindi, verrà sbloccato l'invio dell'atom `bello` verso il processo `dummy1`. Le figure 4.8, 4.9 e 4.10 corrispondono alle viste locali delle funzioni.

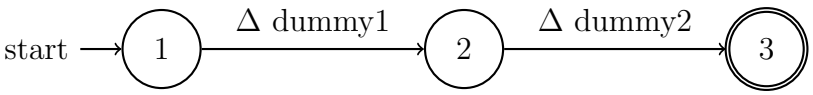

Figura 4.8: Vista locale di `main`

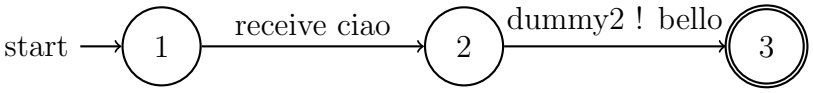

Figura 4.9: Vista locale di `dummy1`

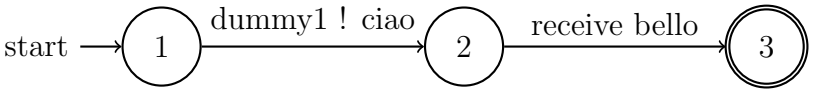

Figura 4.10: Vista locale di `dummy2`

Dopo aver associato le viste locali agli attori, l'algoritmo comporrà la vista globale 4.11.

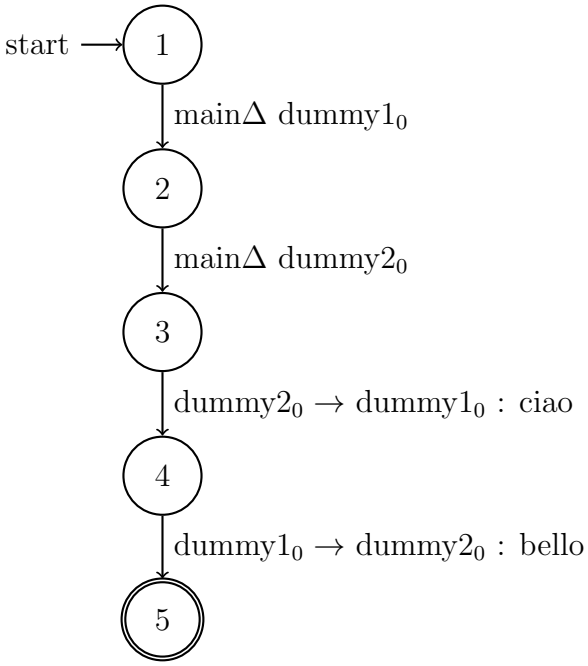

Figura 4.11: Vista globale dell'esempio 4.4.1

Come precedentemente supposto all'inizio dell'esempio, l'automa 4.11 esprime una sola esecuzione possibile del programma.

4.4.2 Esempio con diramazione

In questo esempio, prendiamo l'esempio 4.4.1, modificando l'ordine delle operazioni nella funzione `dummy1`. In questo modo, i due attori si scambieranno un semplice messaggio e non si può sapere quale arriverà prima. Il seguente esempio si trova nella cartella `async` degli esempi. Nel codice 4.6 viene riportata solo la funzione modificata.

```
1 dummy1() ->
2     d2 ! bello,
3     receive
4         ciao -> done
5     end.
```

Listing 4.6: Funzione modificata del codice 4.5

La figura 4.12 rappresenta l'automa modificato della vista locale della funzione `dummy1`.

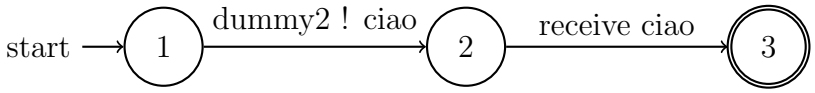

Figura 4.12: Vista locale modificata di `dummy1`

Mentre le viste locali cambiano di poco, la vista globale subisce un cambiamento maggiore. L'automa in figura 4.13 rappresenta la vista globale finale.

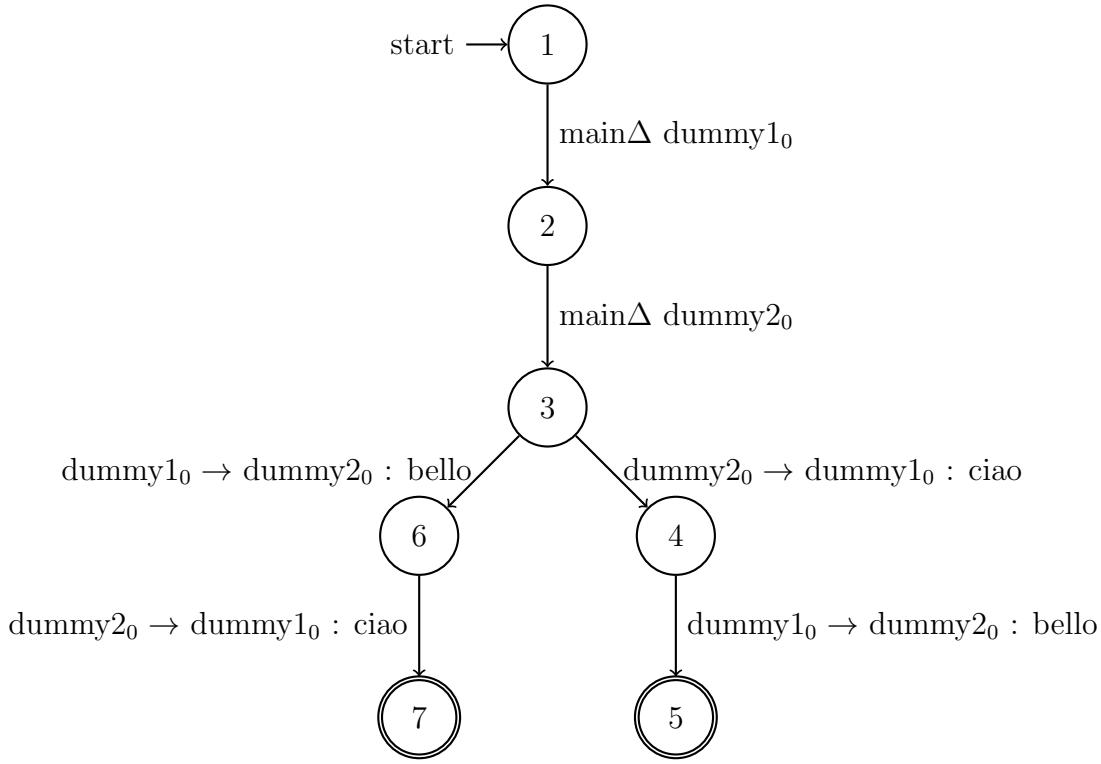

Figura 4.13: Vista globale dell'esempio 4.4.2

Come si può osservare nella figura 4.13, la vista globale dell'esempio segue prima lo sviluppo del `main`. In seguito, i due attori `dummy1`$_0$ e `dummy2`$_0$

eseguono una send, che creerà una diramazione allo stato 3: nel ramo destro viene valutato per primo l'invio del messaggio da parte di $\texttt{dummy1}_0$ e in seguito quello di $\texttt{dummy2}_0$; invece, nel ramo di sinistra, viene valutata prima l'invio di $\texttt{dummy2}_0$ e in seguito quello di $\texttt{dummy1}_0$.

4.4.3 Esempio codice 3.3 tictac

Come primo esempio prendiamo il codice 3.3: gli attori sono `start`, `tic_loop` e `tac_loop`. La funzione di ingresso è `start`. È possibile trovare l'esempio sulla repository nella cartella degli esempi sotto la cartella `tictac`. Dunque, verranno prodotti tre file di viste locali e uno di vista globale.

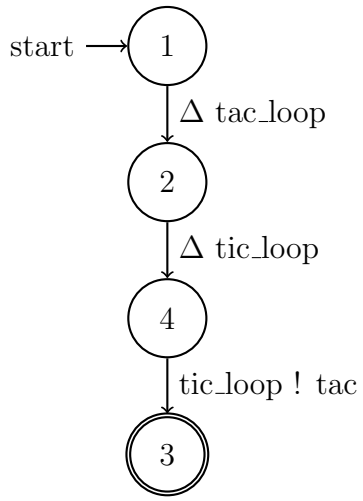

Figura 4.14: Vista locale di `start`

Dalla vista locale 4.14, possiamo vedere che la funzione `start` crea due attori: $\texttt{tac\_loop}_0$ e `tic_loop`, poi manda `tac` a `tic_loop`.

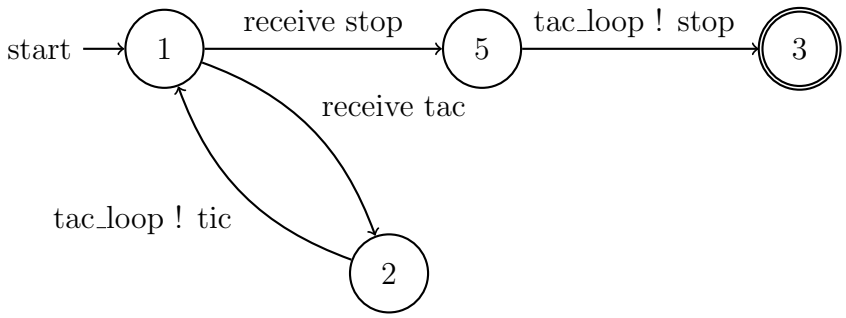

Figura 4.15: Vista locale di `tic_loop`

Dalla vista locale 4.15, si nota come la funzione `tic_loop` se riceve `tac` manda `tic` a $\texttt{tac\_loop}_0$ e, invece, se riceve `stop` manda `stop` a $\texttt{tac\_loop}_0$.

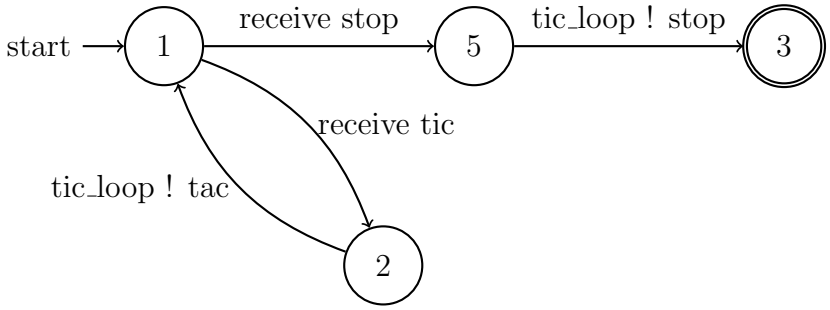

Figura 4.16: Vista locale di `tac_loop`

La vista locale 4.16 di `tac_loop` è speculare alla vista locale 4.15.

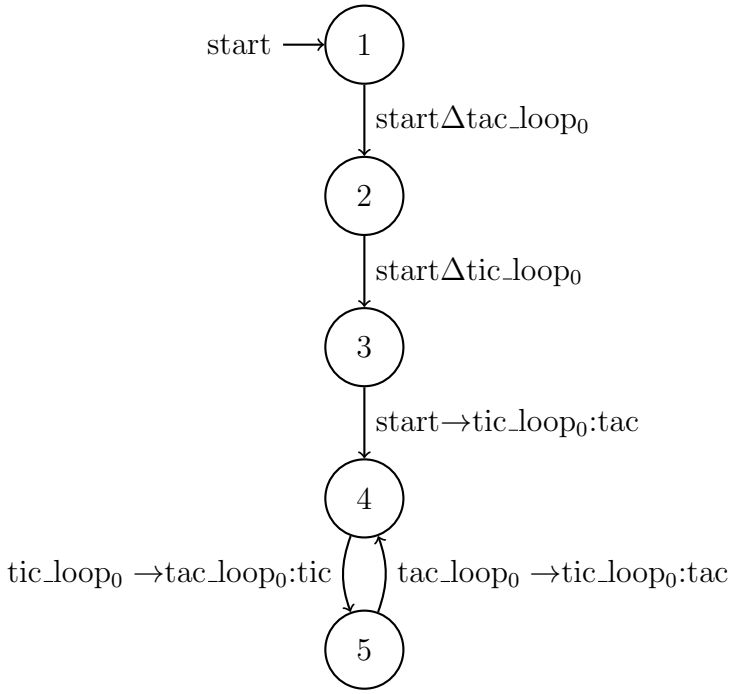

Figura 4.17: Vista globale partendo da `start`

Come viene spiegato nella descrizione del codice 3.3, la vista globale 4.17 descrive come il programma entra in un loop infinito di scambi di messaggi e quindi non valuta il ramo di stop delle figure 4.15 e 4.16.

Dopo aver creato le viste locali, per creare la vista globale verrà creato un processo che simula ogni attore. Ogni processo verrà fatto avanzare fino alla prima *send*. All'inizio ci sarà solo il processo di `start`, che creerà i due processi `tic_loop`$_0$ e `tac_loop`$_0$ e invierà a `tic_loop`$_0$ il messaggio `tac`. All'iterazione seguente, facendo avanzare `tic_loop`$_0$ e `tac_loop`$_0$, entrambi i processi si bloccano sulla `receive`. Solo il processo `tic_loop`$_0$ andrà avanti perché all'iterazione prima è stato inviato `tac` da `start`. Quindi, anche sul grafo, `tic_loop`$_0$ invierà `tic` a `tac_loop`$_0$, entrando poi nel ciclo infinito di comunicazione.

4.4.4 Esempio tictac con stop

Provando a inserire un attore che, *in modo casuale*, manda il messaggio `stop` a `tic_loop`, la vista globale cambia in modo sostanziale.

Di seguito, ecco presentate solo le parti di codice cambiate:

```
1  spawn_process() ->
2      ...
3      spawn(?MODULE, random, []).
4
5  random() ->
6      RandInt = rand:uniform(10),
7      case RandInt > 8 of
8          true -> tic_loop ! stop;
9          false -> random()
10     end.
```

Listing 4.7: Cambi apportati al codice 3.3

Nel codice 4.7, la funzione `spawn_processes` crea un nuovo attore che genera un numero casuale da 1 a 10. Solo se il numero è maggiore di 8, invia a `tic_loop` stop. La vista locale 4.18 è la vista locale di `random`. Viene omessa la vista locale 4.14 modificata di `start`, perché ovvia.

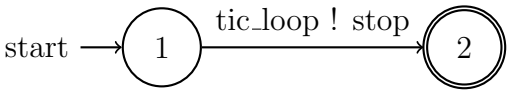

Figura 4.18: Vista locale di `random`

La vista locale 4.18 non tiene conto della chiamata ricorsiva a riga 9, della creazione casuale dell'intero a riga 6 e della verifica dell'intero a riga 7, ma terrà conto solo della *send* perché è un costrutto di comunicazione.

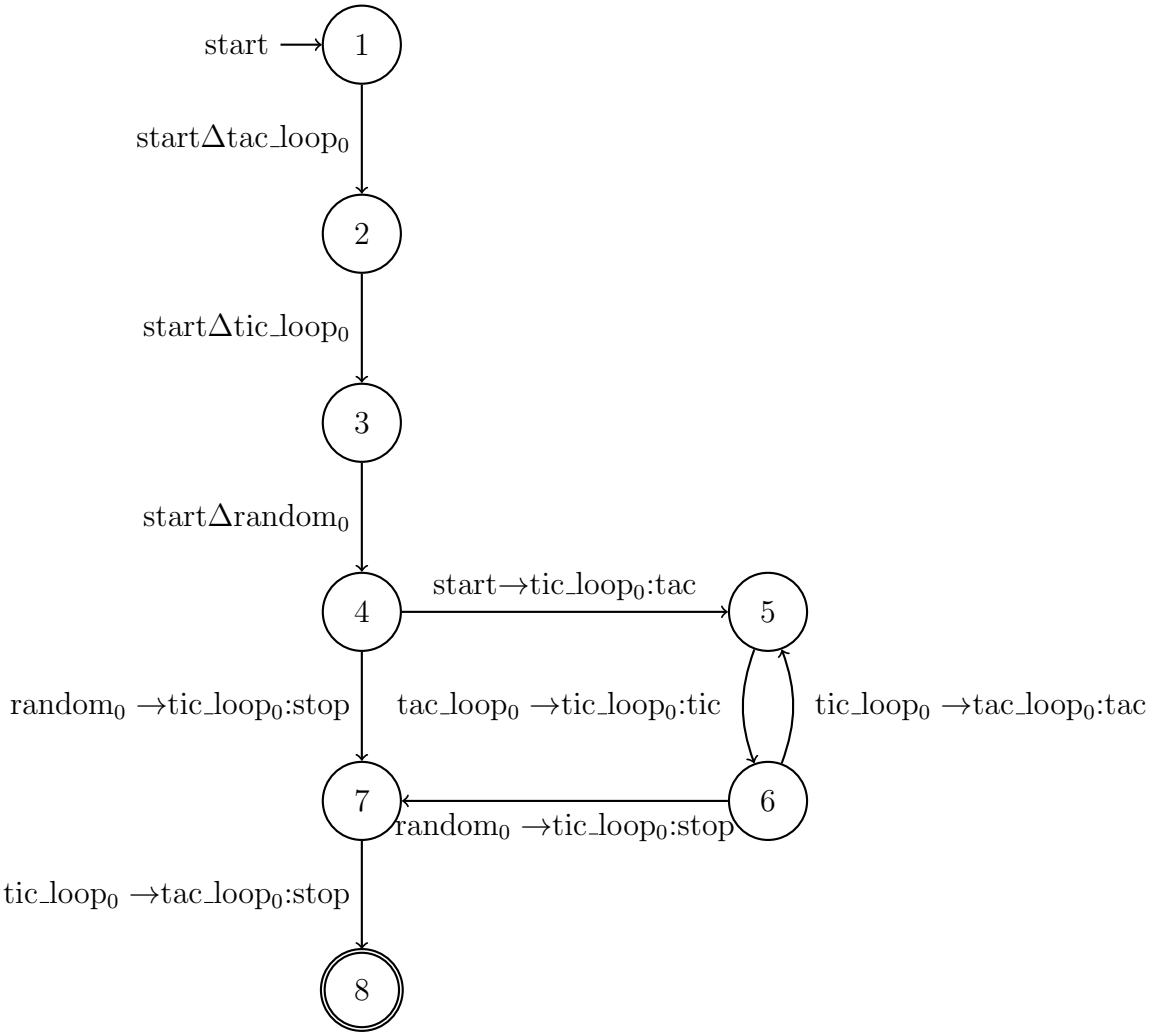

Figura 4.19: Vista globale con il codice 4.7

La vista globale prodotta sarà simile a quella in figura 4.19 (per questioni grafiche è stata modificata). L'automa inizia come quello in figura 4.17, con la differenza che prima di inviare il messaggio `tac` all'attore `tic_loop`$_0$, viene creato l'attore per `random`$_0$. Da questo momento, l'algoritmo vede che ci sono due messaggi inviati: `tac` e `stop`. Allora verrà creata una diramazione allo stato 4. Anche allo stato 6, verrà trovata una diramazione per la presenza dei due messaggi `tac` e `stop`. Infine, entrambi i rami raggiungeranno lo stato

7 dove l'unica operazione rimasta è l'invio di `stop` a $\texttt{tac\_loop}_0$.

Conclusioni

In questo capitolo verranno riassunti i risultati ottenuti in questa tesi, ma anche i limiti del programma e i possibili futuri sviluppi per renderlo più esteso, completo e accurato.

Risultati e progetti simili

Allo stato attuale, il tool è un *proof of concept* ancora grezzo, ma che riesce a produrre ed elaborare con efficacia gli automi coreografici per piccoli progetti. Gli automi prodotti cercano di interpretare l'esecuzione del programma Erlang nel modo più accurato possibile: l'approssimazione che viene effettuata durante conversione tra codice e grafo può essere *più* o *meno* espressiva del programma. Se è più espressivo, allora l'automa rappresenta un *upper bound* del programma. Se è meno espressivo, l'automa rappresenta invece un *lower bound*. Per esempio, se per valutare un `if-then-else` si esplorano tutte i rami possibili dell'esecuzione si parla di un'approssimazione più espressiva; se si esplorano solo alcuni rami sarà meno espressiva. Se si è in grado di valutare con sicurezza quale ramo viene eseguito allora l'approssimazione è ugualmente espressiva.

Nel caso di ChorEr, dipende dal costrutto preso in considerazione: le istruzioni più semplici hanno una corrispondenza uno a uno, ma per alcuni costrutti è impossibile produrre un approssimazione accurata (tipo programmi dove cambia l'esecuzione a seconda di un input dall'utente). Invece, per la creazione delle viste globali viene preso in considerazione qualsiasi tipo di

esecuzione, creando un approssimazione più espressiva del programma. L'obbiettivo futuro è quindi rifinire il tool per rendere più precisa la valutazione delle viste locali (alzare il lower bound) e, di conseguenza, avere viste globali meno generiche (abbassare l'upper bound).

Un problema di difficile risoluzione è la gestione di funzioni ricorsive (molto usate nei linguaggi funzionali al posto dei cicli), che usano come "guardia" per uscire dal loop il pattern matching delle funzioni: a meno che non si possa valutare nel programma, è impossibile predire quante chiamate di funzione vengono eseguite e di conseguenza il rispettivo Automa Coreografico può essere poco accurato.

Il tool prende ispirazione dal progetto di tesi triennale *Choreia* [11], che, similmente a ChorEr, crea Automi Coreografici per programmi scritti in Go. La maggior parte degli analizzatori statici di codice si concentra sull'individuazione di vulnerabilità e bug conosciuti o sulla coerenza con le *best practice* di uno stile di programmazione. Come scritto in [6], uno dei pochi tool simili a ChorEr e Choreia è Gomela [18], un tool per la verifica delle proprietà di comunicazione di programmi in Go dei messaggi scambiati in modo concorrente studiato in [5].

Lavori futuri

Il tool è ancora in fase di sviluppo: è in programma un refactor del codice insieme alla scrittura di una documentazione completa per renderlo più strutturato e chiaro ad eventuali contribuenti al progetto, oltre alla creazione di unit test. Si possono migliorare ed estendere alcune funzionalità già presenti come la valutazione del pattern matching per le funzioni, la valutazione delle guardie oppure aggiungere sempre più dettagli nella funzione 4.2 con più tipi di dato o funzioni *built-in* e funzioni definite in moduli esterni. Altre future funzionalità da aggiungere in programma sono il poter stampare una traccia del programma (utile non solo per debuggare ma anche per far capire

all'utente finale i passaggi del programma) e una maggiore personalizzazione dei grafi da parte dell'utente (formattazione delle etichette e orientamento del grafo).

Con costanza, verranno testati diversi tipi di programmi per migliorare l'approssimazione verso una rappresentazione più accurata, in quanto, allo stato attuale, per alcuni comportamenti particolari di un programma Erlang non verrà prodotto un automa completo o accurato. In futuro, si potrebbe formalizzare con un modello matematico più accurato la corrispondenza tra i costrutti e gli automi; e si potrebbe usare un tool di verificazione per dimostrare la correttezza del programma.

Bibliografia

[1] 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.

[2] Andi Bejleri, Elton Domnori, Malte Viering, Patrick Eugster, and Mira Mezini. Comprehensive multiparty session types. *The Art, Science, and Engineering of Programming*, 3(3), feb 2019.

[3] 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].

[4] 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.

[5] Nicolas Dilley and Julien Lange. Automated verification of go programs via bounded model checking. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1016–1027. IEEE, 2021.

[6] Enea Guidi. Choreia: A Static Analyzer to Generate Choreography Automata from Go Source Code. Bachelor's Thesis.

[7] Ericsson. Erlang/OTP. `https://www.erlang.org/`, 1986. [Online; accessed 26-May-2023].

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

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

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

[11] Enea Guidi. Choreia. `https://github.com/its-hmny/Choreia`. [Online; accessed 02-July-2023].

[12] 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.

[13] 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].

[14] jkrukoffs. Digraph export Erlang Library. `https://github.com/jkrukoff/digraph_export`. [Online; accessed 24-June-2023].

[15] Lightbend. Akka. `https://akka.io/`. [Online; accessed 30-June-2023].

[16] Simone Martini Maurizio Gabbrielli. *Linguaggi di programmazione. Principi e paradigmi. Seconda edizione.* McGraw-Hill, 2011.

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

[18] nicolasdilley. Gomela. `https://github.com/nicolasdilley/Gomela`. [Online; accessed 02-July-2023].

[19] 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.

[20] Graphviz Development Team. Dot. `https://graphviz.org/doc/info/lang.html`. [Online; accessed 26-May-2023].

[21] The Actix Team. Actix. `https://actix.rs/`. [Online; accessed 06-June-2023].

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

[23] The Rust Foundation. Rust. `https://www.rust-lang.org/`. [Online; accessed 06-June-2023].

Ringraziamenti

Ringrazio i miei genitori, Paola e Riccardo, che mi hanno cresciuto con tutto l'affetto e il sostegno possibile, nel corso di tutta la mia vita

Ringrazio Eleonora, che dà colore alla vita di tutti i giorni con il suo affetto e per farmi compagnia accettando ogni parte di me.

Ringrazio Samuele, per le piccole azioni di tutti i giorni che valgono molto più delle parole.

Ringrazio tutti *I guys del Ranzani* per tutti i giorni passati a studiare insieme, per le uscite spensierate e le risate; persone senza le quali tutto sarebbe stato molto più difficile. In particolare, vorrei ricordare Apo, Dan, Fabio e Frau con i quali è stato bellissimo parlare dei massimi sistemi, del futuro ma anche delle cavolate più assurde. Ma anche Luca (capo indiscusso dell'*admstanzetta*) e Gian dai quali ho imparato tantissimo; non riuscirei a ridare un centesimo dell'aiuto che mi hanno dato. Erik, Simo, Yonas (matti folli pazzi imprevedibili) che con carisma e simpatia mi hanno fatto ridere come non mai. Fede e Paolo, compagni di TechWeb fantastici e indimenticabili. Volpe, per essere sempre di ispirazione a fare del mio meglio. Napo e Manu, fratelli a cui voglio un bene assurdo. Andreea e Gaia, con le quali condivido bei ricordi di caffettini e chiacchiere. Ceci e Andreaiaia, che hanno lasciato un segno indelebile in questa esperienza, nonostante il poco tempo passato insieme.

Ringrazio il gruppo scout *Bologna 13*, grazie al quale ho incontrato persone fantastiche che mi hanno aiutato nel cammino della mia crescita personale

come cittadino del mondo. Grazie in particolare a Alf, Amy, Annac, Annam, Ante, Dave, Giulietta, Giulio, Leti, Lucy, Marto, Mary e Tommy per i tanti momenti condivisi.

Infine, ringrazio tantissimo il Professor Ivan Lanese, che mi ha seguito e aiutato moltissimo nella scrittura della tesi, dandomi una fiducia immensa nonostante i tempi stretti.