Alma Mater Studiorum · Università di Bologna

Scuola di Scienze
Dipartimento di Informatica - Scienza e Ingegneria - DISI

Corso di Laurea Triennale in Informatica

Sviluppo di un'applicazione per l'elaborazione di Choreography Automata

Relatore:
Prof. Ivan Lanese

Presentata da:
Simone Orlando

II Appello - II Sessione

Anno Accademico 2018/2019

Dedica corta

Abstract

Le *Choreographies* sono un paradigma emergente per la descrizione top-down di sistemi distribuiti. L'idea chiave è dare al programmatore la possibilità di scrivere una "coreografia" dei partecipanti e di come questi interagiscono tra loro. A partire dalle Choreographies è possibile ottenere il codice di ogni singolo partecipante tramite l'operazione di *proiezione*. Il vantaggio delle Choreographies è che, sotto alcune condizioni ben definite, garantiscono per costruzione l'assenza dei problemi tipici della programmazione concorrente come i *Deadlocks* e le *Race Conditions*. Sono stati proposti diversi modelli di *Choreographies*, tra cui di recente quello dei *Choreography Automata*, basato su automi a stati finiti. In questa tesi viene presentato *Corinne*: un tool per la lettura, composizione e proiezione dei Choreography Automata.

Indice

Capitolo 1

Introduzione

Il progressivo aumento dei sistemi distributi e delle cosiddette architetture orientate ai servizi ha cambiato il modo in cui oggigiorno il software viene prodotto. Le applicazioni sono sviluppate a partire da altre già esistenti e generalmente queste comunicano tra loro tramite scambio di messaggi. Questo trend è dovuto a vari fattori, tra cui esigenze di mercato, come un'azienda che vuole sviluppare un software e decide di riutilizzare componenti già esistenti. Oppure esigenze di rete, dove piuttosto che adottare un'architettura centralizzata che richiede grandi capacità di calcolo, si preferisce distribuire il carico su più calcolatori rendendo il servizio scalabile. Ma sviluppare un sistema distribuito esente da errori è noto per essere un problema difficile [18]. Questo deriva principalmente dalla natura non deterministica del problema. Infatti, a differenza di un sistema sequenziale nel quale i processi vengono eseguiti uno per volta, in un sistema distribuito la varietà di interazioni che si possono verificare tra processi in esecuzione è notevole. Questo rende difficile qualsiasi analisi statica degli errori, perché potrebbe generare uno spazio esponenziale delle possibili esecuzioni. Inoltre, le tecniche usate per rilevare errori durante l'esecuzione, come testing e debugging, sono meno affidabili sui sistemi concorrenti, proprio perché gli errori tendono ad apparire in modo non deterministico. Per programmare tali applicazioni occorre quindi capire come il singolo programma, combinato con altri, possa influenzare il comportamento globale dell'applicazione. Per far fronte a questa difficoltà, nel corso degli anni sono stati proposti diversi paradigmi e modelli per facilitare la scrittura di programmi concorrenti, tra cui quello delle *Choreographies*. L'idea chiave alla base delle Choreographies è semplice: definire la comunicazione che due o più endpoints devono avere durante l'esecuzione, piuttosto che una collezione di programmi, nella quale viene descritto singolarmente il comportamento di ogni endpoint.
Infatti le Choreographies e più nello specifico i *Choreography Languages*, mirano a risolvere questa difficoltà fornendo al programmatore la possibilità di scrivere una "coreografia" dei partecipanti e di come questi interagiscono tra loro.

Tipicamente, nei Choreographies Languages, i processi interagiscono tramite scambio di messaggi. Avremo quindi una primitiva per esprimere ciò, per esempio:

$$Alice \rightarrow Bob : msg$$

descrive l'interazione tra un mittente Alice che manda un messaggio *msg* a Bob. Una caratteristica saliente delle Choreographies è quella di fornire 2 tipi di *view*: una *global* view e una *local* view. La prima è una descrizione del sistema da un punto di vista *olistico*. La seconda, invece, specifica il comportamento di un singolo componente "isolato". Quest'ultima viene ottenuta per *proiezione* della global view rispetto a uno dei componenti della coreografia.

$$Choreography \xrightarrow{\text{projection}} Endpoint\ Code$$

Questo comporta che, sotto alcune condizioni (*well-formedness* [7]), una coreografia è corretta per costruzione, e il codice che da essa viene generato non è soggetto a errori tipici della programmazione concorrente come i *Deadlocks* e le *Race Conditions*. A questo punto fare la composizione delle local views è semplice: basta eseguire i codici dei singoli componenti in parallelo. La situazione invece è più complicata per le global views, la cui composizione potrebbe non garantire in generale la proprietà di well-formedness e rimane tuttora un problema aperto. In questa tesi viene presentato *Corinne*: un tool per la lettura, composizione e proiezione dei *Choreography Automata* (CA) [7]. Quest'ultimo è un modello per descrivere le global views delle Choreographies, basato su automi a stati finiti. Questo ci permette di poter riutilizzare la vasta teoria già esistente in letteratura sugli automi. Verranno introdotti anche le *Communicating-FSA* [8], un altro modello basato su automi, che adotteremo invece per la descrizione delle local views delle Choreographies, e quindi anche per le proiezioni dei CA. Inoltre verrà presentato un algoritmo per la composizione dei CA, basato sul prodotto cartesiano tra automi, che preserva sotto alcune condizioni la proprietà di well-formedness.

Capitolo 2

Teoria alla base del tool

2.1 Automa a stati finiti

Prima di introdurre il modello dei Choreography Automata, occorre fare un breve richiamo della nozione di automa a stati finiti (FSA). Un automa o macchina a stati può essere visto come la descrizione di un sistema dinamico, ovvero un sistema che evolve nel tempo secondo alcune regole. In informatica teorica sono spesso utilizzati per descrivere linguaggi formali, e per questo sono chiamati anche riconoscitori di un linguaggio. Inoltre, va fatto notare che esiste un parallelo tra la struttura dell'automa a stati finiti e la struttura dei moderni elaboratori elettronici. Infatti un automa può essere visto come la rappresentazione astratta della computazione di un programma.
Un'applicazione pratica di questi automi possiamo trovarla in quella che viene chiamata *programmazione basata su automi* [19]. Si tratta di un paradigma di programmazione in cui il software viene sviluppato basandosi proprio sulla logica di funzionamento delle macchine a stati. La programmazione basata su automi trova impiego in settori molto delicati come quello dei compilatori e degli interpreti, nella bioinformatica, ma anche in quello della gestione della concorrenza. Quest'ultimo è proprio il motivo per il quale siamo interessati agli automi in questa tesi. Tipicamente per rappresentare un automa viene utilizzato un diagramma a stati. Vediamo quindi un esempio di automa e partiamo da un problema abbastanza semplice e di uso quotidiano: l'esecuzione di una telefonata.

Come possiamo osservare in figura 2.1, gli elementi chiave per la descrizione di un automa sono:

- un insieme finito di stati;
- un insieme finito di ingressi;
- un meccanismo di transizione che permetta di passare da uno stato all'altro quando si ricevono gli ingressi.

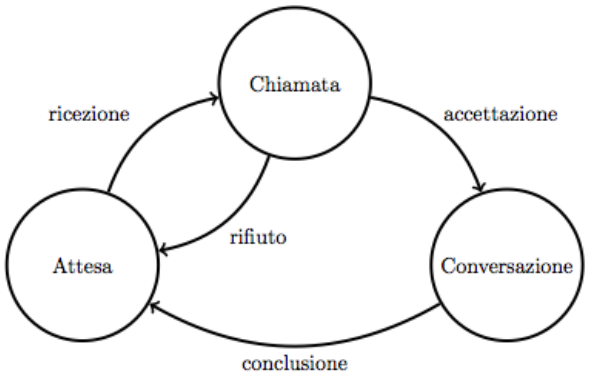


Figura 2.1: Automa per la descrizione di una telefonata

Esistono varie classi di automi a cui corrispondono diverse classi di linguaggi, caratterizzate da diversi livelli di complessità. Per questa tesi siamo interessati a quella classe di automi cosiddetti a *stati finiti*. Questi sono tra gli automi più semplici presenti in letteratura, infatti sono generalmente considerati come il punto di partenza per la discussione sugli automi. La nozione di finito fa riferimento al fatto che il numero di simboli di ingresso e di stati sia rappresentabile da un numero finito. Diamo quindi qui di seguito la definizione formale:

Definizione 2.1. Un automa a stati finito (FSA) è una tupla del tipo $A = \langle \mathcal{S}, \mathcal{L}, \rightarrow, s_0 \rangle$ dove:

- $\mathcal{S}$ è un insieme finito di stati;
- $\mathcal{L}$ è un insieme finito di labels, con $\epsilon \notin \mathcal{L}$ (dove ϵ rappresenta la stringa vuota);
- $\rightarrow \subseteq \mathcal{S} \times (\mathcal{L} \cup \{\epsilon\}) \times \mathcal{S}$ è la funzione di transizione (dove ϵ rappresenta la stringa vuota);

- $s_0 \in \mathcal{S}$ è lo stato iniziale.

Come possiamo notare dalla definizione appena introdotta, è stato definito uno stato s_0 iniziale dal quale inizia la computazione. Tipicamente nella definizione di automa è presente anche un insieme di stati detti di *accettazione* o *finali*. La nostra definizione di FSA non specifica l'insieme degli stati finali dal momento che consideriamo solo FSAs dove ogni stato è di accettazione. Va inoltre sottolineato che la nostra definizione coincide con la definizione di automa a stati finito di tipo *non deterministico*, che in letteratura viene solitamente indicato con la sigla *NFA (Non deterministic Finite Automata)*, per distinguerlo dal *DFA (Deterministic Finite Automata)*.

Sebbene NFA e DFA siano equivalenti, cioè riconoscono la stessa classe di linguaggi, vedremo più avanti che ci tornerà utile poter convertire un NFA in un DFA. In un NFA ci sono due cause di non determinismo. Una è data dalle ϵ-transizioni, un'altra è la possibilità che da uno stato partano più transizioni con la stessa etichetta. Per trasformare un NFA in un DFA equivalente, è utile poter indicare in modo sintetico gli stati raggiungibili a partire da uno stato fissato, esplicitando le scelte non deterministiche. Iniziamo quindi col definire la cosiddetta ϵ-*closure* di uno stato s, cioè l'insieme degli stati che si possono raggiungere da s tramite ϵ-transizioni.

Definizione 2.2. (a) Fissato un NFA N $= \langle \mathcal{S}, \mathcal{L}, \rightarrow, s_0 \rangle$ ed uno stato $s \in \mathcal{S}$, la ϵ-closure di s, indicata con ϵ-clos(s), è il più piccolo insieme di stati tale che

(i) s $\in \epsilon$-clos(s);

(ii) se p $\in \epsilon$-clos(s) allora $\rightarrow (p, \epsilon) \subseteq \epsilon$-clos(s).

(b) Se P è un insieme di stati, definiamo ϵ-clos(P) $= \cup_{p \in P}$ ϵ-clos(p).

Presentiamo qui di seguito un algoritmo in pseudocodice per calcolare la ϵ-*closure* di P, dove P può essere un singolo stato o un insieme di stati.

Per gestire l'altra forma di non determinismo, definiamo la seguente funzione, che, dato un insieme di stati P e un simbolo a, restituisce l'insieme degli stati in cui si può trovare l'automa partendo da uno stato in P e consumando un input a, con $a \in \mathcal{L}$:

$$mossa : P(\mathcal{S}) \times \mathcal{L} \longrightarrow P(\mathcal{S})$$
$$mossa(P, a) = \cup_{p \in P}(\rightarrow (p, a))$$

NOTA: Attenzione a non far confusione con la notazione: la prima freccia (più lunga) denota l'associazione tra il dominio e il codominio di *mossa*, mentre la seconda è la funzione di transizione.

Calcola la ϵ-*closure* di P

```
Inizializza T = P;
Inizializza ϵ-clos(P) = P;
while T ≠ ∅ do
    scegli un qualsiasi r da T e rimuovilo
    for each s ∈→ (r, ϵ);
        if s ∉ ϵ-clos(P)
            add s to ϵ-clos(P);
            add s to T;
end while
```

Dato quindi un NFA N, costruiremo un DFA D che riconosce lo stesso linguaggio. L'idea di fondo è che uno stato di D può essere pensato come un insieme di stati del NFA N. In particolare, se N partendo dallo stato iniziale e consumando l'input $a_1...a_k$ può trovarsi in uno qualsiasi degli stati $s_1, ..., s_m$ (seguendo ovviamente percorsi diversi, visto che si tratta di un NFA), allora D iniziando nel suo stato iniziale e consumando (deterministicamente) lo stesso input $a_1...a_k$ si troverà nello stato che corrisponde all'insieme $s_1, ..., s_m$. Per questo l'algoritmo che vedremo è noto come *Costruzione per Sottinsiemi*.

Sia dunque $N = \langle \mathcal{S}, \mathcal{L}, \rightarrow, s_0 \rangle$ un generico NFA. Applichiamo il seguente algoritmo, nel quale $\mathcal{T}$ conterrà alla fine l'insieme degli stati di D e δ costituirà la funzione di transizione di D.

Converte un NFA N in un DFA D

```
Inizializza S = ϵ-clos(s0);                  ▷ Lo stato iniziale di D
Inizializza T = {S};                         ▷ T insieme di insiemi di stati
while ∃P ∈ T non marcato do
        marca P;
        for each a ∈ L{
            R = ϵ-clos(mossa(P,a));
            if R ∉ T { add R to T; }
            definisci δ(P, a) = R;
        }
end while
```

Il DFA D così ottenuto sarà definito come $D = \langle \mathcal{T}, \mathcal{L}, \delta, \epsilon\text{-clos}(s_0) \rangle$. Di seguito possiamo trovare in figura 2.2 e in figura 2.3 , rispettivamente un NFA e il DFA relativo costruito con l'algoritmo appena introdotto.

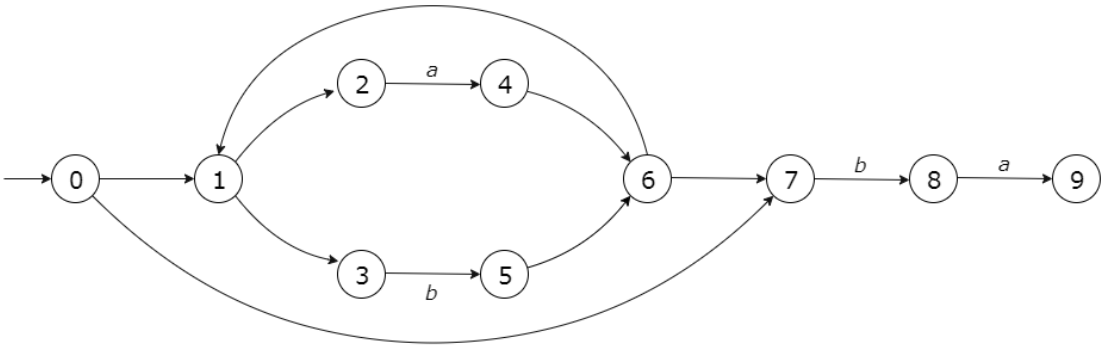
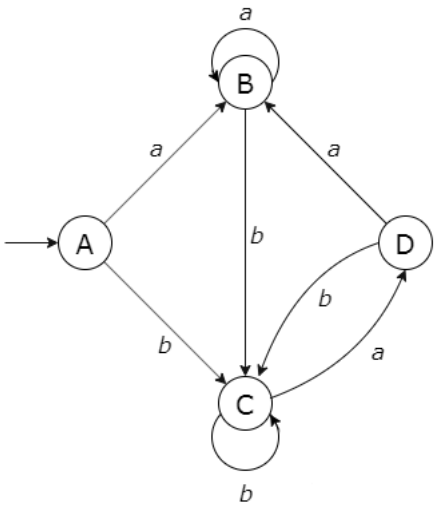


Figura 2.2: Esempio di NFA (gli archi vuoti sono ϵ-transizioni).



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}

Figura 2.3: Esempio di DFA

2.1.1 Minimizzazione

Nell'ambito della teoria degli automi, esistono poi tutta una serie di operazioni che è possibile effettuare su un singolo automa o su più automi. In particolar modo siamo interessati all'operazione di *minimizzazione* di un automa. L'idea che soggiace alla minimizzazione è che in un automa potrebbero esserci degli stati *equivalenti*, cioè che si comportano in modo indistinguibile rispetto al funzionamento dell'automa. Quando questo accade, gli stati equivalenti possono essere fusi tra loro, riducendo così il numero totale di stati e ottenendo un automa *minimo*. Esistono vari metodi per ottenere l'automa minimo a partire da un DFA. Quello che noi useremo prende il nome di *Riempimento della Tabella a Scala* [10].

Prima però di presentare questo metodo occorre aprire una parentesi e fare una precisazione sulla funzione di transizione di un'automa. In breve, il fatto che la funzione di transizione di un automa a stati finiti possa non essere definita per ogni simbolo di ingresso può rappresentare un problema. Infatti, quando andremo a vedere nel dettaglio l'algoritmo di minimizzazione, scopriremo che questo funziona solo se la funzione di transizione è sempre definita. Vista l'importanza dell'argomento, la letteratura distingue gli automi a stati finiti in due categorie: quelli *completi*, in cui la funzione di transizione è definita in ogni punto del suo insieme di partenza e quelli *incompleti*, nei quali la funzione di transizione è parziale. Dal momento che ci troviamo a trattare automi incompleti, questo ci pone di fronte ad un interessante problema: cosa fare se l'automa che ci interessa è incompleto e vogliamo minimizzarlo?

Una tecnica molto semplice per aggirare questo problema consiste nell'aggiunta di uno stato (che potrebbe avere il significato di stato d'errore) su cui facciamo convergere ogni transizione che non è definita nell'automa originale. Per esempio, prendiamo l'automa in figura 2.1. Per completarlo, aggiungiamo uno stato di Errore come mostrato qui di seguito.

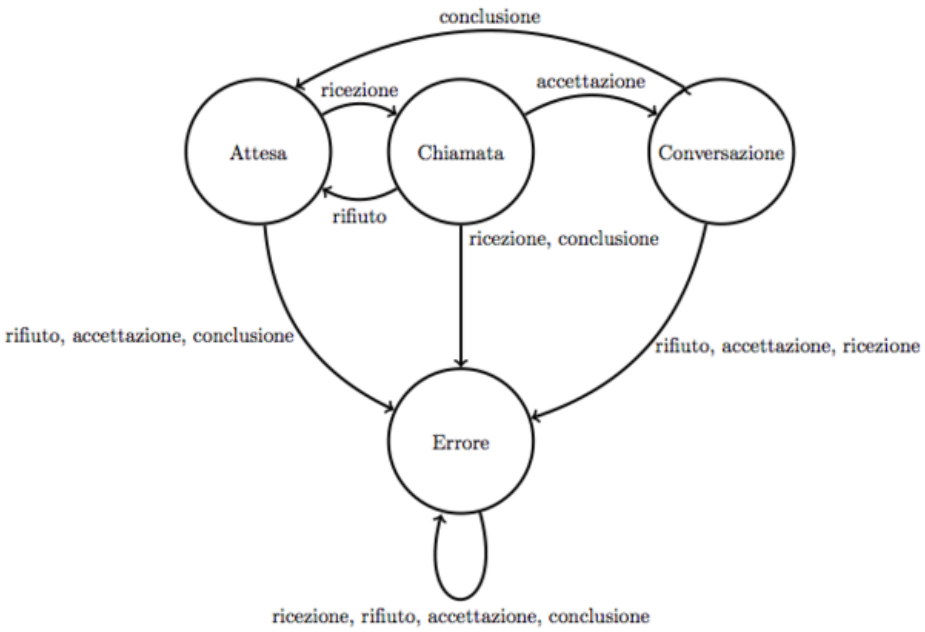


Figura 2.4: Automa della figura 2.1 completato

Passiamo ora a vedere nel dettaglio l'algoritmo di minimizzazione. Consideriamo il DFA in figura 2.3. Come già accennato l'algoritmo è basato su una tabella a scala

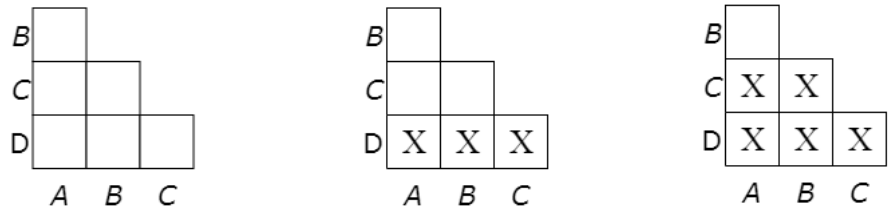


Figura 2.5: Riempimento della tabella a scala

e quella relativa al DFA che stiamo considerando la possiamo vedere in figura 2.5a. In generale la tabella ha un elemento per ogni coppia di stati distinti; l'elemento nella coppia (p,q) è destinato a contenere X se p e q sono distinguibili, altrimenti rimarrà vuoto. Il riempimento inizia quindi mettendo X in ogni coppia (p,q) dove uno dei due stati è finale, mentre l'altro non è finale.

Ricordiamo a tal proposito che la nostra definizione di automa (2.1) specifica automi che hanno solo stati considerati finali. L'unico stato non finale che possono avere è quello che andiamo ad aggiungere in un secondo momento per poterli completare, come mostrato nell'esempio in figura 2.4. Possiamo di conseguenza marcare tutte le coppie del tipo (p,k) o (k,p) dove k è lo stato di errore aggiunto nella fase di completamento dell'automa. Nel caso del DFA in figura 2.3, anche se non esplicitamente dichiarato, stiamo considerando gli stati A,B e C come non finali e lo stato D come finale. Marchiamo quindi tutte le coppie in cui compare lo stato D, figura 2.5b. Adesso si prendono in considerazione, una dopo l'altra, tutte le coppie (p,q): si marca con X la coppia se esiste un simbolo a per cui la coppia $(\rightarrow (p,a), \rightarrow (q,a))$ è già marcata con X. Nel nostro caso, marchiamo (A,C) (perché a li distingue) e (B,C) (distinti anch'essi da a), col risultato in figura 2.5c. Per la coppia (A,B), invece, vediamo che con a si va nella coppia (B,B), mentre con b porta (A,B) in (C,C): in questo caso A e B sono certo indistinguibili e lasciamo vuota al casella (A,B). Avendo esaminato tutte le coppie, l'algoritmo termina. L'automa minimo risultante è quello mostrato in figura 2.6.

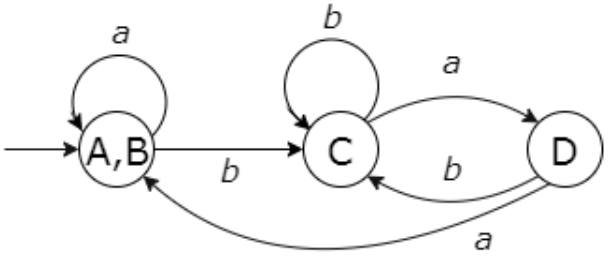


Figura 2.6: Automa minimo

Algoritmo per minimizzare un DFA

Input: DFA
Output: DFA minimizzato
begin

I) Disegna la tabella a scala con ogni coppia di stati (p,q), ogni casella è inizialmente non marcata;

II) Marchia ogni coppia di stati (p,q) del DFA per cui p è finale e q non è finale, e viceversa;

III) Ripeti questo step finché non è più possibile marchiare nessun elemento della tabella - marchia la coppia (p,q) se esiste un elemento a $\in \mathcal{L}$ per cui $(\rightarrow (p,a), \rightarrow (q,a))$ è marchiato;

IV) Al termine del riempimento, considera come un unico stato l'insieme degli elementi di ogni coppia che non è stata marcata nella tabella.

end

2.1.2 Prodotto

Un'ultima operazione sugli automi alla quale siamo interessati è quella di composizione tra automi, e più nello specifico a quella di prodotto tra automi.

Definizione 2.3. (Prodotto di due automi). Siano $A_1 = \langle \mathcal{S}_1, \mathcal{L}_1, \rightarrow_1, s_{01} \rangle$ e $A_2 = \langle \mathcal{S}_2, \mathcal{L}_2, \rightarrow_2, s_{02} \rangle$ due automi a stati finiti.
Definiamo A $= \langle \mathcal{S}, \mathcal{L}, \rightarrow, s_0 \rangle$ come il prodotto $A_1 \times A_2$, dove:

- $\mathcal{S} = \mathcal{S}_1 \times \mathcal{S}_2$;
- $\begin{cases} \rightarrow ((s_1,s_2),a) = \{(x,y) | x \in \rightarrow (s_1,a), y \in \rightarrow (s_2,a)\} & \textit{se } \rightarrow (s_1,a) \textit{ e } \rightarrow (s_2,a) \textit{ definite;} \\ \textit{non definito} & \textit{altrimenti;} \end{cases}$
- $s_0 = (s_{01}, s_{02})$;

Di seguito viene mostrato (figura 2.7) un esempio di prodotto tra due automi. Come si può osservare ogni stato dell'automa risultante è una coppia constituita da uno stato di c1 e c2.

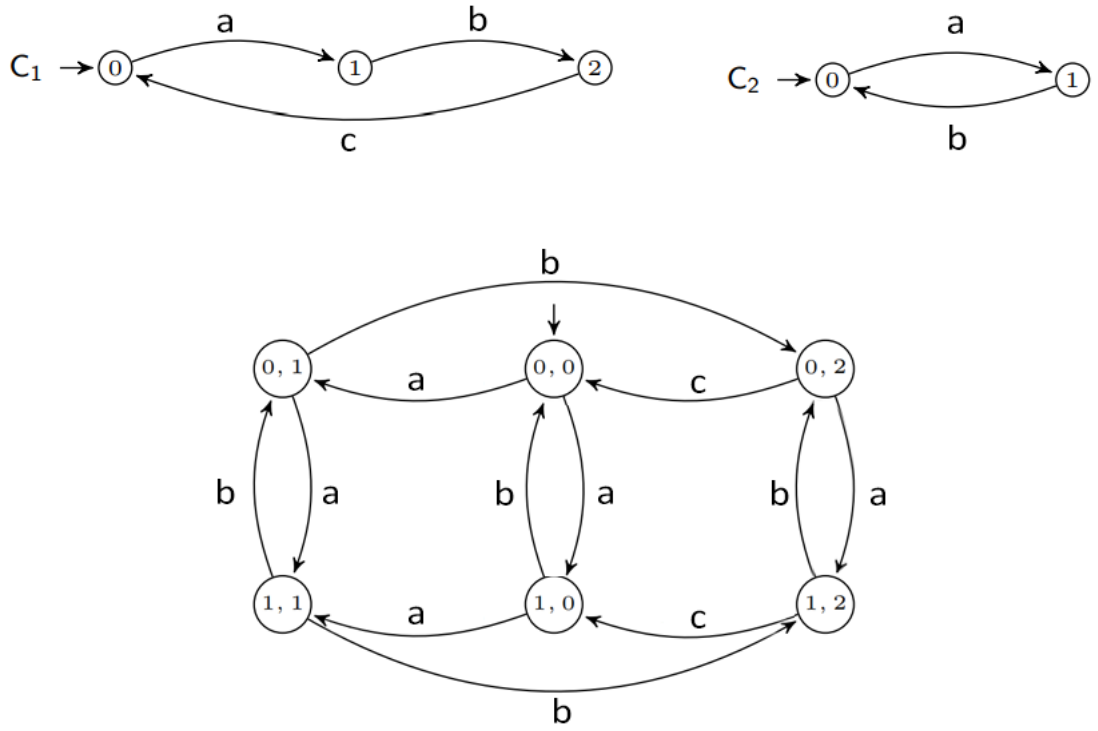


Figura 2.7: Esempio di prodotto tra due automi, c1 e c2

2.2 Choreography Automata

Ora che abbiamo tutti gli elementi di cui abbiamo bisogno, possiamo introdurre i Choreography Automata (CA) [7]. Questi possono essere visti come un modello per specificare le global views delle Choreographies. Una global view non è niente altro che una descrizione del comportamento dei processi (in questo caso li chiameremo *partecipanti*) in esecuzione, da un punto di vista globale. Essendo basati su automi, i CA sono convenienti perché ci permettono di riutilizzare tutta la teoria esistente sugli automi, e già presentata nella precedente sezione. Quindi in breve i CA sono degli automi a stati finiti le cui transizioni (gli archi tra gli stati) specificano delle interazioni tra i partecipanti della coreografia. L'automa presente in figura 2.8 è un esempio di Choreography Automata. Come possiamo osservare su ogni arco dell'automa è specificata un'interazione del tipo $A \rightarrow B : msg$ per descrivere il fatto che A manda un messaggio msg a B, dove A e B sono partecipanti della coreografia.

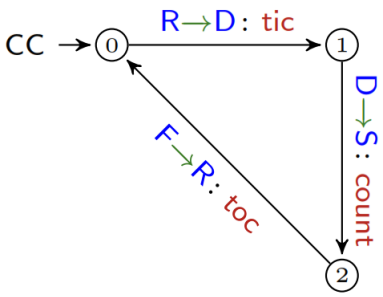


Figura 2.8: Esempio di Choreography Automata

Diamo quindi ora la definizione formale:

Definizione 2.4. (Choreography automata). Un Choreography Automaton (c-automaton) è un ϵ-*free* FSA con insieme di labels:

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

dove:

- $\mathcal{P}$ è un insieme di *partecipanti* (A, B, C, ecc.)
- $\mathcal{M}$ è un insieme di *messaggi* (m, n, x, ecc.)

NOTA: nonostante nella definizione non siano ammesse ϵ-transizioni, il non-determinismo è comunque possibile, ma rientra in una di quelle caratteristiche che si vuole evitare per avere delle composizioni corrette.

2.3 Communicating-FSM e Proiezione

A partire da una global view è possibile ottenere le local views dei partecipanti. Queste ultime a differenza della global view, specificano il comportamento di un singolo partecipante della coreografia rispetto agli altri. L'operazione per ottenere la local view è chiamata *proiezione* e può ossere applicata ad ogni partecipante della coreografia. Prima di definire l'operazione di proiezione occorre introdurre però le *Communicating Finite-State Machines (CFSM)* [8]. Queste sono un modello, sempre basato su automi per la descrizione delle local views di una coreografia. Sostanzialmente sono simili ai CA, con la sola differenza che le labels che possiamo trovare sugli archi dell'automa sono del tipo "A B !m" per dichiarare che A manda un messagio m a B. Oppure "A B ?m" per dichiarare che A riceve un messaggio m da B.

Definizione 2.5. (CFSM). Una Communicating Finite-State Machine (CFSM) è un FSA M con insieme di labels:

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

dove: $\mathcal{P}$ e $\mathcal{M}$ sono sempre l'insieme dei partecipanti e dei messaggi.
Il soggetto di un'azione output A B!m (risp. input A B?m) è A (risp. B). Indichiamo con M_A una CFSM che ha tutte le transizioni con soggetto A.

Un (Communicating) system è una funzione che assegna ad ogni partecipante A $\in \mathcal{P}$ la sua relativa CFSM M_A.

Definizione 2.6. (Proiezione). La proiezione su A di una transizione t = s $\xrightarrow{\alpha}$ s' di un Choreography Automaton (CA), scritta $t{\downarrow_A}$ è definita come:

$$t{\downarrow_A} = \begin{cases} s \xrightarrow{\text{A C !m}} s' & se\ \alpha = B \to C : m \wedge B = A \\ s \xrightarrow{\text{B A ?m}} s' & se\ \alpha = B \to C : m \wedge C = A \\ s \xrightarrow{\epsilon} s' & se\ \alpha = B \to C : m \wedge B, C \neq A \\ s \xrightarrow{\epsilon} s' & se\ \alpha = \epsilon \end{cases}$$

La proiezione di un CA = $\langle \mathcal{S}, \mathcal{L}_{int}, \to, s_0 \rangle$ sul partecipante $A \in \mathcal{P}$, denotata con CA $\downarrow_A$, è ottenuta a partire dall'automa intermedio:

$$A_a = \langle \mathcal{S}, \mathcal{L}_{act}, \{s \xrightarrow{t{\downarrow_A}} s' | s \xrightarrow{\text{t}} s' \in \to\}, s_0 \rangle$$

togliendo le possibili ϵ-transizioni e applicando subito dopo la minimizzazione vista in 2.1.1

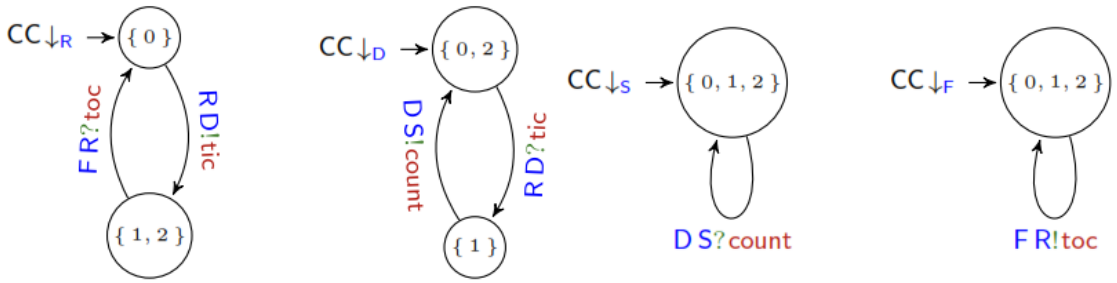


Figura 2.9: Proiezioni del Choreography Automata CC

2.4 Composizione

In questa sezione discutiamo di un problema tipico al quale si va incontro nel momento in cui si hanno più global views e si cerca di farle comunicare tra loro. In poche parole abbiamo bisogno di un metodo per *comporre* due o più global views insieme in modo da ottenere un'unica global view.

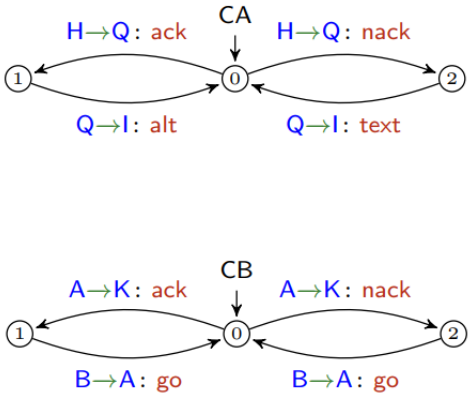


Figura 2.10: Choreography Automata CA e CB

L'operazione di composizione che andremo a presentare in questa sezione parte dal pressupposto che gli insiemi dei partecipanti siano disgiunti. Questo per evitare motivi di ambiguità nella global view risultante. La composizione è ottenuta mediante due operazioni concatenate:

I) Il prodotto dei due Choreography Automata;

II) Una seconda operazione detta di *"Sincronizzazione"*.

Per quanto riguarda il primo punto, si tratta del classico prodotto cartesiano tra due automi, descritto precedentemente in definizione 2.3. Invece per quanto riguarda il secondo punto, l'operazione di *"Sincronizzazione"* prevede che per ogni global view ci sia un partecipante (che in questo caso prenderà il nome di *"interfaccia"*) con il ruolo di *forwarder*. Cioè ogni view può comunicare con l'altra view, inviando o ricevendo messaggi tramite la propria interfaccia. Vediamo un esempio per chiarire meglio l'idea. Prendiamo i due Choreography Automata mostrati in figura 2.10, *CA* (a sinistra) e *CB* (a destra). Consideriamo come interfaccia per CA il partecipante *H*. Mentre come interfaccia per CB consideriamo il partecipante *K*.

Ogni qualvolta un partecipante di CA (risp. CB) vuole mandare un messaggio all'altra coreografia, manderà un messaggio a H (risp. K).

Presentiamo di seguito l'algoritmo informale per la Sincronizzazione. La figura 2.11 mostra la composizione dei due Choreography Automaton CA e CB visti in figura 2.10, usando come interfacce rispettivamente H e K.

Algoritmo di Sincronizzazione

Input: un Choreography Automaton e due partecipanti di esso, H e K.
Output: un Choreography Automaton.
begin

I) Ogni transizione con $p \xrightarrow{A \to H : m} q$ (risp. $p \xrightarrow{A \to K : m} q$) è rimossa, e per ogni transizione $q \xrightarrow{A \to B : m} r$ (risp. $q \xrightarrow{H \to B : m} r$) una transizione $p \xrightarrow{A \to B : m} r$ (risp. $p \xrightarrow{A \to B : m} r$) viene aggiunta verificando che $A \neq B$; se $A = B$ l'operazione di Sincronizzazione non è definita;

II) le transizioni che non coinvolgono né H né K sono mantenute, mentre tutte le altre sono rimosse;

III) gli stati e le transizioni non raggiungibili dallo stato iniziale sono rimossi;

end

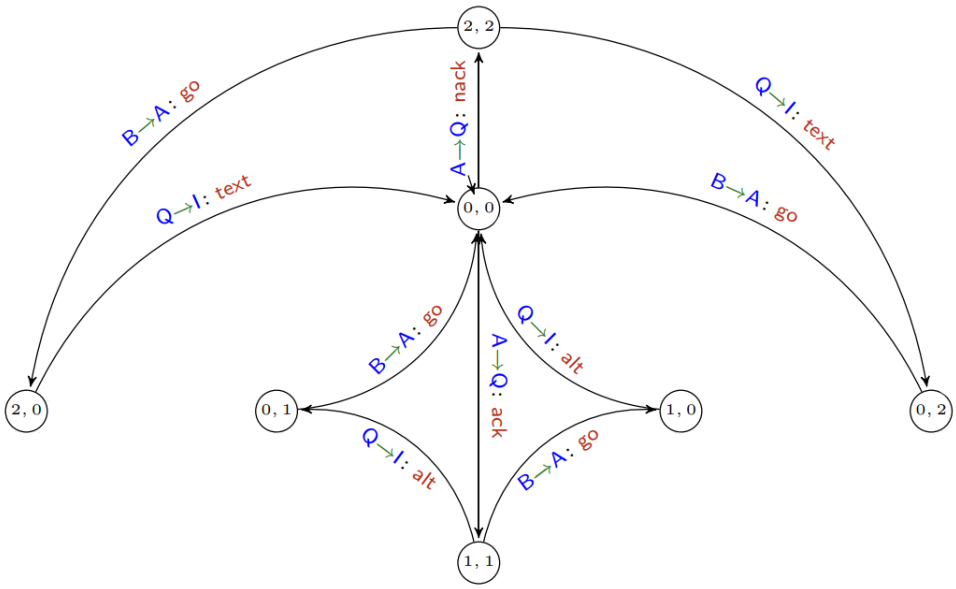


Figura 2.11: Composizione tra CA e CB sulle interfacce H e K

Capitolo 3

Librerie usate

3.1 Graphviz e Linguaggio DOT

Per costruire ed esportare i grafi all'interno di Corinne, è stata utilizzata Graphviz [4]: una libreria open source per disegnare grafi descritti nel linguaggio *DOT* [3].

DOT è un linguaggio di descrizione per grafi, tipicamente con estensione '.gv'. Di seguito viene proposto un esempio di possibile grafo in linguaggio DOT che Corinne può prendere in input:

```
1    digraph cc {
2        s0 [label="" height=0 width=0]
3        s0 -> 0
4        0 -> 1 [label="R->D:tic"]
5        1 -> 2 [label="D->S:count"]
6        2 -> 0 [label="F->R:toc"]
7    }
```

La sintassi di DOT è molto semplice, si comincia indicando il tipo di grafo che si vuole descrivere, in questo caso "digraph" (un grafo bipartito), seguito dal nome del grafo, in questo caso "cc". Poi segue tra parentesi graffe la descrizione di ogni nodo e arco presente nel grafo, generalmente ognuno diviso da un carattere newline. Ogni nodo è specificato da un *id*, tipicamente un intero, ma può essere anche una stringa. Seppur possibile, non è necessario specificare i nodi singolarmente, ma è possibile definire direttamente gli archi che li collegano. Per esempio nel grafo che stiamo considerando, nelle righe 4,5 e 6 piuttosto che definire singolarmente i nodi 0,1 e 2, definisco direttamente gli archi tra loro.

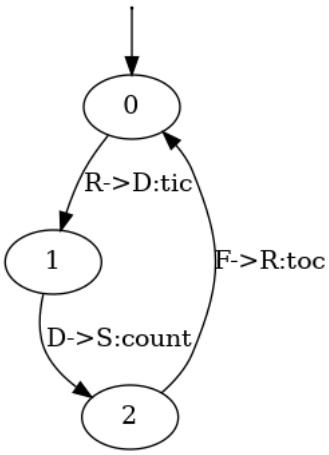


Figura 3.1: Il grafo generato dal codice DOT

NOTA: nell'esempio precedente, oltre a definire i 3 archi di cui è composto il grafo, viene definito un nodo `s0` collegato al nodo iniziale `0`. Come si può osservare nella figura 2.1, questo è in realtà un nodo fittizio che viene creato per visualizzare la freccia entrante nel nodo iniziale `0`. Infatti, i parametri definiti dentro `s0`, cioè `height` e `weight` impostati a zero hanno la funzione di rendere il nodo invisibile, e mostrare solamente la freccia da esso uscente.

3.2 ANTLR

Per prendere in input i DOT files con la sintassi dei Choreography Automata è stato creato un parser utilizzando *ANTLR (Another Tool for Language Recognition)* [16] un noto generatore di parser top-down. A partire da una grammatica (file con estensione '.g4'), ANTLR genera un parser per il linguaggio riconosciuto da quella grammatica. La sintassi per i Choreography Automata è definita dentro il file `DOT.g4` e qui di seguito viene mostrato un breve estratto del file.

```
1 graph   :   'digraph' string '{' '\n' stmt_list+ '}' ;
2 stmt_list:  stmt '\n' ;
3 stmt : node | edge ;
4 node : id_node '[' (attr_list ','?)* ']' ;
5 edge : id_node '->' id_node ('[' 'label' '=' '"' label? '"' ']')? ;
6 ...
```

I vantaggi di creare un parser con ANTLR, piuttosto che scriverlo manualmente sono molteplici: facilità nell'intepretare la grammatica usata, una maggiore flessibilità nel caso di futuri cambiamenti, ottimizzazioni a run-time, ecc. Ma non solo, ANTLR costruisce l'albero di parsing e il *"Tree walker"* per visitare i nodi dell'albero. Nel dettaglio, ANTLR offre 2 tipologie di Tree walker, un *Visitor* e un *Listener*. Entrambi utilizzano una *Depth-First Search (DFS)* per esplorare l'albero. La principale differenza è che il Listener offre due metodi, una *Enter* e una *Exit*, per ogni nodo visitato. La Enter viene chiamata ogni qualvolta si incontra per la prima volta un nodo, mentre la Exit viene chiamata solo dopo aver visitato tutti i nodi children del nodo considerato. Invece con il Visitor viene invocato un solo metodo per ogni singolo nodo. Un'altra differenza tra Visitor e Listener è che il primo può avere un valore di ritorno, mentre il secondo no.

Capitolo 4

Corinne

Corinne è il tool sviluppato per questa tesi. Scritto in Python3, il suo principale scopo è automatizzare le operazioni di composizione e proiezione dei Choreography Automata. Corinne è un software libero con licenza MIT. Disponibile al download all'indirizzo: `https://github.com/simoneorlando/Corinne`.

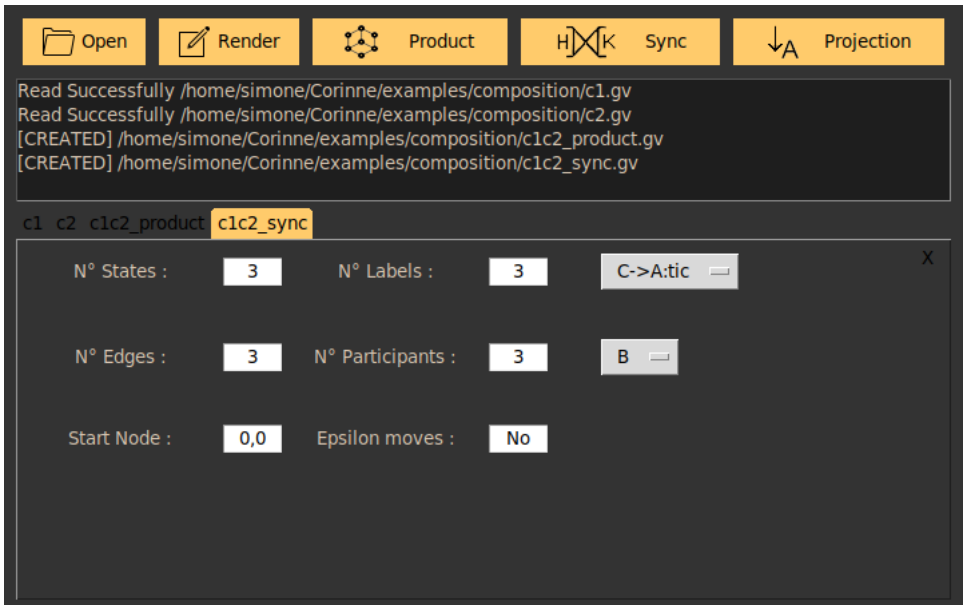


Figura 4.1: Screenshot dell'interfaccia

4.1 Interfaccia

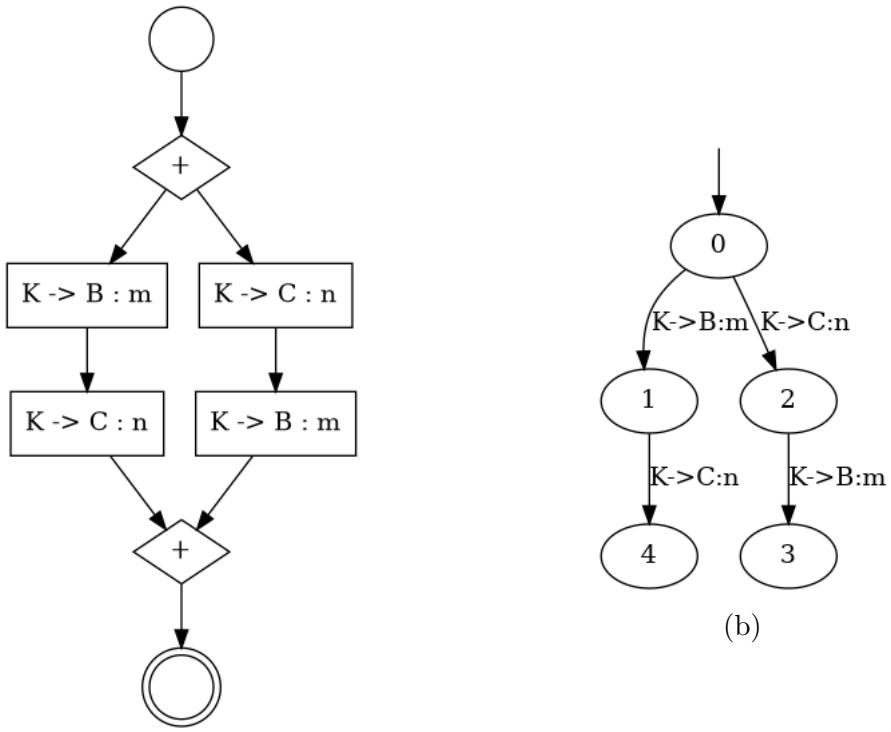
Per sviluppare l'interfaccia è stata utilizzata la libreria *Tkinter* [6]: una libreria multipiattaforma per la creazione di interfacce grafiche. Questo, unito al fatto che Corinne è stato scritto in Python3, assicura la possibilità di poterlo usare su più sistemi operativi senza problemi di compatibilità. Come si può osservare in figura 4.1, il tool si presenta con un'interfaccia piuttosto minimale. Nella parte superiore abbiamo 5 pulsanti, uno per ogni funzionalità principale offerta. Subito sotto c'è una piccola finestra che ospita i messaggi di log del programma. Nella parte inferiore ci sono le tab, una per ogni grafo aperto, su cui sono visualizzate alcune informazioni su di esso come il numero di stati, numero di archi, i partecipanti, ecc.

NOTA: Dato che un Choreography Automata (CA) può essere visto come un grafo, nel seguito potrei quindi riferirmi a un CA usando la parola grafo.

4.2 Input

Il tool prende in input un CA in formato DOT (.gv), ma non solo. Corinne è in grado di prendere in input files con una sintassi diversa da quella dei Choreography Automata. In particolare può prendere in input files provenienti da *Chorgram* [11] e *Domitilla* [2]. Il primo è un framework per *Global Graph* [14]. Ispirati ai *generalised global type* [9] e alle *BPMN choreography* [12], i Global Graph sono un altro formalismo per esprimere global views. Invece, Domitilla è un altro tool, sempre per Global Graph, ma che introduce una distinzione tra *Structured Global Graph* e *Unstructured Global Graph* [17], questi ultimi usati per la *fusione* di due coreografie.

Corinne è quindi dotato di 2 tipi di parser diversi. Uno per i file DOT, di cui abbiamo già visto precedentemente un breve estratto della grammatica. Questo parser, oltre a poter prendere in input i file con la sintassi dei Choreography Automata, può prendere in input anche i file provenienti da Domitilla. Questi ultimi utilizzano lo stesso tipo di etichette usato dai Choreography Automata ma invece di essere memorizzate nei nodi, sono memorizzate sugli archi. Corinne quindi non fa niente altro che proiettare le etichette dai nodi sugli archi e numerare i nodi. In figura 4.2 possiamo osservare l'esempio di un grafo di Domitilla convertito in uno di Corinne.



(a) (b)

Figura 4.2: Un grafo di Domitilla (a) convertito (tramite Corinne) nel formato dei Choreography Automata (b)

Un secondo parser invece prende in input i file provenienti da Chorgram. La sintassi usata da Chorgram è abbastanza semplice e viene mostrata di seguito:

```
1     G:  Ptp '->' Ptp ':' Msg    # interaction
2     | G ';' G                   # sequential
3     | G '+' G                   # choice
4     | G '|' G                   # fork
5     | '*' G '@' Ptp             # loop
6     | '{' G '}'                 # parenthesis
7     ;
8
9     Ptp : [A-Z]+ ;
10    Msg : [a-z]+ ;
```

Corinne prende quindi in input un file nella sintassi di Chorgram (in genere un file 'txt') e costruisce il grafo relativo in formato 'DOT'. Per esempio, prendiamo una stringa di Chorgram, come la seguente:

```
{A->B:m ; C->D:m} + B->Z:n
```

In figura 4.3 possiamo osservare il grafo che Corinne costruisce a partire da questa stringa.

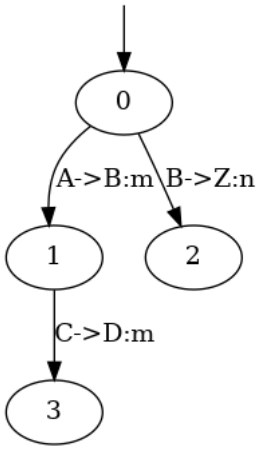


Figura 4.3: Un grafo costruito (tramite Corinne) a partire da una stringa di Chorgram

4.3 Utilizzo

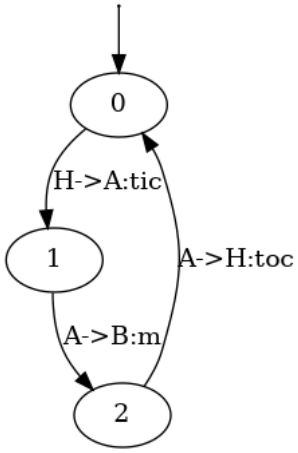
Tramite il tasto **Open** viene selezionato un file da aprire, il quale verrà dato al parser per la creazione del Choreography Automaton corrispettivo. Se non ci sono errori nel file, e il processo va a buon fine, verrà creata una nuova tab nell'elenco tab in basso, contenente le caratteristiche del grafo appena letto.

Tramite il tasto **Render**, invece, viene selezionato un file DOT (.gv) da aprire, di cui si vuole la rappresentazione grafica su file in formato PDF o PNG.

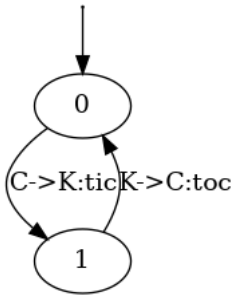
Veniamo ora a descrivere le funzioni principali del tool. Come si può notare in figura 4.1, il tool non prevede un tasto per la composizione così come descritta nel capitolo 2.4. Infatti il tool offre le due operazioni chiave della composizione (prodotto e sincronizzazione) separatamente. Alla base di questa scelta ci sono due casi possibili in cui è preferibile avere le due operazioni separate. Una, meno frequente ma comunque possibile, per sincronizzare due interfacce di un solo grafo. L'altra, decisamente più frequente, per sincronizzare più di due grafi.

In quest'ultimo caso occorre prima effettuare il prodotto di tutti i grafi, e solo in un secondo momento effettuare la sincronizzazione delle interfacce.

NOTA: per poter applicare all'interno del tool le operazioni che andremo a discutere qui di seguito, occorre prima aprire i/il grafi/o tramite il tasto **Open**. I grafi che verranno mostrati da ora in poi sono stati creati tutti utilizzando *Corinne*.



(a) c1



(b) c2

Figura 4.4: Choreography Automata c1 e c2 (creati tramite Corinne)

4.3.1 Prodotto

Il tasto **Product** permette, dati due grafi, di effettuare il prodotto così come già descritto in 2.3. Ricordiamo che può essere effettuato su Choreography Automata che hanno insieme di partecipanti disgiunti, altrimenti non è definito. Nel momento in cui si clicca il tasto, verrà visualizzata una finestra in cui sarà possibile selezionare due grafi precedentemente aperti e di salvare il prodotto risultante nel percorso che si desidera. In figura viene mostrato il prodotto tra c1 (figura 4.4a) e c2 (figura 4.4b)

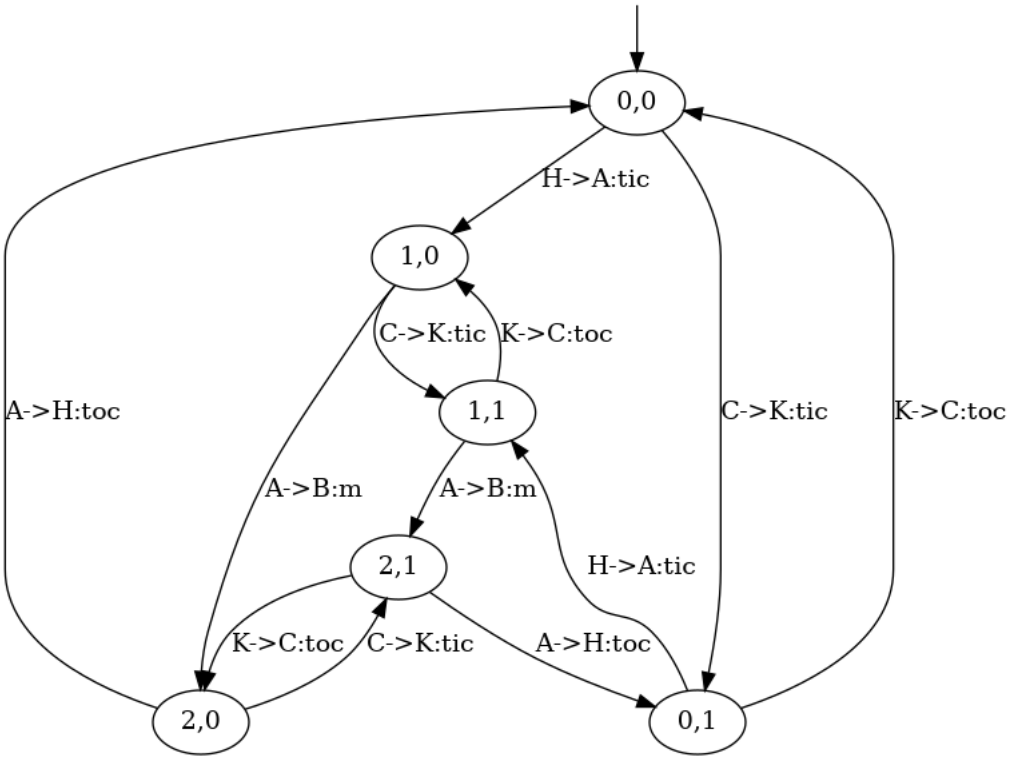


Figura 4.5: Prodotto tra c1 e c2 (usando Corinne)

4.3.2 Sincronizzazione

Con il tasto **Sync** è possibile effettuare la sincronizzazione, così come descritta nell'algoritmo al capitolo 2.4. Nel momento in cui si clicca il tasto verrà visualizzata una finestra in cui sarà possibile selezionare uno tra i grafi aperti, 2 tra i partecipanti del grafo, ed effettuare così la sincronizzazione. Il risultato di questa operazione restituirà un file DOT (.gv) del grafo sincronizzato nel percorso selezionato.

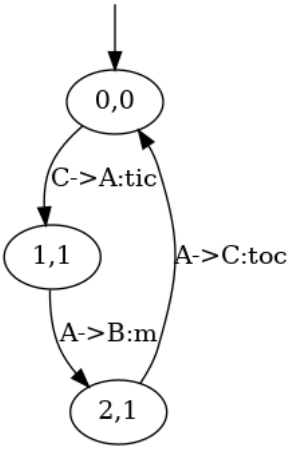
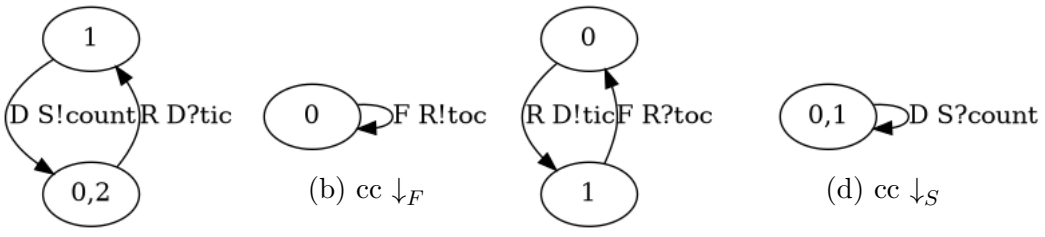


Figura 4.6: Sincronizzazione del prodotto mostrato in figura 4.5 (usando Corinne) sulle interfacce H e K

4.3.3 Proiezione

Tramite il tasto **Projection** è possibile effettuare la proiezione di un grafo (precedentemente aperto) su uno dei suoi partecipanti, come descritto in 2.3. Non appena si clicca sul tasto, sarà visualizzata una finestra nella quale sarà possibile scegliere il grafo e uno dei suoi partecipanti. Verrà così creato il Communicating-FSA relativo al partecipante selezionato in formato DOT (.gv) nel percorso selezionato. A livello implementativo, la funzione di proiezione non fa niente altro che cambiare le labels del Choreography Automata con quelle del Communicating-FSM, aggiustandole a seconda del partecipante considerato. Elimina quindi le ϵ-transizioni usando l'algoritmo presentato precedentemente per convertire un NFA in un DFA. Dopo di che, a partire da questo viene effettuata la classica minimizzazione dei DFA con l'algoritmo descritto nel capitolo 2.1.



(a) cc $\downarrow_D$ (b) cc $\downarrow_F$ (c) cc $\downarrow_R$ (d) cc $\downarrow_S$

Figura 4.7: Proiezioni di cc (create tramite Corinne)

4.4 Implementazione

Corinne è scritto in Python3 e conta all'incirca 3600 linee di codice. La scelta di scriverlo in Python3, piuttosto che in un altro linguaggio, è stata fatta per diversi motivi tra cui semplicità, disponibilità di librerie già esistenti come Graphviz e sicuramente anche la sua portabilità. Nel suo sviluppo si è cercato di applicare il pattern *M.V.C. (Model-View-Controller)*. Esiste quindi un file per la view (`guy.py`), uno per il controller (`controller.py`) e diversi files per il model.
Di seguito, viene mostrato un diagramma UML delle classi informale per descrivere la struttura del programma:

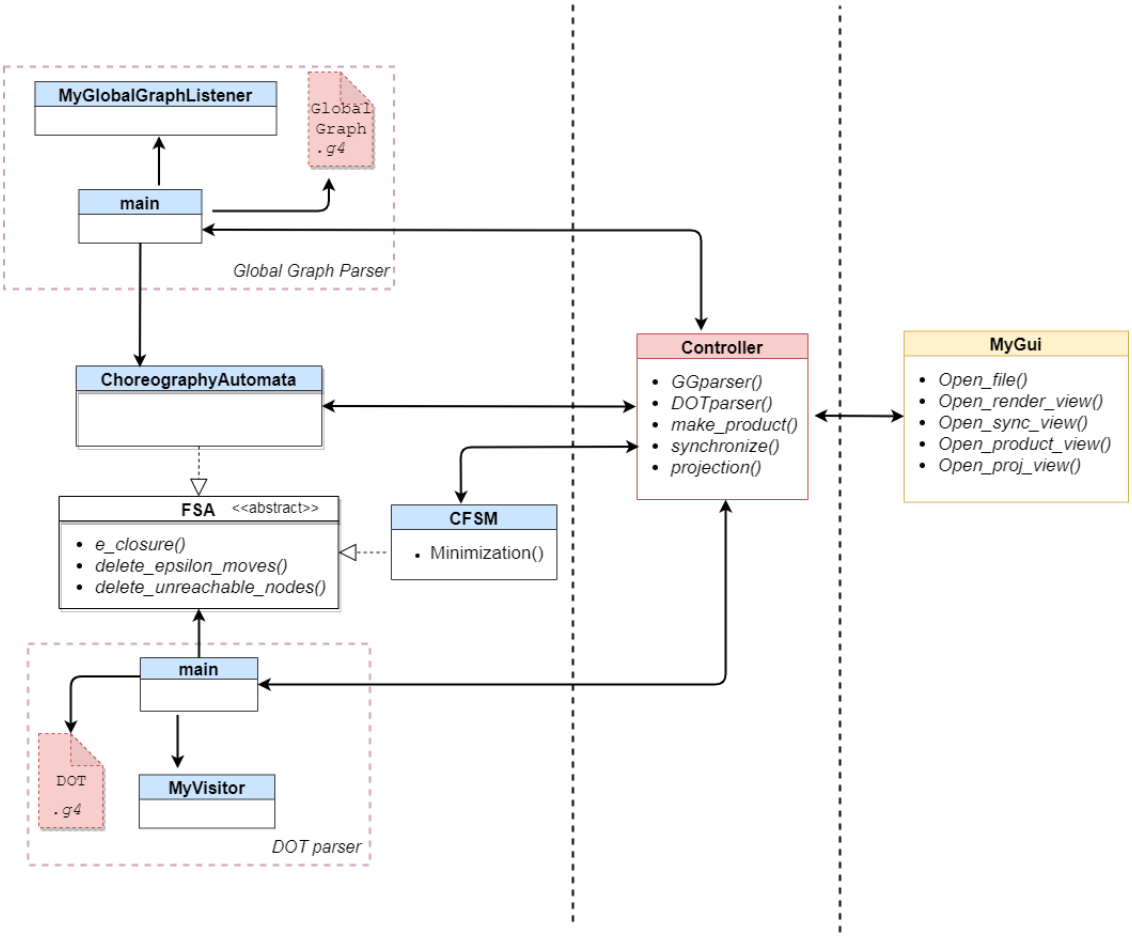


Figura 4.8: Diagramma della struttura del programma

Il controller è il cuore del programma e ospita la maggior parte delle funzioni chiave di Corinne. Qualsiasi richiesta fatta dall'utente, tramite la gui, viene passata al controller. Questo si occupa per esempio, di prendere un file input

(selezionato tramite la gui) e di darlo in input al parser relativo. Per ogni file che è stato aperto, viene quindi creata un'istanza della classe Choreography Automata, memorizzata all'interno del Controller. Nel dettaglio, all'interno del Controller è presente un dizionario di Python (`ca_dict`) che associa al nome del grafo, il relativo Choreography Automata, costruito al momento della lettura. Vediamo qui di seguito un estratto della classe `Controller`:

```
class Controller:
    ca_dict = {}

    def get_participants(self, graph_name):...

    def get_start_node(self, graph_name):...

    def get_states(self, graph_name):...

    ...

    def GGparser(self, path_file, path_store):...

    def DOTparser(self, path_file):...

    def DomitillaConverter(self, graph_name, path_file, ..):...

    def make_product(self, graph_name1, graph_name2, ..):...

    def synchronize(self, graph_name_to_sync, interface1, ..):...

    ...
```

Il model invece è costituito da 3 parti, una parte per il DOT parser, un'altra per il Global Graph parser e infine una per descrivere i Choreography Automata. Per questi ultimi è stata inizialmente creata una classe astratta per descrivere in generale gli automi a stati finiti (`fsa.py`), da cui poi ereditano le classi dei Choreography Automata e delle Communicating-FSM (`cfsm.py`). La classe degli automi a stati finiti, pur essendo una classe astratta, implementa la maggior parte dei metodi usati dagli automi, ad eccezione della minimizzazione. Quest'ultima ho scelto di implementarla nella classe delle CFSM poiché utilizzata solo da loro.

Di seguito viene mostrata la classe astratta degli automi a stati finiti:

```
class FSA:

    def __init__(self, states: set, labels: set,
     edges: set, s0: str):
        self.states = states
        self.labels = labels
        self.edges = edges
        self.s0 = s0

    def __e_closure__(self, nodes):...

    def delete_epsilon_moves(self):...

    def delete_unreachable_nodes(self):...

    @abstractmethod
    def __get_participants_and_message_from_label__(self, label):
        pass
```

Un FSA è definito tramite un insieme di stati, un insieme di labels, un insieme di archi e uno stato iniziale s0. L'insieme degli archi è un insieme di tuple, a differenza dell'insieme degli stati e delle labels che sono invece insieme di stringhe. Nello specifico un arco è definito tramite una tupla di 3 elementi del tipo `(nodo origine, label, nodo destinazione)`. Inoltre, come si può osservare questa classe definisce un metodo astratto per ottenere i partecipanti e il messaggio a partire dalla stringa della label. Questo metodo verrà implementato in modo diverso (per la natura diversa delle label) dalle classe dei CA e delle CFSM.

NOTA: il metodo `delete_unreachable_nodes` corrisponde al terzo step dell'algoritmo di sincronizzazione (2.4) ed è sostanzialmente una visita *Breadth-First Search (BFS)* a partire dal nodo iniziale nella quale vengono memorizzati i nodi visitati e scartati tutti gli altri.

Per quanto riguarda il model invece, come già accennato nel capitolo 3.2, ANTLR mette a disposizione 2 tipi di *"Tree walker": Listener e Visitor*. Ovvero 2 classi astratte che offrono dei metodi (da implementare) invocati al momento della visita dell'albero di parsing.

Vediamo un estratto della classe `MyVisitor` per la costruzione di un Choreography Automata:

```
class MyVisitor(DOTVisitor):

    def __init__(self):
        self.states = set()
        self.labels = set()
        self.edges = set()
        self.s0 = None
        self.participants = set()

    def visitGraph(self, ctx:DOTParser.GraphContext):...

    def visitEdge(self, ctx:DOTParser.EdgeContext):...

    ...
```

`MyVisitor` implementa i metodi della classe astratta `DOTVisitor` la quale viene generata da ANTLR a partire dalla grammatica che gli abbiamo dato. Tramite essa è possibile visitare l'albero di parsing. Quest'ultimo viene costruito automaticamente ogni qualvolta si prova a leggere un nuovo file. A partire quindi dal nodo radice dell'albero di parsing, vengono ricorsivamente visitati tutti i nodi figli e per ognuno viene invocato un metodo apposito. I metodi del `MyVisitor` sono stati implementati in modo tale da riempire gli elementi di cui è composto un Choreography Automata (insieme di stati, insieme di archi, ecc), per poi restituirlo come risultato.

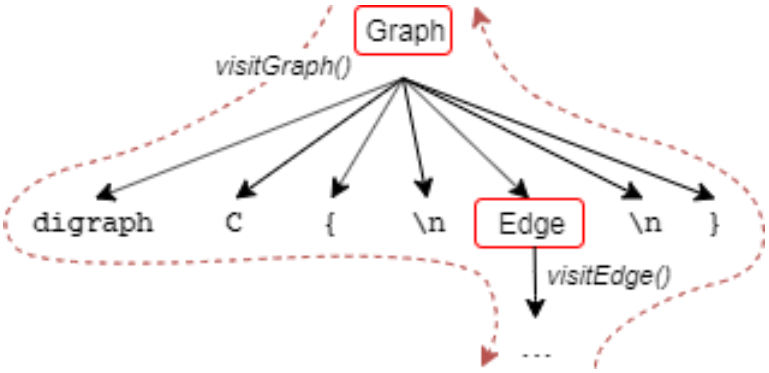


Figura 4.9: Esempio di albero di parsing

Capitolo 5

Conclusioni

Corinne rispecchia le caratteristiche stabilite e permette di fare quello per cui era stato concepito inizialmente; nonostante ciò non è esente da possibili miglioramenti. Ricordo che il tool è concesso in licenza MIT, di conseguenza è un software open source e in futuro chiunque voglia, potrà proporre nuove features da aggiungere o modificare quelle già esistenti.

In particolar modo, una nuova opzione che potrebbe essere aggiunta è quella di identificare i cosiddetti Choreography Automata "*well-formed*", ovvero quella classe di condizioni che devono soddisfare i CAs affinché ogni loro proiezione si comporti come effettivamente stabilito.

Un'altro aspetto che non è stato approfondito per motivi di tempo è un'analisi dei costi computazionali dei principali algoritmi. Infatti il tool è stato testato su grafi di piccole dimensioni, e considerando solamente l'operazione di prodotto cartesiano tra due o più grafi, non è escluso un aumento dei costi computazionali su grafi di dimensioni più importanti.

Concludendo, nonostante la finestra di miglioramenti che è ancora possibile fare, mi sento soddisfatto del lavoro e del cammino intrapreso che mi ha portato allo sviluppo di Corinne. Mi ha permesso di mettere alla prova le mie capacità di programmatore, scontrarmi con nuove problematiche come quella di ideare dall'inizio un'architettura per le varie componenti di un software. Ma non solo, ho potuto approfondire la conoscenza dei generatori di parser, come ANTLR, e di come questi funzionino. Inoltre, nello scegliere una licenza adatta ho approfondito la tematica del software open source, di come questo differisca dal software libero e delle principali licenze software disponibili.

Tutto ciò ha aumentato il mio bagaglio di conoscenze, e queste sicuramente mi saranno utili in futuro quando andrò a confrontarmi con tematiche simili.

Bibliografia

[1] *ANTLR, version(4.7.2),* https://www.antlr.org.

[2] *Domitilla,* https://github.com/dedo94/domitilla.

[3] *DOT language,* https://en.wikipedia.org/wiki/dot_(graph_description_language).

[4] *Graphviz, version(0.10.1),* https://www.graphviz.org/.

[5] *Python3,* https://www.python.org/.

[6] *Tkinter, version(8.6),* https://docs.python.org/3/library/tk.html.

[7] Franco Barbanera, Ivan Lanese, and Emilio Tuosto, *A compositional choreographic framework*, A compositional choreographic framework (submitted), 1–28.

[8] Daniel Brand and Pitro Zafiropulo, *On communicating finite-state machines*, J. ACM **30** (1983), no. 2, 323–342.

[9] Pierre-Malo Deniélou and Nobuko Yoshida, *Multiparty session types meet communicating automata*, pp. 194–213, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[10] M. Gabbrielli and S. Martini, *Linguaggi di programmazione. principi e paradigmi*, Collana di istruzione scientifica, ch. 3, McGraw-Hill Companies, 2011.

[11] Simon Gay and Ravara Antonio, *Behavioural types: from theory to tools*, ch. 6, River Publishers, 2017.

[12] Object Management Group, *Business process model and notation,* http://www.bpmn.org, 2019.

[13] 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** (2016), no. 1, 3:1–3:36.

[14] Julien Lange, Emilio Tuosto, and Nobuko Yoshida, *From communicating machines to graphical choreographies*, SIGPLAN Not. **50** (2015), no. 1, 221–232.

[15] Fabrizio Montesi, *Choreographic programming*, Ph.D. thesis, IT University of Copenhagen, 2013.

[16] Terence Parr, *The definitive antlr 4 reference*, 2nd ed., Pragmatic Bookshelf, 2013.

[17] Davide Schiavi, *Implementazione di un tool per l'elaborazione di unstructured global graph*, Bachelor's thesis, Alma Mater Studiorum, Università di Bologna, 2019.

[18] Gadi Taubenfeld, *Concurrent programming, mutual exclusion (1965; dijkstra)*, The Interdisciplinary Center, Herzliya, Israel (2008).

[19] Wikipedia contributors, *Automata-based programming — Wikipedia, the free encyclopedia*, 2019, [Online; accessed 2-December-2019].

[20] Wikiversità, *Automa a stati finiti — wikiversità,*, 2019, [Online; accesso il 26-novembre-2019].

Elenco delle figure