Alma Mater Studiorum · Università di Bologna

**SCUOLA DI SCIENZE**
**Corso di Laurea in Informatica**

# Choreia: A Static Analyzer to Generate Choreography Automata from Go Source Code

**Relatore:**
**Prof. Ivan Lanese**

**Presentata da:**
**Enea Guidi**

**Sessione III**
**Anno Accademico 2020/2021**

*Alla mia famiglia e agli amici che*
*mi hanno accompagnato in questo viaggio*

**Sommario**

Le coreografie sono un paradigma emergente per la descrizione dei sistemi concorrenti che sta prendendo piede negli utlimi anni. Lo scopo principale è quello di fornire al programmatore uno strumento che permetta di capire in maniera immediata la *coreografia* dei participanti all'interno del sistema e come questi interagiscano tra loro. Partendo dai singoli partecipanti, e le loro *viste locali*, è possibile ricomporre in maniera bottom-up l'intera Choreography (o *vista globale*) del sistema. Un ulteriore vantaggio delle Choreographies è che, quando rispettano alcune proprietà definite, danno garanzie sull'assenza di tipici problemi di concorrenza quali Deadlocks, Liveness e Race Conditions. Esistono vari modelli formali di Choreographies, questa tesi tratta nello specifico i *Choreography Automata*, basati su *Finite State Automata* (FSA). In questa tesi viene presentato Choreia: un tool di analisi statica che, partendo da un codice sorgente Go, ricava il Choreography Automata del sistema concorrente in maniera bottom-up.

# Indice

# Capitolo 1

# Introduzione

# Capitolo 2

# Nozioni preliminari e notazione

## 2.1 FSA non deterministici e deterministici

Prima di introdurre le coreografie e i Choreography Automata è necessario fare un breve richiamo di alcune nozioni fondamentali quali la nozione di Automa a Stati Finiti (FSA)[2] e alcune operazioni possibili sugli stessi. Gli automi a stati finiti sono la descrizione di un sistema dinamico che si evolve nel tempo, esiste un parallelo tra gli automi e i calcolatori moderni, per esempio il flusso d'esecuzione di un programma può essere rappresentato attraverso un automa. Alcune applicazioni pratiche di questi automi possono essere, per esempio, regular expression (RegEx o RegExp), lexer e parser ma possono essere impiegati, come vedremo in questa tesi, anche nel campo dei sistemi concorrenti. Si noti che sebbene per gli scopi di questa tesi gli automi a stati finiti siano dei costrutti sufficentemente potenti esistono tuttavia altre classi di automi, espressivamente più potenti, ai quali corrispondo altrettante classi di linguaggi (si veda, per esempio, gli automi a pila) tuttavia gli automi appartenenti a questa classe sono tra i più semplici e immediati.

**Definition 2.1 (Finite State Automata)** *Un automa a stati finiti (FSA) è una tupla* $A = \langle \mathcal{S}, s_0, \mathcal{F}, \mathcal{L}, \delta \rangle$ *dove:*

- $\mathcal{S}$ *è un insieme finito di stati*
- $s_0 \in \mathcal{S}$ *è lo stato iniziale dell'automa*
- $\mathcal{F}$ *è l'insieme degli stati finali o di accettazione* ($\mathcal{F} \subseteq \mathcal{S}$)
- $\mathcal{L}$ *è l'alfabeto finito, talvolta detto anche insieme di label* ($\epsilon \notin \mathcal{L}$)
- $\delta : \mathcal{S} \times (L \cup \{\epsilon\}) \to \mathcal{P}(\mathcal{S})$ *è la funzione di transizione* ($\epsilon$ *denota la stringa vuota*)

**Remark 2.1.1** *Tipicamente è solito trovare una definizione in cui è presente anche l'insieme degli stati di terminazione (o di accettazione) $\mathcal{F}$. In questo caso non è stato definito e pertanto assumiamo che ogni $s \in \mathcal{S}$ sia uno stato di accettazione.*

**Remark 2.1.2** *Va notato anche che questa definizione coincide con quella di automa a stati finiti* non deterministico*, solitamente indicato in letteratura con la sigla NFA (*Non Deterministic Finite Automata*). Una sottoclasse particolarmente rilevante è quella dei DFA (*Deterministic Finite Automata*) che andremo a definire di seguito.*

**Definition 2.2 (Deterministic Finite Automata)** *Un automa a stati finiti deterministico è una tupla $D = \langle \mathcal{S}, s_0, \mathcal{F}, \mathcal{L}, \delta \rangle$ dove $\delta : \mathcal{S} \times L \to \mathcal{S}$*

Le varianti deterministiche si distinguono dalle loro controparti non deterministiche dal fatto che non ammettono né l'utilizzo di $\epsilon$ transizioni, né l'utilizzo di transizioni *uscenti*, dallo stesso stato, con la medesima etichetta. Sebbene queste due varianti siano tra loro equivalenti, l'utilizzo di una variante rispetto all'altra può essere determinato da fattori come: necessità di una maggiore elasticità (gli NFA sono meno stringenti rispetto ai DFA) o di una migliore chiarezza (i DFA sono più immediati e semplici).
In ogni caso è sempre possibile, dato un NFA qualunque, ottenere un DFA ad esso equivalente anche se quest'ultimo spesso ha un numero maggiore di stati rispetto all' NFA di partenza. L'algoritmo che permette di fare questa trasformazione fa uso estensivo di $\epsilon$ *closure*[2] e della funzione *mossa*[2] che andremo a definire di seguito:

**Definition 2.3 ($\epsilon$ closure)** *Fissato un NFA $N = \langle \mathcal{S}, s_0, \mathcal{F}, \mathcal{L}, \delta \rangle$ ed uno stato $s \in \mathcal{S}$ si dice $\epsilon$ closure di s, indicata con $\epsilon$-clos(s), il più piccolo $\mathcal{R} \subseteq \mathcal{S}$ tale che:*

- *$s \in \epsilon$-clos(s)*
- *se $x \in \epsilon$-clos(s) allora $\delta(x, \epsilon) \subseteq \epsilon$-clos(s)*

**Remark 2.3.1** *Se $\mathcal{X}$ è un insieme di stati definiamo $\epsilon$-clos($\mathcal{X}$) come $\bigcup_{x \in \mathcal{X}} \epsilon$-clos(x).*

**Definition 2.4 (Mossa)** *Dato un insieme di stati $\mathcal{X} \subseteq \mathcal{S}$ e un simbolo $\alpha \in \mathcal{L}$ definiamo la funzione* mossa*: $\mathcal{P}(\mathcal{S}) \times \mathcal{L} \longrightarrow \mathcal{P}(\mathcal{S})$ tale che:* $\text{mossa}(\mathcal{X}, \alpha) = \bigcup_{x \in \mathcal{X}} \delta(x, \alpha)$*, ovvero l'insieme di stati raggiungibili da un dato insieme di stati di partenza, leggendo in input $\alpha$.*

L'algoritmo che permette di ricavare un DFA da un qualsiasi NFA è il seguente:

**Algorithm 2.1** Costruzione per sottoinsiemi

```
x ← ϵ-clos(s_0)                                  ▷ Lo stato iniziale del DFA
T ← {x}                                          ▷ Un insieme di ϵ-clos
while ∃ t ∈ T non marcato do
    marca(t)
    for each α ∈ L do
        r ← ϵ-clos(mossa(t, α))
        if r ∉ T then
            T ← T ∪ {r}
        end if
        δ(t, α) ← r          ▷ Denota che la δ del DFA con input t ed α darà output r
    end for
end while
```

Si noti che $x$, $\mathcal{T}$ e $\delta$ saranno rispettivamente lo stato iniziale, l'insieme degli stati e la funzione di transizione del DFA corrispondente, $\mathcal{F}$ sarà invece l'insieme di tutti i $t \in \mathcal{T}$ che al loro interno contengono almeno uno stato finale dell'NFA di partenza mentre $\mathcal{L}$ rimane invariato. Quindi il DFA ottenuto in output sarà $\mathrm{D} = \langle \mathcal{T}, x, \mathcal{F}, \mathcal{L}, \delta \rangle$.

### 2.1.1 Minimizzazione

Nell'ambito della teoria degli automi esistono una serie di operazioni e trasformazioni che è possibile effettuare, per esempio la composizione di più automi, tuttavia nel nostro caso poniamo particolare riguardo alla minimizzazione. Capita spesso infatti che un automa abbia un numero di stati maggiore del necessario e che alcuni di questi stati siano equivalenti tra loro (e dunque duplicati). Attraverso la minimizzazione è possibile *fondere* insieme questi stati tra loro ottenendo infine un automa più snello (in numero di stati e transizioni) e più facile da comprendere. Si noti questo problema degli stati duplicati non sorge solo dalla progettazione umana ma può anche essere un *side effect* di algoritmi come quello di Costruzione per sottoinsiemi mostrato sopra.
L'algoritmo più conosciuto per minimizzare un automa è detto *Algoritmo di Riempimento a Scala*[2] e, di seguito, vedremo il suo funzionamento. Tuttavia occorre fare un'importante premessa prima di introdurre l'algoritmo, il funzionamento dello stesso è legato al fatto che la funzione di transizione $\delta$ sia definita su ogni $\alpha \in \mathcal{L}$, la letteratura distingue gli automi *incompleti*, che non verificano questa condizione, da quelli *completi*. Negli automi incompleti la funzione di transizione è parziale e dunque sorgono dei problemi nel momento in cui cerchiamo di minimizzarli, una soluzione molto semplice è quella di usare uno *stato di errore* (detto anche *stato pozzo*). Essenzialmente si va a completare la funzione di transizione nei casi mancanti (non definiti) con una transizione

verso questo stato d'errore, allo stesso tempo tutte le transizioni uscenti da questo stato di errore tornano sullo stesso ($\forall_{\alpha \in \mathcal{L}} \delta(E, \alpha) = E$) il nome di stato di pozzo deriva infatti dal fatto che una volta raggiunto non è possibile uscirne.
L'intuizione alla base dell'algoritmo di riempimento a scala è la seguente, valutiamo le singole coppie $(p, q)$ con $p, q \in \mathcal{S}$ e cerchiamo un $\alpha \in \mathcal{L}$ tale che lo stato p si comporti diversamente rispetto allo stato q, questo ci permette di dimostrare che p e q non sono equivalenti e dunque non hanno ragione di essere fusi insieme. Alla fine dell'esecuzione tutte le coppie di stati che non saranno distinte tra loro indicheranno degli stati equivalenti.
L'algoritmo di Riempimento della Tabella a Scala è definito come segue:

**Algorithm 2.2** Riempimento della Tabella a Scala

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

### 2.1.2 Prodotto

L'ultima operazione su automi a stati finiti che introduciamo è il *prodotto* tra automi. Solitamente questa operazione viene usata per ricavare, a partire da due o più linguaggi e rispettivi automi, un automa che riconosca l'unione e/o l'intersezione di tali linguaggi.

**Definition 2.5 (Prodotto di automi)** *Siano* $A_1 = \langle \mathcal{S}_1, s_{01}, \mathcal{L}_1, \delta_1 \rangle$ *e* $A_2 = \langle \mathcal{S}_2, s_{02}, \mathcal{L}_2, \delta_2 \rangle$ *due automi a stati finiti, il loro prodotto* $C = \langle \mathcal{S}, s_0, \mathcal{L}, \delta \rangle$ *è definito come segue:*

- $\mathcal{S} = \mathcal{S}_1 \times \mathcal{S}_2$
- $s_0 = (s_{01}, s_{02})$
- $\mathcal{L} = \mathcal{L}_1 \cup \mathcal{L}_2$
- $\delta$ *è invece definita nel seguente modo:*

$$\begin{cases} \delta((s_1, s_2), a) = \{(x, y) | x \in \delta(s_1, a) \wedge y \in \delta(s_2, a)\} & \textit{se } \delta_1 \textit{ e } \delta_2 \textit{ sono definite} \\ \textit{non definito} & \textit{altrimenti} \end{cases}$$

**Remark 2.5.1** *Per quanto riguarda $\mathcal{F}$ si può procedere in due modi diversi: se siamo interessati all'unione dei due linguaggi allora $\mathcal{F} = \{(x, y)|x \in \mathcal{F}_1 \vee y \in \mathcal{F}_2\}$ mentre se siamo interessati all'intersezione di tali linguaggi prenderemo $\mathcal{F} = \{(x, y)|x \in \mathcal{F}_1 \wedge y \in \mathcal{F}_2\}$.*

### 2.1.3 Esempi

Per concludere questa sezione mostriamo di seguito esempi dei vari concetti definiti in precedenza. Il seguente è un NFA N un grado di riconoscere la Regular Expression $(a|b)^*ba$:

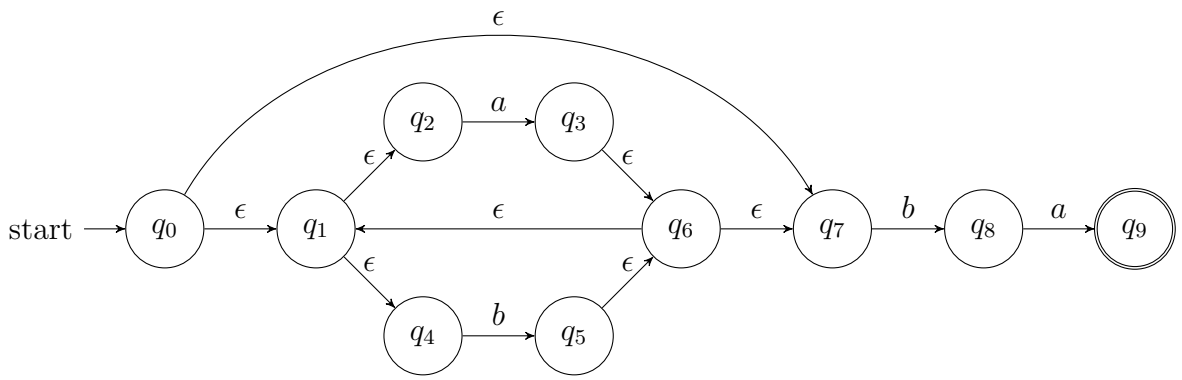
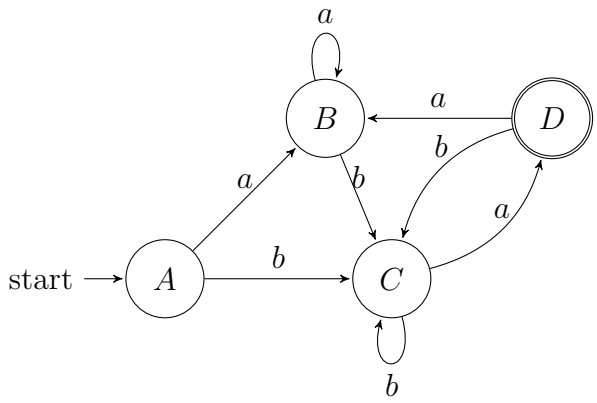



Figura 2.1: Un possibile NFA che riconosce la RegEx $(a|b)^*ba$

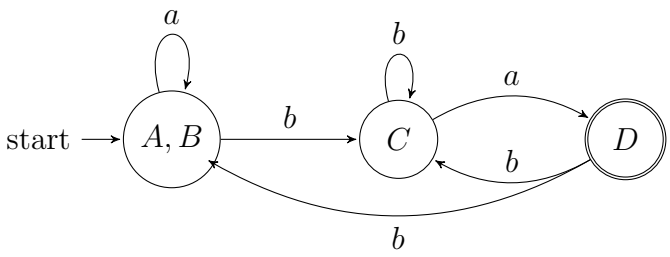
Si noti che questo è solo un *possibile* NFA in grado di riconoscere il linguaggio dato ma ne esistono infiniti altri equivalenti ad esso. Vediamo ora invece il DFA D, equivalente ad N, calcolato tramite l'algoritmo di *Costruzione per sottoinsiemi*




| Stati NFA | Stati DFA |
|---|---|
| { 0,1,2,4,7 } | A |
| { 1,2,3,4,6,7 } | B |
| { 1,2,4,5,6,7,8 } | C |
| { 1,2,3,4,6,7,9 } | D |

Figura 2.2: Il DFA equivalente a quello in figura 2.7

Ora andiamo a minimizzare il DFA ottenuto precedentemente rimuovendo gli stati equivalenti tramite l'algoritmo di *Riempimento della Tabella a Scala*




| | | | |
|---|---|---|---|
| B | | // | // |
| C | $x_1$ | $x_1$ | // |
| D | $x_0$ | $x_0$ | $x_0$ |
| | A | B | C |

Figura 2.3: Il DFA minimizzato ottenuto da quello in figura 2.2

Concludiamo mostrando un esempio di prodotto tra due automi, prendiamo in considerazione i due automi di partenza:

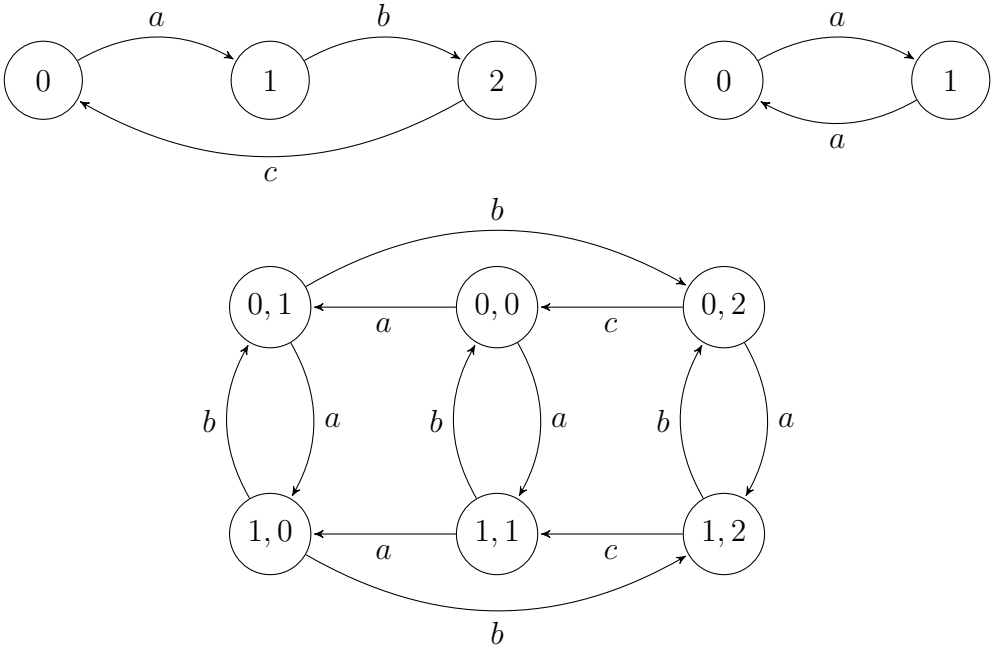



Figura 2.4: Prodotto tra due automi

## 2.2 Choreography Automata

Passiamo ora alla definizione dei *Choreography Automata* (CA); iniziamo diversificando la nozione di *coreografia* e *Choreography Automata*[1] il primo è un modello logico che permette di specificare le interazioni tra più attori (siano essi processi, programmi,

etc.) all'interno di un sistema (concorrente nel nostro caso) mentre i secondi sono invece un'*istanza* possibile per questo modello. In questo caso noi stiamo scegliendo di rappresentare le coreografie tramite degli Automi a Stati Finiti ma questo non esclude altre possibili realizzazioni.
Per prima cosa ricordiamo che le coreografie hanno due tipologie di *view* possibili:

- **Global View**: Che descrive il comportamento dei *partecipanti* "as a whole" specificando anche come questi interagiscono tra loro.
- **Local View**: Che descrive il comportamento di un singolo partecipante in *isolamento* rispetto agli altri.

La *scelta implementativa* di utilizzare gli FSA è dovuta al fatto che gli stessi, oltre ad essere semplici ma espressivi, permettono di utilizzare loop "nested" ed "entangled" e permettono di sfruttare in maniera molto conveniente i risultati e le nozioni descritti in precedenza. I Choreography Automata sono dunque dei *casi particolari* di automi a stati finiti in cui le transizioni specificano le interazioni tra i vari partecipanti della coreografia.
Un esempio di Choreography Automata è visibile nella figura sottostante, la sintassi delle label sulle transizioni è la seguente: *sender*→*receiver*:*message*.

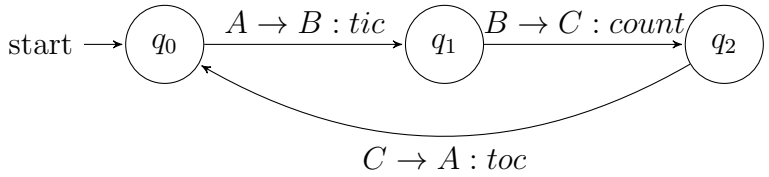



Figura 2.5: Un esempio di Choreogaphy Automata

In questo caso sono rappresentate le interazioni tra gli attori A,B e C, in particolare: A inizia la comunicazione mandando un messaggio *tic* a B, B (dopo aver ricevuto tale messaggio) invia a sua volta *count* a C ed infine C risponde ad A con messaggio *toc*.

**Definition 2.6 (Choreography Automata)** *Un Choreography Automata (c-automata) è un $\epsilon$-free FSA con un insieme di label $\mathcal{L}_{int} = \{A \to B : m | A \neq B \in \mathcal{P}, m \in \mathcal{M}\}$ dove:*

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

**Remark 2.6.1** *Anche se nella definizione non sono ammesse $\epsilon$-transizioni una variante non deterministica rimane sempre possibile, come vedremo anche più avanti in questo lavoro.*

### 2.2.1 CFSM e Local Views

Ora che abbiamo una definizione formale dei Choreography Automata, possiamo concentrarci sull'estrapolazione delle varie view locali a partire dallo stesso. Ricordiamo che le view locali descrivono il comportamento di un singolo partecipante all'interno della coreografia e che sono ottenute attraverso un'operazione di *proiezione* applicata all'intera coreografia (la view globale). Prima di definire però questa operazione di proiezione serve introdurre il concetto di *Communicating Finite-State Machine (CFSM)*. Come il nome suggerisce questo è sempre un modello basato su automi a stati finiti usato specificatamente per la descrizione delle local views. La principale differenza rispetto ai Choreography Automata sta nel fatto che le label sono *direzionali*, ovvero possono essere del tipo "A B ? m" o "A B ! m" per indicare che A riceve (rispettivamente invia) un messaggio m a B.

**Definition 2.7 (Communicating Finite-State Machine)** *Una Communicating Finite State Machine (CFSM) è un FSA C con insieme di labels:*

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

*dove $\mathcal{P}$ e $\mathcal{M}$ sono definiti come in precedenza.*

Dunque il *soggetto* di un'azione in input "A B ? m" è A, lo stesso vale per l'azione di output "A B ! m", indichiamo quindi con $M_a$ la CFSM che ha solo transizioni con soggetto A. Si noti che esiste ed è possibile definire formalmente una funzione *projection* che assegna ad ogni partecipante $p \in \mathcal{P}$ la sua relativa CFSM $M_p$.
Ora che abbiamo introdotto tutti i concetti necessari possiamo definire di seguito l'operazione di *Proiezione* su Choreography Automata.
Definiamo brevemente la notazione $s_1 \xrightarrow{a} s_2$ come abbreviazione per indicare che esiste una transizione da $s_1$ a $s_2$ con label a, formalmente $\exists a \in \mathcal{L}_{act}, s_1, s_2 \in \mathcal{S}.\delta(s_1, a) = s_2$.

**Definition 2.8 (Proiezione)** *La proiezione su A di una transizione $t = s_1 \xrightarrow{a} s_2$ di un Choreography Automata, scritta $t \downarrow_A$ è definita come:*

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

La proiezione di un CA $= \langle \mathcal{S}, s_0, \mathcal{L}_{int}, \delta \rangle$ sul partecipante $p \in \mathcal{P}$, denotata con CA$\downarrow_p$ è ottenuta ricavando in primis l'automa intermedio:

$$A_p = \langle \mathcal{S}, s_0, \mathcal{L}_{act}, \{s \xrightarrow{t\downarrow_p} s' | s \xrightarrow{t} s' \in \delta\} \rangle$$

Tuttavia, come possiamo vedere nella definizione sopra, questo automa intermedio è nondeterministico. è dunque necessario rimuovere le eventuali $\epsilon$ transizioni, ottenendone una versione deterministica e successivamente minimizzare quest'ultima. Entrambe le operazioni sono le medesime definite rispettivamente negli algoritmi 2.1 e 2.2

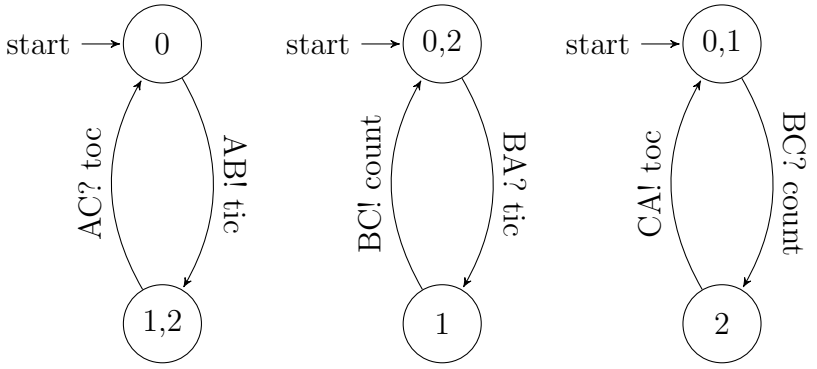



Figura 2.6: Le tre view locali estratte dall'automa in figura 2.5

### 2.2.2 Composizione delle global views

Esiste anche un'operazione opposta alla proiezione, infatti è possible *comporre* più Choreography Automata in uno unico che rappresenti le interazione di tutti gli attori presenti. Questo può essere utile per vari motivi: potremmo avere, per esempio, delle *global views locali* (composte da processi, thread o routine) che talvolta comunicano con altre *global views remote* tramite delle *interfacce* (come endpoint REST, WebSocket o connessioni TCP/IP). Da una situazione come questa può nascere l'esigenza di comporre insieme queste global views in una unica per visualizzare le interazioni (locali e non) che intercorrono tra i vari attori.
Introduciamo dunque l'operazione di *composizione*, in questo caso assumiamo che gli insiemi dei partecipanti delle varie global views di partenza sia disgiunto in questo modo si evitano ambiguità nel risultato finale. La composizione si ottiene concatenando due operazioni:

- **Prodotto** tra tutte le $n$ global views
- **Sincronizzazzione** dell'automa prodotto precedentemente ottenuto

L'operazione di prodotto tra automi è stata definita in 2.5 e rimane pressochè invariata, l'operazione di *sincronizzazione* è invece più particolare: abbiamo appurato che le global views comunicano tra loro attraverso le interfacce, possiamo considerare quest'ultime come partecipanti alla coreografia con il solo ruolo di fare *forwarding* dei messaggi tra

una view e l'altra. Quindi ogni qualvolta che la global view A vorra mandare un messaggio a B, manderà un messaggio all'interfaccia I, lo stesso vale per B quando vorrà ricevere messaggi da A. La sincronizzazzione mira proprio a *rimpiazzare* le interazioni che avvengono tramite interfacce con interazioni tra attori effettivi.

L'operazione di sincronizzazione genera un nuovo automa le cui label sono definite come segue:

$$\mathcal{C}(A \times B) = \begin{cases} p \xrightarrow{A \to B:m} q \text{ se } \exists p \xrightarrow{A \to H:m} q, \exists p \xrightarrow{K \to B:m} q.(A \neq B) \\ p \xrightarrow{A \to B:m} q & \text{se } A, B \in \mathcal{P} \\ \text{nessuna transizione} & \text{altrimenti} \end{cases}$$

Si noti che come step aggiuntivo alla trasformazione di sopra tutti gli stati non raggiungibili da quello iniziale verranno rimossi.
Vediamo di seguito l'esempio di una composizione tra due automi:

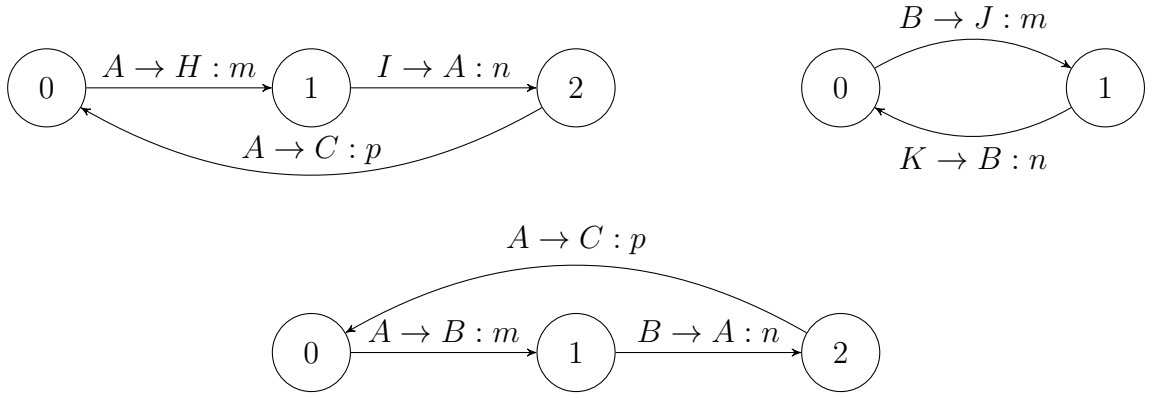



Figura 2.7: Un esempio di composizione tra due Choreography Automata

## 2.3 Analisi statica e dinamica

Ora che abbiamo chiarito le nozioni di base per quanto riguarda la Teoria degli Automi e le Coreografie, passiamo ad un'altro aspetto altrettanto importante per i fini di questi tesi. Considerando che l'obiettivo è quello di ottenere un Choreogaphy Automata partendo da un programma Go dobbiamo determinare in che modo è possibile estrarre delle informazioni da tale programma.
A questo riguardo ricordiamo che un programma può avere due *formati*:

- **Testuale**: ovvero il codice sorgente, un testo scritto in un linguaggio *human readable* con una specifica *grammatica* e specifici *costrutti* che descrivono, ad alto livello, i passi che devono esere intrapresi durante la computazione. Questo formato è quello più utilizzata dagli umani in quanto più facile da comprendere (ed eventualmente modificare), tuttavia non è comprensibile ai calcolatori che, come sappiamo, lavorano con formati binari.

- **Binario**: il codice macchina (o codice eseguibile) generato dal compilatore (come nel nostro caso) o dall'interprete. Questo formato è difficilmente comprensibile da un umano, ma al contrario è perfettamente comprensibile per una macchina tant'è che può essere *eseguito* dalla stessa.

Se consideriamo la definzione di programma come *un insieme di istruzioni per arrivare ad un risultato finale partendo da input forniti* i formati suddetti sono due rappresentazioni equivalenti del medesimo programma e dunque possono essere usati intercambiabilmente e senza alterare la *sostanza* del programma stesso.

Tornando all'estrazione dei dati da un programma esistono due diverse di tecniche, legate al *formato* del programma stesso:

- **Analisi statica**: questo tipo di analisi viene eseguita sul codice sorgente, estrando dei dati dallo stesso ma senza compilarlo ne eseguirlo. Questo tipo di analisi non considera e non è in grado di catturare il *contesto d'esecuzione*, ovvvero i fattori esterni che possono influenzare l'esecuzione di un programma a *runtime*.

- **Analisi dinamica**: questo tipo di analisi invece viene fatta attraverso la *profilazione* del programma mentre lo stesso esegue, il programma è dunque in un formato binario. La profilazione può avvenire attraverso dei log emessi dal programma stesso oppure attraverso l'utilizzo di un'altro programma (detto *tracer*) che controlla le operazioni eseguite dal programma target (il *tracee*)

Entrambe le tecniche presentano i rispettivi vantaggi e svantaggi: l'analisi statica permette una visione più completa in tempo più breve poichè osservando il codice sorgente riesce a catturare tutti i possibili percorsi in cui un programma potrebbe entrare, al contrario l'analisi dinamica non permette di avere sempre una visione completa in quanto è limitata ad osservare solo il percorso che l'esecuzione ha preso in quel momento.
Un esempio di questo comportamento è dato da un semplice costrutto come l'`if-then-else`: tramite l'analisi statica è possibile catturare con facilità entrambi i rami mentre tramite l'analisi dinamica è possibile solo osservare un ramo, quello che a runtime verifica la condizione specificata. Per questo specifico aspetto l'analisi dinamica restituisce dei dati *parziali* e non è possibile fare assunzioni sul ramo che non è stato eseguito, tuttavia l'approssimazione di queste informazioni può essere migliorata eseguendo più profilazioni con input diversi. Si noti però che questo non è sufficente a garantire che le informazioni

siano complete ma solo meglio approssimate e il tempo richiesto per completare l'analisi diventa maggiore (proporzionale rispetto al numero di profilazioni).
In maniera opposta l'analisi statica non riesce a catturare completamente l'evoluzione del programma osservato nel tempo o l'inflluenza che fattori esterni quale il *contesto d'esecuzione* abbiamo sullo stesso, questi aspetti sono invece facilmente osservabili attraverso l'analisi dinamica. Anche in questo caso le informazioni restituite dall'analisi statica sono parziali e vanno approssimate, per esempio usando dei valori predefiniti.

In conclusione, anche se le tecniche mostrate sopra prese singolarmente rappresentano un ottimo strumento per estrarre informazione da un programma, sono fondamentalmente complementari e andrebbero usate in combinazione per ottenre una visione *completa* del programma stesso.

### 2.3.1 Parsing e AST

Appurato che l'analisi statica estrae i dati dal formato *testuale* del programma, serve capire come è possibile ottenre informazioni da del codice sorgente Go. Il *parsing* è l'operazione che permette di trasformare del codice sorgente in una struttura dati appropriata (l'*Abstract Syntax Tree* o AST) dalla quale è poi possibile ricavare informazioni in maniera semplificata rispetto al dover utilizzare e manipolare la stringa iniziale (il contenuto tetsuale del file). Questa operazione non viene solo utilizzata nell'analisi statica ma è anche una fase importante del processo di compilazione (o interpretazione) di qualunque linguaggio di programmazione, il compilatore infatti può utilizzare l'AST per ottimizzare il codice sorgente e, in seguito, per generare il codice binario.
In generale, l'utilizzo di un AST fornisce vari vantaggi: il principale è quello di avere una struttura dati ben definita e gerarchica. Da questo consegue che è possibile navigare l'AST (in maniera molto simile ad una classica *visita* su alberi) e questo permette di estrarre dati in maniera algoritmica dall'AST stesso. Inoltre, sempre grazie alla struttura gerarchica, è possibile definire delle trasformazioni per la stessa o validare che rispetti certe proprietà.

# Capitolo 3

# Tecnologie e librerie utilizzate

## 3.1 Go (golang)

### 3.1.1 Overview

Go[5] (anche chiamato golang) è un linguaggio di programmazione *general purpose* open source sviluppato nel 2007 da Robert Griesemer, Rob Pike e Ken Thompson e poi supportato da Google negli anni a seguire. Fortemente ispirato al C presenta una sintassi minimale e molto semplice, Go è *statically typed* e fornisce un *Garbage Collector* lasciando comunque all'utente la possibilità di interagire con i puntatori e allocare dinamicamente la memoria in modo autonomo.
Alcuni dei problemi che Go mira a risolvere sono

- **Controllo restrittivo delle dipendenze**: Infatti per evitare di appesantire l'eseguibile finale Go rifiuta di compilare moduli o file dove non tutte le dipendenze importate vengono utilizzate
- **Compilazione più veloce**: Grazie a quanto detto sopra e alla sintassi estremamente semplice e snella il compilatore riesce a diminuire drasticamente il tempo richiesto alla compilazione mantenendo tutti i vantaggi dell'avere le eventuali ottimizzazioni a *compile time*
- **Approccio semplificato alla concorrenza**: Il linguaggio utilizza le Goroutine, dei *processi leggeri*, le quali permettono un approccio semplificato ed accessibile alla programmazione concorrente

Altre feature del linguaggio degne di nota sono: il package manager e l'ecosistema di pacchetti totalmente distribuito e decentralizzato, il numero di moduli e librerie disponibili, e la grande varietà di architetture supportate (comprensive di *microcontroller* e *embedded systems*).

Go è stato utilizzato nello sviluppo di tecnologie molto famose e largamente utilizzate come Docker e Kubernetes e attualmente viene regolarmente utilizzato da grandi aziende quali Google, MongoDB, Dropbox, Netflix, Uber e altri.

### 3.1.2 Costrutti di concorrenza

Come accennato sopra Go fornisce un approccio semplificato e built-in alla concorrenza e alla gestione della stessa, il linguaggio permette di avviare dei processi leggeri chiamati Goroutine e scambiare messaggi tra quest'ultimi tramite l'utilizzo di *canali*, i quali permettono sia comunicazione *sincrona* che *asincrona*.
Introduciamo brevemente i principali costrutti di concorrenza messi a disposizione dal linguaggio:

- **Canali**: Go fornisce un tipo di dato built-in `chan` su cui è possibile fare operazioni di *send* e *receive*, i canali possono essere *buffered* e *unbuffered*, i primi permettono una comunicazione asincrona (fino al riempimento del buffer) mentre i secondi permettono solo comunicazione sincrona.

- **Goroutine**: è possibile far partire delle Goroutine anteponendo la keyword `go` ad una qualsiasi function call, questa funzione verrà eseguita in un contesto condiviso (si preservano gli *scope* e le variabili locali) ma parallelo rispetto alla Goroutine che l'ha creato.

- **Select**: Un costrutto particolare che permette eseguire operazioni di invio o ricezione su più canali ed eseguire la prima, tra queste operazioni, che non sia bloccante, oltre a questo è possibile definire anche un blocco da eseguire una volta completata suddetta operazione. Opzionalmente è possibile definire un blocco di default che viene eseguito quando nessuna delle operazioni sopra può essere completata in maniera non bloccante.

Oltre ai costrutti presentati sopra la *standard library* mette a disposizione altri tipi di dato e costrutti *classici* come *Mutex*, *Semafori*, *Monitor* che tuttavia non verranno trattati in questa tesi.

```go
package main

import (
  "fmt"
  "time"
)

func fuzzer(channel chan int, timeout time.Duration) {
  for i := 0; i <= 10; i++ {
    channel <- i
    time.Sleep(timeout * time.Second)
  }

  // From now on sending on this channel will cause an error
  close(channel)
}

func main() {
  // Buffered channel => asynchronous communication
  a := make(chan int, 10)
  // Unbuffered channel => synchronous communication
  b := make(chan int)

  // Starts two "fuzzer" processes
  go fuzzer(a, 4)
  go fuzzer(b, 7)

  for { // Iterates until both channels are closed
    select {
    case data := <-a:
      fmt.Printf("Received from a %d\n", data)
    case data := <-b:
      fmt.Printf("Received from b: %d\n", data)
    default:
      time.Sleep(1 * time.Second)
    }
  }
}
```

Listing 3.1: Esempio di utilizzo dei costrutti di concorrenza forniti da Go

Come possiamo vedere in questo esempio: l'esecuzione parte dalla Goroutine `main` che inizializza i canali `a` e `b` e li passa alle Goroutine `fuzzer`, dopodichè attende di ricevere i vari messaggi da entrambi i canali contemporaneamente fintanto che entrambi non vengano chiusi per poi terminare.

## 3.2 Graphviz e DOT

Vista la necessità di *rappresentare* in qualche modo il Choreography Automata finale e gli eventuali risultati intermedi si è reso necessario l'utilizzo di un qualche tipo di *meccanismo di serializzazione*. Fortunatamente considerando la somiglianza tra Finite State Automata e Grafi (i secondi sono una generalizzazione dei primi) abbiamo potuto riutilizzare tool e strumenti pensati *principalmente* per quest'ultimi.
Abbiamo quindi scelto di usare Graphviz[6], una libreria open source per la visualizzazione di grafi la quale utilizza DOT[4], un formato specificatamente progettato per la descrizione dei grafi.
La scelta è ricaduta su DOT e Graphviz per alcuni motivi principali:

- Il linguaggio DOT è *human readable* e particolarmente facile da comprendere, inoltre Graphviz permette di *convertire* o *esportare* in formati di uso più comune come PNG o SVG
- Permette un utilizzo combinato con *Corinne*[3],un tool grafico per la visualizzazione e manipolazione dei Choreography Automata
- Essendo Graphviz ormai uno standard *de facto* sono presenti librerie e binding che ne permettono l'utilizzo con moltissimi linguaggi di programmazione, tra cui Go

Di seguito un mostriamo un esempio banale di Choreography Automata definito attraverso il linguaggio DOT.

```
1    digraph DOT_Graph_Example {
2        node [shape=circle, fontsize=20]
3        edge [length=100, fontcolor=black]
4
5        q0 -> q1[label="A->B:tic"];
6        q1 -> q2[label="B->C:count"];
7        q2 -> q0[label="C->A:toc"];
8    }
```

Listing 3.2: Rappresentazione in DOT dell'automa in figura 2.5

Chiaramente i Choreography Automata generati da Choreia non saranno così semplici e immediati, ciononostante dovrebbe essere comunque possibile interagirvi e comprenderli.

# Capitolo 4

# Coreografie per Go

## 4.1 Outline

L'obiettivo del progetto, ad alto livello, è quello di prendere del codice sorgente *Go* ed estrarre il Choreography Automata che esprima come le Goroutine interagiscono tra loro durante l'esecuzione del programma, in modo da tale da fornire uno strumento allo sviluppatore per *visualizzare* il sistema concorrente.
Concettualmente possiamo dividere questo obiettivo in 4 fasi:

1. **Validazione e parsing**: Il codice sorgente viene validato e trasformato in un *Abstact Syntax Tree* (AST).

2. **Estrazione dei metadati**: Viene navigato l'AST estraendo tutte le informazioni necessarie (i metadati relativi a funzioni, canali, ecc) e salvandole in strutture dati appropriate.

3. **Derivazione delle local views**: Partendo dai metadati si derivano le local views delle varie Goroutine (gli attori del sistema concorrente).

4. **Generazione della coreografia**: Dalle local view ottenute, è necessario generare un singolo Choreography Automata che rappresenti l'intera Coreografia del sistema (la view globale).

Questo approccio è chiaramente *Bottom-Up* mentre l'approccio delle definizioni nel capitolo 2 è invece *Top-Down*, abbiamo visto infatti come sia possibile, partendo dal Choreography Automata ricavare le singole view locali attraverso l'operazione di *Proiezione*. Come accennato sopra si rende necessaria l'implementazione di un'operazione opposta alla proiezione, chiamata *riconciliazione* che permetta di ottenere un view globale a partire dalle sue singole componenti, ovvero le view locali.
Il punto cruciale della nostra tecnica è quello di *approssimare* ogni funzione trovata

all'interno del codice sorgente in un automa a stati finiti che ne rappresenti il flusso d'esecuzione. In particolare ogni transizione di questo automa rappresenterà un'interazione tra la funzione stessa e il resto della Coreografia.

Per ragioni di chiarezza e semplicità questa tesi si concentra su un sottoinsieme di Go limitato all'utilizzo di canali, iterazione determinata sugli stessi, select statement, canali passati come parametro tra funzioni e creazione di nuove Goroutine. Altre *feature* del linguaggio come *selector expression*, *anonymous functions*, *high order function* o iterazioni su liste e mappe non sono attualmente supportare e potrebbero dunque causare delle inconsistenze durante l'esecuzione o nel Choreography Automata finale.
Alcuni esempi di programmi che non sono supportati includono: programmi utilizzino ricorsione o spawn di Goroutine in ricorsione, programmi che utilizzano la riassegnazione dei canali (i canali sono assunti essere *immutable*) o programmi che utilizzano collection (liste, mappe o altri tipi di strutture) contenenti canali o iterano sulle stesse.

### 4.1.1 Peculiarità di Go

Per gli scopi di questa tesi è bene considerare le particolarità di Go in modo da adattare il modello teorico allo stesso e *risolvere* le eventuali incongruenze.

Mentre aspetti tipici di Go come i canali e il costrutto `select` non generano particolari conflitti con il modello teorico lo stesso non si può dire per le Goroutine: quest'ultime sono intrinsicamente *gerarchiche*, ovvero per ogni programma Go viene avviata sempre e solo una Goroutine (quella che esegue la funzione `main` e che si comporta come *entry point* del programma stesso) sarà poi questa, durante la sua esecuzione, ad avviarne altre, quest'ultime a loro volta potranno avviarne altre ancora e cosi via. Il problema deriva dal fatto che nella definizione di Choreography Automata si assume in qualche modo che tutti i partecipanti siano già avviati e pronti a comunicare tra loro mentre per i nostri scopi servirebbe invece sapere qaundo e da chi è stata avviata una Goroutine in modo da poter definire quando la sua *local view* diventa rilevante nel contesto globale, senza questo ulteriore controllo si potrebbero verificare delle inconsistenze.

Per fare questo possiamo estendere la definizione di Choreography Automata e di Communicating Finite-State Machine date riespettivamente in 2.6 e 2.7 come segue:

**Definition 4.1 (Choreography Automata - Estesa)** *Un Choreography Automata (c-automata) è un $\epsilon$-free FSA con un insieme di label:*

$$\mathcal{L}_{ext} = \mathcal{L}_{int} \cup \{A \triangle B | A, B \in \mathcal{P}\}$$

*con $\mathcal{L}_{int}$ e $\mathcal{P}$ definiti come in 2.6 e $\mathcal{M}$ definito come in 4.2.1*

**Definition 4.2 (Communicating Finite-State Machine - Estesa)** *Una Communicating Finite State Machine (CFSM) è un FSA $C$ con insieme di labels:*

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

**Remark 4.2.1** *Seppur non interessante per gli scopi di questa tesi è possibile adattare la nozione di proiezione in modo che tenga in considerazione di transizione del tipo $A \triangle B$ con $A, B \in \mathcal{P}$*

## 4.2 Estrazione dei metadati

Una volta ottenuto un AST valido, serve estrarre i metadati necessari da quest'ultimo. Per gli scopi di questa tesi siamo interessati ad estrarre informazioni sui canali, per esempio: il nome della variabile associata, il *tipo* di messaggi che possono essere scambiati sullo stesso e la tipologia del canale (*buffered* o *unbuffered*) con particolare distinzione tra canali dichiarati nello scope globale e quelli dichiaratie in uno scope locale. Inoltre siamo interessati ad estrarre metadati dalle *function declarations* come: il nome della funzione, canali dichiarati nello scope della stessa e un FSA che rappresenti il flusso d'esecuzione all'interno della funzione stessa (detto Scope Automata). Inoltre vogliamo memorizzare eventuali *parammetri formali* della funzione che richiedono trattamenti particolari come canali o *callback functions*, infatti i canali passati come parametro dal chiamante dovranno poi essere *sostituiti* nel momento in cui la funzione viene chiamata o avviata come Goroutine da un'altra funzione.

Questa estrazione dei metadati avviene tramite analisi statica, una tecnica di analisi descritta al capitolo 2. Questa tecnica è preferibile rispetto all'analisi dinamica poichè non richiede alcun tipo di esecuzione e quindi protegge da programmi potenzialmente pericolosi e sconosciuti, evita lo scaricamento di eventuali dipendenze per il programma in input e in generale fornisce una visione più ampia e meglio approssimata per i nostri scopi senza richiedere profilazioni multiple. Infatti utilizzando analisi dinamica per questo progetto otterremmo per ogni profilazione eseguita solo un sottografo del Choreography Automata finale che rappresenta il particolare percorso intrapreso dalla Coreografia durante quella esecuzione e non l'intera Coreografia.

### 4.2.1 Limiti dell'analisi statica

Anche per questa fase sorge un'inconsistenza con la definizione formale di Choreography Automata data in precedenza: l'insieme $\mathcal{M}$ dei messagi non è determinabile in maniera precisa attraverso l'analisi statica. La definizione 2.6 sembra suggerire che esista un numero finito e definito di messaggi scambiabili tra i vari attori tuttavia questo insieme non è calcolabile con l'approccio utilizzato. Ricordiamo infatti che questo tipo di

analisi viene effettuata utilizzando solo il codice sorgente e ricavando dei dati senza mai eseguire il codice, in generale non è possibile solo attraverso l'analisi statica ricavare il valore esatto di tutte le variabili (e dunque tutti i messaggi inviati), questo perchè tale valore può essere soggetto a svariati *side effect* durante l'esecuzione o può essere legato a parametri temporali (p.e. timestamp), input forniti dall'utente o altri valori ricavabili solo a runtime. Questi *aspetti* non sono *catturabili* attraverso l'analisi statica e dunque devono essere gestiti in maniera oppurtuna.

L'esempio sottostante mostra un caso di possibile di codice sorgente Go in cui l'analisi statica non riesce a catturare i valori effettivi dei vari messaggi scambiati tra i processi:

```
1  package main
2
3  import (
4    "fmt"
5    "math/rand"
6    "time"
7  )
8
9  type payload struct {
10   data      int
11   timestamp int64
12 }
13
14 func worker(incoming chan int, outgoing chan payload) {
15   for msg := range incoming {
16     // Sends back the results on the out channel
17     outgoing <- payload{msg + 1, time.Now().Unix()}
18   }
19 }
20
21 func main() {
22   // Creates the channels
23   in, out := make(chan int, 10), make(chan payload, 10)
24   // Starts the worker processes
25   go worker(in, out)
26   go worker(in, out)
27   // Infinite loop
28   for {
29     in <- rand.Int()
30     res := <-out
31     fmt.Printf("Received %d at %d \n", res.data, res.timestamp)
32   }
33 }
```

Listing 4.1: Codice sorgente per cui non è possibile calcolare $\mathcal{M}$ tramite analisi statica

Come possiamo vedere il programma mostrato è in realtà alquanto banale: la Goroutine `main` genera un intero random che poi invia su un canale precedentemente condiviso con le due Goroutine `worker`, uno dei due processi riceverà questo intero lo incrementerà per poi lo inviarlo nuovamente con un timestamp aggiuntivo. Attraverso l'analisi statica non solo non riusciamo a determinare il valore inviato sul canale `in` nè entrambi i valori inviati sul canale `out`.

La soluzione da noi adottata è in realtà molto semplice e permette di mantenere una sufficente espressività del modello: generalizziamo $\mathcal{M}$ all'*insieme dei tipi* dei messaggi scambiati, i tipi infatti possono essere inferiti e ricavati senza particolari problemi per mezzo di analisi statica. Nel caso della figura sopra $\mathcal{M}$ sarà definito come: $\mathcal{M} = \{int, payload\}$ e le label nel Choreography Automata associato saranno del tipo $main \xrightarrow{int} worker$ oppure $worker \xrightarrow{payload} main$.

## 4.3 Derivazione delle local views

Una volta ottenuti tutti i metadati necessari sorge la necessità di derivare dagli stessi le local views di ogni Gorountine creata durante l'esecuzione dell'intero programma, da queste poi potremmo procedere alla costruzione della global view.

Fortunamente in questo fase l'esistenza di una gerarchia *intrinseca* nelle Goroutine ci è di aiuto: sappiamo che inizialmente esiste solo una Goroutine, quella che esegue il `main`, e sarà questa ad avviare le altre *in cascata*.

**Algorithm 4.3** Derivazione delle local views

```
grSet ← {main}
while ∃ gr ∈ grSet non marcato do                ▷ Per ogni Goroutine trovata
    for each t ∈ gr do          ▷ Per ogni transizione nell'FSA di questa Goroutine
        if t.kind = Call then
            espande i parametri formali con quelli attuali
            inline del chiamato all'interno del chiamante
        else if t.kind = Spawn then
            espande i parametri formali con quelli attuali
            grSet ← grSet ∪ {t.target}
        end if
    end for
    marca gr
end while
```

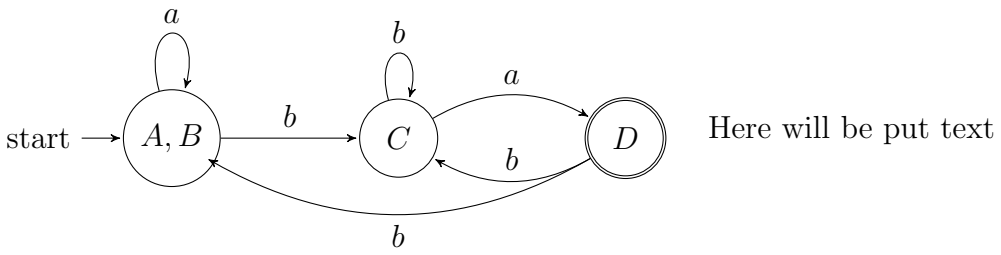



Figura 4.1: La local views per i processoi `main` e `worker` visti in 4.1

## 4.4 Generazione della coreografia

TODO

# Capitolo 5

# Choreia

Choreia è il tool sviluppato come progetto per questa tesi. Il tool si occupa di tutte le fasi descritte al capitolo descritte precedentemente e consente all'utente finale di esportare il Choreogaphy Automata ricavato dal sorgente in formato DOT. Choreia è un software libero con licenza GPL-3.0 scritto completamente in Go, non richiede alcun tipo di setup se non l'installazione iniziale delle dipendenze. è disponibile al download al seguente url: `https://github.com/its-hmny/Choreia`
Il nome *Choreia* deriva dalla medesima parola greca da cui deriva Coreografia parola composta da *choreia*, "danza" e *graphè*, scrittura.

## 5.1 Struttura del progetto

## 5.2 Parametri da linea di comando

Il tool non ha una GUI in quanto, almeno per gli scopi attuali, non è necessaria: infatti non è stato progettato come un tool di uso comune ma come uno strumento per persone interessate e con un minimo di conoscenza pregressa.
Tuttavia è possibile tramite *command line* fornire alcuni parametri e flags per un utilizzo *personalizzato*, di seguito troviamo una spiegazone di ognuno di essi:

| Breve | Esteso | Descrizione |
|---|---|---|
| -i | –input | Il *path* del file .go in input |
| -o | –output | La directory in cui verrano salvati i file i vasri automi |
| -t | –trace | Stampa l'AST sullo `stdout` |
| -h | –help | Mostra un messaggio di aiuto con una breve spiegazione |

Tabella 5.1: La lista di argomenti da linea di comando accettati da Choreia

## 5.3 Flusso d'esecuzione

## 5.4 Esempi pratici

# Capitolo 6

# Conclusioni e lavori futuri

TODO

# Bibliografia


[1] Franco Barbanera, Ivan Lanese e Emilio Tuosto. «Choreography automata». In: (2020), pp. 1–106.

[2] S. Martini M. Gabbrielli. *Linguaggi di programmazione. Principi e paradigmi*. Collana di istruzione scientifica. McGraw-Hill, 2001. ISBN: 8838665737.

[3] Simone Orlando et al. «Corinne, a Tool for Choreography Automata». In: (2021), pp. 1–92.

[4] Wikipedia. *DOT*. Online; Accessed 20-December-2021. 2021. URL: `https://en.wikipedia.org/wiki/DOT_(graph_description_language)`.

[5] Wikipedia. *Go (programming language)*. Online; Accessed 18-December-2021. 2021. URL: `https://en.wikipedia.org/wiki/Go_(programming_language)`.

[6] Wikipedia. *Graphviz*. Online; Accessed 20-December-2021. 2021. URL: `https://en.wikipedia.org/wiki/Graphviz`.