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 ultimi anni. Lo scopo principale è quello di fornire al programmatore uno strumento che permetta di capire in maniera immediata la *coreografia* dei partecipanti all'interno del sistema e come questi interagiscono 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 coreografie 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 coreografie, 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

Negli ultimi anni si è visto un progressivo aumento nell'utilizzo di *sistemi distribuiti* e delle cosidette *architetture orientate ai microservizi* così come un rinnovato interesse verso lo sviluppo concorrente.
Questo nuovo interesse ha portato cambiamenti nei linguaggi di programmazione e nel *language design*, si è cercato infatti di venire incontro alle richieste dei programmatori facilitando l'approccio del linguaggio alla concorrenza. Ricordiamo infatti che alcuni linguaggi non offrivano funzionalità native o talvolta le API fornite erano troppo complesse ed intricate per garantirne un utilizzo chiaro e, più in generale, la concorrenza era un *afterthought* durante la progettazione di un linguaggio. Alcuni esempi pratici di questo rinnovato *focus* verso la concorrenza possono essere visti in nuovi linguaggi di programmazione come Go[12] ed Elixir[10] che offrono un'approccio semplificato e più chiaro alla concorrenza ma anche in linguaggi più consolidati come C++[8], che con lo standard 20 ha introdotto costrutti che facilitano l'utilizzo rispetto alle implementazioni precedenti.

Tuttavia è estremamente complesso sviluppare un sistema concorrente (o distribuito) esente da errori, allo stesso modo si presentano varie problematiche legate al *testing* e *debugging* dello stesso, in entrambi i casi si fatica a riprodurre il bug o a testare tutte le casiste possibili. Questo aspetto di difficoltà è intrinsicamente legato alla natura non deterministica di un sistema concorrente che rende difficile riprodurre una definita situazione. Inoltre questi problemi di riproducibilità e validazione crescono esponenzialmente in difficoltà al crescere della complessità del sistema stesso.

Da queste probllematiche nasce dunque la necessità di avere strumenti che assistano lo sviluppatore, per questo motivo parallelamente al trend sopra descritto, sono stati proposti nuovi *paradigmi* e *modelli teorici* che mirano ad esemplificare la scrittura di programmi concorrenti, tra questi troviamo le *coreografie*[4].
L'idea alla base delle coreografie è quella di definire un metodo standard per rappresentare le interazioni tra due o più processi durante l'esecuzione in un sistema concorrente.

Le coreografie forniscono al programmatore la possibilità di osservare e analizzare le interazioni che avvengono tra i partecipanti in fase di debug e, allo stesso modo, possono fornire uno strumento per la descrizione delle stesse durante la fase progettuale.

Tipicamente nelle coreografie i proocessi comunicano tra loro attraverso *message passing*, avremo quindi una primitiva del tipo:

$$Alice \rightarrow Bob : message$$

per indicare che Alice sta inviando un messaggio *message* a Bob. La caratteristica che distingue le coreografie dagli altri paradigmi consiste nel fatto che quest'ultime forniscono 2 tipi di *view* diverse:

- **View globale**: la descrizione dal punto di visto *olistico* dell'intero sistema
- **View locale**: la descrizione di un singolo componente *in isolamento*

È sempre possibile, partendo dalla global view, ottenere una sua local view attraverso l'operazione di *proiezione* ed è anche possibile unire più global view insieme mediante l'operazione di *composizione*[2]. Oltre a queste operazioni una coreografia che soddisfa certe proprietà ed è dunque *ben formata* permette di fare assunzioni riguardo all'assenza dei classici problemi di programmazione concorrente come *deadlocks*, *race conditions* e *liveness* fornendo al programmatore uno strumento di verifica per il suo codice.

In questa tesi viene presentato *Choreia*, un tool di analisi statica che permette di ricavare da del codice sorgente Go il Choregraphy Automata ad esso associato. I Choreography Automata sono un *modello* per descrivere le coreografie basato su *automi a stati finiti (FSA)*, un particolare vantaggio dei Choreography Automata è la possibilità di poter riutilizzare tutte le nozioni teoriche già esistente in letteratura sugli automi a stati finiti. Oltre ad una descrizioni approfondita del tool e delle nozioni alla base dello stesso verrano introdotte anche le *Communicating Finite State Machine* per la descrizione delle local views e gli algoritmi di proiezione e composizione. Sempre in questa tesi verrà presentato un algoritmo per la composizione di multiple local views in una unica global view, andando difatti a definire un'operazione inversa alla proiezione, quest'ultima sarà direttamente derivata dall'algoritmo di composizone tra global views.

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)[6] e alcune operazioni possibili sugli stessi. Gli automi a stati finiti sono la descrizione di un sistema dinamico che 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 sufficientemente potenti esistono tuttavia altre classi di automi, espressivamente più potenti, ai quali corrispondono altrettante classi di linguaggi (si veda, per esempio, gli automi a pila). Tuttavia gli automi a stati finiti 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* (ϵ *denota la stringa vuota*)

Remark 2.1.1 *È possibile trovare una definizione in cui non è presente l'insieme degli stati di terminazione (o di accettazione)* $\mathcal{F}$*. In tal caso 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.*

Il seguente è un NFA N un grado di riconoscere la Regular Expression $(a|b)^*ba$:

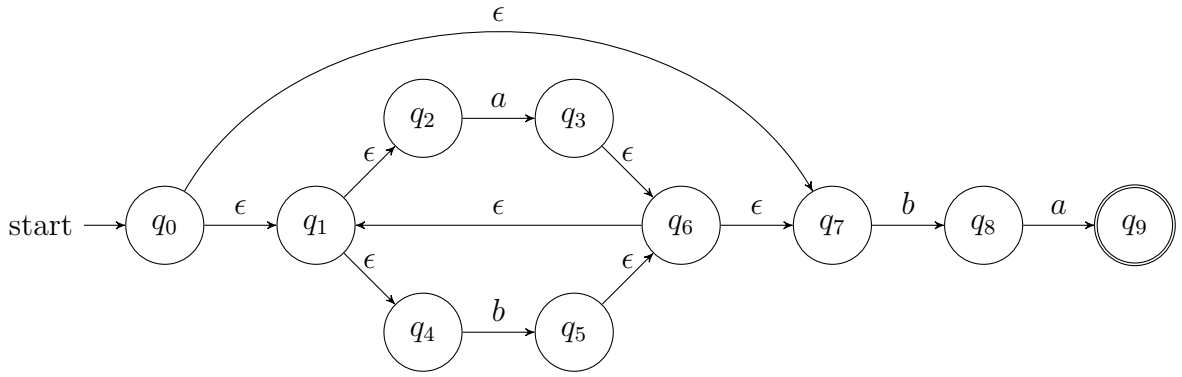


Figura 2.1: Un possibile NFA che riconosce la RegEx $(a|b)^*ba$

Si noti che questo è solo un *possibile* NFA in grado di riconoscere il linguaggio dato ma ne esistono infiniti altri equivalenti ad esso.

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 ϵ 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 ϵ *closure*[6] e della funzione *mossa*[6] che andremo a definire di seguito:

Definition 2.3 (ϵ 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* ϵ *closure di s, indicata con* ϵ*-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 ϵ-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 α.*

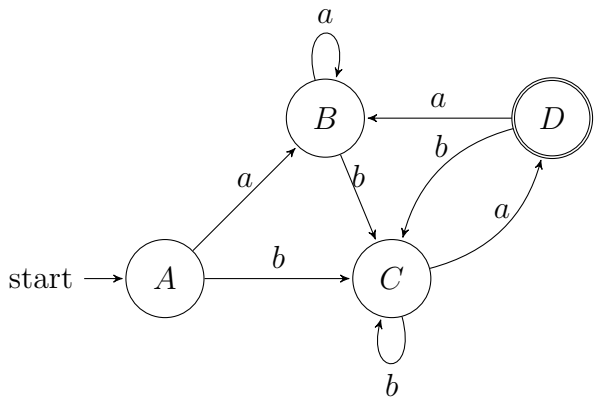
L'algoritmo che permette di ricavare un DFA da un qualsiasi NFA è il seguente:

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 δ 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 $\text{D} = \langle \mathcal{T}, x, \mathcal{F}, \mathcal{L}, \delta \rangle$.

L'automa presentato di seguito riconosce sempre la RegEx $(a|b)^*ba$ ed è del tutto equivalente a quello in figura 2.1, tuttavia è stato ottenuto proprio da quest'ultimo 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 2.1

2.1.1 Minimizzazione

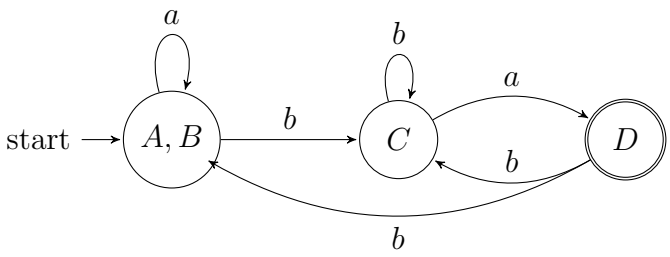
Nell'ambito della teoria degli automi esistono una serie di operazioni e trasformazioni che è possibile effettuare, per esempio la composizione di più automi, tuttavia nel nostro caso poniamo particolare riguardo alla minimizzazione. Capita spesso infatti che un automa abbia un numero di stati maggiore 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*[6] 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 δ 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 di 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 equiva-

lenti.
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 con marca x_0 le coppie (m, n) con $m \in \mathcal{F}$ e $n \notin \mathcal{F}$
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 (p, q) non è marcata $\wedge\ \delta(p, \alpha) \neq \delta(q, \alpha)$ **then**
    Marca (p, q) con marca x_i
  end if
  Considera all'iterazione seguente solo gli stati non marcati
end while

Di seguito troviamo la versione *minimizzata* dell'automa in figura 2.1 con l'algoritmo di *Riempimento della Tabella a Scala*. Anche questo automa riconosce sempre la RegEx $(a|b)^*ba$.



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 2.1

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

- $\mathcal{S} = \mathcal{S}_1 \times \mathcal{S}_2$
- $s_0 = (s_{01}, s_{02})$
- $\mathcal{F} = \{(s_1, s_2) | s_1 \in \mathcal{F}_1 \wedge s_2 \in \mathcal{F}_2\}$

- $\mathcal{L} = \mathcal{L}_1 \cap \mathcal{L}_2$
- δ *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 *Nella definizione data sopra abbiamo definito l'automa prodotto che riconosce l'*intersezione *dei linguaggi riconosciuti dai due automi di partenza. Tuttavia è possibile anche definire l'automa prodotto che riconosce l'*unione *dei due linguaggi invertendo oppurtunamente le varie congiunzioni (insimistiche e logiche) con delle disgiunzioni.*

Per concludere mostriamo un esempio di prodotto tra due automi: i primi due automi in figura sono quelli di partenza mentre il terzo è l'automa prodotto.

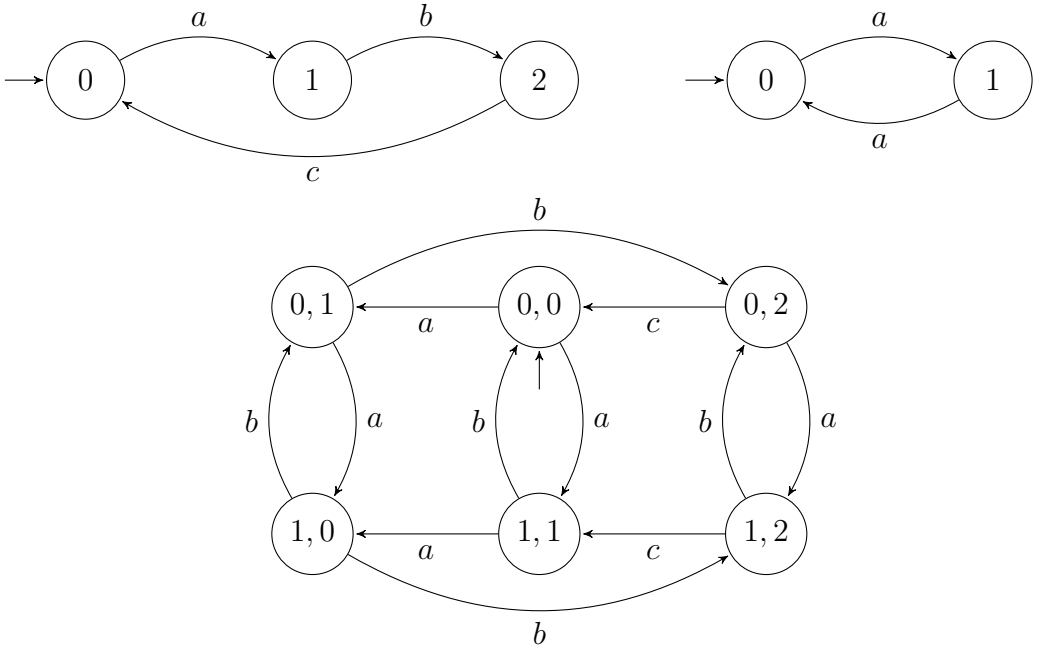


Figura 2.4: Prodotto tra due automi

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*.

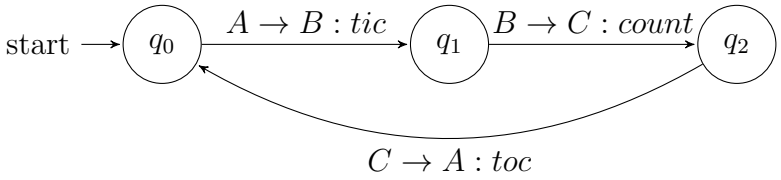


Figura 2.5: Un esempio di Choreography Automata

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 ϵ-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 ϵ-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)*[3]. 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 B, l'opposto vale per l'azione di output "A B ! m", indichiamo quindi con M_a la CFSM che ha solo transizioni con soggetto A.
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:*

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

Remark 2.8.1 *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.*

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 non deterministico. è dunque necessario rimuovere le eventuali ϵ 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

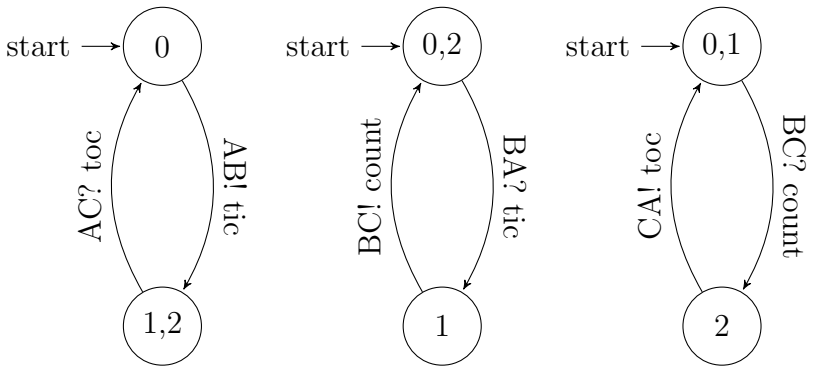


Figura 2.6: Le tre view locali estratte dall'automa in figura 2.2

2.2.2 Composizione delle global views

Esiste anche un'operazione opposta alla proiezione, infatti è possibile *comporre*[2] più Choreography Automata in uno unico che rappresenti le interazioni 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
- **Sincronizzazione** 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 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 la global view A vorrà mandare un messaggio a B, manderà un messaggio all'interfaccia I, lo stesso vale per B quando vorrà ricevere messaggi da A. La Sincronizzazione 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:

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

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

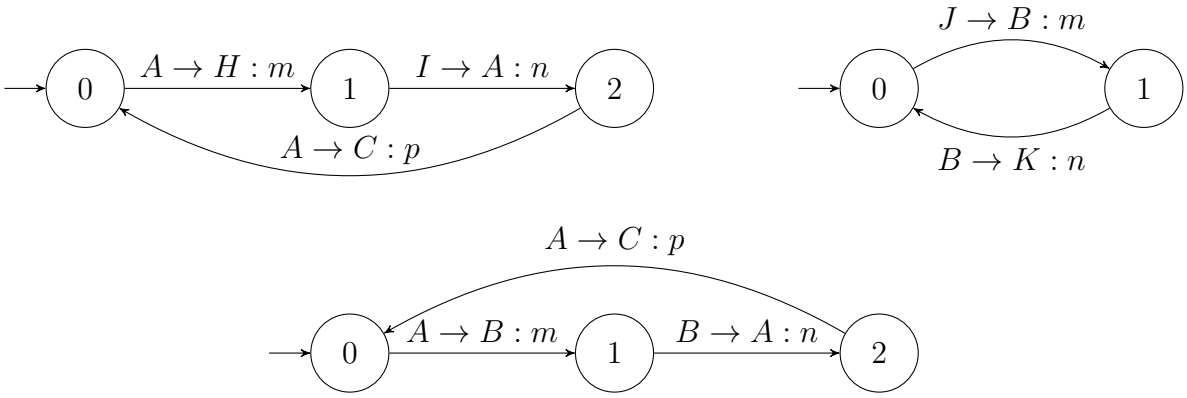


Figura 2.7: Un esempio di composizione tra due Choreography Automata

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 questa tesi. Considerando che l'obiettivo è quello di ottenere un Choreography 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*:

- **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 essere intrapresi durante la computazione. Questo formato è quello più utilizzato dagli esseri umani in quanto più facile da comprendere (ed eventualmente modificare), tuttavia non è comprensibile ai calcolatori che, come sappiamo, lavorano con formati binari.

- **Codice binario**: detto anche codice macchina o codice eseguibile generato dal compilatore. 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 definizione 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**[5]: questo tipo di analisi viene eseguita sul codice sorgente, estraendo dei dati dallo stesso ma senza compilarlo nè eseguirlo. Questo tipo di analisi non considera e non è in grado di catturare il *contesto d'esecuzione*, ovvero i fattori esterni che possono influenzare l'esecuzione di un programma a *runtime*.

- **Analisi dinamica**[16]: 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 sufficiente 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'influenza che fattori esterni quale il *contesto d'esecuzione* abbiano 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 ottenere 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 ottenere informazioni dal 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[11]) dalla quale è poi possibile ricavare informazioni in maniera semplificata rispetto al dover utilizzare e manipolare la stringa iniziale (il contenuto testuale 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 verificare che rispetti certe proprietà.

Capitolo 3

Tecnologie e librerie utilizzate

3.1 Go (golang)

3.1.1 Overview

Go[12] (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[9] e Kubernetes[15] 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 di 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 verrànno 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, mentre le due nuove Goroutine inviano 10 messaggi ciascuna (con un timeout tra un invio e l'altro), la Goroutine `main` attende i vari messaggi tramite la `select` su entrambi i canali, questo significa che la Goroutine `main` eseguirà la prima operazione *non bloccante* oppure il `default` branch se nessun canale ha un messaggio in coda. Il for loop terminerà solamente

quando entrambi i canali saranno chiusi dalle rispettive Goroutine `worker` con l'apposita primitiva `close`.

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[14], una libreria open source per la visualizzazione di grafi la quale utilizza DOT[13], 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*[7],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 2.2

Il seguente esempio definisce un grafo direzionato con tre nodi (q0, q1, e q2) e altrettanti archi, rispettivamente da q0 a q1, da q1 a q2 e da q2 a q0 con le rispettive label. DOT fornisce anche la possibilità all'utente di definire prima tutti i nodi che fanno parte del grafo e poi tutti gli archi.
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 allo sviluppatore uno strumento 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*.
Si rende dunque necessaria l'implementazione di un'operazione opposta alla proiezione che permetta di ottenere una view globale a partire dalle sue singole componenti, ovvero le view locali, questo tipo di operazione riprenderà alcuni concetti dall'operazione

di *composizione* (illustrata precedentemente) oppurtunamente adattati alla nostra situazione.
Il punto cruciale della nostra tecnica è quello di *approssimare* ogni funzione dichiarata all'interno del codice sorgente in un automa a stati finiti che 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 argomento 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 nel Choreography Automata finale.
Alcuni esempi di programmi che non sono supportati includono: programmi che utilizzano ricorsione o spawn di Goroutine in ricorsione e 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 intrinsecamente *gerarchiche*: 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 così via. Il *conflitto* con il modello teorico 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 tenere traccia del momento in cui una Goroutine viene avviata 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 esempio potremmo avere interazioni tra Goroutine prima ancora che queste siano avviate.

Estendiamo dunque le nozioni di Choreography Automata e di Communicating Finite-State Machine date rispettivamente nelle definizioni 2.6 e 2.7 come segue:

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

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

con $\mathcal{L}_{int}$, $\mathcal{P}$ e $\mathcal{M}$ definiti come nella definizione 2.6

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 dichiarati all'interno del programma, in particolare: 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 dichiarati 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). Vogliamo anche memorizzare eventuali *parametri formali* della funzione che richiedono trattamenti particolari come canali o *callback functions*, infatti i canali passati come parametro dal chiamante dovranno poi essere *sostituiti* con i parametri formali nel momento in cui la funzione viene chiamata o avviata come Goroutine da un'altra.

Questa estrazione dei metadati avviene tramite analisi statica, una tecnica descritta nel 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. 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/profilazione 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 messaggi non è determinabile in modo preciso 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 opportuna.

L'esempio sottostante mostra un caso di possibile 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 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 ma permette di mantenere una sufficiente 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 $worker \leftarrow main : int$ oppure $main \leftarrow worker : payload$.

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 del programma, da queste poi potremmo procedere alla costruzione della global view.

Partiamo definendo la struttura dell'automa che verrà prodotto durante queste fase, rispetto alla fase precedente dove l'automa è associato ad una singola funzione qui l'automa ottenuto sarà associato ad un intera Goroutine. Per i nostri scopi non siamo interessati all'intero flusso d'esecuzione bensì solo alle interazioni con i canali, allo *spawn* di nuove Goroutine e alle *function call* effettuate.

Definition 4.3 (Goroutine FSA) *Un Goroutine FSA è un fsa G con insieme di labels:*

$$\mathcal{L} = \{\leftarrow c, \rightarrow c, \triangle f | c \text{ è un canale ed } f \text{ è una funzione}\}$$

le label indicano rispettivamente la ricezione da un canale, l'invio su un canale e lo spawn di una nuova Goroutine con entrypoint f

Un pattern tipico di Go è quello di avviare Goroutine passando alle stesse il canale/i su cui comunicare, questo comporta un meccanismo di sostituzione dei *parametri formali* con i *parametri attuali*.
Un altro aspetto da tenere in considerazione riguarda le chiamate di funzione effettuate, in questo caso per avere una migliore approssimazione serve che l'automa associato al *chiamato* venga *fuso* all'automa del chiamante. Questo tipo di operazione è simile all'*inlining* un'ottimizzazione usata dai compilatori per evitare l'overhead della *function call* quando possibile, esattamente come nell'inlining sostituiamo la transizione con la chiamata di funzione con l'automa associato alla funzione chiamata. In questo modo otteniamo un automa che rappresenta il flusso d'esecuzione delle Goroutine tenendo in considerazione anche le eventuali function call fatte.

Vediamo ora l'algoritmo per l'estrazione delle local views. In questo caso l'esistenza di una gerarchia *intrinseca* nelle Goroutine ci è di aiuto poichè ci permette di assumere un input iniziale per l'algoritmo, ovvero la Goroutine che esegue la funzione `main`.

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
```

Applicando l'algoritmo di sopra al codice sorgente in figura 4.1 otteniamo i seguenti automi rispettivamente per le Goroutine $main_0$, $worker_1$ e $worker_2$:

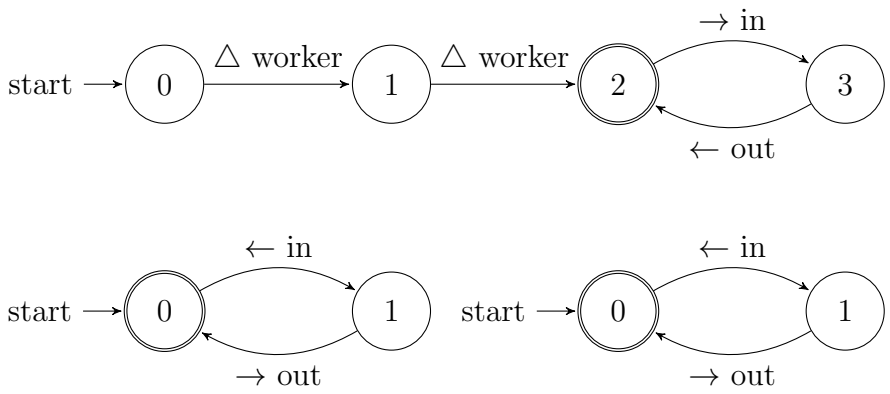


Figura 4.1: La local views per i processi `main` e `worker` visti in 4.1

4.4 Generazione della coreografia

Ottenuti gli automi associati alle singole Goroutine, che ricordiamo essere nel nostro caso le local views, ci si presenta il problema di capire come *fondere* questi in un unico automa che rappresenti la coreografia e rispetti la definizione data in 4.1.

Il problema che si presenta in questa fase è il seguente: in Go un canale può essere condiviso tra n Goroutine ed ognuna di queste può inviare e ricevere a suo piacimento sullo stesso. Prendendo per esempio la figura precedente i dati inviati dalla Goroutine $main_0$ sul canale `in` possono essere ricevuti o da $worker_0$, o da $worker_1$ e non ci è possibile determinare in esattamente quale delle due Goroutine sia il destinatario effettivo di uno specifico messaggio. Inoltre la complessità del problema peggiora se generalizziamo ad un caso con n sender ed m receiver o, peggio ancora, con n attori che eseguono sia operazione di send che di receive sullo stesso canale condiviso (questo caso apre anche alla possibilità che una Goroutine riceva un messaggio inviato precedentemente da lei stessa).

La nostra soluzione considera i canali come delle *interfacce*, quindi tutte le transizioni con operazioni su canali all'interno di un Goroutine FSA sono considerate come interazioni tra un attore (la Goroutine) e un'interfaccia (il canale). Nel capitolo 2 però abbiamo già definito l'operazione di *composizione* che permette di unire molteplici Choreography Automata, comunicanti tra loro mediante interfacce, in uno unico.
Adattiamo dunque la nozione di composizione ai Goroutine FSA, anzichè ai Choreography Automata: l'operazione di prodotto tra automi rimane invariata rispetto a quanto detto nella definizione 2.5 mentre definiamo ora l'operazione di sincronizzazione come

segue:

$$\mathcal{S}(A \times B) = \begin{cases} p \xrightarrow{A \to B:m} q \text{ se } \exists p_A \xrightarrow{\to c} q_A, p_B \xrightarrow{\leftarrow c} q_B \\ p \xrightarrow{A \triangle B} q & \text{se } A, B \in \mathcal{P} \\ \text{nessuna transizione} & \text{altrimenti} \end{cases}$$

Remark 4.3.1 *Con la notazione $p_A \xrightarrow{\to c} q_A$ indichiamo che esiste una transizione dallo stato p allo stato q con label $\to c$ nell'automa associato alla Goroutine A.*

In questo modo l'automa prodotto ci permette di considerare tutti le *possibili evoluzioni* della computazione parallela mentre l'operazione di sincronizzazione ci permette di *filtrare* e *unire* solo le evoluzioni che portano ad una interazione tra due Gouroutine sullo stesso canale. Queste tipo di interazioni, assieme a quelle di spawn, saranno le sole transizioni ammesse nell'automa finale dal momento che tutte le transizioni che non rispettano i casi definiti sopra vengono eliminate. Così facendo otteniamo in output dall'operazione di composizione un automa che rispetta la definizione 4.1.

Concludiamo questo capitolo con il Choreography Automaton ottenuto a partire dalle local views mostrate in figura 4.3 mediante l'algoritmo di composizione descritto in questa sezione.

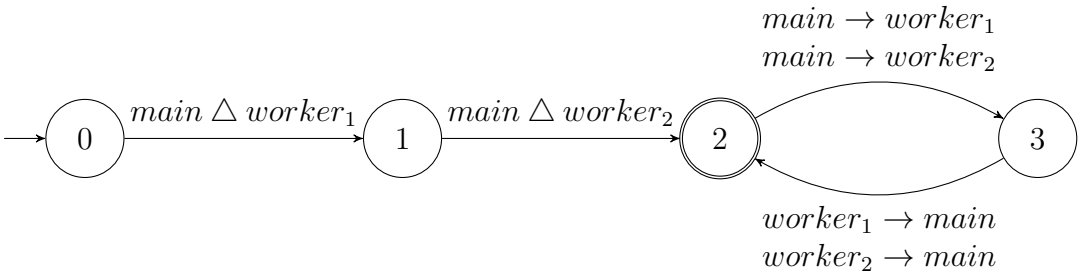


Figura 4.2: Il Choreography Automaton associato al codice sorgente in 4.1

Capitolo 5

Choreia

Choreia è il tool sviluppato come progetto per questa tesi. Il tool si occupa di tutte le fasi descritte nel capitolo precedente e consente all'utente finale di esportare il Choreography Automata ricavato dal codice sorgente in formato DOT.. È un software *open source* 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`

5.1 Parametri da linea di comando

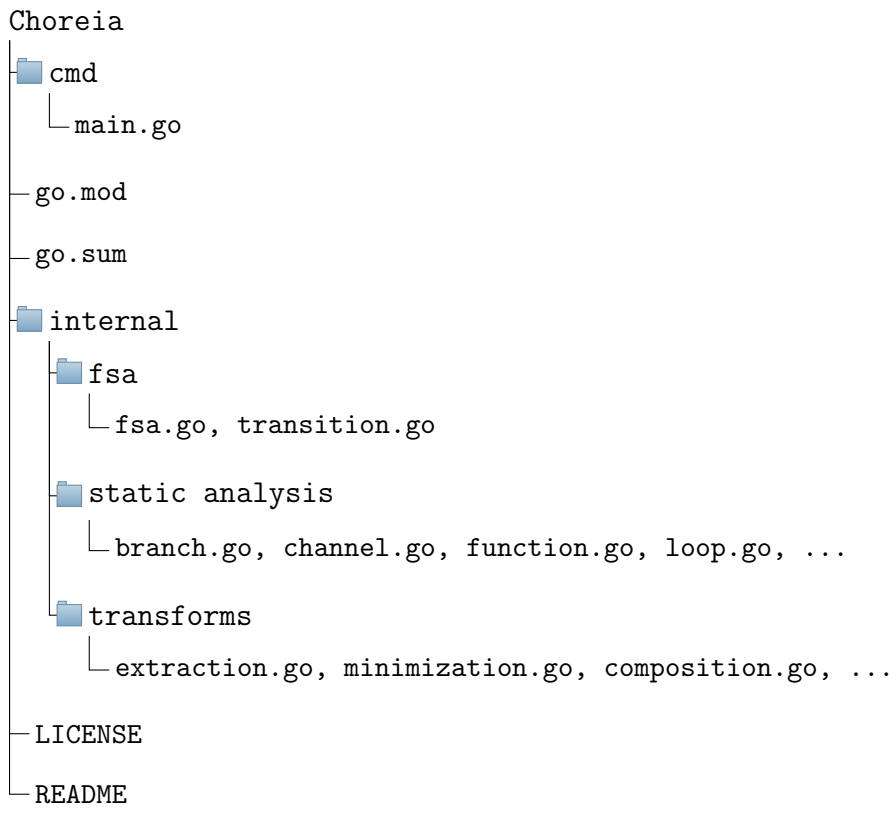
Il tool non ha una GUI in quanto, per gli scopi attuali, non è necessaria: infatti non è stato progettato come un tool di uso comune ma come uno strumento per persone interessate e con un minimo di conoscenza pregressa.
In ogni caso è possibile tramite *command line* fornire alcuni parametri e flags per un utilizzo *personalizzato*, i parametri disponibili sono:

Breve	Esteso	Descrizione
-i	–input	Il *path* del file .go in input
-o	–output	La directory in cui verranno salvati i file i vari 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

5.2 Struttura del progetto

Il progetto è strutturato su 3 moduli principali, ognuno con uno specifico compito: Il modulo `fsa` fornisce un'implementazione per gli automi a stati finiti (implementati attraverso un multigrafo), il modulo `static analysis` gestisce il parsing (fatto tramite la *standard library* golang) e l'estrazione dei metadati dall'AST ed infine il modulo `transforms` implementa varie operazioni su FSA come per esempio: determinizzazione e minimizzazione (per FSA generici) o la composizione (per automi associati alle local views).
La validazione dei dati e l'orchestrazione delle funzionalità fornite dai vari moduli sono gestite nel `main` che agisce come entry point del programma.



5.3 Flusso d'esecuzione

Il flusso d'esecuzione riprende le quattro macro fasi definite nella sezione 4.1.

Come prima cosa il file in input viene validato e parsato, generando un AST (queste funzionalità sono fornite dal modulo `go/ast` della standard library), l'AST viene poi navigato da un *visitor* che si occupa di raccogliere e organizzare i dati in apposite strutture

dati. Durante questa fase vengono generati degli automi a stati finiti non deterministici associati ad ogni funzione dichiarata all'interno del file in input, questi NFA descriveranno l'evoluzione della computazione nello scope di funzione.

Una volta ottenuti gli automi e i metadati si procede con lo step successivo, ovvero ricavare gli automi associati alle Goroutine presenti nel programma. Per fare questo si utilizza l'algoritmo definito in 4.3 che attraverso l'*inlining* delle chiamate di funzione e la sostituzione dei parametri formali con quelli attuali ricava in maniera ricorsiva degli automi non deterministici che rappresentano il flusso d'esecuzione completo della Goroutine. Come fase finale di questo step gli automi vengono determinizzati e minimizzati tramite gli algoritmi mostrati al capitolo 2.

L'ultimo step dell'esecuzione è quello di generare la global view a partire dalle view locali (e rispettivi automi), esattamente come detto in 4.4 generiamo l'automa prodotto di tutte le local views e poi eseguiamo l'operazione di sincronizzazione su quest'ultimo, infine esportiamo questo automa *finale* in formato DOT.

5.4 Esempi pratici

Di seguito mostriamo alcuni esempi di programmi Go e il rispettivo Choreography Automata associatogli da Choreia.

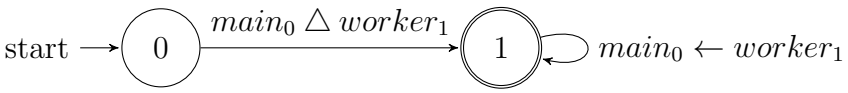
5.4.1 Loop determinato su canale

In questo esempio possiamo vedere che la Goroutine `main` inizializza il canale `channel` ed avvia la Goroutine `worker`, quest'ultima invia 100 messaggi numerati sul canale condiviso conludendo con la chiusura dello stesso. La chiusura di un canale è importante poichè blocca ogni futuro utilizzo dello stesso da entrambi i lati della comunicazione e permette al costrutto `for range` di interrompere il loop una volta ricevuti tutti i messaggi in coda. Se `worker` non chiudesse il canale `main` si bloccherebbe, alla 101-esima iterazione, aspettandosi di ricevere un messaggio da un canale *abbondanato* e questo creerebbe un problema di *liveness*.

```go
package main

import (
  "fmt"
)

func worker(shared chan string) {
  // Sends 100 numerated messages on the shared channel
  for i := 0; i < 100; i++ {
    shared <- fmt.Sprintf("This is message number: %d", i)
  }
  close(shared) // Close the cannel before returning
}

func main() {
  // Creates the shared channel
  channel := make(chan string, 10)
  // Starts the "worker" process
  go worker(channel)
  // Receives until the channel is closed by worker
  for msg := range channel {
    fmt.Printf("Received message: '%s'\n", msg)
  }
}
```

Di seguito troviamo gli automi delle local views associate rispettivamente alle Goroutine $\texttt{main}_0$ e $\texttt{worker}_1$ ottenuti tramite l'algoritmo di estrazione delle local views. Come si può facilmente osservare rispecchiano perfettamente il flusso d'esecuzione descritto dal codice che in questo caso è particolarmente semplice.
Si noti che l'automa della Goroutine $\texttt{worker}_1$ non presenta una transizione del tipo $\rightarrow$ *shared*, come si potrebbe essere indotti a pensare, bensì la label della transizione è $\rightarrow$ *channel* poichè l'algoritmo di estrazione gestisce la sostituzione dei parametri formali con quelli attuali.

Il seguente è invece l'automa associato alla global view ottenuto con l'algoritmo di composizione a partire dai due precedenti. Chiaramente in questo caso la sincronizzazione è banale e avviene sulle sole due transizione di invio e ricezione presenti.

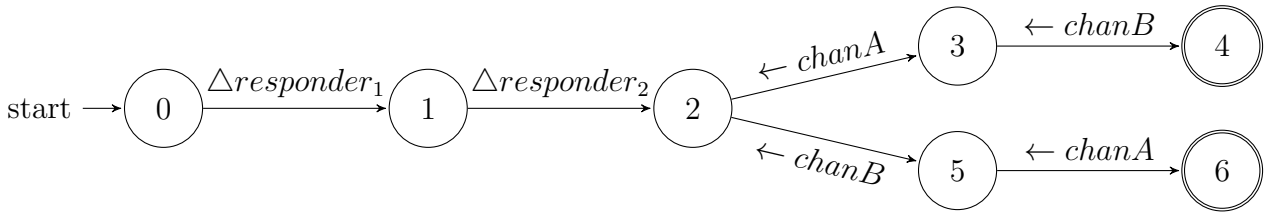


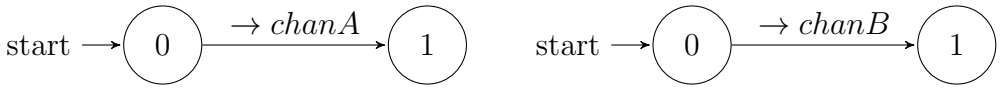
5.4.2 Operazioni condizionali con `select`

In questo esempio la Goroutine `main` crea due canali `chanA` e `chanB` e avvia due Goroutine `responder`, ciascuna con uno dei due canali, dopodichè tramite il costrutto `select` esegue il primo ramo che presenta un'operazione *non bloccante*, nel caso in cui nessuno dei due canali abbia dei messaggi si mette *in ascolto* aspettando che uno dei due abbia almeno un messaggio in coda. Nel nostro caso le Goroutine `responder` inviano solo un messaggio e non eseguono altre operazioni tuttavia possiamo assumere che portino avanti computazioni più complesse, inviando alla fine il risultato.

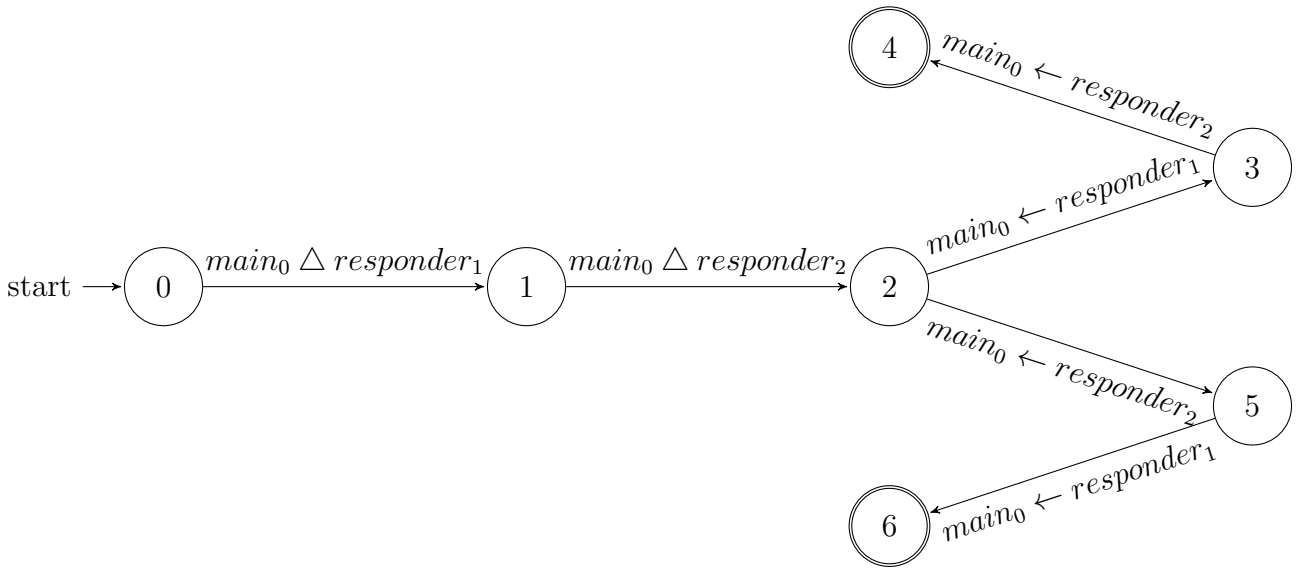
```
1  package main
2
3  import "math/rand"
4
5  func responder(channel chan int) {
6    // ... do something else
7    channel <- rand.Int()
8    // ... do something else
9  }
10
11 func main() {
12   // Creates the channels
13   chanA, chanB := make(chan int), make(chan int)
14
15   // Starts the "responder" processes
16   go responder(chanA)
17   go responder(chanB)
18
19   // Select from both channels
20   select {
21   case <-chanA:
22     <-chanB
23     // ... do something else
24   case <-chanB:
25     <-chanA
26     // ... do something else
27   }
28 }
```

Gli automi associati alle Goroutine $\texttt{main}_0$, $\texttt{responder}_1$, $\texttt{responder}_2$ sono i seguenti. Si noti come viene *mappato* il costrutto `select` attraverso gli automi, dal momento che il costrutto permette di *valutare* operazioni su più canali eseguendo solo la prima tra queste che non sia bloccante nell'automa a stati finiti questo viene mappato come una biforcazione del flusso di esecuzione che poi procede nel suo *sottografo* dando eventualmente la possibilità di ricongiungersi con gli altri.
Nel Choreography Automaton generato possiamo vedere come in questo caso l'utilizzo





del *prodotto tra automi* permette di considerare tutte le possibili evoluzioni del sistema concorrente che poi verrano elaborate ulteriomente e poste nella forma attuale dalla *sincronizzazione*, che in questo caso genera correttamente entrambi i rami d'esecuzione.

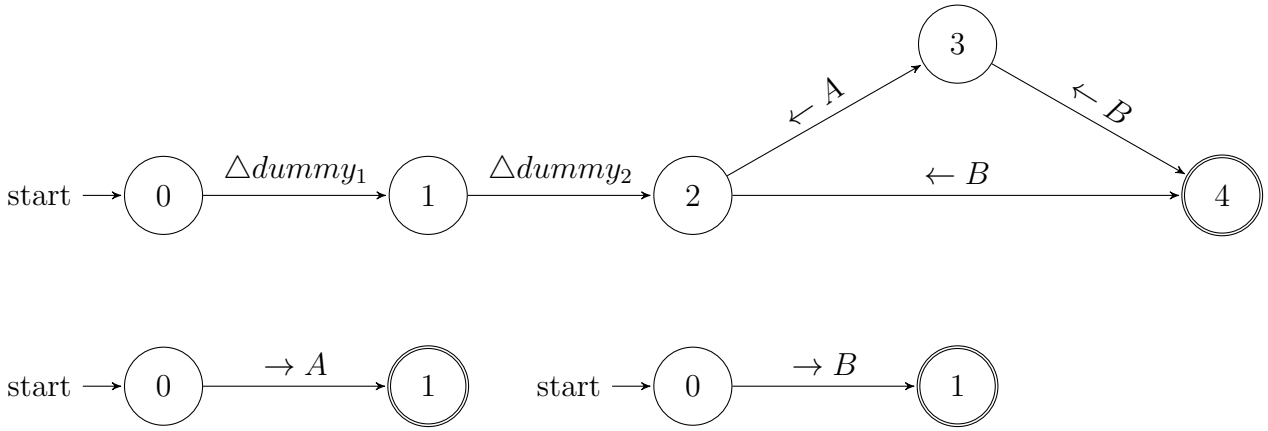


5.4.3 Branching con `if-then-else`

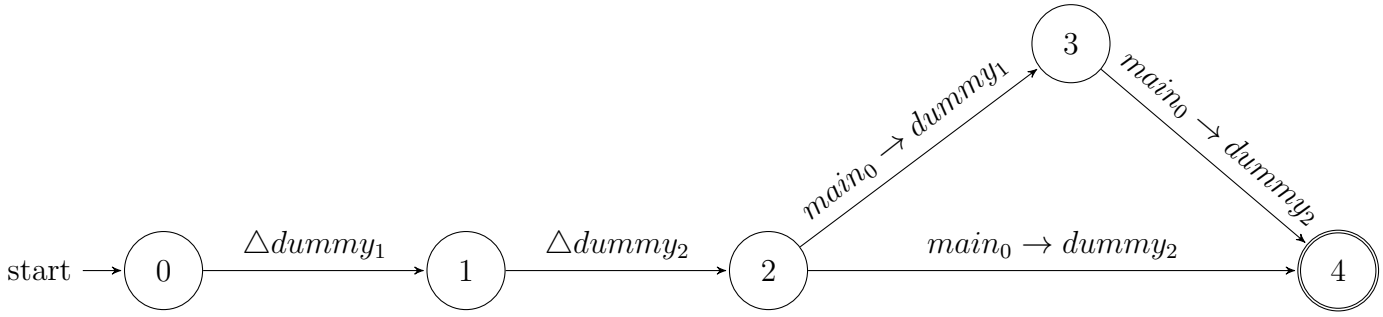
In questo esempio la Goroutine `main` inizializza due canali `A` e `B` ed avvia due Goroutine `dummy` ciascuna con il proprio canale: Una volta completata questa prima fase di setup semplicemente riceve un messaggio da `A` e uno da `B`. La receive su `A` è sempre effettuata poichè la condizione dell'`if` è sempre verificata.

```go
package main

import (
  "math/rand"
  "time"
)

func dummy(channel chan int) {
  channel <- rand.Int() // Send some random message
  close(channel)        // Closes the cannel before returning
}

func main() {
  // Creates the shared channel
  A, B := make(chan int), make(chan int)

  // Starts the "dummy" processes
  go dummy(A)
  go dummy(B)

  // Sleeps in order to let the other Goroutine send messages
  time.Sleep(5 * time.Second)

  if true { // Since the conditions is always verified
    <-A // Always receive from A
  }

  <-B // Receives a message from B
}
```

Gli automi associati alle Goroutine $\texttt{main}_0$, $\texttt{dummy}_1$, $\texttt{dummy}_2$ sono i seguenti. Si noti come viene *mappato* l'`if-then-else` attraverso gli automi: la biforcazione del ramo then viene mappata correttamente ma allo stesso tempo procede un'esecuzione *lineare* che rappresenta il caso in cui la condizione dell'if non sia verificata. Sempre riguardante la condizione dell'if possiamo banalmente osservare le limitazioni dell'analisi statica descritte al capitolo 2: nonostante la condizione sia sempre verificata Choreia assume del *non determinismo* e biforca il flusso d'esecuzione.



Nel Choreogaphy Automaton generato possiamo vedere che la struttura dello stesso è del tutto simile a quella dell'automa associtato alla Goroutine $\texttt{main}_0$. Questo è dovuto al fatto che almeno per questi esempi il sistema concorrente è particolarmente semplice e *gerarchico*: il `main` è la Goroutine dominante mentre le altre Goroutine sono piuttosto banali e immediate, dunque non aggiungono complessità al Choreography Automaton finale. Questo esempi *semplicizzati* sono dovuti alla difficoltà di rappresentazione degli automi e all'incremento nella complessità degli stessi al crescere della complessità del sistema concorrente.

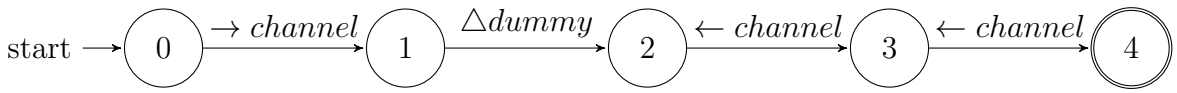


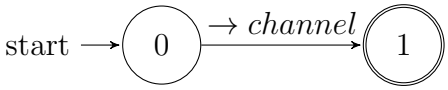
5.4.4 Function call con sostituzione dei parametri

In questo esempio la Goroutine `main`, dopo aver creato il canale `channel` chiama la funzione `f` passandogli sudetto canale come argomento, questa funziona esegue le seguenti operazione: invia un messaggio sul canale, avvia una Goroutine `dummy` condividendo con essa il canale e infine riceve il messaggio inviato precedentemente prima di restituire il controllo a `main` che conclude ricevendo per la seconda volta dal canale.

```go
package main

import "fmt"

func dummy(channel chan string) {
  channel <- "Hello from dummy" //Sends a message on the shared channel
}

func f(channel chan string) {
  channel <- "Hello from nested" // Send a message on channel
  go dummy(channel)              // Spawns a new "dummy" Goroutine
  fmt.Println(<-channel)         // Receives the message sent by itself
}

func main() {
  // Creates the shared channel
  channel := make(chan string, 1)
  // Call the "f" function
  f(channel)
  // Receives something from "channel"
  fmt.Println(<-channel)
}
```

Gli automi associati alle Goroutine $\texttt{main}_0$ e $\texttt{dummy}_1$ sono i seguenti. Si noti che la funzione `f` subisce l'*inlining* durante la fase di estrazione delle local views, difatti possiamo vedere che alcune delle transizioni nel primo automa sono proprio derivanti dall'esecuzione della funzione `f`.

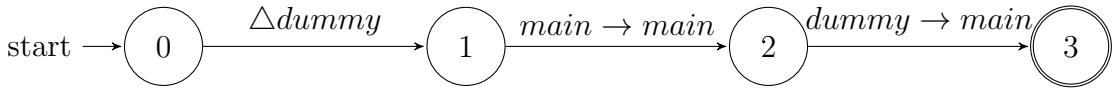




Choreia, allo stato attuale, non è in grado di generare un automa per il seguente codice, questo è dovuto al fatto che sono presenti diverse criticità all'interno dello stesso: prima di tutto la Goroutine `main` si sincronizza con se stessa inviando il primo messaggio su `channel` e ricevendolo poi dallo stesso. Questo, seppur possibile teoricamente in Go, non è *catturabile* dal nostro algoritmo in quanto richiederebbe un adattamento della nozione di prodotto tra automi. Attualmente infatti il prodotto tra automi $A \times B$ non considera il caso con A utilizzato da entrambi i lati come fattore $(A \times A)$ e questo comporta una *perdita d'informazione.*

Inoltre anche ammettendo che fosse implementato quanto detto sopra a questo punto avremmo nel Choreorgaphy Automata ben due sincronizzazione del tipo $main \rightarrow main$ sia nella transizione da 0 a 1, sia nella transizione da 2 a 3 e andrebbe quindi implementata un qualche tipo di algoritmo e/o validazione che eviti di incappare in questi casi particolari.
Un'ulteriore problematica legata a questo esempio invece è dovuta al fatto che il nostro algoritmo genererebbe due transizioni da 2 a 3 rispettivamente con label $main \rightarrow main$ e $dummy \rightarrow main$ questo perchè l'algoritmo di composizione attuale non tiene in considerazione che i canali sono gestiti con un politica di *First In First Out (FIFO)* e dunque in questo caso l'unica sincronizzazione possibile tra gli stati 2 e 3 è quella tra main e sè stesso.

Il Choreogaphy Automaton corretto è il seguente:



Capitolo 6

Conclusioni e lavori futuri

TODO

Bibliografia

[1] Franco Barbanera, Ivan Lanese e Emilio Tuosto. «Choreography Automata». In: Lecture Notes in Computer Science 12134 (2020). A cura di Simon Bliudze e Laura Bocchi, pp. 86–106.

[2] Franco Barbanera, Ivan Lanese e Emilio Tuosto. «Composition of choreography automata». In: *CoRR* abs/2107.06727 (2021).

[3] Daniel Brand e Pitro Zafiropulo. «On Communicating Finite-State Machines». In: *J. ACM* 30.2 (1983), pp. 323–342.

[4] Mario Bravetti e Gianluigi Zavattaro. «Towards a Unifying Theory for Choreography Conformance and Contract Compliance». In: Lecture Notes in Computer Science 4829 (2007). A cura di Markus Lumpe e Wim Vanderperren, pp. 34–50.

[5] Roman Haas et al. «Is Static Analysis Able to Identify Unnecessary Source Code?» In: *ACM Trans. Softw. Eng. Methodol.* 29.1 (2020), 6:1–6:23.

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

[7] Simone Orlando et al. «Corinne, a Tool for Choreography Automata». In: Lecture Notes in Computer Science 13077 (2021). A cura di Gwen Salaün e Anton Wijs, pp. 82–92.

[8] C++ Development Team. *C++'s website*. Online; Accessed 07-February-2022. 2021. URL: `https://isocpp.org/`.

[9] Docker Devolopment Team. *Docker's website*. Online; Accessed 07-February-2022. 2021. URL: `https://www.docker.com/`.

[10] Elixir Development Team. *Elixir's website*. Online; Accessed 07-February-2022. 2021. URL: `https://elixir-lang.org/`.

[11] Go Development Team. *Abstract Syntax Tree*. Online; Accessed 26-January-2022. 2021. URL: `https://en.wikipedia.org/wiki/Abstract_syntax_tree)`.

[12] Go Development Team. *Golang's website*. Online; Accessed 18-December-2021. 2021. URL: `https://go.dev/`.

[13] Graphviz Development Team. *DOT Language*. Online; Accessed 20-December-2021. 2021. URL: https://graphviz.org/doc/info/lang.html.

[14] Graphviz Devlopment Team. *Graphviz's website*. Online; Accessed 20-December-2021. 2021. URL: https://graphviz.org/.

[15] Kubernetes Devolopment Team. *Kubernetes' website*. Online; Accessed 07-February-2022. 2021. URL: https://kubernetes.io/.

[16] Wikipedia. *Dynamic program analysis*. Online; Accessed 26-January-2022. 2021. URL: https://en.wikipedia.org/wiki/Dynamic_program_analysis).