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:
Chiar.mo Prof.
Ivan Lanese

Presentata da:
Enea Guidi

Sessione III
Anno Accademico 2020/2021

Agli amici che ho conosciuto e che
mi hanno accompagnato in questo viaggio

Sommario

Le Choreographies 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 il quale permetta di capire in maniera immediata la *coreografia* dei participanti nel sistema e come questi interagiscano tra loro. Partendo dai singoli partecipanti, e le loro Choreographies *locali*, è possibile ricomporre in maniera bottom-up l'intera Choreography *globale* del sistema. Un ulteriore vantaggio delle Choreographies è che, quando rispettano alcune proprietà definite, permettono di fare assunzioni sull'assenza di tipici problemi di concorrenza quali Deadlocks 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 sisitema concorrente in maniera bottom-up.

Indice

Capitolo 1

Introduzione

Negli ultimi anni si è visto un interesse sempre più pronunciato verso lo sviluppo di sistemi concorrenti e distribuiti, questo si riflette nella progettazione a microservizi o in linguaggi di programmazione più recenti come Go, Rust o C++ 20 i quali implementano tutti, in forme disparate, un approccio alla concorrenza facilitato e univoco evitando la complessità e frammentazione che può derivare dall'utilizzo di librerie esterne e non standardizzate.
Prendendo per esempio Go, talvolta chiamato anche golang, un linguaggio di programmazione creato nel 2009 da Robert Griesemer, Rob Pike e Ken Thompson, che negli anni successivi prende una forte trazione, fino a essere supportato da Google, sopratutto nel settore del web-development e system programming. Alcuni dei fattori da successo sono una compilazione ed esecuzione veloce, una grammatica semplice e snella e un approccio *built-in* alla concorrenza con tipi e primitive fornite direttamente dalla standard library. Quest'ultima infatti fornisce un solo tipo di dato *chan* che può essere *buffered* o *unbuffered*, la comunicazione che avviene su questi canali è rispettivamente asincrona e sincrona.
Parallelamente a questo rinnovato interesse verso la programmazione concorrente e distribuita nasce l'esigenza di formalizzare dei modelli teorici che possano supportare la progettazione, gestione e manutenzione di sistemi concorrenti. Uno di questi modelli sono le *Choreographies*, le quali facilitano la rappresentazione e descrizione di sistemi concorrenti in cui i singoli *attori* comunicano tra di loro. Un aspetto distintivo delle Choreographies è la coesistenza di due *view*:

- Vista **globale**: descrive la coordinazione necessaria tra i componenti di un sistema
- Vista **locale**: descrive il comportamento di un singolo componente *in isolamento*

L'obiettivo di questa tesi è quello di progettare e implementare un tool che permetta, preso un sorgente Go e per mezzo di analisi statica, di estrarre un Choreography Automata, ovvero un automa a stati finiti che rappresenta la Choreography in modo grafico, più chiaro e immediato rispetto alla lettura del codice sorgente.

Capitolo 2

Nozioni preliminari e notazione

2.1 FSA non deterministici e deterministici

Prima di introdurre le Choreography e i Choreographies Automata è necessario fare un breve richiamo di alcune nozioni fondamentali quali la nozione di Automa a Stati Finiti (FSA) e alcune operazioni possibili sugli stesssi. 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 (come vedremo più avanti). Alcune applicazioni pratiche di questi automi possono essere, per esempio, regular expression (RegEx o RegExp), lexer e parser ma possono anche 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 sono 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 e forniscono una sufficente espressività per gli scopi di questa tesi.

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)*
- $\mathcal{L}$ *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 *Tipicamente è solito trovare anche una definizione alternativa ma equivalente in cui l'insieme degli stati di terminazione $\mathcal{F}$ non è presente, 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*) e distinto dalla nozione di DFA (*Deterministic Finite Automata*).*

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 ne l'utilizzo di ϵ transizioni, ne 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. L'algoritmo che permette di fare questa trasformazione fa uso estensivo di ϵ *closure* e della funione *mossa* 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 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 $\mathrm{D} = \langle \mathcal{T}, x, \mathcal{F}, \mathcal{L}, \delta \rangle$.

2.1.1 Esempi

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

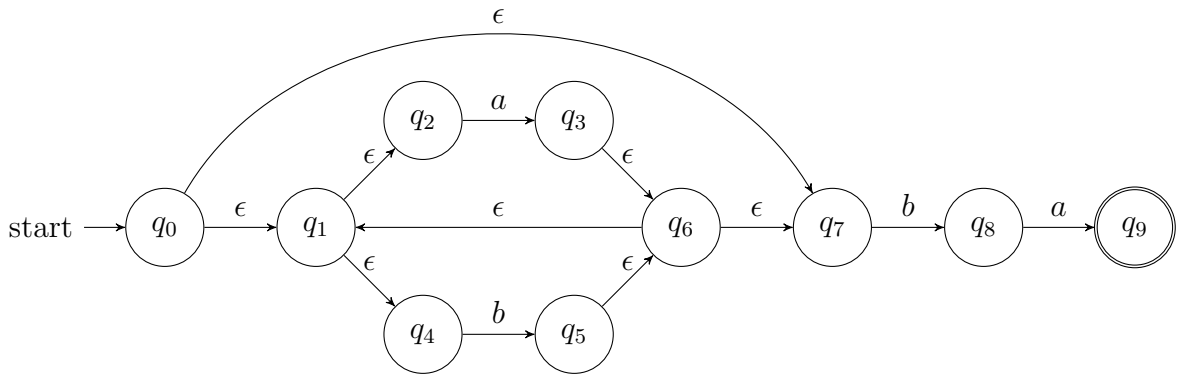
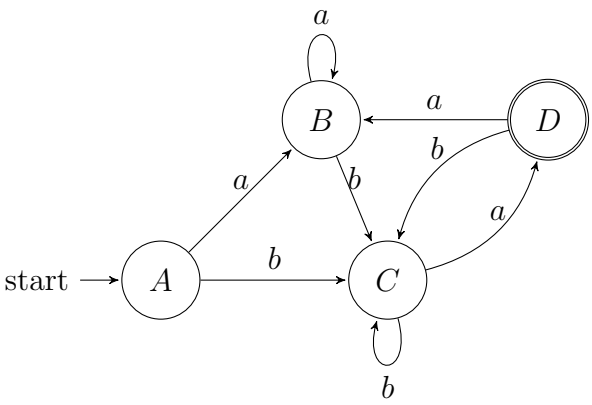


Figura 2.1: Un possibile NFA che riconosce la RegEx $(a|b)^*ba$

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

2.2 Choreography Automata

Passiamo ora alla definizione dei *Choreography Automata* (CA) iniziamo diversificando la nozione di *Choreographies* e *Choreography Automata* 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 sono inece un'*implementazione* possibile per questo modello. In questo caso noi stiamo scegliendo di implementare le Choreographies tramite degli Automi a Stati Finiti ma questo non esclude altre possibili implementazioni equivalenti alla nostra.

Per prima cosa ricordiamo che le Choreographies hanno due tipologie di *view* possibili:

- **Global View**: Che descrive il comportamento dei *partecipanti* (siano essi processi, programmi o routine) "as a whole" specificando anche come 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 e immediati, permettono di sfruttare in maniera molto conveniente i risultati e le nozioni desccritti precedentemente. I Choreography Automata sono dunque dei *casi particolari* di autmi 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: $sender \rightarrow receiver : message$.

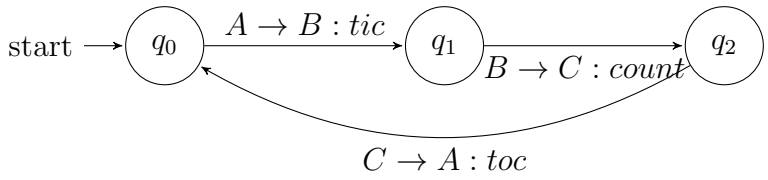


Figura 2.3: Un esempio di Choreogaphy Automata

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 2.5 (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 2.5.1 *Anche se nella definizione non sono ammesse ϵ-transizioni una variante non deterministica rimane sempre possibie, come vedremo anche più avanti in questo lavoro, ma si tende ad evitare per avere delle composizioni tra automi piu corrette.*

Capitolo 3

Adattazioni e modifiche

TODO

Capitolo 4

Tecnologie utilizzate

TODO

Capitolo 5

Descrizione del tool

TODO

Capitolo 6

Conclusioni e lavori futuri

TODO