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

Alla mia famiglia e agli amici 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 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 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, 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 sisitema concorrente in maniera bottom-up.

Indice

Capitolo 1

Introduzione

TODO

Capitolo 2

Nozioni preliminari e notazione

2.1 FSA non deterministici e deterministici

Prima di introdurre le Choreography e i Choreography 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. 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 i nostri scopi.

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

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 rigurado sulla minimizzazzione. 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* 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 lettereatura 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 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 $(a|b)^*ba$:

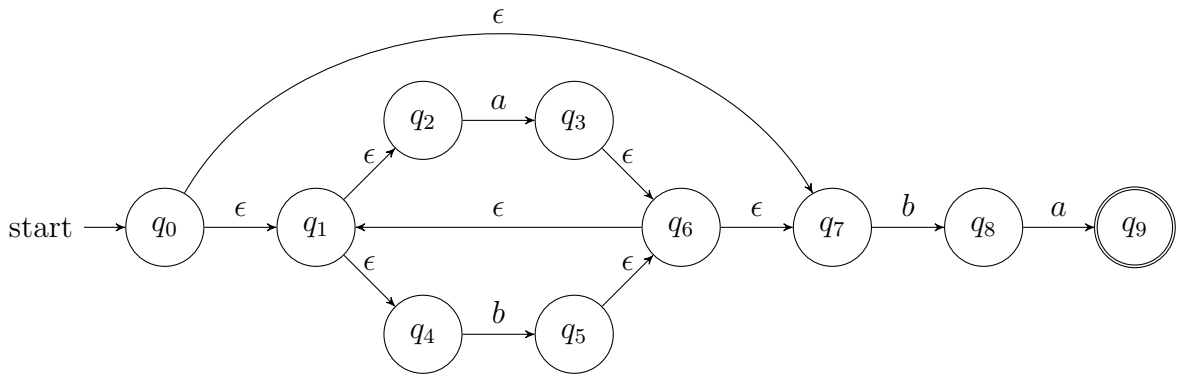
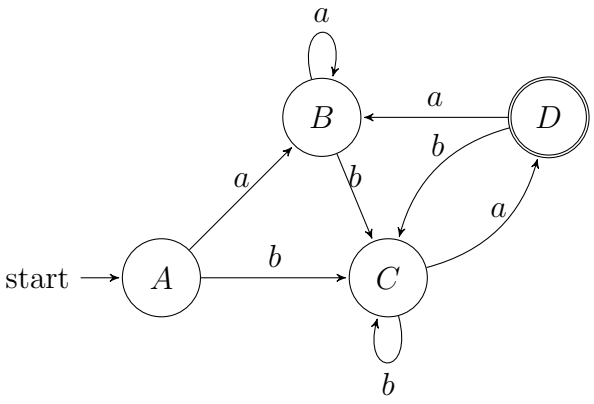


Figura 2.1: Un possibile NFA che riconosce la RegEx $(a|b)^*ba$

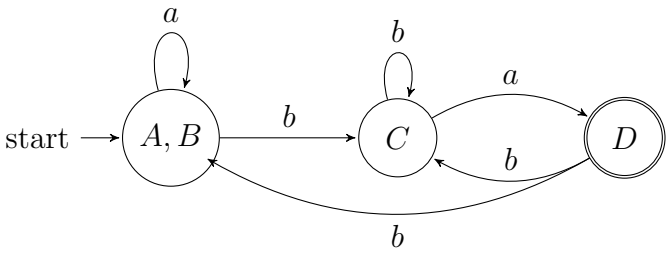
Si noti che questo è solo un *possibile* NFA in grado di riconscere il linguaggio dato ma ne esistono infiniti altri equivalenti ad esso. Vediamo ora invece il DFA D, equivalente ad N, calcolato tramite l'algorimo di *Costruzione per sottoinsiemi*



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

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

2.2 Choreography Automata

Passiamo ora alla definizione dei *Choreography Automata* (CA); iniziamo diversificando la nozione di *Choreography* 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* "as a whole" specificando anche come questi interagiscano 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*.

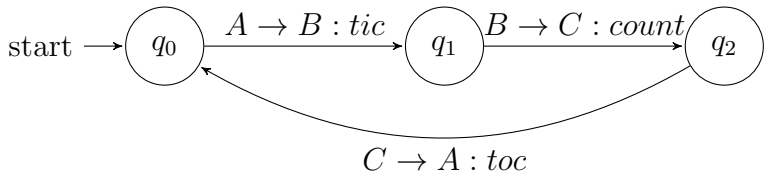


Figura 2.4: Un esempio di Choreogaphy Automata

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.*

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 dalla view globale. 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 2.6 (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 con cha ha tutte e sole le 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.

Definition 2.7 (Proiezione) *La proiezione su A di una transizione $t = s \xrightarrow{a} s_2$ 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 ϵ transizioni, ottenendone una versione deterministica e successivamente minimizzare quest'ultima. Entrambe le operazioni sono le medesime definite rispettivamente in 2.1 e 2.2

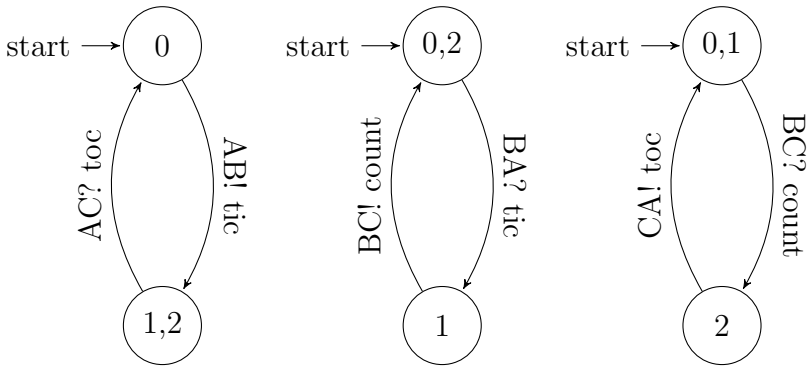


Figura 2.5: Le tre view locali estratte dall'automa in figura 2.4

Capitolo 3

Adattamento del modello

3.1 Obiettivi e problematiche

L'obiettivo del progetto, ad lato livello, e' quello di prendere del codice sorgente *Go* ed estarre in qualche modo il Choreography Automata associato a quel programma e che esprima in particolare come i processi interagiscono tra loro durante l'esecuzione.
L'esecuzione del nostro programma si divide in quattro fasi:

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

2. **Analisi statica**: Viene navigato l'AST estraendo tutte le informazioni necessarie (i metadati) e salvandole in strutture dati appropriate.

3. **Estrazione delle local views**: Partendo dai metadati si generano le local views dei vari processi (o attori).

4. **Riconcialiazione**: Ottenere partendo dalle view locali un singolo Choreography Automata che rappresenti l'intera coreografia del sistema (view globale).

Questo approccio e' chiaramente *Bottom-Up* mentre l'approccio delle definizioni nel capitolo 2 e' invece *Top-Down*, infatti abbiamo visto come, partendo dal Choreography Automata possiamo 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.

3.1.1 Limiti dell'analisi statica

Per quanto riguarda la fase di estrazione dei metadati, esplicata brevemente sopra, sorge una inconsistenza con la definizione formale di Choreography Automata data in pre-

cedenza: l'insieme $\mathcal{M}$ dei messagi non e' determinabile in maniera precisa attraverso l'analisi statica. Questo tipo di analisi viene effettuata infatti utilizzando solo il codice sorgente e ricavando dei dati senza eseguire in alcun modo il codice stesso (per questo motivo e' detta *statica*) e nel caso di Go e altri linguaggi non e' possibile solo attraverso l'analisi statica ricavare il valore esatto di tutti i messaggi, questo perche' detto valore puo' essere soggetto a vari tipi di *side effect* durante l'esecuzione, puo' essere legato a parametri temporali (timestamp o chron) o input forniti dall'utente. Tutti 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:

```go
package main

import (
  "fmt"
  "math/rand"
  "time"
)

type payload struct {
  data      int
  timestamp int64
}

func worker(incoming chan int, outgoing chan payload) {
  for msg := range incoming {
    // Sends back the results on the out channel
    outgoing <- payload{msg + 1, time.Now().Unix()}
  }
}

func main() {
  // Creates the channels
  in, out := make(chan int, 10), make(chan payload, 10)
  // Starts the worker processes
  go worker(in, out)
  go worker(in, out)
  // Infinite loop
  for {
    in <- rand.Int()
    res := <-out
    fmt.Printf("Received %d at %d \n", res.data, res.timestamp)
  }
}
```

Come possiamo vedere il programma mostrato e' in realta' alquanto banale il processo *main* genera un intero random che poi invia su un canale precedentemente condiviso con i due processi *worker*, uno dei due processi ricevera' questo intero lo incrementera' e poi lo reinviera' su un canale di output con in aggiunta un timestamp del momento dell'invio. Attraverso l'analisi statica non solo non riusciamo a determinare il valore inviato sul canale "in" ne' entrambi i valori inviati sul canale "out".

Una possibile soluzione a questo tipo di problemi puo' essere quello di utilizzare un tipo di analisi detta dinamica dove essenzialmente si osserva il programma mentre questo esegue e si raccolgono dati di esecuzione a runtime, questo tipo di analisi risolverebbe il problema posto sopra ma allo stesso tempo ne introrubbe alcuni nuovi. L'analisi dinamica presenta per esempio problemi di *incompletezza dei dati* per esempio: prendiamo un costrutto *if-then-else*, sappiamo che a *runtime* l'esecuzione entrera' in uno o nell'altro ramo questo tuttavia significa che saremo in grado di estrarre informazioni solo riguardanti il ramo in cui l'esecuzione si e' svolta e non conosceremo nulla di quanto succede nell'altro. Questo problema diventa ancora piu' evidente per codice che non e' *centrale* nel nostro programma, ovvero quel codice che non essendo *core* viene eseguito di rado e solo al sussistere di condizioni particolari.

La soluzione adottata qui e' in realta' molto semplice: prendiamo $\mathcal{M}$ non come l'insieme dei messaggi scambiati tra i partecipante ma come l'*insieme dei tipi* dei messaggi scambiati, i tipi infatti possono essere inferiti e ricavati senza particolari problemi per mezzo di analisi statica e non limitano l'espressivita' del modello. Nel caso della figura sopra $\mathcal{M}$ sara' definito come segue: $\mathcal{M} = \{int, payload\}$ e le label nel Choreography Automata associato saranno del tipo $main \xrightarrow{int} worker$ oppure $int \xrightarrow{payload} main$.

3.1.2 Peculiarita' di Go

Visto che il progetto deve gestire del codice sorgente Go e' bene considerare delle particolarita' del linguaggio in modo da adattare il modello teorico al linguaggio stesso. Introduciamo brevemente gli strumenti di concorrenza messi a disposizione dal linguaggio:

- **Canali**: Go fornisce un tipo di dato built-in *chan* su cui e' possibile fare operazioni di *send* e *receive*, la comunicazione sugli stessi puo' essere sia sincrona che asincrona.
- **Goroutine**: E' possibile far partire delle Goroutine anteponendo la keyword go ad una qualsiasi function call, questa funzione verra' eseguita in un contesto separato e parallelo rispetto al thread che l'ha creato.

- **Select**: Un costrutto particolare che permette eseguire operazioni di invio o ricezione su piu' canali ed eseguire la prima, tra queste operazioni, che non sia bloccante.

Mentre i canali e il costrutto sintattico "select" non generano particolari problemi o conflitti con il modello teorico attuale lo stesso non si puo' dire per le Goroutine, in particolare il problema sta nel fatto che le Goroutine siano intrinsicamente *gerarchiche*, ovvero per ogni programma Go viene avviata sempre e solo una Goroutine (quella che esegue la funzione *main*), sara' poi questa durante la sua esecuzione a farne partire altre, le quali a loro volta potranno avviarne altre ancora e cosi via. Il problema che sorge da questo approccio deriva dal fatto che nella definizione di Choreography Automata si assume in qualche modo che tutti i partecipanti siano gia' avviati e pronti a comunicare tra loro mentre per i nostri scopi servirebbe invece sapere qaundo e da chi e' stata avviata una Goroutine in modo da poter definire quando la sua *local view* diventa rilevante, ci serve determinare il momento esatto in cui una view locale (risp. Goroutine) inizia a interagire con le altre.

Per fare questo possiamo estendere la definizione di Choreography Automata e di Communicating Finite-State Machine date riespettivamente in 2.5 e 2.6 come segue:

Definition 3.1 (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 $\mathcal{L}_{int}$ e $\mathcal{P}$ definiti come in 2.5 e $\mathcal{M}$ definito come in 3.1.1

Definition 3.2 (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 3.2.1 *Seppur non interessante per gli scopi di questa tesi e' possibile adattare la nozione di proiezione in modo che tenga in considerazione di transizione del tipo $A \triangle B$ con $A, B \in \mathcal{P}$*

3.2 Algoritmo di Riconciliazione

TODO

Capitolo 4

Tecnologie e librereie utilizzate

4.1 Go (golang)

Go (anche chiamato golang) e' un linguaggio di programmazione *general purpose* open source sviluppato nel 2007 da Robert Griesemer, Rob Pike e Ken Thompson e poi supportato da Google stesso negli anni a seguire. Fortemente ispirato al C presenta una sintasssi minimale e molto semplice, Go e' *statically typed* e fornisce un *Garbage Collector* lasciando comunque all'utente la possibilita' 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 appensantire l'eseguibile finale Go rifiuta di compilare moduli o file dove non tutte le dipendenze importate vengono utilizzate
- **Compilazione piu 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 della compilazione
- **Approccio semplificato alla concorrenza**: Il linguaggio utilizza le Goroutine, dei *processi leggeri*, le quali permettono un utilizzo semplificato e piu' accessibile della programmazione concorrente

Altre feauture del linguaggio degne di nota sono: il package manager della *standard library* e l'ecosistema di pacchetti totalmente ditribuito e librerie disponibili, e la grande varieta' di architetture supportate (comprensive di *microcontroller* e *embedded systems*). Go e' stato utilizzato nello sviluppo di tecnologie celeberrime come Docker e Kubernetes e attualmente viene utilizzato da grandi aziende quali Google, MongoDB, Dropbox, Netflix, Uber e altri.

4.2 Graphviz e Linguaggio DOT

TODO

Capitolo 5

Descrizione del tool

TODO

Capitolo 6

Conclusioni e lavori futuri

TODO