Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Dipartimento di Informatica - Scienza e Ingegneria - DISI
Corso di Laurea Triennale in Informatica

Verifica di Well-Formedness in un Tool per Choreography Automata

Relatore:
Prof. Ivan Lanese

Presentata da:
Vairo Di Pasquale

II Sessione - I Appello
Anno Accademico 2019/2020

A Spotty,
che fu una persona migliore
di quanto mai potrò essere io

e a Ralph,
per non averci nemmeno mai provato

Abstract

In questo elaborato viene presentato un lavoro di estensione e miglioramento di Corinne, un tool per la lettura, composizione e proiezione dei *Choreography Automata*. Questi sono un modello emergente basato sul concetto di automi a stati finiti impiegato principalmente per la descrizione top-down di coreografie di sistemi distribuiti. In particolare il progetto si concentra nell'implementazione della verifica di *Well-formedness*, caratteristica fondamentale di un c-automaton al fine di garantirne una correttezza strutturale, priva dei tipici errori della programmazione concorrente, come *Deadlocks* e le *Race Conditions*.

Indice

Elenco delle figure

Capitolo 1

Introduzione

Nel corso degli ultimi anni si è assistito ad un notevole sviluppo e ad una repentina evoluzione delle architetture dei sistemi informativi. Queste, infatti, sono progredite passando da schemi centralizzati, in cui i dati e le applicazioni risiedono in un unico nodo elaborativo a modelli distribuiti, basati genericamente su un un insieme di processi interconnessi tra loro. In questo caso le comunicazioni avvengono esclusivamente tramite lo scambio di opportuni messaggi [3]. I modelli distribuiti risultano pertanto maggiormente rispondenti alle necessità di decentralizzazione e di cooperazione delle moderne organizzazioni. Questo andamento è dovuto a diverse esigenze a livello globale tra cui esigenze economiche ed esigenze tecnologiche. Rispettivamente per quanto riguarda le esigenze a livello di mercato, i sistemi distribuiti permettono di velocizzare il processo di creazione e messa sul mercato di un prodotto mentre per quanto riguarda le esigenze tecnologiche si ricorre ad un'architettura centralizzata in modo da distribuire il carico su più calcolatori rendendo così il servizio scalabile. Dato che un sistema distribuito è fisicamente più complesso da gestire rispetto ad uno centralizzato e richiede quindi complicate tecniche di comunicazione non sarà semplice svilupparne uno esente errori [4]. In un sistema distribuito possono palesarsi infatti, una serie di problematiche proprio per la varietà di interazioni che si verificano da parte di più processi in esecuzione. Bisognerà quindi isolare il singolo er-

rore e capire come la combinazione del nostro programma con altri influenzi il comportamento complessivo dell'applicazione. In questo elaborato viene analizzato nello specifico l'argomento delle *Choreographies* [6] le quali mirano a risolvere tali complicazioni. Queste si basano sull'idea di descrivere la comunicazione che due o più processi devono avere durante l'esecuzione nella quale viene delineato nello specifico il comportamento di ognuno. Lo scopo delle *Choreographies* è quindi proprio quello di rappresentare in modo chiaro e diretto i partecipanti e le loro interazioni attraverso una vera e propria "coreografia". Una volta realizzata, sarà possibile verificarne la correttezza attraverso il concetto di *Well-formedness*, insieme di condizioni che una coreografia deve rispettare per dirsi corretta per costruzione e avere la garanzia che la stessa sia esente dagli errori tipici della programmazione concorrente come i *Deadlocks* e le *Race Conditions*. Il lavoro svolto in questo elaborato si focalizza nell'estensione di Corinne [5], un tool preesistente per la lettura e manipolazione dei *Choreography Automata* [2] basato sugli automi a stati finiti. L'implementazione consiste proprio nella verifica delle condizioni di *Well-formedness* dell'automa che forniremo in input a tale programma.

Nel capitolo *Teoria Di Base* vengono riportati i concetti alla base di Corinne, è pertanto introdotto il concetto di Automi a Stati Finiti, il concetto di Choregoraphy Automata fino ad arrivare all'operazione della proiezione. Nel terzo capitolo, *Corinne*, è invece mostrato il funzionamento del tool. Viene dunque descritto il suo funzionamento soffermandoci in particolare sulle funzioni preesistenti e introducendo le nuove implementazioni. L'ultimo capitolo *Well-formedness*, oggetto principale dell'elaborato, contiene una dettagliata descrizione delle condizioni necessarie per il suo corretto verificarsi, vengono quindi definiti i concetti di *Well-sequencedness* e *Well-branchedness*, fornendo un approfondimento del modo in cui è stata eseguita la loro implementazione in Corinne.

Capitolo 2

Teoria Di Base

2.1 Automi a stati finiti

Per arrivare a comprendere meglio i concetti che verranno affrontati, si dovrà prima introdurre il concetto di automi a stati finiti. Gli automi a stati finiti (FSA, Finite State Automata) sono un modello matematico di computazione basato sul concetto di "stati" e "transizioni". Questi non sono nient'altro che macchine astratte le quali possono trovarsi in uno stato alla volta a seconda di un dato input. Tipicamente rappresentati da dei diagrammi a stati, gli FSA aiutano a descrivere circostanziatamente e formalmente il comportamento di un dato sistema a patto che esso sia composto da una predeterminata sequenza di azioni innescate a loro volta da una predeterminata sequenza di eventi. Possiamo rappresentare per esempio secondo questo modello processi come il funzionamento di un ascensore, di un sistema di riscaldamento o più semplicemente il funzionamento di un distributore automatico.

Come possiamo infatti vedere dalla Figura 2.1, il diagramma è un grafo orientato (o digrafo) costituito da stati, rappresentati dai nodi, e da transizioni, a loro volta rappresentate dagli archi tra i nodi stessi. Gli stati rappresentano la "posizione", o meglio, il "valore" corrente del sistema. Come si può notare sempre in Figura 2.1 nell'esempio del funzionamento di un distri-

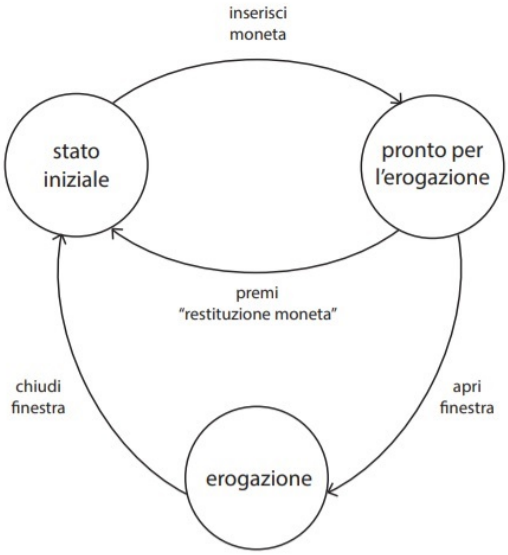

Figura 2.1: Automa per la descrizione di un distributore automatico

butore automatico gli stati sono lo stato "iniziale","pronto per l'erogazione", al quale si accede inserendo la moneta all'interno, e lo stato di "erogazione". Considerando l'esempio del caso di un semaforo, più semplicemente, lo stato "verde","giallo" o "rosso". La computazione ha inizio attraverso uno stato iniziale, rappresentato dalla freccia entrante nel grafo e termina con uno o più stati di accettazione o finali. Nella nostra definizone di FSA non viene specificato l'insieme degli stati finali dal momento che consideriamo solo FSAs dove ogni stato è di accettazione. Prendendo invece in analisi un caso più complesso e prestando particolarmente attenzione all'interazione tra transizioni e stati, tramite gli FSA si riesce a rappresentare anche il comportamento dell'"intelligenza artificiale" del fantasmino del celebre videogioco Pac-Man come si può osservare in Figura 2.2.
Diamo quindi ora la definizione formale di FSA:

Definizione 1. *Un automa a stati finiti (FSA) è una tupla del tipo* $T = < S, L, \delta, s_0 >$ *dove* :

- S *è un insieme finito di stati*
- L *è un insieme finito di labels con* $\epsilon \notin L$ (*input vuoto*)
- $\delta \subseteq S \times (L \bigcup \{\epsilon\}) \times S$ *è la funzione di transizione*
- $s_0 \in S$ *è lo stato iniziale*

Il sistema quindi cambia di stato in stato seguendo un semplice sistema di transizioni in input ed output predeterminato ma non per questo deterministico: gli FSA infatti possono essere del tipo NFA (Non deterministic Finite Automata) o DFA (Deterministic Finite Automata). In termini generali la principale differenza tra queste due tipologie è che un DFA può essere solamente in uno stato alla volta durante l'esecuzione, mentre un NFA può esserne in più di uno nello stesso momento. Ciò è dato principalmente dal fatto che un NFA può avere più transizioni con la stessa label che partono dallo stesso stato o che eseguono una transizione senza che sia strettamente necessario alcun tipo di input (epsilon-transitions).

Queste due tipologie di FSA riconoscono la stessa classe di input, pertanto sono equivalenti per definizione. Tuttavia, facendo uso di un particolare algoritmo per la conversione di NFA in DFA troveremo più comodo lavorare esclusivamente con gli automi deterministici, i DFA.

2.2 Choreography Automata

Tenendo in mente la definizione e composizione di un FSA possiamo finalmente introdurre l' argomento nucleo di questo elaborato, i choreography automata [2]. Questi non sono altro che un modello di coreografie di sistemi che comunicano tra di loro. Questo modello delinea e definisce il modo in cui avviene la comunicazione tra due o più servizi, in modo tale da controllare e gestire efficientemente il rapporto tra i suoi partecipanti e, rispettando

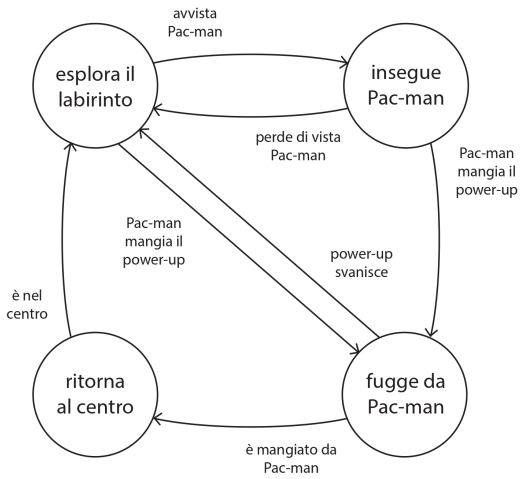

Figura 2.2: FSA per il comportamento di un fantasmino del gioco Pac-Man

alcune regole e condizioni, garantire al programmatore un codice esente da errori propri della programmazione concorrente come le *Race Conditions* e *Deadlocks*. Un esempio di Choreography Automata è l'automa in Figura 2.3, nella quale possiamo notare tutti gli elementi descritti in precedenza per un generale FSA deterministico, dove però le labels descrivono il comportamento di due processi che comunicano tra di loro, ovvero che inviano e ricevono un messaggio. Siamo quindi pronti per dare una definizione formale di Choreography Automata:

Definizione 2. *Un Choreography Automaton (c-automaton) è un* $\epsilon - free$ *FSA con insieme di labels:*

$$L_{int} = \{\ A \to B : m \mid A \neq B \in P,\ m \in M\ \}$$

dove:

- *P è l'insieme dei partecipanti*
- *M è l'insieme dei messaggi*

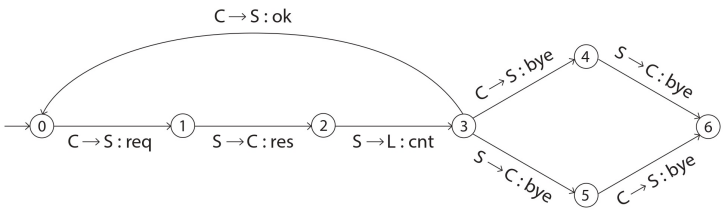

Figura 2.3: Esempio Choreography Automata (Automa di riferimento)

2.3 Local views e Communicating-FSM

Una caratteristica importante e costante per ogni coreografia è quella di possedere due modalità distinte di visione della comunicazione dei partecipanti: le global e le local views. La global view non è nient'altro che la rappresentazione del sistema secondo un punto di vista olistico del diagramma, ovvero secondo un punto di vista intero, globale. La local view descrive, invece, il comportamento isolato di un singolo componente ed è un modello alla base di alcune metodologie importanti che vedremo in questo elaborato. La local view è ottenuta tramite l'operazione della proiezione e può essere applicata ad ogni partecipante della coreografia. Prima di definire l'operazione di proiezione è necessario introdurre tuttavia le *Communicating Finite-State Machines* (Communicating-FSM, o CFSM). Queste sono un modello che si basa su FSA e ha come unica differenza dai Choreography Automata le labels dell'automa che risultano composte in maniera differente:

- A B !m viene adottato per dichiarare che il processo A invia un messaggio m a B
- A B ?m viene adottato per dichiarare che il processo A riceve un

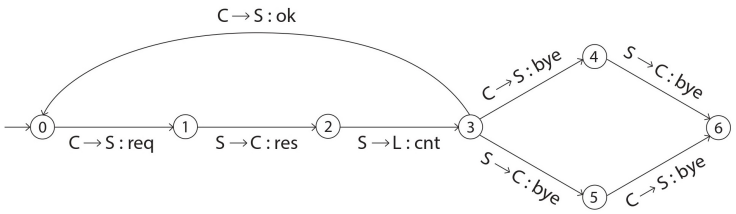

Figura 2.4: Esempio Communicating Finite-State Machine

messaggio m da B

Il c-automaton già osservato precedentemente in Figura 2.3, una volta trasformato in CFSM cambia le label come in Figura 2.4, dove si prende per soggetto il partecipante S. Il soggetto di un azione output A B!m è infatti A. Definiamo tutto ciò formalmente:

Definizione 3. *Una Communicating Finite-State Machine (CFSM) è un FSA M con l'insieme di labels:*

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

dove:

- *A e B sono i partecipanti*
- *P è l'insieme dei partecipanti*
- *M è l'insieme dei messaggi*

2.4 Proiezione

Per ottenere da una global view una local view di un partecipante della coreografia si adotta il metodo delle proiezione, il quale a partire dalle *Choreographies* permette di ottenere il codice di ogni singolo componente.

Definizione 4. *La proiezione su A di una transizione $t = q \xrightarrow{\alpha} q'$ di un c-automaton, scritta $t\downarrow_A$ è definita come:*

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

La proiezione di un Choreography Automata $= < S,\ L,\ \delta,\ s_0 >$ sul partecipante $A \in P$ è ottenuta a partire dall'automa intermedio.

$$A_\alpha = < S,\ L,\ \{ s \xrightarrow{t\downarrow_A} s' | s \xrightarrow{t} s' \ \in\ \delta \},\ s_0 >$$

Questo procedimento è ottenuto togliendo le possibili ϵ-transizioni e applicando l'operazione di minimizzazione, che permette di trasformare un automa in un altro equivalente ma con un minimo numero di stati.

Capitolo 3

Corinne

3.1 Tool preesistente

Corinne è un tool per la lettura, composizione e proiezione dei Choreography Automata. Tale piattaforma risulta essere la base perfetta per un ideale lavoro di verifica delle condizioni di Well-Formedness poichè presenta già al suo interno alcune funzionalità fondamentali per la manipolazione e lo studio dei c-automaton.
Scritta in Python3 [1], che permette inoltre di poter essere utilizzata su più sistemi operativi, si presenta dunque come la scelta perfetta per il lavoro che si intende sviluppare.

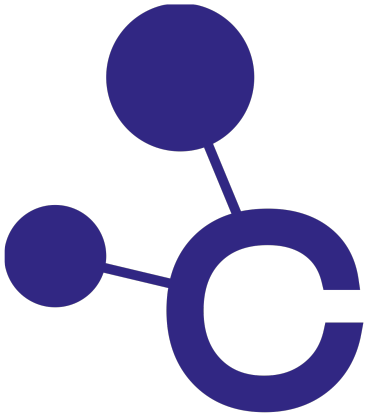

Figura 3.1: Logo Corinne

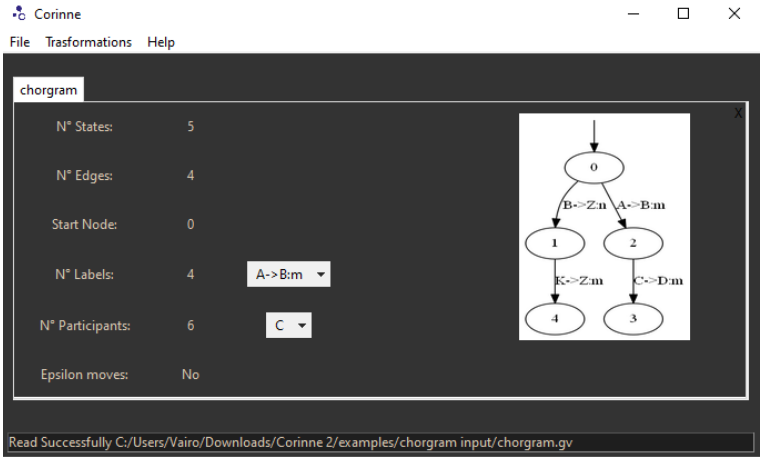

Figura 3.2: Interfaccia Corinne Preesistente

3.2 Interfaccia

L'applicazione si presenta con un design minimale, con 3 menù a tendina nella parte superiore e un box per il log dei messaggi nella parte inferiore.

Una volta aperto un file nel programma con l'apposito tasto **Open**, nella parte centrale del tool verrà illustrata un'interfaccia a schede per la visualizzazione delle caratteristiche del grafo appena letto, con tanto di immagine a destra per la rappresentazione del suo diagramma.

3.3 Funzionalità

In Corinne, oltre alla principale funzione di lettura di file in formato DOT (.gv) convertiti in Choreography Automata, possono essere effettuate diverse operazioni alla base della manipolazione dei c-automaton.
Tramite il tasto **Trasformations** è possibile accedere alle principali funzionalità del tool quali Prodotto, Sincronizzazione e Proiezione.

3.3.1 Prodotto

Attraverso il tasto **Product** è possibile effettuare il prodotto tra due Choreography Automata. Questa operazione coincide con il prodotto cartesiano che è alla base della composizione tra uno o più automi.

Definizione 5. *Siano* $A_1 = < S_1,\ L_1,\ \delta_1,\ s_{01} >$ *e* $A_2 = < S_2,\ L_2,\ \delta_2,\ s_{02} >$ *due automi a stati finiti. Definiamo* $A = < S,\ L,\ \delta,\ s_0 >$ *come il prodotto* A_1 *e* A_2 *dove:*

- $S = S_1 \times S_2$;
- $\begin{cases} \delta((s_1 s_2, a) = \{(x,y) | x \in \delta(s_1, a), y \in \delta(s_2, a)\} & se\ \delta(s_1, a)\ e\ \delta(s_2, a)\ definite \\ non\ definito & altrimenti; \end{cases}$
- s_0 = (s_{01},s_{02});

3.3.2 Sincronizzazione

Con il tasto **Sync** è possibile fare l'operazione della sincronizzazione. La sincronizzazione, eseguita idealmente dopo aver composto due global view con l'operazione del prodotto trattata poc'anzi, consente la comunicazione di ciascuna global view mediante un proprio partecipante designato allo scambio di messaggi con un altro partecipante che svolge la stessa funzione di mediatore nell'altra global view. In altri termini, la sincronizzazione prevede che per ciascuna global view vi sia un componente ("interfaccia") con il ruolo di *forwarder* al fine di permettere la comunicazione tra più views inviando o ricevendo messaggi tramite la propria interfaccia.

3.3.3 Proiezione

Con il tasto **Projection** è possibile fare l'operazione così come descritta in Sezione 1.4. Viene visualizzata una finestra nel quale viene a sua volta scelto il grafo e uno dei suoi partecipanti ed eseguire la funzione di proiezione su quest'ultimo, cambiando le labels del c-automaton con quelle del CFSM.

Capitolo 4

Well-formedness

Avendo delineato, dunque, i concetti teorici alla base dei *Choreography Automata* e avendo analizzato il funzionamento del tool che ci permette di lavorare su tali concetti, possiamo finalmente soffermarci nello specifico sul concetto di *Well-formedness*. Se una coreografia rispetta alcuni dei presupposti, che a breve andremo ad intrudurre, allora avremo la certezza che la coreografia in questione risulta essere *Well-behaved* ovvero che si comporta in modo tale da garantire un'assenza totale di alcuni dei classici errori della programmazione concorrente.

Definizione 6. *Un Choreography Automata rispetta le condizioni di Well-formedness se rispetta le condizioni di Well-sequencedness e Well-branchedness contemporaneamente.*

Tenendo dunque in mente le finalità per cui la *Well-formedness* viene utilizzata, si avrà quindi la necessità di analizzare nello specifico le altre due proprietà che la costituiscono. Una volta definito l'aspetto teorico di quest'ultime verrà mostrata una panoramica delle implementazioni eseguite all'interno di Corinne.

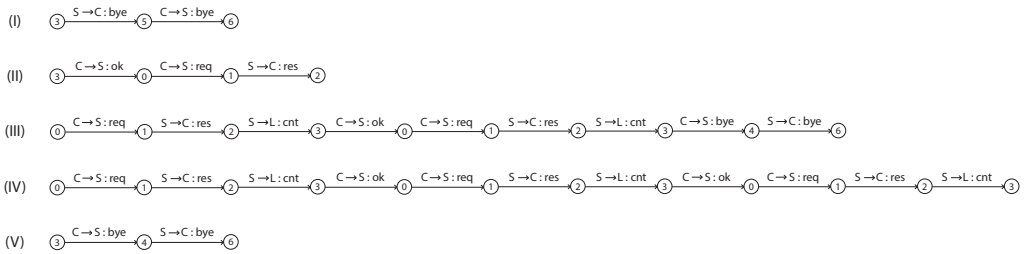

Figura 4.1: Cammini dell'Automa di riferimento

4.1 Well-branchedness

La *Well-branchedness* è la prima proprietà che andremo ad analizzare. Questo concetto chiave afferma come ogni partecipante sia effettivamente consapevole delle scelte fatte nella coreografia quando il suo comportamento dipende proprio da quelle scelte. La consapevolezza di tale scelta viene verificata su *span*, vale a dire su coppie di percorsi che possono costituire rami di scelta alternativi. Questi *span* sono formalizzati partendo dalla nozione base di *Candidate branch* che a sua volta è definito in termini di *pre-Candidate branch.*

Definizione 7. *Sia q uno stato di un Choreography Automata. Un pre-candidate q-branch del c-automaton è un suo q-cammino tale che ogni ciclo abbia al massimo un'occorrenza all'interno dell'intera visita. Un candidate q-branches è un pre-candidate q-branch massimale rispetto all'ordine dei prefissi.*

In Figura 4.1 sono elencati alcuni dei cammini possibili nel nostro automa di riferimento di Figura 2.3, possiamo notare tra questi come tutti i cammini elencati siano candidate q-brach tranne per il cammino (IV) poichè il ciclo 0 - 1 - 2 - 3 - 0 appare almeno due volte. D'altro canto, si può notare come i cammini (I), (III) e (V) siano candidate q-branch massimali rispetto

all'ordine dei prefissi.
Come si è osservato per q-branch passeremo dunque a dare una definizione di q-span per averne un quadro più chiaro.

Definizione 8. *Dato uno stato q di un Choreography Auomata, una coppia* (σ, σ') *di pre-candidate q-branches del c-automaton è un q-span se* σ *e* σ' *sono:*

- *Entrambi cofinali, con nessun nodo in comune se non q e l'ultimo nodo*
- *O candidate q-branches con nessun nodo in comune se non q*
- *1 candidate q-branch e un loop su q con nessun altro nodo in comune*

Un partecipante $A \in P$ *sceglie a q-span* (σ, σ') *solo se la prima transizione di entrambi* σ *e* σ' *ha A come mittente.*

Facendo nuovamente riferimento al nostro esempio di Figura 2.3, l'unico stato con la presenza di *q-span* risulta evidente sia il 3. Un *q-span* da 3, seguendo la condizione di cofinalità, può essere la coppia ((I),(V)) poichè i due cammini hanno in comune solamente lo stato iniziale e quello finale. Un'ultima semplice nozione necessaria alla comprensione della definizione di *Well-branchedness* è quella delle "transizioni concorrenti". Due transizioni infatti sono concorrenti, come possiamo vedere chiaramente in Figura 4.2, se provenendo da uno stesso stato q eseguono una transizione verso due stati differenti q_1 e q_2, dai quali effettuano entrambi un'ulteriore transizione verso uno stato q' eseguendo uno la precedente label dell'altro. Possiamo capire meglio il procedimento appena descritto dando una definizione formale.

Definizione 9. *Dato un Choreography Automata* $= < S,\ L,\ \delta,\ s_0 >$. *Due transizioni* $q \xrightarrow{l_1} q_1$ *e* $q \xrightarrow{l_2} q_2$ *sono concorrenti se e solo se esiste uno stato* $q' \in S$ *e due transizioni* $q_1 \xrightarrow{l_2} q'$ *e* $q_2 \xrightarrow{l_1} q'$.

Avendo quindi passato in rassegna le precedenti premesse, e avendo dunque tutti gli strumenti necessari per comprenderla a fondo, è possibile ora dare una definizione chiara e completa di *Well-branchedness*.

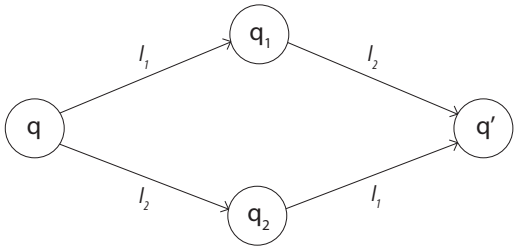

Figura 4.2: Esempio transizioni concorrenti

Definizione 10. *Un c-automaton rispetta le condizioni di Well-branchedness se per ogni stato q di un Choreography Automata deterministico e $A \in P$ è il mittente in una transizione da q, devono valere nello stesso momento:*

- *Tutte le transizioni da q che coinvolgono A, hanno mittente A*
- *Per ogni transizione t da q il quale mittente non sia A e ogni transizione t' da q il quale mittente sia A, t e t' sono concorrenti*
- *Per ogni q-span (σ, σ') dove A sceglie e ogni partecipante $B \neq A \in P$, la prima coppia di labels differenti nei cammini $\sigma \downarrow_B$ e $\sigma' \downarrow_B$ (se esistono) sono nella forma (CB?m, DB?n) con $C \neq D$ o $m \neq n$.*

Per familiarizzare con le condizioni di questa definizione proviamo ad applicarle al nostro automa di riferimento in Figura 2.3. Andando ad analizzare fin da subito lo stato più critico ovvero lo stato 3, ci si può rendere conto come questo non rispetti la prima condizione e di conseguenza non sia *well-branched*. Infatti prendendo A = C, oppure allo stesso modo A = S possiamo constatare come nelle transizioni uscenti da 3, il partecipante C si possa trovare sia con il ruolo di sender che con quello di receiver come anche S. Siccome l'automa non rispetta la prima condizione, è inutile proseguire

con la verifica delle altre due, poichè sappiamo fin da subito chiaramente, che esso non è *well-branched* e di conseguenza nemmeno *well-formed*. Intuitivamente possiamo dire che una scelta risulta essere *well-branched* quando i partecipanti, diversi dal partecipante che opta per cammini alternativi, si comportano uniformemente in ogni ramo oppure sono in grado di stabilire quale ramo è stato scelto dai messaggi che ricevono.

4.2 Well-sequencedness

La seconda ed ultima proprietà essenziale per completare la definizione di *Well-formedness* è la *Well-sequencedness*

Definizione 11. *Un c-automaton rispetta le condizioni di Well-sequencednes se per ogni due consecutive transizioni, vale almeno una delle due condizioni:* $q \xrightarrow{A \to B:m} q' \xrightarrow{C \to D:n} q''$

- *condividono un partecipante, che sia* $\{A, B\} \cap \{C, D\} \neq \emptyset$
- *sono concorrenti, ovvero che esista uno stato q"' tale che* $q \xrightarrow{C \to D:n} q \xrightarrow{A \to B:n} q''$

Date queste condizioni risulterà dunque facile verificare che il nostro Choreography Automa di riferimento in Figura 2.3 sia *Well-sequenced*. Considerando la prima condizione notiamo infatti che per ogni due transizioni consecutive condivide almeno un partecipante (banalmente intuibile anche per fatto che in questo caso vi siano solo 3 partecipanti). Se la prima condizione di *Well-sequencedness* risulta verificata allora possiamo subito dire che l'automa in questione rispetta la definizione senza doverne verificare la seconda.

Si può notare pertanto che non risulta molto difficile verificare che un c-automaton rispetti effettivamente questa condizione. Questo quindi può essere "completato", in caso non lo fosse, in uno che rispetta le condizioni

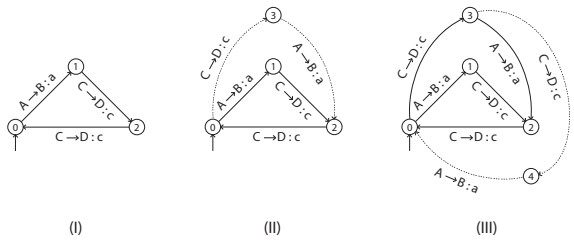

Figura 4.3: Fallimento operazione di Well-sequencedness

della *Well-sequencedness*. Questo processo può, apparentemente, risultare semplice tuttavia si può constatare come effettivamente non lo sia. Prendiamo ora come riferimento l'automa in Figura 4.3 (I), il quale si può verificare facilmente non sia well-sequenced per le transizioni dallo stato 0 allo stato 1, e dallo stato 1 allo stato 2. Aggiungendo, come in Figura 4.3 (II) un nuovo stato 3 con le rispettive transizioni che mirano a rispettare le condizioni di *Well-sequencedness*, otteniamo un nuovo automa equivalente all'automa di partenza ma comunque non ancora well-sequenced. Lo stesso accade se aggiungiamo ancora una volta un nuovo stato e due nuove transizioni come in Figura 4.3 (III). Di conseguenza è facilmente constatabile come questo procedimento possa andare avanti indefinitamente. Non è quindi sempre possibile eseguire un lavoro di completamento di un automa ad un altro che rispetti la *Well-sequencedness*.

4.3 Estensione dell'Interfaccia

Il lavoro eseguito per l'estensione di Corinne si concentra prevalentemente nell'aggiunta di un quarto e nuovo drop down menù: **Properties** . Tramite questo tasto è possibile verificare in maniera chiara ed esplicita la correttezza

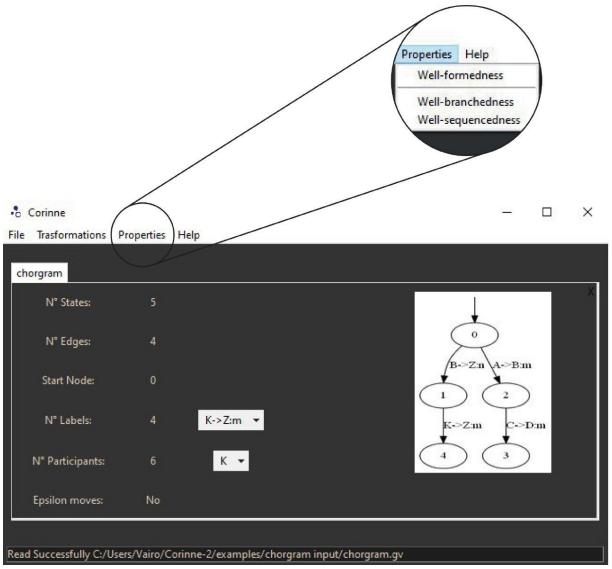

Figura 4.4: Azioni tasto Properties

di un qualsiasi automa che il tool sia in grado di leggere. Come si può notare in Sezione 4.4, cliccando il tasto è possibile effettuare la scelta di eseguire tre diverse azioni tra:

- Eseguire direttamente la verifica della *Well-formedness*
- Eseguire unicamente la verifica della *Well-brenchness*
- Eseguire unicamente la verifica della *Well-sequencedness*

Come si è visto in modo specifico in questo Capitolo, la *Well-formedness* non risulta essere nient'altro che l'insieme delle rispettive altre due proprietà: *Well-brenchness* e *Well-sequencedness*. Se il nostro automa infatti verifica queste due proprietà di conseguenza risulterà esatto per definizione e coerente quindi con le caratteristiche di un automa che rispetti la *Well-formedness* e le sue implicazioni. Al termine dell'operazione verrà quindi visualizzato nel box dei log di Corinne se per l'appunto l'automa verifichi o meno tali condizioni.

4.4 Implementazione del Controllo di Well-Formedness

L'applicazione preesistente di Corinne è stata sviluppata seguendo il pattern *M.V.C. (Model-View-Controller)*. Sono presenti pertanto un file per la view (`guy.py`), diversi files per il model e uno per il controller (`controller.py`), il quale costituisce il cuore del programma. Esso infatti contiene la maggior parte delle funzioni chiave di Corinne. Mi sono dunque servito di questo file per ospitare le mie implementazioni di *Well-formedness*, *Well-branchedness* e di *Well-sequencedness*.
Qualsiasi richiesta fatta dall'utente tramite il drop-down menu **Properties** viene gestita dai rispettivi metodi per selezionare un Choreography Automata già aperto nell'applicativo. Una volta terminata questa semplice azione la richiesta viene passata poi al controller tramite altri altrettanti rispettivi metodi designati. Di seguito verrà mostrato un estratto della classe (`MyGui`):

```
class MyGui(ttk.Frame):

    def open_well_formedness(self):...

    def open_well_branchedness(self):...

    def open_well_sequencedness(self):...

    def __exec_well_formedness__(self,combo_value):

        result = self.controller.make_well_branchedness(combo_value)
        self.log(result[0])
        return

    def __exec_well_branchedness__(self,combo_value):

        result = self.controller.make_well_branchedness(combo_value)
        self.log(result[0])
        return

    def __exec_well_sequencedness__(self,combo_value):

        result = self.controller.make_well_sequencedness(combo_value)
```

```
        self.log(result[0])
        return
```

Tutti i metodi di verifica delle condizioni di *Well-formedness*, *Well-branchedness* e di *Well-sequencedness* risiedono quindi nella classe `Controller`.

```
class Controller:
    ca_dict = {}

    def make_well_sequencedness(self,graph_name):

        ca = self.ca_dict.get(graph_name)

        ret = self.well_sequencedness_conditions(ca)

        if ret == None:
            result = ['Verified: Well-sequenced']
            return [result]
        else:
            result = ['Verified: NO Well-sequenced, not verified in ' + ret]
            return [result]

   def make_well_branchedness(self,graph_name):

        ca = self.ca_dict.get(graph_name)


        res1 = self.well_branchedness_first_condition(ca.edges,ca.states)

        if res1 != None:
            result = ['Verified: NO Well-branched in first condition: ' +
    res1]
            return [result]


        res2 = self.well_branchedness_second_condition(ca.edges,ca.states,ca
    .participants)

        if res2 != None:
            result = ['Verified: NO Well-branched in second condition: ' +
    res2]
            return [result]
```

```
        res3 = self.well_branchedness_second_condition(ca.edges,ca.states,ca
    .participants)

        if res3 != None:
            result = ['Verified: NO Well-branched in third condition: ' +
    res3]
            return [result]

        else:
            result = ['Verified: Well-branched']
            return [result]

    def make_well_formedness(self.graph_name):

        resultWS = self.make_well_sequencedness(graph_name)

        if resultWS[0] == 'Verified: Well-sequenced':

            resultWB = self.make_well_branchedness(graph_name)

            if resultWB[0] == 'Verified: Well-branched':

                result = ['Verified: Well-formed']
                return [result]

            return [resultWB[0]]

        return [resultWS[0]]
```

Come si può osservare nel codice appena mostrato, nella classe `Controller`, il metodo implementato per la verifica delle condizioni di *Well-formedness*, `make_well_formedness`, richiama banalmente il metodo delle altre due condizioni. Nel caso vi fosse un errore in una delle verifiche, esso, ritorna in output nel punto critico dove avviene l'errore del Choreography Automata preso in input. Come quindi già appurato nella teoria, esso confermerà che l'automa sia *Well-formed* solamente quando non ci siano errori nel metodo per la verifica di *Well-branchedness* e sia quando non ci siano errori nel metodo per la verifica di *Well-sequencedness*. Analizziamo ora questi due casi.

Il metodo `make_well_sequencedness` gestisce le condizioni di *Well-sequencedness* in un unico algoritmo strettamente legato alla definizione

nella funzione di verifica: `well_sequencedness_conditions`. Tale algoritmo prende come input la struttura dati per definire l'automa di cui si desidera verificarne le condizioni. Nello specifico, di ogni coppia di transizioni consecutive, in esso presenti, viene accertato se almeno una delle due condizioni risulti vera (come abbiamo visto nella Sezione 4.2); in caso contrario, come output verrà restituito il segmento specifico nel quale si è verificato l'errore.

Il metodo per verificare le condizioni di *Well-branchedness* viene invece gestito dal metodo `make_well_branchedness`. Esso segue lo stesso modus operandi del metodo per la verifica della *Well-sequencedness* ma divide la verifica delle condizioni in tre funzioni differenti:

- `well_branchedness_first_condition`
- `well_branchedness_second_condition`
- `well_branchedness_third_condition`

Come abbiamo già appurato nella Sezione 4.1, queste tre condizioni devono valere contemporaneamente, pertanto il processo di verifica, per ottimizzare il tempo di calcolo, terminerà non appena trovato un errore. Quest'ultimo verrà, come nel metodo `make_well_sequencedness`, segnalato nell'apposito box dei log, evidenziando all'utente la posizione dell'errore strutturale.

4.5 Esempi di casi d'uso

Come detto in precedenza, l'estensione di Corinne riguarda l'aggiunta di un nuovo menù a tendina denominato **Properties** che ci permette la verifica della corettezza strutturale del nostro automa, scegliendo quale operazione eseguire tra quelle indicate: *Well-formedness*, *Well-branchedness* e di *Well-sequencedness*. Il tool risulterà dunque in grado di confermare o meno se l'automa preso in considerazione rispetta effettivamente tali condizioni; in caso contrario verrà segnalato nel box dei log in basso il motivo dell'errore. Di seguito sono riportati alcuni esempi di casi d'uso.

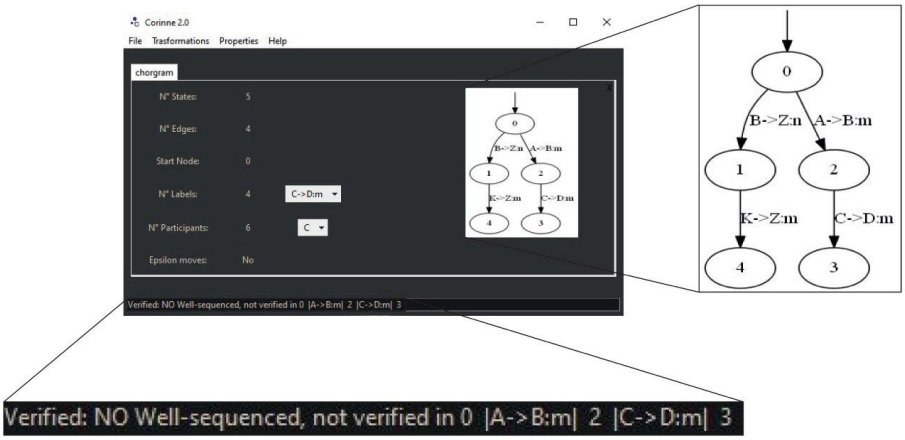

Figura 4.5: Esempio No Well-sequenced Automata

Nel primo mostrato in Figura 4.5, si vuole verificare se, dato un automa come input, risulti *Well-sequenced*: in basso viene segnalato come esso sia stato definito "NO Well-sequenced", seguito subito dopo dal punto esatto del grafo in cui le due condizioni di *Well-sequencedness* non si sono verificate. Sarà possibile quindi notare come le transizioni da 0 a 2 e da 2 a 3 non abbiano partecipanti in comune, o come dallo stato 0 non ci sia un cammino che rispetti le condizioni precedentemente osservate nella teoria delle transizioni concorrenti.

Nel secondo esempio in Figura 4.6 si vuole invece verificare se il nostro automa di riferimento di Figura 2.3 sia invece *well-branched*. Abbiamo già constato nella Sezione 4.1 come esso non sia *well-branched* e il nostro tool difatti conferma le nostre intuizioni segnalando anch'esso come il partecipante C abbia il ruolo sia di mittente che di destinatario nelle transizioni uscenti dal nodo 3.

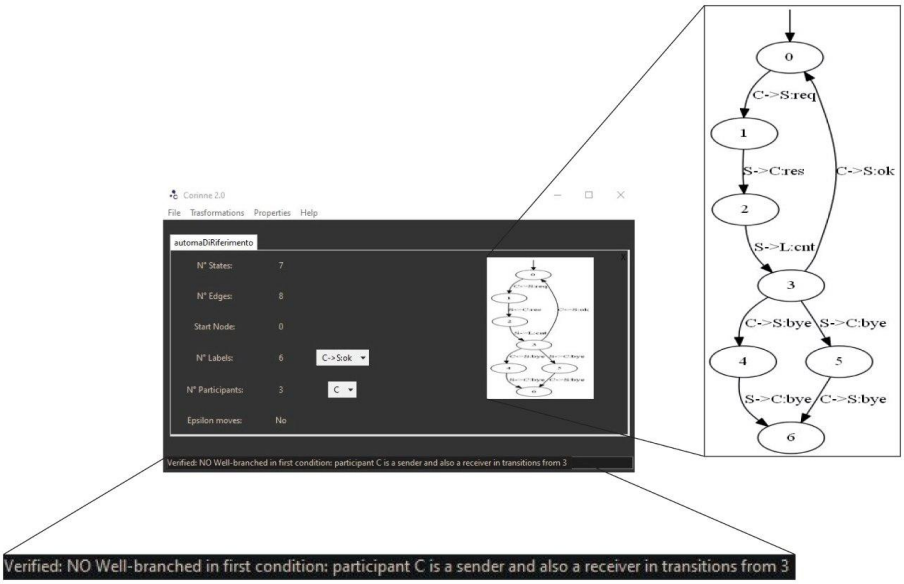

Figura 4.6: Verifica Well-brenchness dell'Automa di Riferimento

Conclusioni

In questo lavoro di tesi è stato proposto un lavoro di estensione e miglioramento dell'applicazione preesistente di Corinne. Tale applicativo è risultato essere la base perfetta per un ideale lavoro di verifica delle condizioni di Well-formedness poichè, come si è potuto constatare, presenta già al suo interno alcune funzionalità fondamentali per la manipolazione e lo studio dei Choreography Automata. Inoltre, il tool risulta concesso in licenza MIT ed è dunque un software open source, proposto pertanto a nuove modifiche. Nel mio lavoro di espansione (+500 loc) mi sono concentrato sull'identificazione dei c-automaton che effettivamente rispettano le condizioni di Well-formedness in modo da garantirne una correttezza strutturale. L'obiettivo preposto, facilitare all'utente la verifica degli automi tramite semplici passaggi, risulta pertanto conseguito e posso considerarmi soddisfatto del lavoro eseguito. Corinne è un'applicazione dalle molteplici possibilità di estensione e posso dirmi felice di aver collaborato all'espansione di questo tool. Data questa sua natura versatile, Corinne può essere oggetto di innumerevoli estensioni da qui la decisione di rendere questo aggiornamento dell'applicazione open source per le le proposte di nuove features e progetti futuri.

Download all'indirizzo **https://github.com/vairodp/Corinne-2**

Bibliografia

[1] *Python3*, `https://wwww.python.org`

[2] Franco Barbanera, Ivan Lanese e Emilio Tuosto *Choreography Automata*, COORDINATION 2020: 86-106, `http://www.cs.unibo.it/~lanese/work/TR/choreography_automata.pdf` (2020)

[3] George F. Coulouris, Jean Dollimore *Distributed Systems: Concepts and Design*, Addison-Wesley; Prima edizione (1988)

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

[5] Simone Orlando, *Sviluppo di un'applicazione per l'elaborazione di Choreography Automata*, Bachelor's thesis, Alma Mater Studiorum, Università di Bologna (2019)

[6] Fabrizio Montesi, *Choreographic programming*, Ph.D. thesis, IT University of Copenhangen (2013)

Ringraziamenti

Ringrazio in primo luogo il mio relatore, il prof. Ivan Lanese per l'infinita disponibilità e pazienza concessa.
Ringrazio la mia famiglia per tutto il supporto, per esserci sempre stata e per non avermi mai fatto mancare nulla. È solo merito vostro, vi voglio bene.
Ringrazio tutti i miei amici e colleghi di Università, persino Ravaglia
Ringrazio tutti gli amici del Sasha, siete fondamentali ma non ve lo dimostro mai, giuro cambio.
Ringrazio ovviamente Fazi, lo sappiamo perchè. Però smettila di cercare easter egg in questa pagina, ***non*** *ci sono.*
Ringrazio tutte le persone che in qualche modo mi sono state vicino, per chi mi ha dato sempre una mano e specialmente chi ha fatto le 6 del mattino per starmi vicino.
Ringrazio Zelda per la consistenza e costanza nelle spike.
Non ringrazio chi mi ha rubato la bici il primo anno. Infame.
Vorrei poter dedicare più parole ad ognuno di voi perchè in qualche modo avete reso possibile questa mia impresa impossibile, manca però solamente 1 ora alla consegna e ho le ansie.
Grazie