ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Dipartimento di Informatica - Scienza e Ingegneria

Corso di Laurea Triennale in Informatica

JolieGraph: uno strumento per l'analisi delle comunicazioni di servizi Jolie

Relatore:
Prof.
Ivan Lanese

Presentata da:
Francesco Licchelli

Sessione marzo 2025
Anno Accademico 2024/2025

microservizi

DOT

IDFA

analisi statica

Jolie

Abstract

Negli ultimi anni, il panorama informatico ha subito una significativa evoluzione grazie alla diffusione dei paradigmi orientati ai servizi e alle architetture a microservizi. Questi approcci hanno reso necessaria la disponibilità di strumenti avanzati per l'analisi delle comunicazioni tra servizi distribuiti. La presente tesi introduce JolieGraph, uno strumento progettato per analizzare staticamente il codice sorgente dei servizi Jolie e generare automi finiti che rappresentano in modo chiaro e rigoroso il flusso comunicativo fra microservizi.

JolieGraph utilizza tecniche di parsing e genera alberi di sintassi astratta (AST) sfruttando la libreria `jolie.lang`. Gli AST vengono analizzati e, attraverso la rappresentazione degli automi nel formato DOT, si permette agli sviluppatori di individuare rapidamente anomalie e potenziali criticità nei flussi comunicativi, difficilmente rilevabili con tecniche tradizionali di test dinamici.

Lo strumento proposto costituisce inoltre un primo passo significativo verso la generazione automatica di coreografie, modelli globali capaci di descrivere le interazioni tra servizi distribuiti.

Indice

Elenco delle figure

Capitolo 1

INTRODUZIONE

Negli ultimi anni il panorama informatico ha subito una significativa evoluzione, caratterizzata dal crescente utilizzo di paradigmi orientati ai servizi e dall'affermazione delle architetture a microservizi. Questi approcci hanno radicalmente trasformato il modo in cui si progettano, sviluppano e distribuiscono le applicazioni software, favorendo modularità, scalabilità e flessibilità. In tale contesto emerge il linguaggio Jolie, specificamente progettato per supportare lo sviluppo di servizi e facilitare la definizione esplicita delle interazioni tra microservizi tramite interfacce e porte di comunicazione.

Tuttavia, con l'aumentare della complessità dei sistemi distribuiti, diviene sempre più critica la necessità di strumenti capaci di effettuare analisi statiche rigorose dei flussi comunicativi. Questa tesi si concentra sulla progettazione e realizzazione di JolieGraph, uno strumento che analizza il codice sorgente di servizi Jolie[1, 2] generando automi finiti in grado di modellare in modo formale e chiaro le interazioni fra i diversi microservizi coinvolti. Attraverso tecniche di parsing e la costruzione di alberi di sintassi astratta (AST), JolieGraph produce rappresentazioni visuali intuitive utilizzando il linguaggio DOT[3], facilitando così l'individuazione precoce di anomalie e problematiche comunicative altrimenti difficilmente rilevabili mediante tecniche di testing dinamico.

L'obiettivo finale del lavoro è fornire uno strumento concreto che supporti non solo la validazione e il debugging di sistemi distribuiti, ma che costituisca anche un primo passo verso la generazione automatica di coreografie, cioè modelli globali che descrivono e orchestrano l'intero flusso comunicativo tra servizi. Si rimanda al capitolo delle conclusioni per maggiori dettagli circa il supporto alla generazione di coreografie.

1.1 Motivazioni

La corretta modellazione e analisi delle comunicazioni tra servizi rappresenta un aspetto cruciale per garantire la robustezza e l'affidabilità delle applicazioni. L'approccio tradizionale, basato su test e verifiche dinamiche, risulta spesso insufficiente nel catturare tutte le possibili criticità presenti in sistemi eterogenei e distribuiti. L'impiego di tecniche formali, come gli automi, e l'utilizzo di linguaggi di descrizione grafica (ad es. DOT) permette di esplicitare in modo rigoroso i flussi comunicativi e di evidenziare anomalie che potrebbero sfuggire a metodi meno strutturati.

1.2 Obiettivi e Contributi

La presente tesi si propone di sviluppare un framework per l'analisi statica dei codici sorgente dei servizi, con particolare riferimento al linguaggio *Jolie*. In particolare, il lavoro si concentra sulla progettazione e implementazione di JolieGraph, uno strumento che:

- Estrae e analizza le strutture dei servizi Jolie, sfruttando tecniche di parsing e rappresentazione tramite Abstract Syntax Tree (AST).
- Genera automi che modellano il flusso di comunicazione tra i microservizi, facilitando la verifica della correttezza e l'identificazione di potenziali criticità.
- Integra il formato DOT per la visualizzazione dei modelli, rendendo più immediata la comprensione delle interazioni e delle dipendenze presenti nel sistema.

Il contributo principale di questa ricerca risiede nell'unione di approcci formali e pratici, che consente di applicare teorie ben consolidate (come quella degli automi e dei linguaggi formali) a problematiche concrete di sviluppo e debug in ambienti distribuiti. In questo modo, il framework non solo supporta il processo di validazione dei servizi, ma offre anche un valido strumento per il reperimento rapido di eventuali anomalie nei flussi comunicativi.

1.3 Innovazioni e Implicazioni Pratiche

Il framework proposto si distingue per la sua capacità di coniugare teoria e pratica. Tradizionalmente, le tecniche formali sono state impiegate prevalentemente in contesti accademici; il lavoro presentato qui ne dimostra l'applicabilità concreta, offrendo agli sviluppatori uno strumento che agevola il debugging e la validazione dei microservizi. Tra le innovazioni principali si evidenziano:

- **Automatizzazione dell'analisi statica:** l'integrazione di parsing, generazione dell'AST e costruzione degli automi permette di ottenere un modello formale del sistema, riducendo il rischio di errori non rilevati dai test dinamici.
- **Visualizzazione intuitiva dei flussi di comunicazione:** la traduzione dei modelli formali in grafi tramite DOT favorisce una rapida identificazione delle dipendenze e dei possibili colli di bottiglia nelle interazioni.
- **Modularità ed estendibilità:** il framework è progettato in modo da poter essere facilmente integrato in ambienti di sviluppo esistenti e da supportare future estensioni, anche verso altri linguaggi orientati ai servizi.

Queste caratteristiche rendono il lavoro non solo un contributo accademico, ma anche uno strumento potenzialmente utile in contesti industriali, dove la gestione della complessità e la robustezza dei sistemi distribuiti sono requisiti imprescindibili.

1.4 Struttura della Tesi

La tesi è articolata in diversi capitoli, ognuno dei quali affronta un aspetto specifico del lavoro svolto:

- Nel **Capitolo 2** vengono introdotti i concetti fondamentali degli automi finiti, illustrando la loro struttura e le proprietà rilevanti per la modellazione dei flussi informativi. Viene inoltre presentato in dettaglio il linguaggio DOT, che consente di tradurre queste strutture in rappresentazioni grafiche chiare ed intuitive, facilitando la comprensione dei modelli complessi. Infine vengono illustrati il modelli *Service-oriented computing* e l'architettura a microservizi.
- Il **Capitolo 3** è dedicato all'analisi del linguaggio Jolie, evidenziandone il ruolo centrale nello sviluppo di microservizi e nella definizione di interfacce e porte di comunicazione. Viene descritta anche la libreria `jolie.lang`, che permette di trasformare il codice sorgente di un servizio Jolie in un albero di sintassi astratta, elemento essenziale per l'analisi statica e la successiva generazione degli automi.
- Nel **Capitolo 4** viene descritto il funzionamento di JolieGraph, lo strumento per la generazione degli automi sviluppato per questa tesi. Vengono elencati i diversi statement supportati e per ognuno di questi viene mostrato come JolieGraph procede con la creazione dell'automa relativo.
- Infine, il **Capitolo 5** raccoglie una serie di casi di studio ed esempi applicativi in cui vengono analizzate le prestazioni e l'efficacia del framework in contesti reali. In questo capitolo, ciascun esempio è accompagnato da commenti detta-

gliati che spiegano la struttura del grafo generato e le comunicazioni effettuabili o ricevibili dai servizi.

Capitolo 2

Background

In questo capitolo verranno fornite le basi teoriche necessarie alla comprensione e contestualizzazione del lavoro sviluppato nel seguito della ricerca. Verranno introdotti i concetti fondamentali degli automi a stati finiti, elementi chiave per l'analisi dei linguaggi formali.

Si analizzeranno in dettaglio la struttura e le proprietà degli automi, strumento essenziale per modellare e visualizzare relazioni e dipendenze in contesti tanto accademici quanto applicativi.

Verrà introdotto `DOT`, un linguaggio di descrizione grafica che consente di rappresentare in modo chiaro e leggibile le strutture dei grafi, rendendo immediata la comprensione visiva dei sistemi studiati.

Infine, si presenterà il paradigma Service-oriented Computing, con una breve digressione sui suoi punti di forza.

2.1 Fondamenti sugli automi

Questa sezione della tesi è stata strutturata con l'obiettivo di eliminare ambiguità sulle definizioni utilizzate, chiarire i concetti fondamentali necessari per la comprensione del resto del lavoro e dimostrare l'equivalenza tra le tre classi di automi utilizzate. Vengono citati, tradotti e/o adattati [4, p. 53, 58-62], [5, p. 145], [6, p. 139].

Definizione 2.1.1. Un *alfabeto*, denotato con Σ, è un insieme finito e non vuoto di simboli.

Definizione 2.1.2. Un *linguaggio* sull'alfabeto A è un insieme di stringhe su A.

Definizione 2.1.3. Un *Automa Finito Non Deterministico* (NFA, *Non-deterministic Finite Automaton*) è una quintupla

$$(\Sigma, Q, \delta, q_0, F)$$

dove:

- Σ è un alfabeto finito;
- Q è un insieme finito di stati;
- $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ è la funzione di transizione che, ad ogni coppia (q, a), associa un insieme (eventualmente vuoto) di stati successivi;
- $q_0 \in Q$ è lo stato iniziale;
- $F \subseteq Q$ è l'insieme degli stati finali (o accettanti).

Questa definizione, che consente transizioni multiple (o anche nessuna transizione) per una data coppia stato-simbolo, è ampiamente adottata in letteratura.
Definizione 2.1.4. Un *Automa Finito Deterministico* (DFA, *Deterministic Finite Automaton*) è una quintupla

$$(\Sigma, Q, \delta, q_0, F)$$

dove:

- Σ, Q, q_0 e F sono definiti come nella definizione del NFA;
- la funzione di transizione δ è totale e definita come $\delta : Q \times \Sigma \to Q$, ossia, esiste esattamente uno stato in Q associato ad ogni coppia (q, a).

La prossima definizione ha particolare rilevanza in quanto gli automi generati da JolieGraph sono riconducibili a questa categoria.
Definizione 2.1.5. Un *Automa Finito Deterministico Incompleto* (IDFA, *Incomplete Deterministic Finite Automaton*) è una quintupla

$$(\Sigma, Q, \hat{\delta}, q_0, F)$$

dove:

- Σ, Q, q_0 e F sono definiti come nel DFA;
- la funzione di transizione $\hat{\delta}$ è parziale, ovvero $\hat{\delta} : Q \times \Sigma \rightharpoonup Q$, permettendo che per alcune coppie (q, a) non sia definita alcuna transizione.

2.1.1 Equivalenza tra NFA, DFA e IDFA

Definizione 2.1.6. Un NFA $N = (\Sigma, Q, \delta, q_0, F)$ accetta la stringa $x = a_1 \dots a_n$ sse nel diagramma di transizione esiste un cammino, che inizia in q_0 e termina in uno stato di F, nel quale la stringa che si ottiene concatenando le etichette degli archi percorsi è esattamente x.
Definizione 2.1.7. Il linguaggio accettato da un NFA N è l'insieme $\mathcal{L}[N] = \{x \in \Sigma^* |$ N accetta x $\}$.

Prima di enunciare il teorema di equivalenza tra NFA e DFA è necessario introdurre la definizione di due funzioni ausiliarie:
Definizione 2.1.8 (ϵ-chiusura)**.** (a) Fissato un NFA $N = (\Sigma, Q, \delta, q_0, F)$ ed uno stato $q \in Q$, la ϵ-chiusura di q, indicata con ϵ-clos(q), è il più piccolo insieme di stati tale che (i) $q \in \epsilon$-clos(q); (ii) se $p \in \epsilon$-clos(q) allora $\delta(p, \epsilon) \subseteq \epsilon$-clos$(q)$.
(b) Se P è un insieme di stati, definiamo ϵ-clos$(P) = \cup_{p \in P} \epsilon$-clos$(p)$.

Questa operazione viene eseguita sulle transizioni etichettate con ϵ poichè questo simbolo rappresenta la stringa vuota, quindi percorrere una transizione etichettata con questo simbolo non consuma simboli dall'input dell'automa.

Come ultima definizione preliminare al teorema di equivalenza, si definisce la funzione mossa:

$$\text{mossa} : \mathcal{P}(Q) \times \Sigma \to \mathcal{P}(\mathcal{Q})$$

$$\text{mossa}(P, a) = \cup_{p \in P}\ \delta(p, a)$$

Teorema 2.1.1 (Equivalenza tra NFA e DFA)**.** Sia $N = (\Sigma, Q, \delta, q_0, F)$ un NFA e sia M_N l'automa ottenuto attraverso il Listing 2.1. Allora M_N è un DFA e si ha $\mathcal{L}[N] = \mathcal{L}[M_N]$.

Listing 2.1: Costruzione per sottoinsiemi. Dato un NFA $N = (\Sigma, Q, \delta, q_0, F)$, si ottiene il DFA $M = (\Sigma, \mathcal{T}, \Delta, \epsilon\text{-clos}(q_0), \mathcal{F})$, dove $\mathcal{F} \subseteq \mathcal{T} = \{q \in \text{R} \mid q \in \text{F}\}$

```
1 Inizializza S = ϵ-clos(q0)
2 Inizializza T = {S}
3
4 finché c'è un P ∈ T non marcato:
5     marca P
6     R = ϵ-clos(mossa(P, a))
7     se R ∉ T a:
8         aggiungi R a T
9     definisci Δ(P, a) = R
```

Proposizione 2.1.1. Per ogni DFA incompleto I esiste un DFA completo M che accetta il medesimo linguaggio.

Dal Teorema 2.1.1 e dalla Proposizione 2.1.1 concludiamo che le tre classi di automi sono equivalenti, ovvero, riconoscono la stessa classe di linguaggi.

Nei riferimenti futuri, salvo diversa ed esplicita indicazione, per automa andranno sempre intesi gli automi di classe IDFA.

2.2 DOT

Nelle sezioni precedenti sono state fornite le definizioni matematiche degli automi, le quali costituiscono un solido punto di partenza per l'analisi e lo studio formale di questi sistemi. Tuttavia, per poter effettuare elaborazioni successive e facilitare la condivisione dei modelli, è necessario adottare una struttura che consenta di salvare gli automi su file.

A questo scopo, è stato scelto di utilizzare il formato *DOT*[3], un linguaggio di descrizione testuale dei grafi sviluppato nell'ambito del progetto *Graphviz*[7]. DOT permette di rappresentare in maniera semplice e intuitiva le strutture grafiche, come quelle degli automi, tramite una sintassi leggibile e facilmente modificabile. I vantaggi principali derivanti dall'utilizzo di DOT sono:

- **Elaborazioni successive**: il formato testuale consente l'interazione con diversi strumenti software per l'analisi, la trasformazione e la visualizzazione automatica dei grafi;
- **Condivisione**: essendo uno standard ampiamente adottato, DOT facilita lo scambio dei modelli tra ricercatori e l'integrazione con altri sistemi e piattaforme.

Un file DOT per la rappresentazione di un automa tipicamente definisce i nodi (stati) e gli archi (transizioni) che li collegano. Ad esempio, il seguente snippet di codice DOT illustra la rappresentazione di un semplice automa:

```
digraph Automa {
    q0 [shape = doublecircle];
    q1 [shape = circle];
    q0 -> q1 [label = "a"];
    q1 -> q0 [label = "b"];
}
```

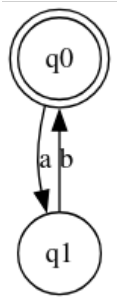


Figura 2.1: Esempio di codice DOT che descrive l'automa presente sulla destra

Nella Figura 2.1 il nodo `q0` è evidenziato come stato finale grazie alla forma `doublecircle`, mentre le transizioni sono rappresentate da archi etichettati con i simboli corrispondenti.

L'adozione del formato DOT non solo semplifica la memorizzazione e la successiva elaborazione degli automi, ma permette anche di utilizzare strumenti di visualizzazione, come Graphviz, che automatizzano la generazione di rappresentazioni grafiche, rendendo immediatamente comprensibili le strutture complesse. Inoltre, l'utilizzo di un formato standardizzato garantisce la portabilità dei dati e favorisce l'integrazione con altri sistemi di analisi e verifica, rendendo l'approccio proposto particolarmente versatile ed estendibile.

2.3 Libreria `JGraphT`

La libreria `JGraphT`[8] è una libreria Java open-source per la gestione di grafi, che fornisce strutture dati e algoritmi per la rappresentazione e l'elaborazione di grafi sia orientati che non orientati (anche pesati). Nel contesto di questo lavoro, JGraphT è stata utilizzata per modellare il grafo degli *stati* del sistema in esame. Di seguito si presentano due componenti chiave di JGraphT impiegate: la classe `DirectedDefaultGraph`, per rappresentare archi diretti fra vertici, e la classe `DOTExporter`, per esportare il grafo nel formato **DOT**.

2.3.1 DefaultDirectedGraph

La classe `DefaultDirectedGraph`[9] rappresenta un **grafo diretto** in JGraphT. Essa consente di gestire in maniera semplice i vertici e gli archi, garantendo che ciascun arco abbia una direzione definita (dal vertice sorgente a quello destinazione). Tale implementazione è particolarmente indicata per la modellazione di sistemi in cui le transizioni tra gli *stati* devono seguire una sequenza logica.

La classe `DefaultDirectedGraph` è una classe generica parametrizzata da due tipi: il primo, tipicamente indicato come `V`, rappresenta il tipo dei vertici, mentre il secondo, indicato come `E`, rappresenta il tipo degli archi. Il parametro `V` permette di definire il tipo degli elementi che costituiranno i nodi del grafo (ad esempio, `String`, oggetti personalizzati, ecc.), garantendo flessibilità nella scelta dei dati. Il parametro `E` specifica invece il tipo degli archi, che devono aderire a determinati requisiti (come l'implementazione di interfacce predefinite) per assicurare una corretta gestione degli archi. Questa struttura parametrica consente di adattare il grafo a svariate esigenze applicative in modo modulare ed efficiente.

Per esempio, si consideri un grafo composto da tre stati, `A`, `B` e `C`. Gli archi sono creati in modo da connettere `A` a `B` e `B` a `C`. Il seguente esempio di codice mostra come creare tale grafo:

```
Graph<String, DefaultEdge> graph =
    new DefaultDirectedGraph<>(DefaultEdge.class);
graph.addVertex("A");
graph.addVertex("B");
graph.addVertex("C");

graph.addEdge("A", "B");
graph.addEdge("B", "C");
```

2.3.2 DOTExporter

La classe `DOTExporter`[10] offre una funzionalità complementare: permette di esportare un grafo JGraphT nel formato testuale **DOT** (di GraphViz), come spiegato in precedenza. In altre parole, questa classe attraversa la struttura del grafo e ne produce una rappresentazione testuale secondo la sintassi DOT. Ciò è utile per visualizzare il grafo con strumenti esterni o per includerne uno schema nella documentazione.

Proseguendo con l'esempio precedente, l'utilizzo di `DOTExporter` sul grafo contenente A, B e C produce il seguente output in formato DOT (si noti l'uso di `digraph` per indicare un grafo diretto):

```
strict digraph G {
  A;
  B;
  C;
  A -> B;
  B -> C;
}
```

2.4 Service-Oriented Computing (SOC)

Il termine *Service-Oriented Computing* (SOC), traducibile come "computazione orientata ai servizi", si riferisce a un paradigma informatico che utilizza i **servizi** come elementi fondamentali per lo sviluppo di applicazioni software [11, 12]. In questo approccio, le funzionalità di un sistema sono incapsulate in componenti riutilizzabili (servizi) che espongono interfacce standard, accessibili tramite la rete. SOC promuove l'idea di assemblare tali componenti in una rete di servizi a *basso*

accoppiamento, anziché in un'unica applicazione monolitica, permettendo così una maggiore flessibilità nella composizione e nella riconfigurazione delle applicazioni.

Per implementare il SOC, si adotta tipicamente un'*Architettura Orientata ai Servizi* (Service-Oriented Architecture, SOA) che fornisce i meccanismi necessari per pubblicare, scoprire e integrare i servizi. Questo modello di progettazione consente di superare le tradizionali problematiche di interoperabilità tra sistemi eterogenei e di favorire il riutilizzo dei componenti software [12].

Perché il SOC è interessante

Il paradigma SOC è particolarmente interessante in quanto offre significativi benefici in termini di flessibilità e riuso del software. Grazie all'architettura a servizi autonomi e debolmente accoppiati, le applicazioni basate su SOC risultano più agili e adattabili: è possibile riconfigurare o sostituire i servizi per modificare i processi di business, ottenendo sistemi capaci di reagire rapidamente ai cambiamenti delle esigenze. Inoltre, l'adozione di interfacce standard e protocolli aperti favorisce l'interoperabilità tra piattaforme e organizzazioni diverse, facilitando l'integrazione di sistemi eterogenei su larga scala. Tali caratteristiche permettono di ridurre i tempi e i costi di sviluppo, promuovendo un uso efficiente delle risorse software [12].

Architettura a Microservizi

L'architettura a microservizi rappresenta un'evoluzione del paradigma SOC tradizionale, suddividendo le applicazioni in piccoli servizi autonomi che comunicano tramite interfacce leggere (es. API REST). Questo approccio consente una maggiore flessibilità e scalabilità, poiché ogni servizio può essere sviluppato, testato e deployato in modo indipendente, facilitando l'adozione di tecnologie e linguaggi differenti. Inoltre, l'isolamento dei componenti permette di aggiornare o sostituire funzionalità specifiche senza riprogettare l'intero sistema, contribuendo a una rapida evoluzione e a una migliore gestione degli aggiornamenti.

Parallelamente, i microservizi favoriscono una maggiore resilienza: il malfunzionamento di un servizio isolato non compromette l'intera applicazione, garantendo continuità operativa anche in presenza di errori. Tuttavia, l'adozione di questo paradigma introduce nuove sfide, come la gestione della comunicazione tra servizi distribuiti e la garanzia della consistenza dei dati, che richiedono l'uso di pattern avanzati (ad es. l'eventual consistency e il saga pattern). In sintesi, pur offrendo notevoli vantaggi in termini di agilità, scalabilità e manutenibilità, l'architettura a microservizi necessita di una progettazione attenta e di strumenti dedicati per gestire la complessità operativa dei sistemi altamente distribuiti [13, 14].

Capitolo 3

Jolie

3.1 Introduzione a Jolie

Nel presente capitolo viene presentato il linguaggio Jolie, illustrandone le caratteristiche principali e motivando la sua adozione in JolieGraph. Infine verranno illustrati gli aspetti di sintassi, semantica e della relativa implementazione, in `jolie.lang`, dei costrutti interessanti per JolieGraph. Le informazioni esposte sono state raccolte da [2], [15] e [16]. In particolare, dal lavoro del Prof. Montesi [2] sono state tratte le definizioni informali dei vari costrutti di Jolie, che costituiscono il fondamento per la trattazione degli argomenti che seguiranno.

3.1.1 Jolie: Il linguaggio per i microservizi

Jolie è un linguaggio di programmazione imperativo concepito per la definizione di microservizi, il loro coordinamento e la loro esecuzione. Ogni programma Jolie definisce un **servizio**, componente fondamentale del linguaggio. Esso include blocchi che vanno a definirne il *deployment* e il *behaviour*, ovvero, rispettivamente, il comportamento del servizio e la modalità di interazione con l'esterno. La separazione dei due concetti consente di poter modificare il deployment, anche a runtime, mantenendo costante l'aspetto di behaviour. Più nello specifico, il *service deployment* consiste nelle **porte**, di input e di output, attraverso le quali il servizio può ricevere o inviare richieste. Il *service behaviour*, invece, caratterizza il comportamento del servizio rispetto all'esecuzione di procedure, tra cui `main`. JolieGraph si avvale di Jolie per diversi motivi. In primo luogo, il linguaggio esprime in maniera esplicita l'interfaccia offerta dai servizi, permettendo così una più facile estrazione del modello delle interazioni presenti nel programma.

Listing 3.1: Esempi di servizi Jolie

```
from console import Console
from console import ConsoleInputInterface
service Greeter {
    execution: sequential
    inputPort input {
        location: "socket://localhost:8001"
        protocol: http
        requestResponse:
            greetUser( string )( string )
    }
    main {
        greetUser( request )( response ){
            response = "Hello "+request
        }
    }
}
service Client {
    embed Console as Console
    inputPort ConsoleInput {
        location: "local"
        Interfaces: ConsoleInputInterface
    }
    outputPort Greeter {
        location: "socket//localhost:8001"
        protocol: http
        requestResponse:
            greetUser( string )( string )
    }
    main {
        registerForInput@Console( )( )
        println@Console( "Insert your name:" )( )
        in( request )
        greetUser@Greeter( request )( response )
        println@Console( response )( )
    }
}
```

Nel Listato 3.1 sono presenti due servizi: Greeter e Client. Greeter risponde a richieste su porta 8001 di tipo `greetUser` con un saluto. Il servizio `Client` si collega con una porta di output al servizio `Greeter`. Dopo aver chiesto all'utente di

inserire il proprio nome, questo viene inviato al servizio `Greeter` tramite l'operazione dedicata. Ricevuta la risposta, questa verrà mostrata a schermo all'utente.

3.1.2 `jolie.lang`

Il pacchetto `jolie.lang` (versione 1.12.x) rappresenta l'elemento centrale su cui si basa l'analisi dei programmi Jolie. Questa libreria, realizzata in Java, offre un'API strutturata per il parsing e l'analisi sintattica dei programmi, permettendo di rappresentare il codice come un albero di sintassi astratta.

Grazie a queste funzionalità, JolieGraph è in grado di identificare in maniera automatica e precisa i costrutti interessanti per le comunicazioni.

I programmi Jolie vengono processati dalla libreria `jolie.lang` in due fasi:

- **Scanner**: suddivide il codice in token secondo le regole lessicali di Jolie.
- **Parser**: utilizza i token per costruire l'AST del programma.

AST Jolie e OLSyntaxNode

Tutti i nodi degli alberi di sintassi astratta prodotti dal parser di Jolie sono istanze di classi ereditate dalla classe astratta `jolie.lang.parse.ast.OLSyntaxNode`. L'interfaccia esposta da questa classe è composta da due metodi principali:

- `context()`: che ritorna informazioni relative al nodo rispetto al testo del codice del programma;
- (astratto) `accept(OLVisitor<C,R> v, C ctx)`: questo metodo, assieme al metodo `go(OLSyntaxNode n, C ctx)` dichiarato nell'interfaccia `OLVisitor` definisce il supporto al design pattern Visitor[17], che consente di esplorare l'AST in maniera semplice ed estendibile, poichè per aggiungere il supporto ad un nuovo costrutto, basterà aggiungere alla classe del visitatore lo specifico metodo `visit`.

3.1.3 Obiettivi del capitolo

Nel proseguo di questo capitolo verranno approfonditi:

- **Sintassi dei costrutti Jolie**: verranno analizzati esempi di codice che evidenziano come vengono definiti i servizi, le porte, le interfacce, le operazioni e i tipi, entità necessarie al funzionamento di JolieGraph.

- **Rappresentazione tramite** `jolie.lang`: saranno presentate le classi e le API offerte dalla libreria, spiegando in che modo ciascun costrutto sintattico viene rappresentato e gestito.

3.2 Analisi del blocco Deployment

Come anticipato, in questo blocco vengono definite le modalità tramite le quali un servizio Jolie può comunicare con l'esterno. Le richieste vengono scambiate tramite le primitive di comunicazione, diversi tipi di operazioni che permettono di inviare o ricevere messaggi tipati.

3.2.1 Il sistema dei tipi in Jolie

In Jolie, i messaggi scambiati attraverso le operazioni sono rappresentati come alberi di dati. Jolie adotta una tipizzazione dinamica: le variabili non richiedono dichiarazioni esplicite di tipo e possono contenere valori di qualsiasi tipo base. Il controllo sui tipi avviene a runtime, durante l'invio o la ricezione di messaggi, verificando che il dato scambiato corrisponda con quanto atteso.

Tipi di base

Jolie supporta un insieme di tipi base, o primitivi, per i valori elementari. I tipi primitivi includono: `bool`, `int`, `long`, `double`, `string`, `raw`[1] e `void`. Viene inoltre definito un tipo `any` che funge da super-tipo: una variabile di tipo `any` può assumere i valori di qualunque tipo nativo.

Tipi custom

Jolie permette di definire tipi complessi combinando tipi esistenti. Il costrutto `type` consente di dichiarare un nuovo tipo, assegnandogli un nome e specificandone la struttura interna. In Jolie ogni nodo di qualsiasi albero è un vettore, ed i nodi che compongono l'albero di definizione di un tipo non fanno eccezione. Questa struttura, associata all'indicazione della cardinalità, ovvero il numero minimo e massimo (`[min, max]`) di occorrenze per quel nodo, consente di ottenere array di tipi e di limitarne la dimensione ad un intervallo. La cardinalità di default per un nodo è `[1, 1]`.

Oltre alla possibilità di definire tipi strutturati, Jolie supporta anche i *tipi somma* (o union types), che permettono di definire un tipo come l'unione di più possibili alternative. Questo viene realizzato tramite l'operatore `|`, che indica che un valore

[1] il tipo `raw` indica una sequenza grezza di byte

può appartenere a uno dei tipi specificati. Jolie offre inoltre la possibilità di definire tipi complessi i cui nodi non sono specificati attraverso la definizione `any{ ? }` o con la parola chiave `undefined`.

Listing 3.2: Definizione dei tipi per il servizio WeatherForecast: coordinate, richieste pubbliche e private e la risposta della previsione.

```
1 type Coord: void {
2     .lat: double
3     .lon: double
4 }
5 type Films: void {
6     .favourite[0,10]: string
7     .public: bool
8 }
9 type UnknownStructure: undefined
```

Nel Listing 3.2 vengono definiti tre tipi:

- `Coord`: rappresenta le coordinate geografiche con latitudine e longitudine;
- `Films`: permette di indicare la Top 10 dei propri film preferiti e scegliere se renderla pubblica o meno;
- `UnknownStructure`: non vengono specificati i sottonodi.

Listing 3.3: Definizione di un tipo somma `ForecastResponseOrError` che può essere la risposta alla richiesta fatta oppure un messaggio d'errore.

```
1 type ForecastResponseOrError: ForecastResponse |
    ↪ AddressNotFoundError
```

Rappresentazione dei tipi in `jolie.lang.parse.ast.types`

In questo paragrafo si descrive come le definizioni dei tipi del linguaggio Jolie siano rappresentate internamente tramite il package `jolie.lang.parse.ast.types`. Questo package fornisce un'API strutturata per il parsing e l'analisi semantica degli alberi di sintassi relativi esclusivamente alle definizioni di tipi.

Struttura del package Il package implementa classi per:

- Le definizioni di tipi elementari e complessi.
- I tipi inline, che rappresentano direttamente la struttura di un tipo.
- I tipi somma, che definiscono un tipo come l'unione di alternative.

- I riferimenti a tipi esterni, gestiti tramite link.

Gerarchia delle classi e metodi principali La rappresentazione interna dei tipi si fonda su una gerarchia di classi. La classe base `TypeDefinition` definisce l'interfaccia comune a tutte le definizioni di tipo e comprende i metodi `name()` che ritorna il qualificatore del tipo, `node()` che ritorna l'AST della definizione del tipo e `cardinality()` che ritorna la cardinalità del tipo.

La classe base, `TypeDefinition`, fornisce metodi comuni quali:

- `getName()`: restituisce il nome del tipo;
- `validate()`: metodo astratto che ogni classe derivata implementa per verificare la correttezza strutturale del tipo.

Le classi derivate da `TypeDefinition` sono:

- `TypeChoiceDefinition`: viene utilizzato per implementare i tipi somma. Implementa i metodi:
  - `left()`: ritorna la definizione del tipo alla sinistra dell'operatore `|`;
  - `right()`: ritorna la definizione del tipo alla destra dell'operatore `|`.
- `TypeDefinitionLink`: per il riferimento a tipi composti all'interno di altri tipi composti; la sua interfaccia include i metodi:
  - `linkedTypeName()`: ritorna il nome del tipo riferito;
  - `linkedType()`: ritorna la definizione del tipo riferito.
- `TypeInlineDefinition`:
  - `subTypes()`: ritorna un insieme di definizioni di tipi, accoppiando ciascuna definizione al nome del tipo stesso;
  - `basicType()`: ritorna il tipo di base.

  Questa classe viene estesa a sua volta da `TypeDefinitionUndefined`, che permette il supporto alle variabili di tipi `undefined`.

3.2.2 Primitive di comunicazione

Le operazioni in Jolie possono essere divise in due categorie:

- Input

- Output

Primitive di input

Riguardano le invocazioni ricevute e, a loro volta si distinguono in due tipi:

- OneWay: Permette la ricezione asincrona di un'invocazione;
- RequestResponse: Permette la ricezione, la gestione e la risposta alle invocazioni.

Primitive di output

Analogamente alle primitive di input, esistono due tipi di operazioni di output:

- Notification: Permette l'invio di una richiesta;
- SolicitNotification: Permette l'invio di una richiesta, ma blocca il servizio fino all'arrivo della risposta.

Rappresentazione delle primitive di comunicazione in `jolie.lang.parse.ast`

In questa sottosezione vengono descritte le primitive di comunicazione di Jolie, modellate tramite le classi del package `jolie.lang.parse.ast`. Tali classi definiscono le modalità con cui un servizio Jolie interagisce, distinguendo in particolare tra operazioni one-way e request-response.

Gerarchia delle classi e metodi principali La classe astratta `OperationDeclaration` rappresenta la base comune per tutte le operazioni di comunicazione e fornisce metodi generici quali:

- `id()`: restituisce il nome dell'operazione.

Le due sottoclassi derivanti da `OperationDeclaration` sono:

- `OneWayOperationDeclaration`:
  - `requestType()`: ritorna la definizione al tipo della richiesta;
- `RequestResponseOperationDeclaration`:
  - `requestType()`: ritorna la definizione al tipo della richiesta;
  - `responseType()`: ritorna la definizione al tipo della risposta;
  - `process()`: valido solo nelle operazioni di tipo RequestResponse. Contiene il riferimento all'albero sintattico per il calcolo della response.

3.2.3 Interfacce

Le interfacce sono una collezione di operazioni. Poiché la distinzione tra operazione di input e operazione di output è legata all'inserimento dell'operazione in una porta di input o di output, le interfacce distinguono solamente tra operazioni OneWay e operazioni RequestResponse. In un'interfaccia, ogni operazione è caratterizzata da un nome e da una coppia di tipi: uno per la richiesta in ingresso e uno per l'eventuale risposta in uscita

Listing 3.4: Definizione di due interfacce per il servizio WeatherForecast. Nell'interfaccia WeatherInterface viene dichiarata un'operazione RequestResponse `getForecast` con tipi `ForecastPublicRequest` in ingresso e `ForecastResponse` in uscita. Nell'interfaccia `LogInterface` viene dichiarata un'operazione OneWay `save` che accetta stringhe.

```
1 interface WeatherInterface {
2     RequestResponse:
3         getForecast( ForecastPublicRequest )(
              ↪ ForecastResponse )
4 }
5 interface LogInterface {
6     OneWay:
7         save ( string )
8 }
```

In fase di esecuzione, quando un servizio Jolie riceve una richiesta sull'operazione `getForecast`, l'ambiente verificherà che il tipo del messaggio in ingresso corrisponda a quanto definito in ForecastPublicRequest. Allo stesso modo, prima di inviare la risposta, controllerà che questa corrisponda al tipo dichiarato in ForecastResponse.

Rappresentazione delle interfacce in `jolie.lang.parse.ast`

La rappresentazione delle interfacce in Jolie è gestita dalla classe `InterfaceDefinition` all'interno del package `jolie.lang.parse.ast`. Questa classe modella un'interfaccia come una collezione di operazioni, distinguendo implicitamente tra operazioni di tipo `OneWay` e `RequestResponse`.

Struttura e metodi principali La classe `InterfaceDefinition` offre metodi fondamentali per:

- `name()`: restituisce il nome dell'interfaccia;
- `operationsMap()`: ritorna la mappa contenente le operazioni. Queste vengono accedute tramite il loro nome.

Le operazioni contenute nell'interfaccia sono rappresentate come istanze di classi derivate da `OperationDeclaration` (`OneWayOperationDeclaration` o `RequestResponse OperationDeclaration`), discusse nella Sottosezione 3.2.2.

Utilità nell'analisi del codice L'utilizzo di `InterfaceDefinition` permette a JolieGraph di estrarre in modo automatico il modello delle interazioni di un servizio, verificando:

- La correttezza dei tipi associati alle operazioni, sia per le richieste che per le risposte.
- La distinzione tra operazioni che non richiedono risposta e quelle che prevedono un ciclo di richiesta e risposta.
- L'integrazione coerente delle interfacce nel modello complessivo dei servizi.

3.2.4 Porte

Le porte sono costrutti fondamentali per definire i canali di comunicazione tra i servizi. Esse rappresentano gli endpoint attraverso i quali un servizio può conversare e vengono identificate da un nome. Le porte specificano diversi parametri, quali:

- **Location**: l'indirizzo URI dell'endpoint (ad es. un indirizzo socket o URL);
- **Protocol**: il protocollo utilizzato per la comunicazione;
- **Interfaces**: l'insieme delle interfacce che definiscono le operazioni accessibili attraverso quella porta.

Esistono due tipi di porte:

- **Porte di input**: vengono utilizzate per ricevere messaggi in ingresso, definendo come il servizio si espone verso l'esterno;
- **Porte di output**: sono utilizzate per l'invio di messaggi verso servizi esterni, specificando dove e come i messaggi devono essere recapitati.

Questa distinzione permette di separare le responsabilità di ricezione e invio.

Listing 3.5: Definizione di tre porte: nella prima, di input, si fa riferimento esplicito all'interfaccia; nella seconda e nella terza porta, entrambe di output, l'interfaccia viene definita implicitamente e vengono direttamente elencate le operazioni al suo interno.

```
1 inputPort PublicInput {
2     location: "socket://localhost:8000"
3     protocol: http
4     interfaces:
```

```
5           WeatherInterface
6 }
7 outputPort LocationResolver {
8     location: "socket://localhost:8005"
9     protocol: http
10    requestResponse:
11        resolve( string )( Coord )
12 }
13 outputPort ForecastProvider {
14    location: "socket://localhost:8010"
15    protocol: http
16    requestResponse: getForecast ( ForecastPrivateRequest
          ↪ )( ForecastResponse)
17 }
```

Embedding dei servizi

L'embedding dei servizi in Jolie permette di integrare, in modo trasparente, le porte di un servizio *embedded* all'interno del servizio *embedder*. Questo meccanismo consente di riutilizzare le definizioni di porte definite nel servizio embeddato, rendendoli accessibili dal servizio che le include. La sintassi di questa istruzione è:

```
embed <nomeServizio> ( valore ) [ in <nomePortaEsistente> | as
                    <nomeNuovaPorta> ] }
```

Se la parte di binding (sezione tra parentesi quadre) viene omessa, il servizio verrà avviato senza effettuarne un collegamento automatico ad una porta del servizio embedder. Il servizio embedded sarà comunque raggiungibile con i soliti meccanismi. Indicando, invece, l'argomento tra parentesi quadre, il programmatore Jolie puó specificare verso quale porta collegare il servizio. Utilizzando la parola chiave **in** il servizio embedded verrà collegato alla porta esistente `nomePortaEsistente`, mentre con **as** il servizio viene collegato ad una nuova porta di `nomeNuovaPorta`.

Rappresentazione delle Porte in `jolie.lang.parse.ast`

La rappresentazione delle porte nel codice Jolie è gestita tramite le classi `PortInfo`, `InputPortInfo` e `OutputPortInfo` presenti nel package `jolie.lang.parse.ast`. Queste classi modellano gli endpoint di comunicazione, definendo proprietà essenziali come location, protocollo e interfacce associate.

Gerarchia delle classi e metodi principali La classe astratta `PortInfo` funge da base comune per entrambe le tipologie di porte e definisce metodi generali quali:

- `id()`: restituisce l'identificatore della porta;
- `operationsMap()`: dato il nome di un'operazione ne ritorna la sua descrizione;
- `getInterfaceList()`: ritorna la lista delle interfacce di cui la porta implementa le operazioni.

Le classi che estendono `PortInfo` sono:

- `InputPortInfo`;
- `OutputPortInfo`.

Specializzate rispettivamente per le porte in ingresso e in uscita. Entrambe le classi implementano i metodi

- `location()`: ritorna l'endpoint della porta;
- `protocolId()`: ritorna il nome del protocollo utilizzato;
- `protocol()`: ritorna un albero sintattico contenente i parametri del protocollo specificati.

3.3 Analisi del Service Behaviour

Il *service behaviour*, behaviour da qui in poi, definisce il comportamento esecutivo del servizio Jolie, ovvero la logica che regola l'elaborazione delle richieste e la gestione delle interazioni con l'esterno. In questa sezione vengono analizzati i principali costrutti del blocco behaviour, utili alla comprensione di JolieGraph (illustrato nel prossimo capitolo), corredati da esempi esplicativi. Per ciascun costrutto viene inoltre illustrata la relativa rappresentazione tramite la libreria `jolie.lang`.

3.3.1 Assegnamento

L'assegnamento in Jolie segue la sintassi:

```
1 id = expression
```

dove `id` è il nome della variabile che conterrà il risultato della valutazione di `expression`.

3.3.2 Invocazioni di operazioni

Ognuna della quattro primitive di comunicazione utilizza una sintassi diversa dalle altre per la propria invocazione:

```
1 notification@outputPort( request )
2 solicitNotification@outputPort( request ) ( response )
3 oneWay( request )
4 requestResponse( request )( response ) { process }
```

Per le operazioni in uscita (righe 1, 2) bisogna specificare la porta di output a cui appartengono. Questa informazione viene omessa per le porte di input (righe 3, 4). Infine, nelle richieste di tipo `requestResponse` è presente un blocco `process` che contiene le istruzioni per il calcolo della risposta alla richiesta ricevuta.

3.3.3 Sequenza

Il costrutto di **sequenza** permette di eseguire una serie di istruzioni in ordine, garantendo che ciascuna operazione venga completata prima dell'invocazione della successiva.

Listing 3.6: Esempio di sequenza nel blocco behaviour

```
1 main {
2     println@Console("1")();
3     println@Console("2")();
4     println@Console("3")();
5 }
```

Rappresentazione della sequenza in `jolie.lang`

La libreria `jolie.lang` rappresenta il costrutto di sequenza tramite la classe `SequenceStatement` (disponibile nel package `org.jolie.lang.parse.ast`), che implementa l'interfaccia `Statement`. Il metodo principale di quest'interfaccia è:

- `children()`: restituisce la lista delle istruzioni che compongono la sequenza.

3.3.4 Scelta condizionale deterministica (if)

Il costrutto `if` permette di eseguire un ramo specifico in base al risultato della valutazione di un'espressione booleana. A differenza del costrutto riportato nella Sottosezione 3.3.6, questo costrutto garantisce un comportamento deterministico.

Listing 3.7: Esempio di scelta condizionale deterministica

```
1 main {
2     if ( temperature > 25 ) {
```

```
3         println@Console("Giornata calda")();
4     } else {
5         println@Console("Giornata non troppo calda")();
6     }
7 }
```

Rappresentazione della scelta condizionale in `jolie.lang`

Nella libreria `jolie.lang`, il costrutto `if` è rappresentato dalla classe `IfStatement` (nel package `org.jolie.lang.parse.ast`). I principali metodi messi a disposizione da questa classe per rappresentare le strutture condizionali sono:

- `children()`: restituisce una lista di coppie (`Pair<OLSyntaxNode, OLSyntaxNode>`). In ciascuna coppia il primo elemento contiene l'espressione condizionale da verificare (per il ramo `if` e per gli eventuali rami "`else if`"), mentre il secondo elemento contiene il blocco di codice associato a tale condizione. Questo meccanismo consente di rappresentare, in un unico nodo, una catena di condizioni alternative, come indicato nel Listing 3.8.
- `elseProcess()`: restituisce il riferimento al nodo dell'albero in cui è contenuto il codice del blocco else. Se questo blocco non viene specificato, il metodo ritorna `null`.

Listing 3.8: Esempio di scelta condizionale multipla.

```
1 main {
2     if ( temperature > 35 )
3         println@Console("Giornata molto calda")()
4     else if ( temperature > 25 )
5         println@Console("Giornata calda")()
6     else if ( temperature > 15 )
7         println@Console("Giornata mite")()
8     else
9         println@Console("Giornata fredda")()
10 }
```

3.3.5 Costrutti Iterativi

In Jolie sono previsti diversi costrutti per l'iterazione, che permettono di eseguire ripetutamente un blocco di codice in funzione di condizioni o di strutture dati. In particolare, sono disponibili:

- Iterazione indefinita:

- **While**:

```
while ( condition ) {
    body
}
```

- **For**: La sintassi del costrutto `for` in Jolie è strutturata in modo analogo a quella di linguaggi come C e Java. Sebbene l'identificatore `for` possa far presupporre l'impiego di un iteratore per gestire cicli indefiniti, in Jolie esso assume prevalentemente la funzione di una utility sintattica che consente di definire in maniera compatta sia il blocco di inizializzazione sia quello di post-esecuzione.

```
for ( init ; condition ; post ) {
    body
}
```

- Iterazione definita: In jolie è possibile iterare con il costrutto `foreach` su liste o su sottonodi di un nodo.

```
for ( element in array ) {
    body
}
foreach ( subnode: node ) {
    body
}
```

Rappresentazione dei costrutti iterativi in `jolie.lang`

La libreria `jolie.lang` rappresenta i diversi costrutti iterativi mediante classi specifiche:

- `ForStatement` per il ciclo `for`. La classe `ForStatement`, che implementa questo costrutto, espone quattro metodi, utili per accedere alle diverse sezioni del blocco, questi sono: `init()`, `condition()`, `post()` e `body()`.
- `WhileStatement` per il ciclo `while`. L'interfaccia della classe espone due metodi: `condition()` e `body()` per ottenere i riferimenti ai relativi nodi dell'AST.
- `ForEachArrayItemStatement` per l'iterazione su array. Il riferimento all'AST del blocco di codice iterato è ottenibile dal metodo `body()`.

- `ForEachSubNodeStatement` per l'iterazione sui sotto-nodi. Anche in questo caso è presente un metodo `body()` che ritorna il nodo dell'albero sintattico iterato.

3.3.6 Scelta non deterministica

La scelta non deterministica (o *input choice*) permette al servizio di attendere in parallelo l'arrivo di input su operazioni differenti, eseguendo il branch associato all'operazione che riceve per prima un input. In altre parole, non essendoci un ordine fisso per la ricezione degli input, il comportamento del servizio risulta non deterministico. Ad ogni condizione, che corrisponde all'attesa di una specifica operazione, è associato un branch: il blocco di istruzioni che viene eseguito subito dopo la gestione dell'operazione ricevuta.

Listing 3.9: Esempio di scelta non deterministica. L'esecuzione del servizio rimane bloccata finchè questo non riceve una richiesta `op1` o `dammi42`. L'invio della richiesta `println` (riga 8) avviene successivamente all'invio della *response* dell'operazione `dammi42` (riga 6).

```
1  main {
2      [op1()] {
3          println@Console("Operazione 1 ricevuta")()
4      }
5      [dammi42( )( response ) {
6          response = 42
7      }] {
8          println@Console("Operazione 2 ricevuta")()
9      }
10 }
```

Rappresentazione della scelta non deterministica in `jolie.lang` (NDChoiceStatement)

La classe `NDChoiceStatement`, definita nel package `org.jolie.lang.parse.ast`, rappresenta il costrutto non deterministico che gestisce le input choices. In questo contesto, la struttura del costrutto è composta da una serie di coppie, ognuna delle quali associa una condizione (cioè l'operazione in attesa di input) al branch, ovvero il blocco di istruzioni da eseguire una volta ricevuto l'input corrispondente.

Il metodo principale offerto dalla classe è:

- `children()`: restituisce una lista di coppie (`Pair<OLSyntaxNode, OLSyntaxNode>`). In ciascuna coppia il primo elemento rappresenta l'operazione che specifica

l'input da attendere, mentre il secondo elemento contiene il *branch* associato, ovvero il blocco di codice da eseguire in seguito.

3.3.7 ProvideUntil

Il blocco `provide until` è una struttura del linguaggio Jolie che consente di definire due gruppi distinti di operazioni:

- **Il blocco `provide`:** contiene una o più operazioni rese continuamente disponibili. Quando la gestione di una di queste operazioni termina, essa viene automaticamente ripristinata e rimessa a disposizione per eventuali nuove invocazioni.
- **Il blocco `until`:** contiene una o più operazioni che, una volta invocate, interrompono il ciclo di disponibilità delle operazioni definite in `provide`.

Il comportamento complessivo risulta simile a una scelta non deterministica (input choice): il sistema è in ascolto simultaneamente per le invocazioni provenienti da entrambi i blocchi e la prima richiesta ricevuta, sia essa relativa a un'operazione in `provide` oppure in `until`, determina il flusso di esecuzione. In particolare, le operazioni definite nel blocco `provide` vengono continuamente riproposte dopo ogni gestione, a meno che non venga eseguita un'operazione definita nel blocco `until` che interrompa definitivamente tale disponibilità.

Listing 3.10: In questo esempio l'operazione `dammi42` sarà disponibile e invocabile più volte fino alla ricezione dell'operazione `shutdown`.

```
1 main {
2     provide
3         [dammi42( )( response ) { response = 42 }]
4     until
5         [shutdown( )]{ exit }
6 }
```

Rappresentazione in `jolie.lang`

Dal punto di vista della rappresentazione interna, il blocco `provide until` viene modellato in Jolie attraverso la classe `ProvideUntilStatement`. Questa classe fa parte dell'*Abstract Syntax Tree* (AST) generato durante il parsing del codice Jolie e la sua interfaccia fornisce due metodi principali:

- `provide()`: ritorna il riferimento all'AST delle operazioni presenti nel blocco provide;

- `until()`: ritorna il riferimento all'AST contenente le operazioni presenti nel blocco omonimo.

3.3.8 Le procedure in *Jolie*

Le **procedure** in Jolie sono blocchi di codice riutilizzabili, simili a sottoprogrammi interni al servizio, che consentono di strutturare e organizzare la logica dell'applicazione. A differenza di molti linguaggi mainstream, in Jolie le procedure non prevedono un proprio scope locale: non è possibile definire parametri di ingresso né restituire valori, e le variabili utilizzate al loro interno appartengono allo *stesso ambiente globale* del servizio. In altre parole, tutte le procedure condividono lo stato (le variabili) dell'istanza di servizio in cui vengono eseguite, e ogni modifica a una variabile all'interno di una procedura è visibile anche all'esterno di essa (nel `main` o in altre procedure).

Per definire una procedura personalizzata si utilizza la sintassi:

```
define <nomeProcedura> { <processo> }
```

dove `<processo>` rappresenta una sequenza di istruzioni Jolie (eventualmente vuota). Una procedura così definita può essere richiamata semplicemente usando il suo nome come istruzione. Jolie supporta la dichiarazione di un numero arbitrario di procedure ausiliarie (*auxiliary procedures*) oltre alla procedura principale, permettendo al programmatore di suddividere il comportamento del servizio in parti più modulabili e riutilizzabili.

Due procedure hanno un ruolo particolare nel ciclo di vita di un servizio Jolie: `init` e `main`. La procedura principale `main` è il punto di ingresso dell'esecuzione di ogni servizio Jolie: il suo blocco di codice viene eseguito all'avvio del microservizio e ne rappresenta il flusso di controllo primario. La procedura iniziale `init`, se presente, è una procedura speciale che permette di effettuare operazioni di inizializzazione, in quanto viene eseguita automaticamente all'avvio, subito prima del `main`. Tipicamente `init` viene utilizzata per effettuare operazioni di inizializzazione globali (ad esempio impostare variabili di configurazione, inizializzare connessioni, ecc.). Sia `init` che `main` si definiscono con sintassi dedicata (`init {...}`, `main {...}`) invece di usare 'define', ma al pari delle altre procedure non hanno parametri espliciti. Dopo l'esecuzione di `init` (quando presente), il runtime Jolie avvia la procedura `main`; al suo interno, il codice può eseguire operazioni di I/O, creare nuovi processi concorrenti e anche invocare eventuali procedure ausiliarie definite dal programmatore.

Listing 3.11: Esempio di un servizio Jolie con procedura `init`, procedura `main` e una procedura ausiliaria definita dal programmatore. Il servizio, una volta lanciato, stamperà 40 e terminerà.

```
1 from console import Console
2 service Raddoppia{
3     embed Console as Console
4     init {
5         x = 10
6     }
7     main {
8         raddoppia
9         raddoppia
10        println@Console("Valore finale di x: " + x)
11    }
12    define raddoppia {
13        x = x * 2
14    }
15 }
```

Rappresentazione delle procedure in `jolie.lang.parse.ast.DefinitionNode`

All'interno dell'implementazione di Jolie, le procedure definite nel codice vengono rappresentate nel *Abstract Syntax Tree* (AST) tramite istanze della classe `DefinitionNode` (parte del package `jolie.lang.parse.ast`). In particolare, ogni costrutto `define` di una procedura genera un nodo `DefinitionNode` nell'AST. La classe `DefinitionNode` estende `OLSyntaxNode` (la classe base per tutti i nodi sintattici di Jolie) e contiene le informazioni essenziali sulla procedura, tra cui:

- **Identificatore** – il nome della procedura (`id`), ad esempio `"raddoppia"` oppure `"init"`.
- **Corpo** – il nodo AST che rappresenta il corpo della procedura (`body`), ossia la sequenza di istruzioni racchiuse nelle sue parentesi graffe.

I metodi principali offerti da `DefinitionNode` permettono di accedere a tali componenti: `id()` restituisce il nome della procedura, mentre `body()` restituisce il nodo corrispondente al suo corpo.

3.3.9 Modalità di esecuzione di un servizio

Un servizio Jolie può essere eseguito in tre modalità, ciascuna delle quali influisce sul comportamento del servizio:

- **Single**: il blocco behaviour viene eseguito una sola volta; al termine dell'esecuzione del `main` il servizio termina e non vengono effettuate ulteriori invocazioni.
- **Sequential**: il blocco behaviour viene eseguito ripetutamente in sequenza. Una volta terminata l'esecuzione del `main`, il servizio riparte dall'inizio di tale procedura, garantendo così una nuova iterazione del comportamento.
- **Concurrent**: il servizio è in grado di gestire più richieste contemporaneamente. Per ciascuna richiesta ricevuta viene istanziato un nuovo contesto di esecuzione, permettendo l'elaborazione parallela dei comportamenti.

Rappresentazione delle modalità di esecuzione in `jolie.lang`

La libreria `jolie.lang` rappresenta le modalità di esecuzione del blocco behaviour utilizzando un'enumerazione, in cui:

- **Single** corrisponde a `ExecutionMode.SINGLE`;
- **Sequential** corrisponde a `ExecutionMode.SEQUENTIAL`;
- **Concurrent** corrisponde a `ExecutionMode.CONCURRENT`.

3.4 Gestione dei *fault* in Jolie

In Jolie, i *fault* rappresentano un meccanismo di gestione degli errori, analogo alle eccezioni in altri linguaggi. Un fault è definito come un segnale di errore, è identificato da un nome, che viene sollevato (*thrown*) da un comportamento verso lo *scope* contenitore quando si verifica una condizione di errore, allo scopo di permettere il recupero (*recovery*) da tale errore. Il costrutto sintattico fornito da Jolie per lanciare esplicitamente un fault è l'istruzione `throw`. Un fault lanciato ma non gestito all'interno dello scope corrente viene automaticamente propagato (*re-throw*) allo scope esterno (quello che ha creato lo scope corrente); in ultima istanza, un fault non intercettato nel `main` (lo scope principale) viene segnalato come errore all'esterno del servizio.

La gestione dei fault avviene definendo dei fault handler (gestori di fault) associati dinamicamente agli scope durante l'esecuzione. Jolie mette a disposizione l'istruzione `install` per installare uno o più gestori di fault all'interno di uno scope. La sintassi generale è la seguente:

```
1 install(
2     FaultName_1 => /* handler code*/,
3     ...,
```

```
4     FaultName_N => /* handler code*/
5 )
```

In pratica, `install` associa a ciascun nome di fault specificato un blocco di codice (il *processo* di gestione) che deve essere eseguito qualora quello specifico fault venga sollevato all'interno dello scope corrente. Quando un fault viene lanciato tramite `throw`, l'interprete Jolie verifica se nello scope corrente sia presente un handler installato per quel nome di fault: in caso affermativo, esegue il codice di gestione associato e considera il fault come gestito (impedendo al fault di propagarsi oltre lo scope); se invece non esiste un handler per quel fault, il segnale di errore viene propagato allo scope esterno. È possibile anche che un handler, durante la sua esecuzione, decida di rilanciare il fault (mediante un ulteriore `throw`) per propagare l'errore allo scope superiore, dopo aver eventualmente effettuato delle operazioni di recupero locali. In presenza di più handler (anche su scope diversi) per lo stesso fault, l'ordine di esecuzione dei gestori segue la gerarchia degli scope annidati: prima viene eseguito l'handler definito nello scope più interno dove il fault è stato lanciato, e solo se questo rilancia il fault si attiverà l'handler definito a livello superiore (come illustrato in seguito nell'esempio). Inoltre, Jolie risolve possibili condizioni di competizione tra la registrazione di un handler e il lancio concorrente di un fault assegnando priorità alle istruzioni `install` rispetto ai `throw` in parallelo: ciò garantisce che in un costrutto parallelo del tipo `throw(...)|install(...)` il gestore venga installato prima che il fault sia propagato allo scope, assicurando un comportamento deterministico.

Nel listato 3.12 è mostrato un esempio di codice Jolie relativo alla gestione dei fault. Nello `scope` principale `main` viene installato un gestore per un fault chiamato `CustomFault`; all'interno di uno scope annidato (`innerScope`), si installa un altro gestore per lo stesso fault. Infine, viene lanciato il fault `CustomFault` nello scope interno.

Listing 3.12: Esempio di installazione di un fault handler e lancio di un fault in Jolie

```
1 from console import Console
2
3 main {
4     install(
5         CustomFault => println@Console( "Gestore␣fault␣
              ↪ CustomFault␣in␣main" )()
6     )
7     scope( innerScope ){
8         install(
9             CustomFault => println@Console( "Gestore␣
                  ↪ fault␣CustomFault␣in␣innerScope" )()
```

```
10          )
11          throw ( CustomFault )
12      }
13      throw ( CustomFault )
14  }
```

Nel codice sopra, quando l'istruzione `throw(CustomFault)` viene eseguita all'interno di `innerScope`, il fault `CustomFault` viene catturato dal gestore installato nello stesso `innerScope`, il quale stampa un messaggio su console. All'uscita dallo scope interno, è presente un ulteriore fault, il cui gestore, stavolta, è quello definito all'inizio della procedura `main`.

3.4.1 Rappresentazione in `jolie.lang`

Internamente, il linguaggio Jolie rappresenta le operazioni di gestione dei fault attraverso specifiche classi nella propria implementazione. In particolare, le classi responsabili della gestione dei fault in `jolie.lang` sono `InstallStatement` e `ThrowStatement`, che fungono da nodi dell'*Abstract Syntax Tree* per le istruzioni `install` e `throw` rispettivamente.

La classe `InstallStatement` modella un'istruzione `install` e contiene al suo interno la struttura dei fault handler associati. Tale struttura è rappresentata da un oggetto `InstallFunctionNode`, il quale incapsula un insieme di coppie (`<fault_name>, <fault_handler_code>`). Ciascuna coppia associa il nome di un fault (di tipo stringa) al corrispondente blocco di istruzioni che costituisce il gestore per quel fault, rappresentato anch'esso come un nodo sintattico (derivato dalla classe base `OLSyntaxNode`). In questo modo, durante la fase di esecuzione, l'interprete può registrare dinamicamente questi handler nell'ambito dello scope corrente: il nodo `InstallStatement` fornisce le informazioni necessarie per collegare ogni nome di fault al relativo codice di gestione.

Analogamente, la classe `ThrowStatement` rappresenta nel parsing tree ogni occorrenza dell'istruzione `throw`. In essa vengono memorizzati: (i) l'identificatore (nome) del fault da lanciare e (ii) un eventuale parametro o valore associato al fault, qualora il lancio del fault includa un'espressione (come in `throw(FaultName, expr)`). L'eventuale espressione fornita viene mantenuta come nodo figlio di tipo `OLSyntaxNode` all'interno del `ThrowStatement`. Questa rappresentazione interna consente all'interprete Jolie di identificare il tipo di segnale di errore da generare (tramite il nome del fault) e di valutare/trasportare un dato aggiuntivo se presente, al momento in cui l'istruzione `throw` viene eseguita.

Capitolo 4

JolieGraph

4.1 Introduzione

JolieGraph[1] è un'applicazione Java composta da circa 2500 righe di codice progettata per eseguire un'analisi statica del codice sorgente di servizi Jolie, al fine di generare automi rappresentanti il flusso di comunicazione tra i servizi coinvolti. Per comprendere meglio JolieGraph vengono presentati un servizio **ForecastService** e il relativo output.
Il servizio fa parte di un sistema che consente agli utenti di rilevare le condizioni meteo di un indirizzo in uno specifico giorno. Diverse porte di input vengono usate al fine di filtrare l'accesso verso operazioni privilegiate (`shutdown` nell'esempio). Una volta avviato il servizio si entra nel blocco **ProvideUntil**, permettendo al servizio di gestire la ricezione dell'operazione `getForecast` fino all'arrivo di una richiesta di tipo `shutdown` che interrompe l'esecuzione del programma.

```
1 from .logs import Logs
2 from .locationResolver import LocationResolver
3 type Coord:void {
4     .lat: double
5     .lon: double
6  }
7 type ForecastPublicRequest:void {
8     .date: string
9     .address: string
10 }
```

[1]JolieGraph è disponibile sotto licenza GPL 2.0 su `https://github.com/francesco-licchelli/joliegraph/tree/master`.

```
type ForecastPrivateRequest:void {
    .coord: Coord
    .date: string
}
type ForecastResponse:void {
    .temperature: double
    .humidity: double
}
service WeatherForecast {
    embed Logs as Logs
    embed LocationResolver as LocationResolver
    inputPort PublicInput {
        location: "socket://localhost:8000"
        protocol: http
        requestResponse:
            getForecast( ForecastPublicRequest )(
              ↪ ForecastResponse )
    }
    inputPort ManagementInput {
        location: "socket://localhost:8001"
        protocol: http
        oneWay:
            shutdown( void )
    }
    outputPort ForecastProvider {
        location: "socket://localhost:8010"
        protocol: http
        requestResponse:
            getForecast ( ForecastPrivateRequest )(
              ↪ ForecastResponse )
    }
    main {
        provide
            [getForecast( request )( response ) {
                resolve@LocationResolver( request.address
                  ↪  )( coord )
                getForecast@ForecastProvider( { coord =
                  ↪ coord, date = request.date } )(
                  ↪ response )
            }]{
```

```
46                      save@Logs( request )
47                  }
48              until
49                  [shutdown( )]{
50                      exit
51                  }
52      }
53  }
```

All'avvio del servizio `WeatherForecast` questo si trova nello stato 1. Se riceve una richiesta di tipo `shutdown` percorre l'arco $(1, 3)$ arrivando ad uno stato finale. Altrimenti, quando riceve una richiesta di tipo `getForecast` (arco $(1, 2)$ il servizio procede alla risoluzione dell'indirizzo in coordinate (archi $(2, 4)$, $(4, 5)$). Una volta ottenute le coordinate, si procede con la richiesta delle previsioni meteo per il giorno e il luogo specificato e l'invio della risposta all'utente (archi $(5, 6)$, $(6, 7)$, $(7, 8)$). Infine, `WeatherForecast` invia un'operazione `save` (arco $(8, 1)$). Nell'automa presentato vengono anche indicati i tipi delle richieste. Nel caso di tipi composti, vengono mostrati anche i tipi che li compongono.

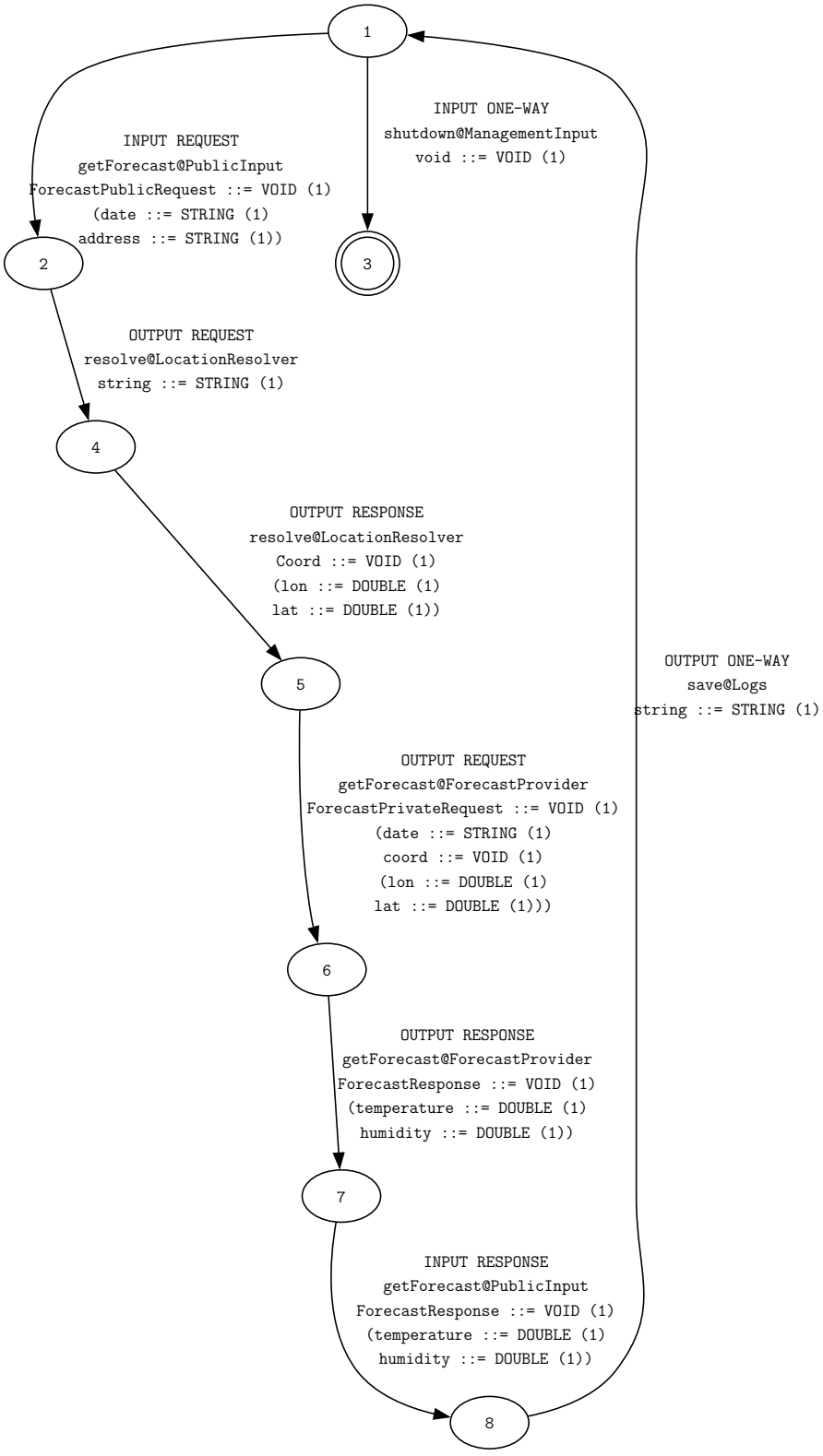


Figura 4.1: Automa generato dal servizio `ForecastService`. JolieGraph è stato eseguito con l'opzione `-T`.

L'analisi di un servizio viene condotta attraversando l'albero di sintassi astratta (AST) generato dal parser presente in `jolie.lang`. L'esecuzione di JolieGraph è scomponibile in due fasi:

1. **Fase di raccolta dei simboli**: In questa prima fase vengono ricercate le dichiarazioni di tipi, operazioni, interfacce, porte e servizi presenti all'interno del programma. Le informazioni contenute in tali definizioni vengono riorganizzate per accedervi in maniera più semplice durante la fase successiva;
2. **Fase di generazione dell'automa**: per ogni servizio rilevato ne si esplora l'albero di sintassi astratta e sulla base di questo viene generato un automa che ne rappresenta il flusso di comunicazione.

Questo procedimento consente di ricostruire il modello di comunicazione tra servizi Jolie, facilitando lo studio delle loro interazioni e l'individuazione di potenziali criticità nel flusso informativo.

4.1.1 Installazione e Dipendenze

L'applicazione JolieGraph è basata su Java ed è gestita tramite Maven. Per installarla e utilizzarla, è necessario disporre delle seguenti dipendenze:

- 'commons-cli' (1.5.0) - Per la gestione delle opzioni a riga di comando.
- 'junit' (3.8.1) - Per i test.
- 'libjolie' (1.12.x) - Per l'interazione con Jolie.
- 'jgrapht-core' (1.5.2) e 'jgrapht-io' (1.5.2) - Per la gestione e la visualizzazione dei grafi.
- 'slf4j-api' (2.0.16) e 'logback-classic' (1.4.12) - Per la gestione dei log.

4.1.2 Utilizzo del Tool

Il tool JolieGraph è sviluppato in Java e sfrutta Maven per la compilazione e la gestione delle dipendenze.

Compilazione

Per compilare il tool, una volta raggiunta la root del progetto, si esegue il seguente comando:

```
mvn clean package
```

Questo comando esegue il processo di pulizia, compila il codice sorgente e crea un file JAR eseguibile nella cartella `target`.

Esecuzione

Una volta compilato, JolieGraph può essere eseguito come un'applicazione Java standalone. Il comando di base per avviare l'applicazione è:

```
java -jar JolieGraph.jar [options]
```

4.1.3 Opzioni Disponibili

Parametri Obbligatori

- `-i, --input`: Specifica il percorso del file contenente il servizio da analizzare.

Parametri Opzionali

- `-T, --full-type`: Visualizza le etichette dei nodi in modo ricorsivo, mostrando i sottotipi dei tipi compositi.
- `-S, --stdlib`: Include anche i servizi definiti nella standard library di Jolie.
- `-o, --output`: Specifica il percorso in cui verranno salvate le viste generate. Se non indicato, il percorso di default è `./generated`.

4.2 Costruzione dell'Automa

La creazione dell'automa avviene in maniera induttiva sui nodi dell'AST del servizio da analizzare. Nei casi base rientrano tutti quei nodi da cui non possono avere origine comunicazioni, non interessanti, poichè ritornano grafi vuoti, e dalle quattro primitive di comunicazione messe a disposizione da Jolie. I passi induttivi sono invece composti da tutti quei costrutti Jolie al cui interno possono essere presenti chiamate alle operazioni sopracitate.

Nel proseguo della sezione verrà illustrato il procedimento adottato da JolieGraph per la creazione degli automi relativi a ciascuno statement supportato. In particolare, verrà presentato l'NFA creato per ogni statement e, successivamente, mostrato un esempio specifico. Nei casi in cui l'automa di uno statement sia ottenuto dalla composizione di altri automi, i nodi finali di questi ultimi non vengono più considerati come tali.

4.2.1 Primitive di Comunicazione

Primitiva `OneWay`

La primitiva `OneWay` consente la ricezione di richieste unidirezionali. L'automa corrispondente è rappresentato nella Figura 4.2:

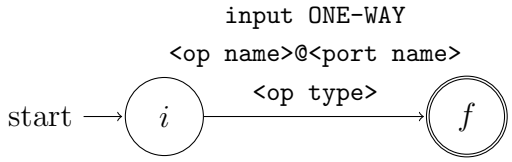


Figura 4.2: NFA associato a `OneWayOperationStatement`

Primitiva `Notification`

La primitiva `Notification` è speculare a `OneWay`, in quanto permette l'invio di richieste unidirezionali verso altri servizi. L'automa generato da questa classe di operazioni viene illustrato nella Figura 4.3:

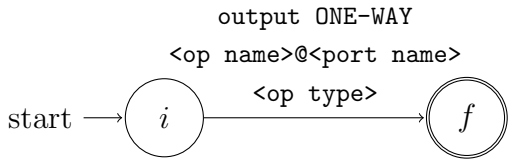


Figura 4.3: NFA associato a `NotificationOperationStatement`

Primitiva `RequestResponse`

La primitiva `RequestResponse` consente di ricevere richieste e di inviare le relative risposte. Sia G_p l'automa del calcolo della response dell'operazione, allora l'automa associato ad una richiesta di questo tipo è:

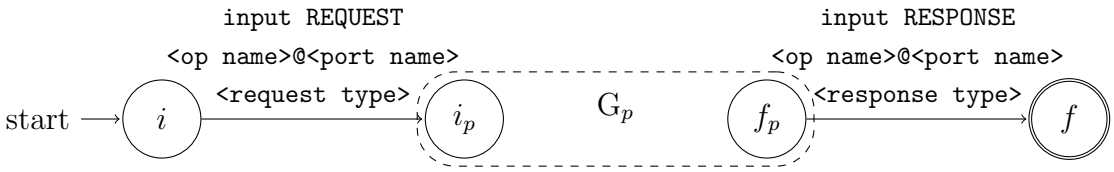


Figura 4.4: NFA associato a `RequestResponseOperationStatement`

Primitiva `SolicitNotification`

La primitiva `SolicitNotification` permette, in maniera sincrona, l'invio di richieste e la ricezione delle relative risposte. L'automa corrispondente è illustrato nella Figura 4.5:

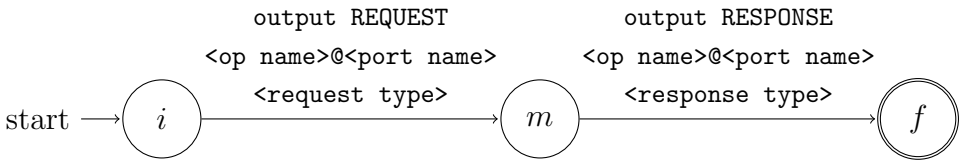


Figura 4.5: NFA associato a `SolicitNotificationOperationStatement`

4.2.2 Operatori di controllo

Sequenza

Siano G_1, ..., G_n gli automi generati dalle istruzioni I_1, ..., I_n presenti nello statement, l'automa generato è:

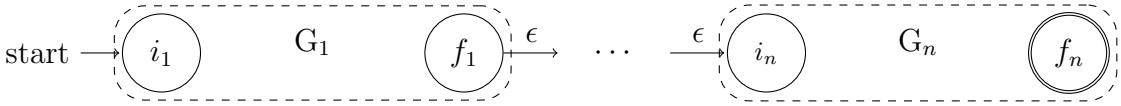


Figura 4.6: NFA associato a `SequenceStatement`

ovvero, si concatenano in sequenza. Nel Listing 4.1 viene proposta una modifica del servizio WeatherForecast in cui si definisce la procedura `init`, la quale inizializza, tramite diverse chiamate in sequenza, il servizio.

Listing 4.1: WeatherForecast con procedura `init`

```
1  service WeatherForecast{
2      ...
3      outputPort Logs {
4          location: "socket://localhost:8002"
5          protocol: http
6          oneWay:
7              initialize ( void )
8              logStart ( void )
9              save ( ForecastPrivateRequest )
10     }
11     outputPort ForecastProvider {
```

```
12          location: "socket://localhost:8002"
13          protocol: http
14          oneWay:
15              setResponseTimeFrame ( string )
16          requestResponse:
17              addRequester ( void )( string ),
18              getForecast ( ForecastPrivateRequest )(
                    ↪ ForecastResponse )
19
20      }
21      init{
22          initialize@Logs( )
23          addRequester@ForecastProvider( )( SECRET_KEY )
24          setResponseTimeFrame@ForecastProvider( "1 DAY" )
25          logStart@Logs( )
26      }
27  }
```

In Figura 4.7 è presente l'automa prodotto da JolieGraph utilizzando l'esempio modificato. Essendo `init`, quando definita, la prima procedura ad essere eseguita, i nodi aggiunti sono stati inseriti prima dell'ingresso nel ciclo creato dal `ProvideUntil`.

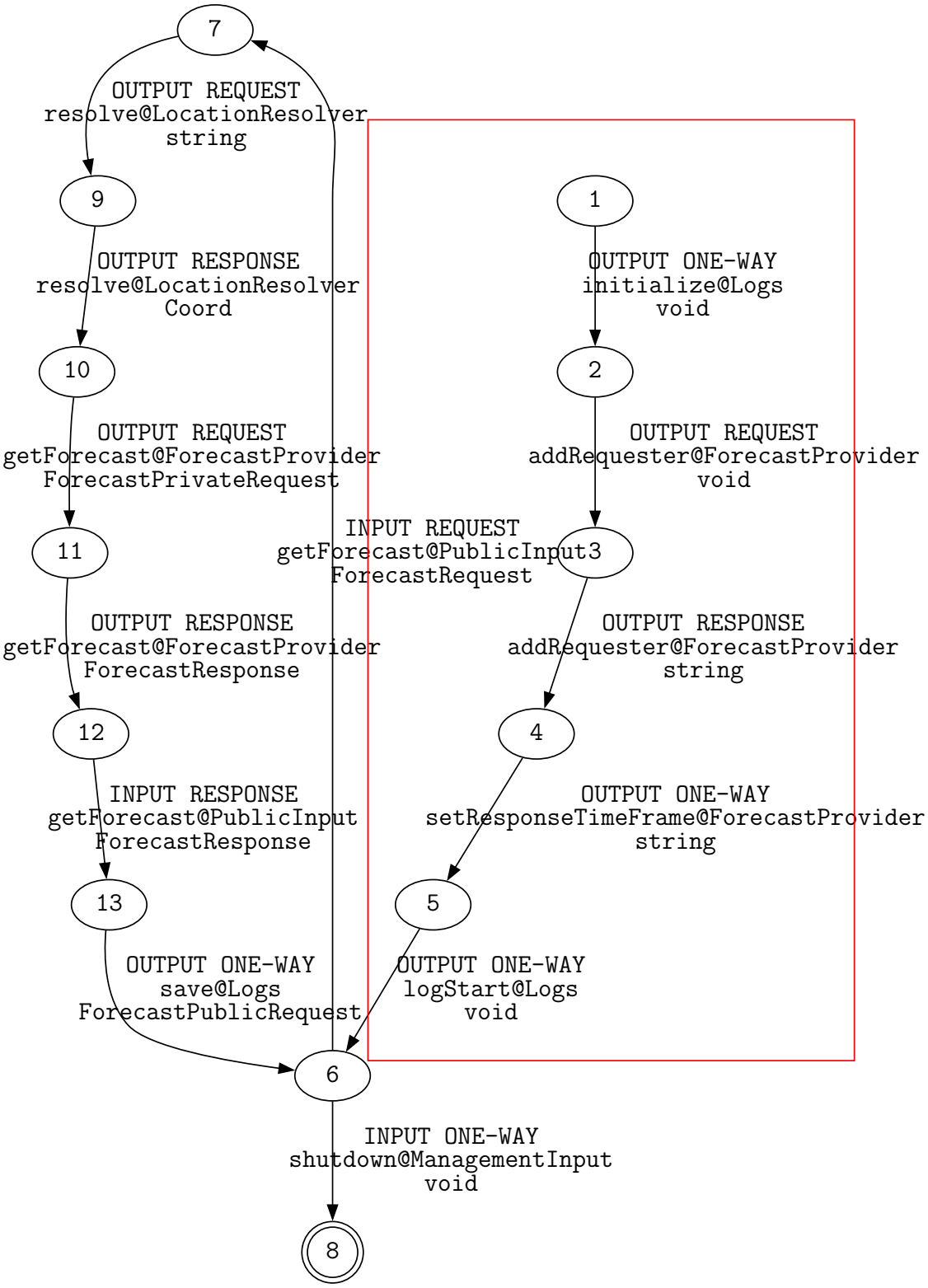


Figura 4.7: Automa del servizio **WeatherForecast** con procedura `init`. JolieGraph è stato eseguito senza opzione **-T**.

Scelta deterministica condizionata

Per la costruzione dell'automa associato ai nodi di tipo `IfStatement` si procede visitando i grafi generati dai vari rami del costrutto, per poi unirli in parallelo. Considerato un blocco `if` con N rami condizionali ed un ramo `else`, siano B_1, ..., B_n gli automi ottenuti dai N rami condizionali e B_e il grafo prodotto dal ramo `else`. L'automa costruito per la gestione di questo grafo è rappresentato in Figura 4.8:

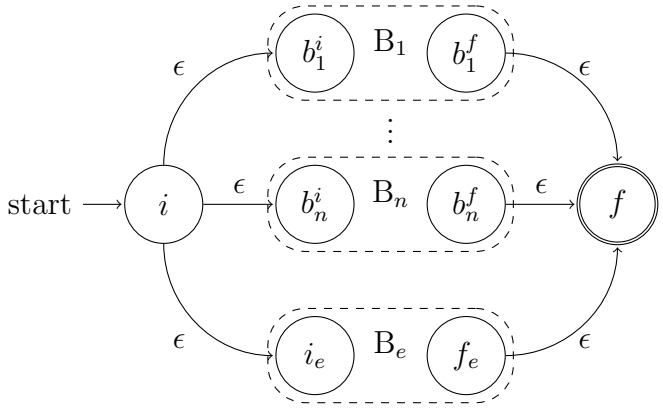


Figura 4.8: NFA associato a `IfStatement`

Per mostrare un esempio di gestione da parte di JolieGraph di un blocco if viene proposto nel listato sottostante una versione modificata di WeatherService, che permette all'utente di ottenere il risultato finale espresso in gradi sia Celsius che Fahrenheit. Nel caso in cui l'utente decida di utilizzare i gradi Fahrenheit, WeatherForecast convertirà la temperatura espressa in Celsius ottenuta dall'operazione `getForecast` attraverso l'operazione `cel2fah`.

```
1 type ForecastPublicRequest:void {
2     .date: string
3     .address: string
4     .inFahrenheit: bool
5 }
6 ...
7 service WeatherForecast {
8     ...
9     outputPort ForecastProvider {
10        location: "socket://localhost:8010"
11        protocol: http
12        requestResponse:
13            cel2fah ( double )( double ),
```

```
14              getForecast ( ForecastPrivateRequest )(
                    ↪ ForecastResponse )
15      }
16      main {
17          provide
18              [getForecast( request )( response ){
19                  resolve@LocationResolver( request.address
                        ↪  ) ( coord )
20                  getForecast@ForecastProvider ( { coord =
                        ↪ coord, date = request.date } )(
                        ↪ response )
21                  if ( request.inFahrenheit )
22                      cel2fah@ForecastProvider( response.
                            ↪ temperature )( response.
                            ↪ temperature )
23              }]{ save@Logs( request )}
24          until
25              [shutdown( )]{ exit }
26      }
27  }
```

Come evidenziato nella sezione riquadrata in Figura 4.9, sono previsti due possibili scenari. Nel primo, la risposta relativa all'operazione `getForecast` viene inviata direttamente, come indicato dall'arco (7,9). Nel secondo, invece, prima di restituire il risultato (arco (10,9)), quest'ultimo viene convertito da gradi Celsius a gradi Fahrenheit tramite l'operazione `cel2fah` (archi (7,8) e (8,10)).

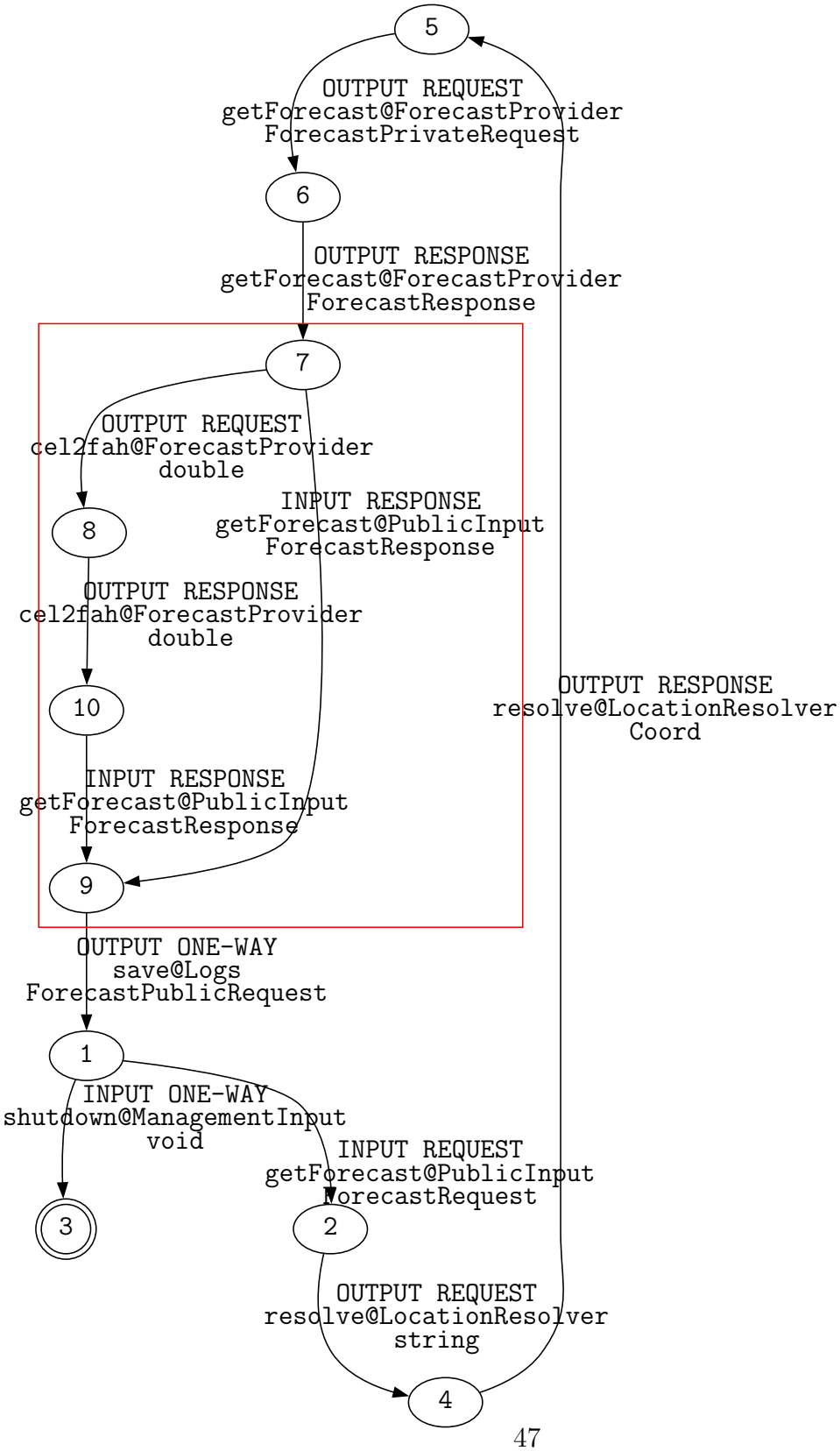


Figura 4.9: Automa del servizio **WeatherForecast** con costrutto *if*.

Costrutti iterativi

In Jolie l'iterazione può essere eseguita con quattro statement:

- `WhileStatement`: iterazione indefinita;
- `ForStatement`: iterazione indefinita;
- `ForEachArrayItemStatement`: iterazione sugli elementi di un array;
- `ForEachSubNodeStatement`: iterazione sui sottonodi di un nodo;

Gli NFA prodotti da nodi di tipo `ForEachArrayItemStatement`, `ForEachSubNodeStatement` e `WhileStatement` vengono descritti in Figura 4.10 fissando G_B come l'automa generato dal corpo dell'iterazione.

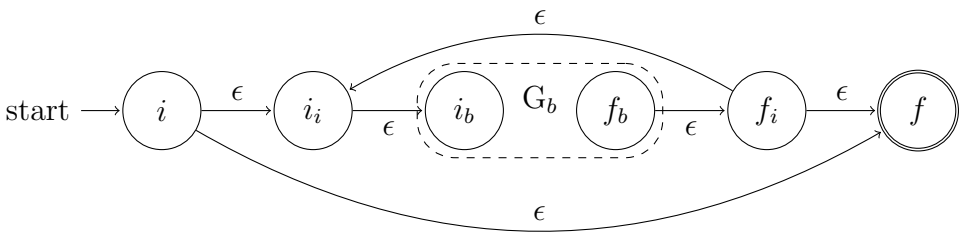


Figura 4.10: NFA associato ai costrutti iterativi definiti

In Figura 4.11 si mostra l'NFA definito per la gestione dei nodi di tipo `ForStatement`. Nella figura G_i rappresenta l'NFA del blocco di inizializzazione, ottenuto con `init()`, G_p l'NFA del blocco di aggiornamento post-iterazione, G_b ha lo stesso significato definito per gli altri costrutti iterativi.

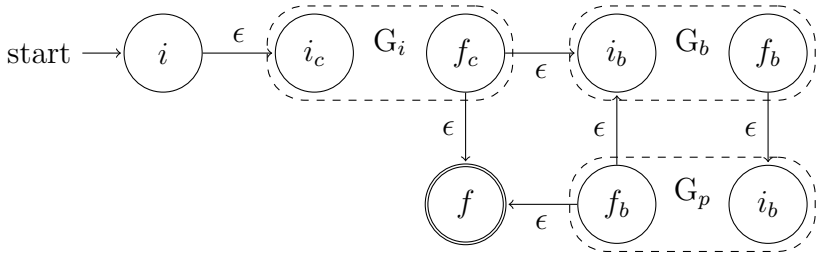


Figura 4.11: NFA associato ai cicli for

Viene mostrata nel Listing 4.2.2 una variante di WeatherForecast in cui l'utente può richiedere informazioni su più indirizzi, fissato un giorno.

```
type ForecastPublicRequest:void {
    .date: string
    .addresses[1, *]: string
}
...
service WeatherForecast {
    ...
    main {
        provide
            [getForecast( request )( response ) {
                for ( address in request.addresses ){
                    resolve@LocationResolver( address )(
                        ↪ coord )
                    getForecast@ForecastProvider( { coord
                        ↪  = coord, date = request.date }
                        ↪ ) ( response.(address) )
                }
            }]{ save@Logs( request ) }
        until
            [shutdown( )]{ exit }
    }
}
```

Gli stati evidenziati in Figura 4.12 $(2, 4, 6, 7)$ riguardano la gestione del ciclo foreach aggiunto nel codice. Il ciclo formato da questi nodi dimostra come sia possibile ripetere un numero arbitrario di volte (anche nessuna) le operazioni per la conversione in coordinate e per la risoluzione della query.

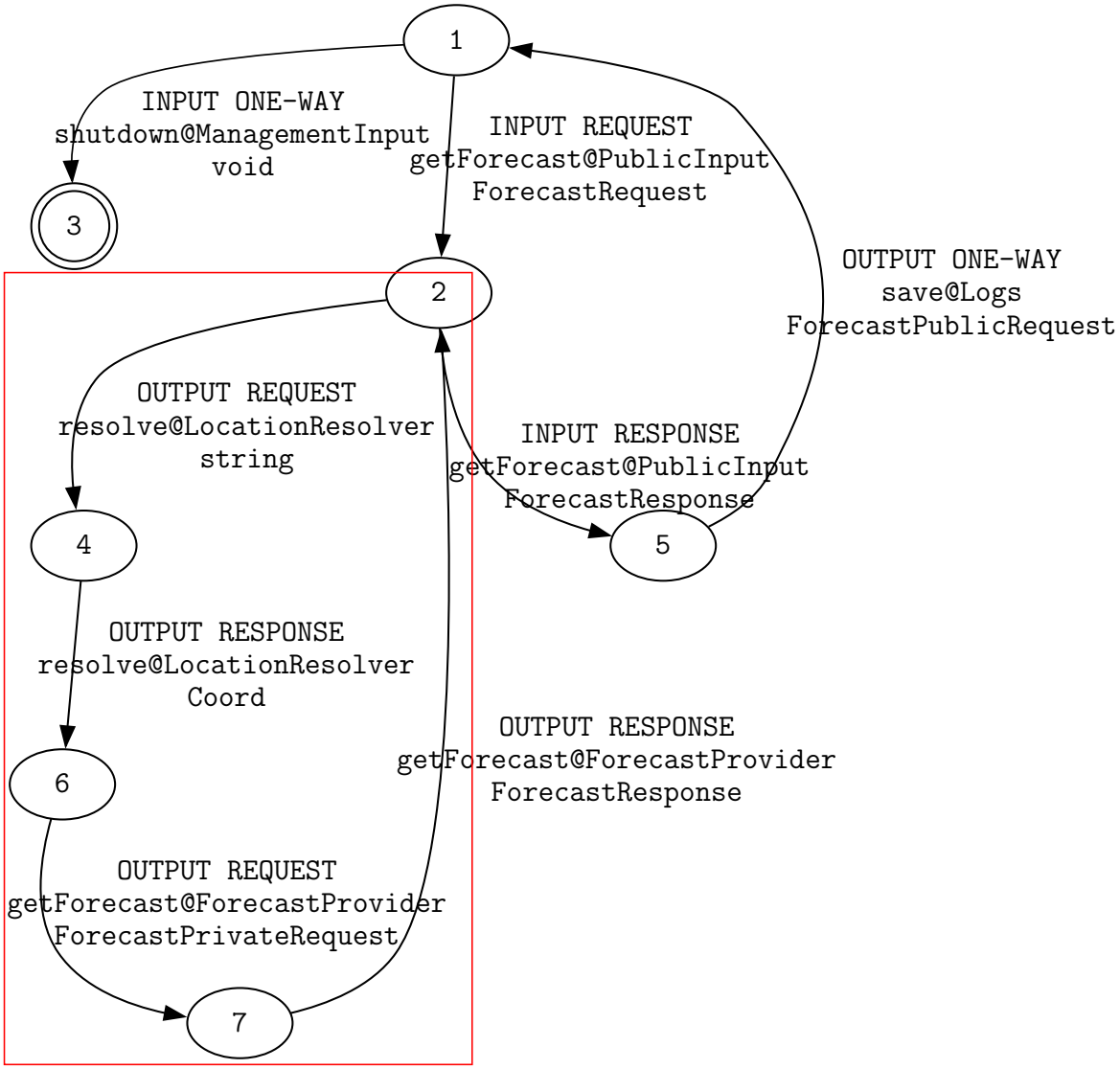


Figura 4.12: Automa del servizio **WeatherForecast** con costrutto *foreach*.

Viene inoltre presentato il Listato 4.2.2 contenente la definizione di un servizio all'interno del quale è presente un blocco for.

```
1 from console import Console
2 from .queue import Queue
3 service QueueHandler {
4     embed Console as Console
5     embed Queue as Queue
6     main {
```

```
7           for ( isEmpty@Queue()( empty ), !empty,
                ↪ isEmpty@Queue()( empty ) ){
8               extract@Queue()( message )
9           }
10      }
11      println@Console( "finito" )()
12  }
```

Nella Figura 4.13 è presente l'automa generato dal servizio appena mostrato. Gli archi $(1, 2)$, $(2, 3)$, $(3, 4)$, $(4, 1)$ rappresentano le operazioni svolte all'interno del for. Quando la coda è vuota la condizione non sarà più verificata, quindi si uscirà dal for e si terminerà l'esecuzione del servizio (archi $(3, 5)$, $(5, 6)$).

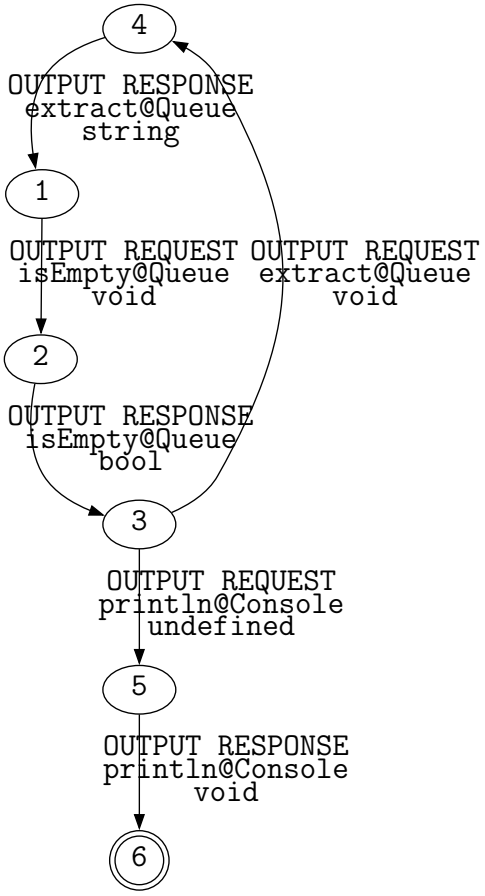


Figura 4.13: Automa del servizio `QueueHandler`.

Scelta non deterministica condizionata

Per implementare la scelta non deterministica, Jolie implementa gli `NDChoiceStatement`. Per ogni *input choice* presente vengono generati due automi: il primo riguarda la gestione della richiesta ottenuta, il secondo riguarda il relativo *branch*.

Si descrive la creazione di un costrutto generico con N input choice, dove per ognuna K_i è l'NFA di gestione della richiesta, B_i rappresenta l'NFA del branch.

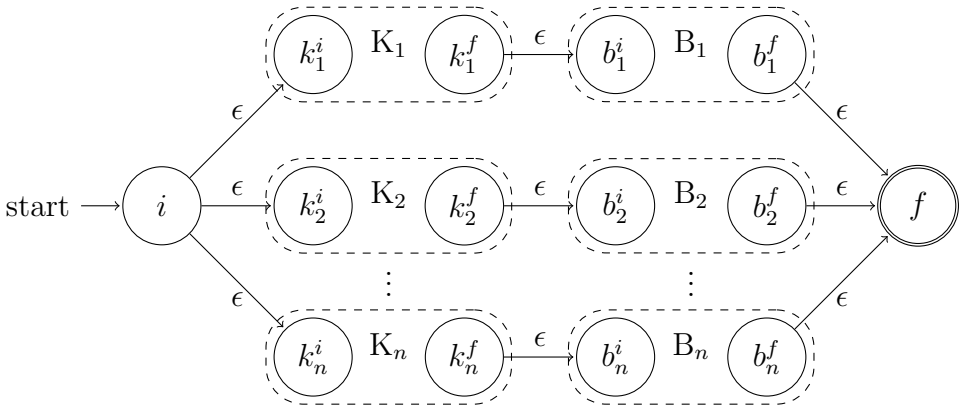


Figura 4.14: NFA associato a `NDChoiceStatement`

Nel Listing 4.2.2 si modifica il servizio d'esempio WeatherForecast sostituendo il blocco ProvideUntil con la scelta non deterministica. Viene inoltre impostata l'*execution mode* a sequential, permettendo la riesecuzione del `main` una volta che questo è terminato.

```
1 ...
2 service WeatherForecast {
3     execution: sequential
4     ...
5     main {
6         [getForecast( request )( response ) {
7             resolve@LocationResolver( request.address )(
                ↪ coord )
8             getForecast@ForecastProvider( { coord = coord
                ↪ , date = request.date } )( response )
9         }]{ save@Logs( request )}
10        [shutdown( )]{ exit }
11    }
12 }
```

Si noti come l'automa generato da questa versione di WeatherForecast, mostrato in Figura 4.15, è uguale all'automa mostrato in Figura 4.1, dimostrando come con i due costrutti sia possibile ottenere lo stesso grafo delle comunicazioni.

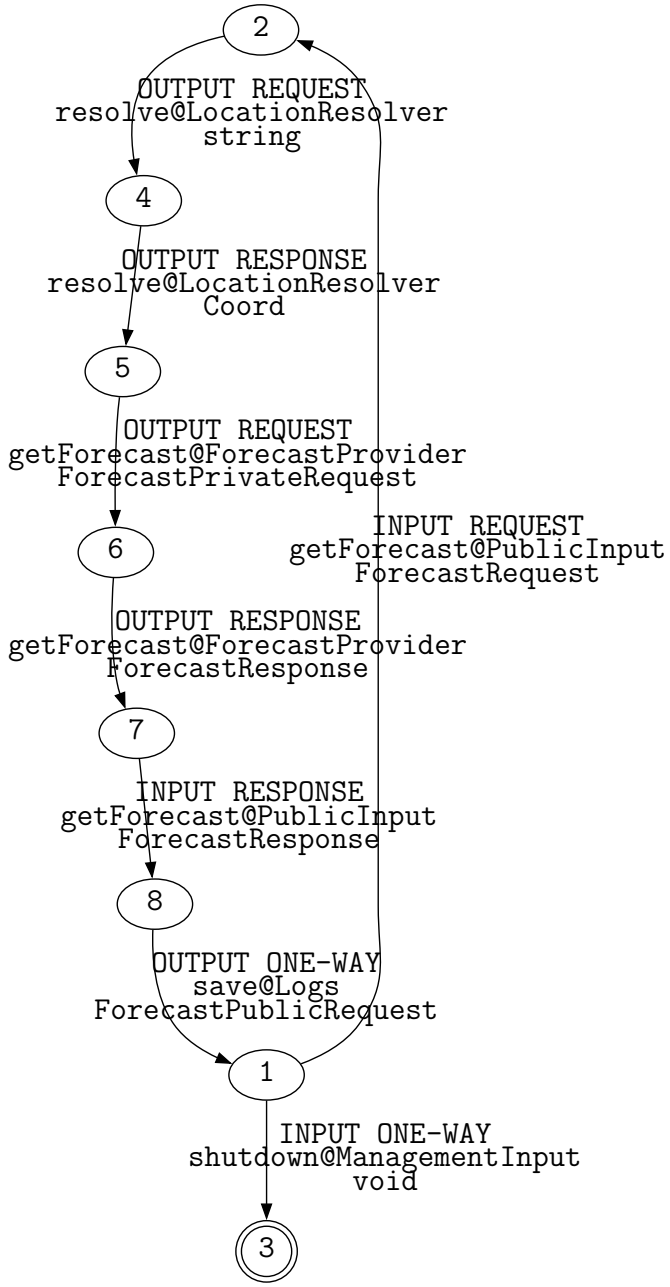


Figura 4.15: Automa del servizio **WeatherForecast** realizzato mediante *input choices*.

ProvideUntil

I due blocchi che compongono il costrutto ProvideUntil sono, concettualmente, delle scelte non deterministiche, con la differenza che al termine della gestione di una richiesta del blocco provide, sarà possibile ricevere nuovamente qualsiasi delle richieste. Mentre a seguito della ricezione di una richiesta del blocco until si uscirà dal costrutto, terminando la ricezione di tutte le operazioni specificate in esso.

Siano P e U gli NFA generati rispettivamente dal blocco Provide e dal blocco Until, si mostra in Figura 4.16 l'NFA definito per la gestione del ProvideUntil.

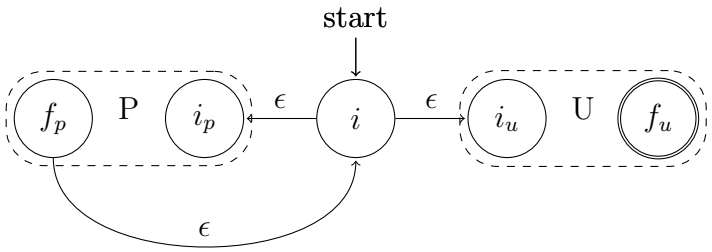


Figura 4.16: NFA associato a `ProvideUntilStatement`

Lancio dei fault

Poiché in Jolie i gestori dei fault vengono installati dinamicamente, non è possibile ottenere l'handler di un fault basandosi esclusivamente su quest'ultimo: è necessario tenere traccia delle operazioni di `install` effettuate all'interno dello scope in cui si trova l'operazione di `throw` e di quelli degli scope esterni. I nodi di tipo `InstallStatement` non generano direttamente alcun grafo, tuttavia vengono memorizzati nel contesto[2] gli handler installati. Quando viene incontrato un nodo di tipo `ThrowStatement`, il sistema procede alla ricerca dell'handler corrispondente tra quelli installati e in scope.

Per chiarire il concetto, viene presentata l'ultima versione del servizio **WeatherForecast** (Listing 4.2.2), nella quale si considera la possibilità che la conversione da indirizzo a coordinate possa fallire. Per gestire tale eventualità, viene eseguito un primo handler che tenta nuovamente la conversione utilizzando un'altra operazione. Nel caso in cui anche questa dovesse fallire, il sistema registra nei log il fallimento e invia una risposta d'errore all'utente.

```
1 ...
2 service WeatherForecast {
```

[2]vedi Sezione 4.3.2

```
3       ...
4       outputPort LocationResolver {
5           location: "socket://localhost:8005"
6           protocol: http
7           requestResponse:
8               resolve ( string )( Coord ),
9               resolveProbable ( string )( Coord )
10      }
11      outputPort Logs {
12          location: "socket://localhost:8001"
13          protocol: http
14          oneWay:
15              invalidAddress ( string ),
16              unrecoverableInvalidAddress ( string ),
17              save ( ForecastPrivateRequest )
18      }
19      define invalidAddressHandler {
20          invalidAddress@Logs( request.address )
21          resolveProbable@LocationResolver( request.address
              ↪  )( coord )
22          if ( coord.lat != -1 || coord.lon != -1 )
23              getForecast@ForecastProvider( { coord = coord
                  ↪ , date = request.date } )( response )
24          else
25              throw InvalidAddressError
26      }
27      main {
28          provide
29              [getForecast( request )( response ) {
30                  install(InvalidAddressError =>
31                      unrecoverableInvalidAddress@Logs(
                            ↪ request.address );
32                      response = { coord = -1, date = -1}
33                  )
34                  scope(s){
35                      install(InvalidAddressError =>
                            ↪ invalidAddressHandler)
36                      resolve@LocationResolver( request.
                            ↪ address )( coord )
37                      if ( coord.lat != -1 || coord.lon !=
```

```
                        ↪ -1 )
38                        getForecast@ForecastProvider( {
                              ↪ coord = coord, date =
                              ↪ request.date } )( response )
39                    else
40                        throw InvalidAddressError
41                }
42            }]{ save@Logs( request ) }
43        until
44            [shutdown( )]{ exit }
45    }
46 }
```

Nella Figura 4.17 vengono evidenziati gli stati e le transizioni inerenti il fault e i relativi handler. Questo esempio dimostra come JolieGraph supporti la gestione di fault handler ricorsivi. Infine, si nota come l'applicazione di una chiamata a procedura (`invalidAddressHandler`) sia trasparente a JolieGraph. L'esecuzione del servizio `WeatherForecast` procede normalmente fino all'arrivo sullo stato 5. Qui, se la conversione ha buon fine si procede all'ottenimento delle previsioni (archi $(5, 6)$, $(6, 8)$, $(8, 10)$) altrimenti verrà eseguito l'handler definito in riga 35 del programma. In quest'ultimo caso, il servizio notifica il servizio `Logs` della fallita conversione (arco $(5, 7)$) e tenta di risolverlo nuovamente utilizzando un'operazione di risoluzione più lasca (archi $(7, 9)$, $(9, 11)$). Se questo tentativo ha successo, come nel caso precedente, l'esecuzione procede con l'ottenimento delle previsioni e l'invio del risultato all'utente (archi $(11, 6)$, $(6, 8)$ e $(8, 10)$). Se anche questo tentativo fallisce entrerà in azione l'handler definito in riga 30, il quale informerà il sistema di log dell'impossibilità nella conversione e invierà un valore d'errore all'utente (archi $(11, 8)$, $(8, 10)$).

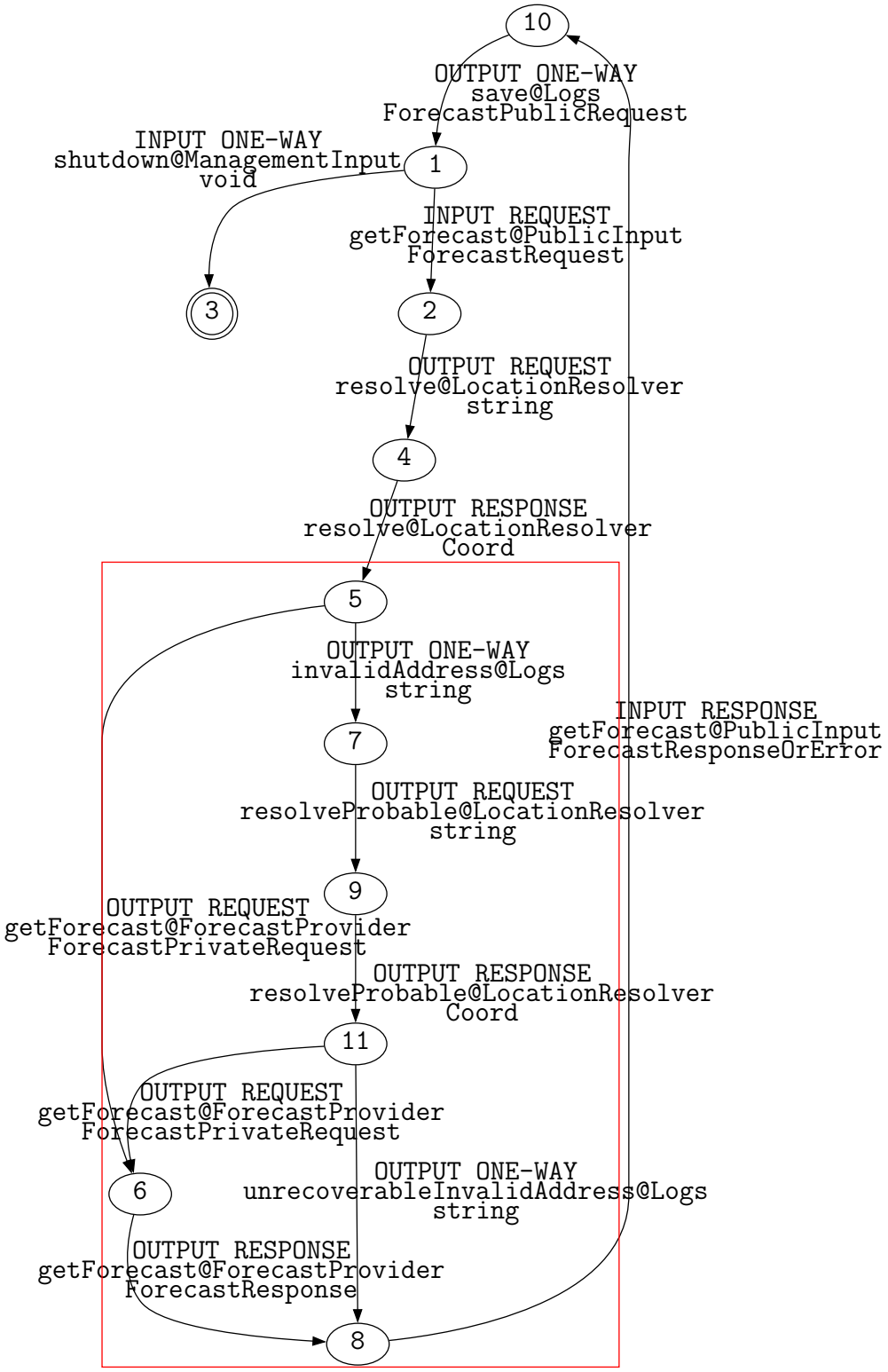


Figura 4.17: Automa del servizio **WeatherForecast** con gestori dei fault.

4.3 Implementazione di JolieGraph

Come già detto all'inizio del capitolo, l'esecuzione di JolieGraph è divisa in due fasi, al termine delle quali viene prodotto l'automa, ovvero la vista locale del servizio analizzato. Dal punto di vista dell'implementazione, questa divisione si riflette nella struttura di JolieGraph, mostrata in Figura 4.18. Nel pacchetto `symbols` sono presenti le classi per il caricamento delle definizioni, la principale delle quali è `SymbolManager`. La fase di generazione del grafo è gestita nel pacchetto `flow`, la cui classe principale è `FlowVisitor`. Esiste inoltre un terzo pacchetto, `utils`, in cui sono state definite diverse classi helper, tra cui `GraphUtils` (responsabile dell'ottimizzazione dell'automa, discussa nella Sezione 4.3.4) e `OutputSettings`, che contiene i parametri per la stampa dell'automa nel file DOT. Infine, `Application` funge da punto d'ingresso dell'applicazione.

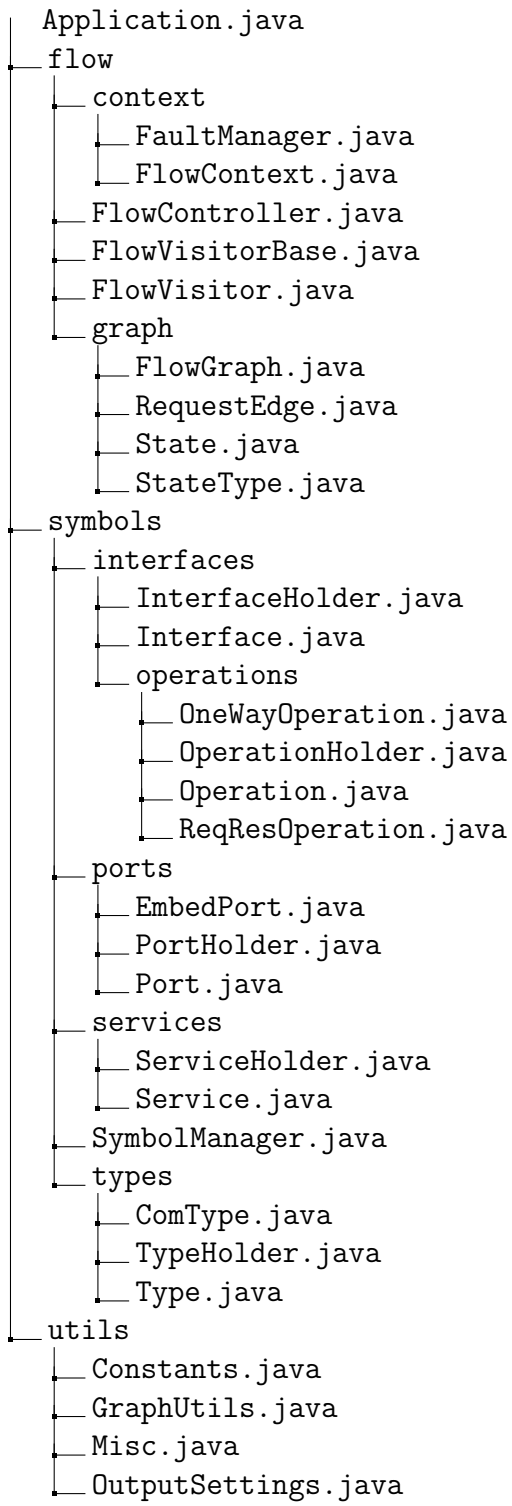


Figura 4.18: Struttura di JolieGraph.

4.3.1 Pacchetto `symbols`

L'obiettivo delle classi definite in `symbols` è il caricamento delle definizioni utilizzate dal servizio dato in input. Queste definizioni possono essere presenti nel file stesso del servizio, oppure possono essere importate da questo.

Architettura del pacchetto `joliegraph.symbols`

I tipi sono gestiti tramite `TypeHolder`, che memorizza una mappa di `Type`. Esistono due categorie principali di tipi:

- `Type`, che rappresenta un tipo generico con un nome;
- `ComType`, che estende `Type` e permette di definire tipi composti.

Le interfacce sono modellate dalle classi `InterfaceHolder` e `Interface`. Un'interfaccia contiene una lista di operazioni (`Operation`), che possono essere di due tipi:

- `OneWayOperation`, che rappresenta un'operazione unidirezionale senza risposta;
- `ReqResOperation`, che estende `Operation` aggiungendo un `responseType`, per gestire le operazioni di richiesta-risposta. In questa classe sono presenti anche i meccanismi di gestione dei fault.

I servizi sono organizzati in `ServiceHolder`, che tiene traccia di più servizi (`Service`). Ogni servizio può avere:

- `inputPorts`, che rappresentano le porte di comunicazione in ingresso;
- `outputPorts`, che rappresentano le porte di comunicazione in uscita.

Entrambi i tipi di porte (`Port`) fanno riferimento alle interfacce e alle operazioni associate.

Infine, `EmbedPort` estende `Port` e viene utilizzata per rappresentare servizi incorporati.

Risoluzione degli import

Il primo compito eseguito da `SymbolManager` consiste nell'esplorazione dei file di import, al fine di ottenere la collezione contenente tutti i percorsi relativi al servizio da analizzare. Tale collezione includerà, oltre al percorso del servizio stesso, anche i percorsi di tutti i file importati da qualsiasi file presente nella collezione. Per realizzare ciò, SymbolManager si affida alla classe `ModuleFinderImpl` della libreria `jolie.lang`, che offre un metodo `find` per la risoluzione di import relativi e assoluti

all'interno di file Jolie. `ModuleFinderImpl` utilizza due path per la risoluzione degli import:

- `workingDirectoryPath`: il path della directory in cui si trova il file di cui si stanno risolvendo gli import;
- `packagePaths`: il path della directory contenente la stdlib di Jolie.

Successivamente, verranno elaborate prima le definizioni presenti più "interni", per evitare il riferimento a definizioni non ancora catturate.

Gestione delle Definizioni

Dopo aver risolto gli import, `SymbolManager` si occupa di elaborare le definizioni di tipi, interfacce e servizi presenti nei file Jolie analizzati. Questo processo avviene attraverso la costruzione di una tabella dei simboli, che viene popolata utilizzando la classe `SymbolTableGenerator` della libreria `jolie.lang`. In particolare, vengono estratti e classificati i seguenti elementi:

- **Tipi**: Vengono raccolti e gestiti attraverso la classe `TypeHolder`, che distingue tra tipi semplici e tipi composti (`ComType`).
- **Interfacce**: La classe `InterfaceHolder` si occupa di registrare tutte le interfacce dichiarate, permettendo in seguito di associare operazioni a specifiche entità.
- **Servizi**: I nodi di tipo `ServiceNode` vengono salvati all'interno di una struttura di dati (`ServiceHolder`) che consente di navigare la gerarchia dei servizi definiti nel codice Jolie.

Il codice di `SymbolManager` assicura che l'analisi venga eseguita con un ordine gerarchico, elaborando prima i servizi più interni per garantire che ogni riferimento venga risolto correttamente.

Associazione delle Interfacce

Una volta caricate le definizioni, `SymbolManager` chiama il metodo `bindInterfaces` della classe `ServiceHolder`. Questo passaggio è fondamentale per associare le interfacce ai relativi servizi e porte di comunicazione:

- Ogni servizio viene collegato alle interfacce dichiarate nei suoi input e output port.
- Gli oggetti `Port` e `EmbedPort` mantengono una mappa delle operazioni disponibili e le risolvono interrogando il rispettivo `InterfaceHolder`.

- Le operazioni, rappresentate dalla classe `Operation`, sono divise in due categorie principali: `OneWayOperation` e `ReqResOperation`, a seconda che la comunicazione avvenga con o senza risposta.

L'associazione tra interfacce e servizi è cruciale per permettere al programma di JolieGraph di navigare e analizzare la comunicazione tra le varie componenti Jolie.

Struttura delle Porte

Nel pacchetto `symbols`, le porte di comunicazione sono gestite tramite la classe `Port`, che si occupa di:

- Analizzare il protocollo utilizzato (`setProtocol()`).
- Determinare la posizione della porta nel codice (`setLocation()`).
- Mantenere un riferimento alle interfacce utilizzate e alle operazioni definite.

Le porte possono essere di due tipi:

- **InputPort**: Definisce i canali di ricezione delle richieste.
- **OutputPort**: Gestisce le comunicazioni in uscita.

Inoltre, esiste una variante speciale chiamata `EmbedPort`, utilizzata per rappresentare i servizi incorporati (`EmbedServiceNode`).

Conclusione

Il pacchetto `symbols` fornisce un'infrastruttura solida per la gestione delle definizioni Jolie all'interno del programma. Grazie a una gerarchia ben strutturata di classi, JolieGraph è in grado di risolvere le dipendenze tra file, estrarre informazioni sui servizi e fornire un'analisi dettagliata delle interfacce e dei tipi utilizzati nel codice analizzato.

4.3.2 Pacchetto `flow`

Il pacchetto `flow` si occupa dell'analisi del flusso di esecuzione dei servizi Jolie. Il suo obiettivo principale è trasformare il codice Jolie in un grafo di flusso, rappresentando il comportamento del servizio in termini di stati ed eventi. Questa rappresentazione è utile per analisi statiche, ottimizzazione e visualizzazione del comportamento del programma.

Architettura del pacchetto `joliegraph.flow`

La classe **FlowGraph** rappresenta la struttura dati centrale, estendendo `DefaultDirectedGraph` per modellare il flusso esecutivo. Essa collega gli stati (**State**) attraverso archi etichettati (**RequestEdge**) e supporta operazioni di manipolazione del grafo. Gli stati sono categorizzati tramite `StateType`, distinguendo quelli normali, finali o di errore.

Il controllo del flusso è gestito da **FlowController**, che interagisce con il simbolismo Jolie tramite `SymbolManager`. Si occupa di esportare il grafo generato in formato DOT, utilizzando `DOTExporter`. **FlowVisitor**, invece, implementa un'analisi strutturale del codice Jolie, visitando l'AST ed estraendone il flusso di comunicazioni. Questo è reso possibile grazie all'uso di `FlowContext`, che conserva il riferimento al servizio in esame e gestisce le eccezioni attraverso **FaultManager**.

Generazione del Grafo di Flusso

Il componente centrale del pacchetto è la classe `FlowController`, che avvia l'analisi del codice Jolie e genera i grafi di flusso corrispondenti. Il processo avviene attraverso i seguenti passi:

- Viene istanziato un oggetto `SymbolManager` per analizzare i simboli presenti nei file Jolie.
- Ogni servizio viene analizzato da un'istanza di `FlowVisitor`, che visita il codice e costruisce un oggetto `FlowGraph`.
- I grafi ottenuti vengono memorizzati in una struttura dati e, se necessario, salvati in formato DOT per la visualizzazione.

Per la generazione dei file DOT, `FlowController` utilizza la libreria `JGraphT`, che fornisce strumenti per manipolare e visualizzare grafi.

Visita della Struttura del Servizio

L'analisi del codice Jolie avviene attraverso il pattern Visitor, implementato nella classe `FlowVisitor`. Questo componente definisce il comportamento per ogni nodo dell'AST rilevante. In particolare, il visitatore:

- Analizza le procedure principali (`init` e `main`) per costruire il flusso base dell'esecuzione.
- Identifica le dichiarazioni di operazioni e associa le chiamate agli stati corrispondenti nel grafo.

- Gestisce costrutti di controllo come sequenze (`SequenceStatement`), scelte (`IfStatement`, `NDChoiceStatement`), iterazioni (`WhileStatement`, `ForStatement`, `ForEachArrayItemStatement`, `ForEachSubNodeStatement`) e di gestione delle eccezioni (`ThrowStatement`, `InstallStatement`, `Scope`).

Struttura del Grafo di Flusso

Il grafo di flusso generato è rappresentato dalla classe `FlowGraph`, che è un'estensione della struttura `DefaultDirectedGraph` di `JGraphT`. Ogni stato del programma è rappresentato dalla classe `State`, mentre le transizioni tra stati sono modellate con `RequestEdge`. Gli stati possono essere di diversi tipi, definiti nell'enumerazione `StateType`:

- `NORMAL`: stato di esecuzione standard;
- `END`: stato terminale che indica il completamento del processo;
- `EXIT`: stato di uscita forzata.

Le transizioni tra stati possono rappresentare sia chiamate a operazioni interne che comunicazioni tra servizi, distinguendo tra operazioni `OneWayOperation` e `ReqResOperation`.

Metodi `joinBetween` e `joinAfter` Durante la costruzione del grafo, il pacchetto `flow` offre due metodi fondamentali per combinare automi parziali:

- `joinBetween(FlowGraph o, String label)`: permette di inserire un automa all'interno di un altro, posizionandolo tra lo stato di inizio e lo stato di fine del grafo principale. Se il grafo principale non ha uno stato iniziale o finale, questi vengono creati. Se una transizione diretta tra stato iniziale e stato finale già esiste, questa viene rimossa prima dell'inserimento dell'automa intermedio.
- `joinAfter(FlowGraph o)`: concatena un automa alla fine di un altro, aggiornando lo stato finale. Se il grafo principale non ha stati iniziali, viene inizializzato con uno stato vuoto. Inoltre, il metodo assicura che il nuovo stato finale corrisponda al nodo finale dell'automa concatenato.

Questi metodi risultano particolarmente utili per rappresentare strutture sequenziali o nidificate, permettendo di comporre grafi in maniera modulare.

4.3.3 Installazione dei Fault Handlers

`FaultManager` è la classe responsabile della gestione degli handler dei fault e del loro corretto recupero. La sua struttura si basa su una pila di scope, in cui ciascun scope

è rappresentato da una lista di mappe (`HashMap`) che associano chiavi (`String`) a fault, espressi come istanze della classe `FlowGraph`.

L'impiego delle liste è necessario per garantire la compatibilità con le installazioni effettuate all'interno dei blocchi `if-then-else`: per ogni ramo del costrutto viene aggiunta una specifica `HashMap` alla lista, e i fault handler definiti in ciascun ramo vengono memorizzati nella mappa corrispondente.

Una volta visitati tutti i rami, viene eseguita un'operazione di merge sulle `HashMap` presenti nella lista in cima alla pila. In caso di conflitto su un fault tra due mappe, il conflitto viene risolto fondendo gli handler e memorizzando il risultato.

Per spiegare meglio i meccanismi di caricamento e ricerca all'interno della pila, vengono mostrati due esempi: il primo ha come scopo illustrare il supporto offerto da JolieGraph a scope annidati, il secondo mostra il supporto all'installazione di fault handler all'interno di blocchi if.

Installazione dei fault handlers in scope annidati

Nel Listato 4.3.3 è definita la procedura `main` di un servizio Jolie:

```
1 main {
2     install ( f => println@Console("in main scope" )() )
3     scope(A){
4         install ( f => println@Console("in A scope" )() )
5         ...
6         throw(f)
7     }
8     throw(f)
9 }
```

1. **Stato iniziale:** All'inizio l'esecuzione la pila dei fault contiene una lista con un solo elemento: un'hashmap vuota.
2. **Installazione nel Main:** L'istruzione in riga 2 esegue un `install` che aggiunge, nella hashmap in posizione 0 della lista in cima alla pila, l'automa del fault handler definito da `println@Console("in main scope" )()`. Tale operazione associa la chiave `f` a questo gestore.
3. **Ingresso nello scope annidato `A`:** L'entrata nello scope `A` comporta un'operazione di *push* sulla pila, creando una nuova lista in cima. All'interno di questo scope, l'istruzione in riga 5 esegue un'altra `install`, inserendo il fault handler (che esegue `println@Console("in A scope" )()`) nella hashmap in posizione 0 della nuova lista, sempre associato alla chiave `f`.

4. **Prima `throw` (dentro lo scope `A`):** Quando in riga 9 viene eseguita `throw(f)`, il sistema cerca nella hashmap in posizione 0 della lista in cima alla pila il fault handler associato alla chiave `f`. Poichè in tale hashmap è presente il gestore installato in riga 5, l'operazione `throw` restituisce l'automa di questo handler, attivando il fault handling locale allo scope `A`.

5. **Uscita dallo scope `A` e seconda `throw`:** Al termine dello scope `A`, la lista relativa viene rimossa dalla pila (operazione di *pop*), lasciando in cima la lista originaria del main. Quando viene eseguita la seconda `throw(f)` (alla fine della procedura `main`), la ricerca del fault handler avviene ora nella hashmap in posizione 0 della lista del main, che contiene il gestore installato in riga 2. Di conseguenza, la `throw` restituisce l'automa del fault handler definito nel main, attivando il fault handling a livello globale del servizio.

Installazione dei fault handler nei blocchi condizionali

Poiché JolieGraph è uno strumento di analisi statica, in molti casi non è possibile determinare quale ramo di un blocco `if` verrà eseguito. Pertanto, in caso di installazione di fault handler all'interno di blocchi `if`, si è deciso di generare gli automi di tutti i gestori definiti.

```
1 main {
2     ...
3     if ( ... )
4         install ( f => println@Console("in if")() )
5     else
6         install ( f => println@Console("in else")() )
7     throw(f)
8 }
```

- **Ingresso nel ramo `if`**:
  All'ingresso del ramo `if` (riga 3) viene aggiunta una nuova hashmap alla lista in cima alla pila. Essa conterrà l'automa del fault handler installato in riga 4.

- **Ingresso nel ramo `else`**:
  Analogamente, all'ingresso del ramo `else` (riga 5) viene aggiunta una seconda hashmap alla lista in cima alla pila. Tale hashmap conterrà il gestore del fault `f` definito in riga 6.

- **Uscita dal blocco condizionale**:
  All'uscita dal blocco `if` viene eseguita un'operazione di *merge* sulle hashmap della lista in cima alla pila. In caso di conflitti, come in questo esempio, gli automi in collisione vengono uniti in un unico automa (come mostrato in

Figura 4.19, G_1 e G_2 rappresentano gli automi dei gestori dei fault in collisione) e quest'ultimo viene salvato. Al termine di tale operazione, nella lista sarà presente una sola hashmap.

In nessun momento la dimensione della pila è stata modificata, in quanto si è sempre operato all'interno del main scope.

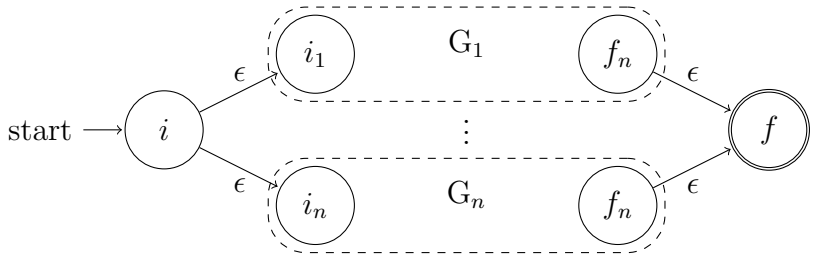


Figura 4.19: NFA utilizzato per l'unione di gestori in collisione

In Jolie, dopo l'esecuzione di un'operazione di `throw` l'esecuzione riprende dal termine dello scope che contiene l'handler corrispondente. Tuttavia, questo comportamento è supportato solo parzialmente da JolieGraph: nella versione presentata per questo lavoro è richiesto che i `throw` vengano eseguiti come ultima istruzione dello scope.

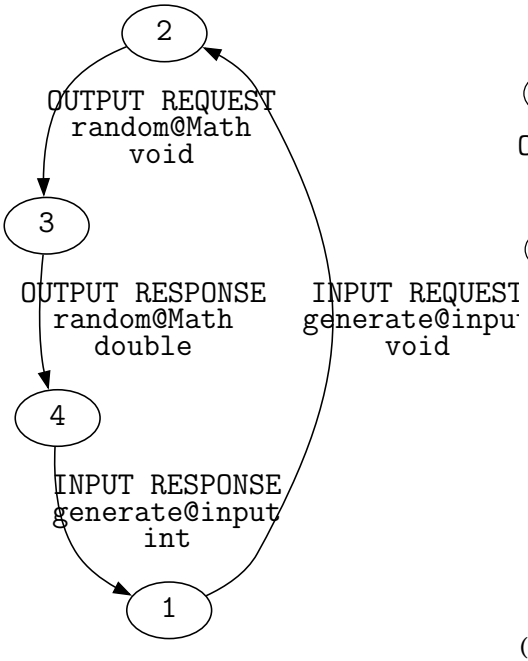
4.3.4 Ottimizzazione del Grafo

Come dimostrato nella Sezione 4.2, durante la creazione del grafo vengono aggiunte diverse *ϵ-transition*, ovvero archi con etichetta ϵ o vuota. Per la trasformazione in IDFA viene utilizzato l'algoritmo di costruzione per sottoinsiemi, nel quale vengono individuati insiemi di stati equivalenti e sostituiti agli stati originali. Una volta ottenuto l'IDFA, si procede alla sua minimizzazione attraverso il partizionamento iterativo e la sostituzione delle partizioni stabili con stati unici, riducendo il numero di nodi.

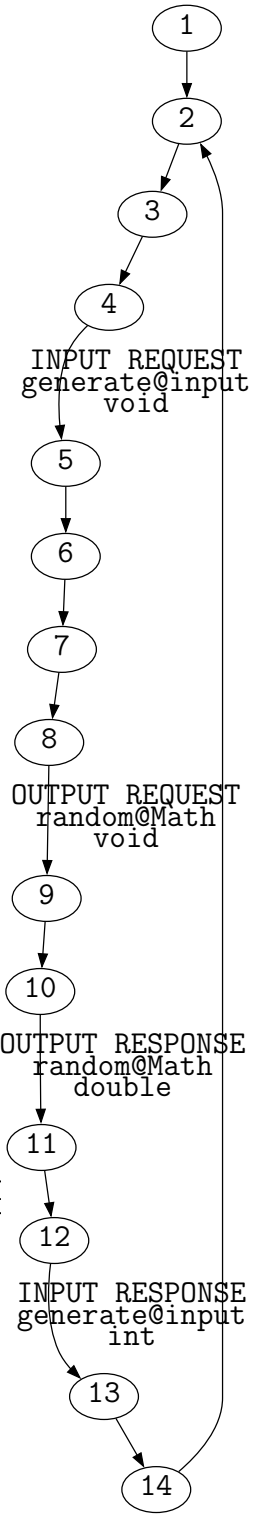
A titolo esemplificativo dell'operazione, vengono mostrati due grafi prodotti da JolieGraph a partire dallo stesso programma, presente nel Listato 4.3.4.

```
1 service NumberGenerator{
2     execution: sequential
3     inputPort input{
4         location: "socket://localhost:8000"
5         protocol: http
6         requestResponse: generate ( ) ( int )
7     }
```

```
8       main{ generate( ) ( r ) { random@Math()( r ) } }
9 }
```



(a) Automa del servizio `NumberGenerator` ottimizzato



(b) Automa del servizio `NumberGenerator` non ottimizzato

4.3.5 Architettura del pacchetto `symbols`

In Figura 4.21 viene mostrata l'architettura del pacchetto `joliegraph.symbols`, responsabile del caricamento delle definizioni in `JolieGraph`.

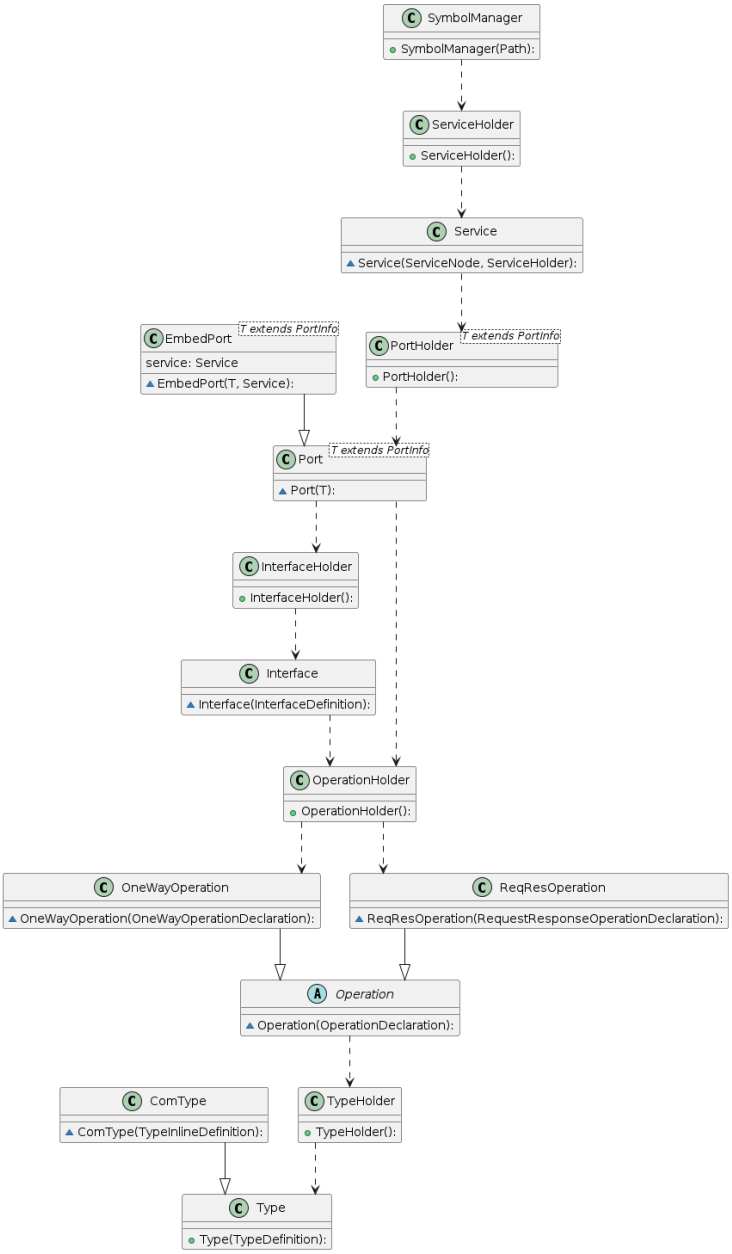


Figura 4.21: Schema UML del pacchetto `joliegraph.symbols`.

4.3.6 Architettura del pacchetto `flow`

In Figura 4.22 viene mostrata l'architettura del pacchetto `joliegraph.flow`, responsabile della generazione degli automi in `JolieGraph`.

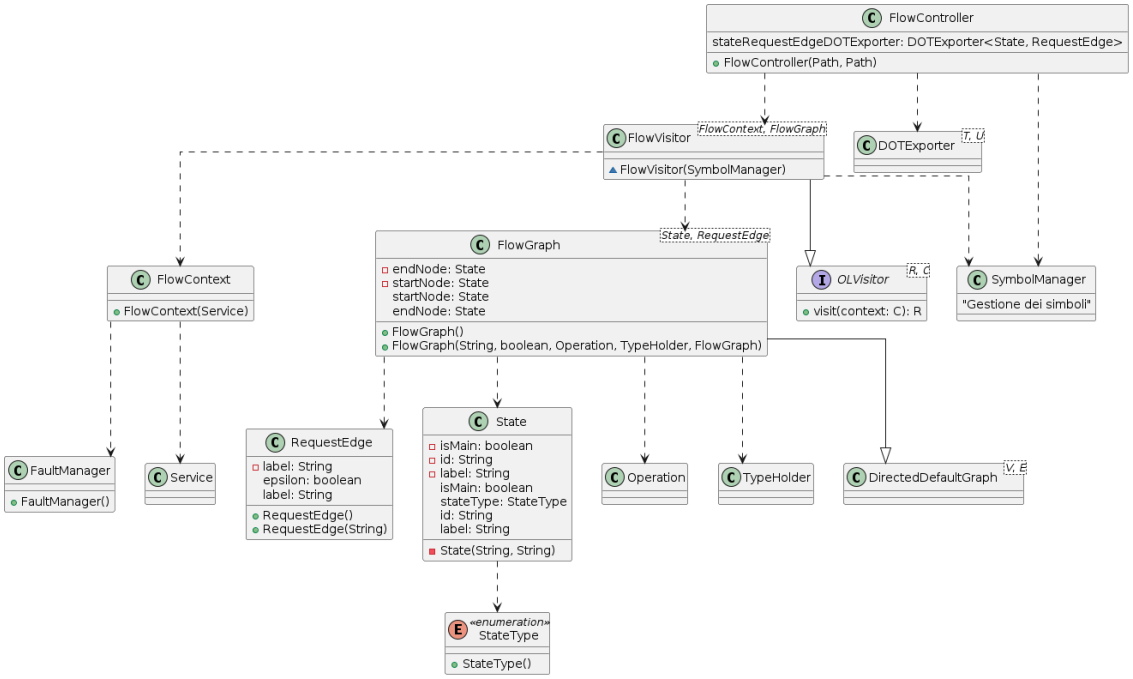


Figura 4.22: Schema UML del pacchetto `joliegraph.flow`.

4.4 Supporto ai nodi dell'AST

Nella Tabella 4.1 vengono considerate le sottoclassi di `OLSyntaxNode` che eseguono operazioni oppure i cui discendenti nell'albero sintattico potrebbero effettuarne. Per ciascuna tipologia di nodo ne viene indicato lo stato di supporto in JolieGraph.

Costrutto	**Supporto**	**Costrutto**	**Supporto**
DefinitionNode	Completo	Program	Completo
SequenceStatement	Completo	NDChoiceStatement	Completo
OneWayOperationStatement	Completo	RequestResponseOperationStatement	Completo
NotificationOperationStatement	Completo	SolicitResponseOperationStatement	Completo
IfStatement	Completo	WhileStatement	Completo
NullProcessStatement	Completo	ForEachArrayItemStatement	Completo
ForStatement	Completo	ForEachSubNodeStatement	Completo
ExitStatement	Completo	DefinitionCallStatement	Completo
ProvideUntilStatement	Completo	Scope	Parziale
ThrowStatement	Parziale	ImportStatement	Parziale
ServiceNode	Parziale	InstallStatement	Parziale
ParallelStatement	Assente	SpawnStatement	Assente
CompensateStatement	Assente	CorrelationSetInfo	Assente
PointerStatement	Assente	RunStatement	Assente
CurrentHandlerStatement	Assente	SolicitResponseForwardStatement	Assente
CourierDefinitionNode	Assente	CourierChoiceStatement	Assente
NotificationForwardStatement	Assente	LinkInStatement	Assente
LinkOutStatement	Assente		

Tabella 4.1: Tabella del supporto ai costrutti

Capitolo 5

Esempi di uso

Nel presente capitolo si vogliono dimostrare i risultati che JolieGraph riesce ad ottenere analizzando servizi più complessi. Verranno descritti i servizi **CalculatorService**[1] ed una sua variante più avanzata **AdvancedCalculatorService**[2] e i servizi del sistema roulette: **Table**, **Player**, **Croupier**[3].

5.1 CalculatorService

Il servizio `CalculatorService`, mostrato nel Listato 5.1, è stato implementato in Jolie per eseguire operazioni aritmetiche di base.

- **sum**: Questa operazione riceve in input una lista di termini e, mediante un ciclo foreach, li somma per ottenere il risultato finale.
- **sub**: Viene eseguita la sottrazione calcolando la differenza tra il *minuend* e il *subtraend* presenti nella richiesta.
- **mul**: L'operazione inizia con un valore iniziale pari a 1 e, iterando su ogni fattore fornito, calcola il prodotto totale.
- **div**: Infine, la divisione è realizzata dividendo il *dividend* per il *divisor*.

[1] Il codice completo del servizio, comprensivo anche delle definizioni delle interfacce è disponibile su `https://github.com/jolie/examples/tree/master/v1.10.x/tutorials/getting_started`.

[2] L'esempio completo è disponibile su `https://github.com/jolie/examples/tree/master/v1.10.x/tutorials/using_more_than_one_dependency`.

[3] I tre servizi, assieme ai loro file di configurazione, sono disponibili su `https://github.com/jolie/examples/tree/master/Tutorials/roulette/simple-version`.

La modalità di esecuzione di `CalculatorService` è impostata su `sequential`, quindi, ogni volta che verrà terminata la gestione di una delle operazioni appena elencate, l'esecuzione ripartirà dall'inizio del `main`, rendendo nuovamente il servizio disponibile ad accettare operazioni.

Listing 5.1: Codice del servizio **CalculatorService**

```
1 from .CalculatorInterfaceModule import
    ↪ CalculatorInterface
2  service CalculatorService {
3      execution: sequential
4      inputPort CalculatorPort {
5          location: "socket://localhost:8000"
6          protocol: http { format = "json" }
7          interfaces: CalculatorInterface
8      }
9      main {
10         [ sum( request )( response ) {
11             for( t in request.term ) {
12                 response = response + t
13             }
14         }]
15         [ sub( request )( response ) {
16             response = request.minuend - request.
                 ↪ subtraend
17         }]
18         [ mul( request )( response ) {
19             response = 1
20             for ( f in request.factor ) {
21                 response = response * f
22             }
23         }]
24         [ div( request )( response ) {
25             response = request.dividend / request.
                 ↪ divisor
26         }]
27     }
28 }
```

Appena avviato, il servizio si trova nello stato 1 e rimane in attesa di ricevere una delle operazioni tra `sum`, `sub`, `mul` o `div`. Una volta ricevuta una di queste operazioni, il servizio si troverà negli stati 4, 2, 5 o 3 rispettivamente. Durante la gestione di

nessuna di queste operazioni vengono inviate altre richieste, pertanto, una volta calcolata la response, inviandola si ritorna allo stato iniziale 1.

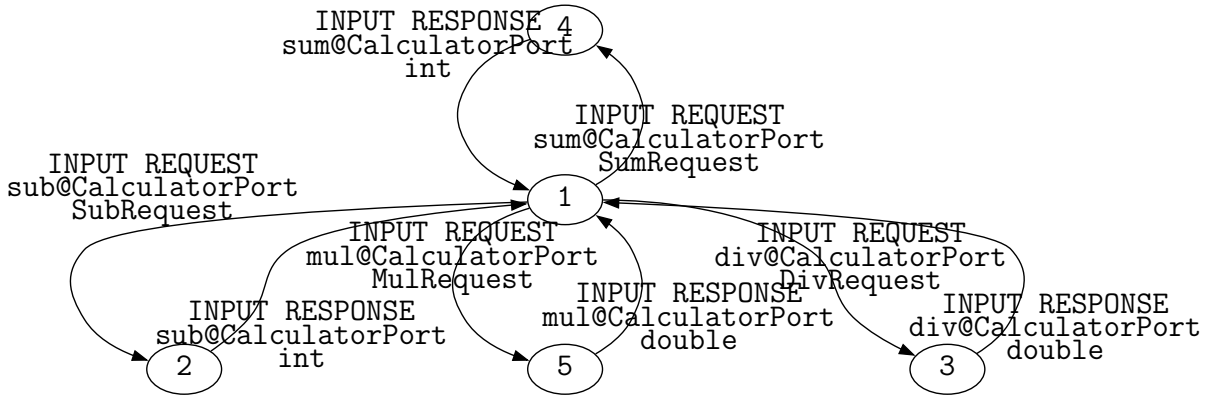


Figura 5.1: Automa del servizio **CalculatorService**.

5.1.1 AdvancedCalculatorService

Il servizio `AdvancedCalculatorService` è un'estensione del precedente esempio `CalculatorService` e introduce operazioni per il calcolo del fattoriale, della media e della percentuale. Per fare questo, `AdvancedCalculatorService` si basa sulle API esterne definite in `CalculatorService`.

Nel servizio vengono implementate tre operazioni:

- **factorial**: Calcola il fattoriale di un numero utilizzando un ciclo decrementale per preparare i valori da moltiplicare. La moltiplicazione vera e propria viene eseguita invocando il servizio di moltiplicazione definito in `CalculatorService`. Al termine dell'elaborazione, il servizio integra un messaggio pubblicitario ottenuto tramite l'API esterna di Chuck Norris.
- **average**: Determina la media dei termini forniti. In primo luogo, somma tutti gli elementi e poi divide il risultato per il numero degli elementi, ricorrendo alle operazioni del servizio base. Anche in questo caso, viene aggiunto un messaggio pubblicitario tramite la stessa API esterna.
- **percentage**: Calcola una percentuale applicata a un valore. L'operazione esegue prima una divisione per normalizzare il termine e poi una moltiplicazione per applicare la percentuale, utilizzando le operazioni di base già definite. Infine, viene arricchita la risposta con un messaggio pubblicitario proveniente dall'API di Chuck Norris.

Anche questo servizio, come `CalculatotService`, utilizza una modalità di esecuzione *sequential*.

```
from .AdvancedCalculatorServiceInterfaceModule import
    ↪ AdvancedCalculatorInterface
from .CalculatorInterfaceModule import
    ↪ CalculatorInterface
interface ChuckNorrisIface {
    RequestResponse: random( undefined )( undefined )
}
service AdvancedCalculatorService {
    execution: concurrent
    outputPort Chuck {
        location: "socket://api.chucknorris.io:443/"
        protocol: https {
            .osc.random.method = "get";
            .osc.random.alias = "jokes/random"
        }
        interfaces: ChuckNorrisIface
    }
    outputPort Calculator {
         location: "socket://localhost:8000"
         protocol: http { format = "json" }
         interfaces: CalculatorInterface
    }
    inputPort AdvancedCalculatorPort {
         location: "socket://localhost:8001"
         protocol: http { format = "json" }
         interfaces: AdvancedCalculatorInterface
    }
    main {
        [ factorial( request )( response ) {
            for( i = request.term, i > 0, i-- ) {
                req_mul.factor[ #req_mul.factor ] = i
            }
            mul@Calculator( req_mul )( response.factorial
                ↪  )
            random@Chuck()( chuck_res )
            response.advertisement = chuck_res.value
        }]
        [ average( request )( response ) {
```

```
36              sum@Calculator( request )( sum_res )
37              div@Calculator( { dividend = double( sum_res
                    ↪ ), divisor = double( #request.term ) })(
                    ↪  response.average )
38              random@Chuck()( chuck_res )
39              response.advertisement = chuck_res.value
40          }]
41          [ percentage( request )( response ) {
42              div@Calculator( { dividend = request.term,
                    ↪ divisor = 100.0 })( div_res )
43              mul@Calculator( { factor[0] = div_res, factor
                    ↪ [1] = request.percentage })(
                    ↪ response_mul )
44              response = response_mul
45              random@Chuck()( chuck_res )
46              response.advertisement = chuck_res.value
47          }]
48      }
49  }
```

Il servizio appena avviato si trova sullo stato 1. Da qui può andare verso gli stati 2, 3, 4 attraverso la ricezione di una richiesta, `average`, `factorial` o `percentage` rispettivamente. Se l'operazione ricevuta è `average`, `AdvancedCalculatorService` richiederà al `CalculatorService` il calcolo di una somma e di una divisione (archi $(2, 5)$, $(5, 8)$, $(8, 11)$, $(11, 14)$). Dopo aver ricevuto le risposte corrispondenti verrà inviata una richiesta attraverso la porta `Chuck` per l'inclusione dell'advertisement nella risposta (archi $(14, 17)$, $(17, 19)$). Se `AdvancedCalculatorService` riceve una richiesta `factorial`, il servizio utilizzerà nuovamente `CalculatorService` per eseguire le operazioni matematiche di base, quali il calcolo del prodotto (archi $(3, 6)$, $(6, 9)$). Infine, (archi $(9, 12)$, $(12, 15)$) viene aggiunto il campo a response contenente l'advertisement. L'ultima operazione disponibile è la `percentage`, la cui gestione prevede l'invio di richieste di tipo `div` e `mul` verso `CalculatorService` (archi $(4, 7)$, $(7, 10)$, $(10, 13)$, $(13, 16)$). Una volta ricevuta anche la risposta all'operazione di moltiplicazione, si effettua la solita richiesta verso la porta `Chuck` per l'inclusione dell'advertisement (archi $(16, 18)$, $(18, 20)$).

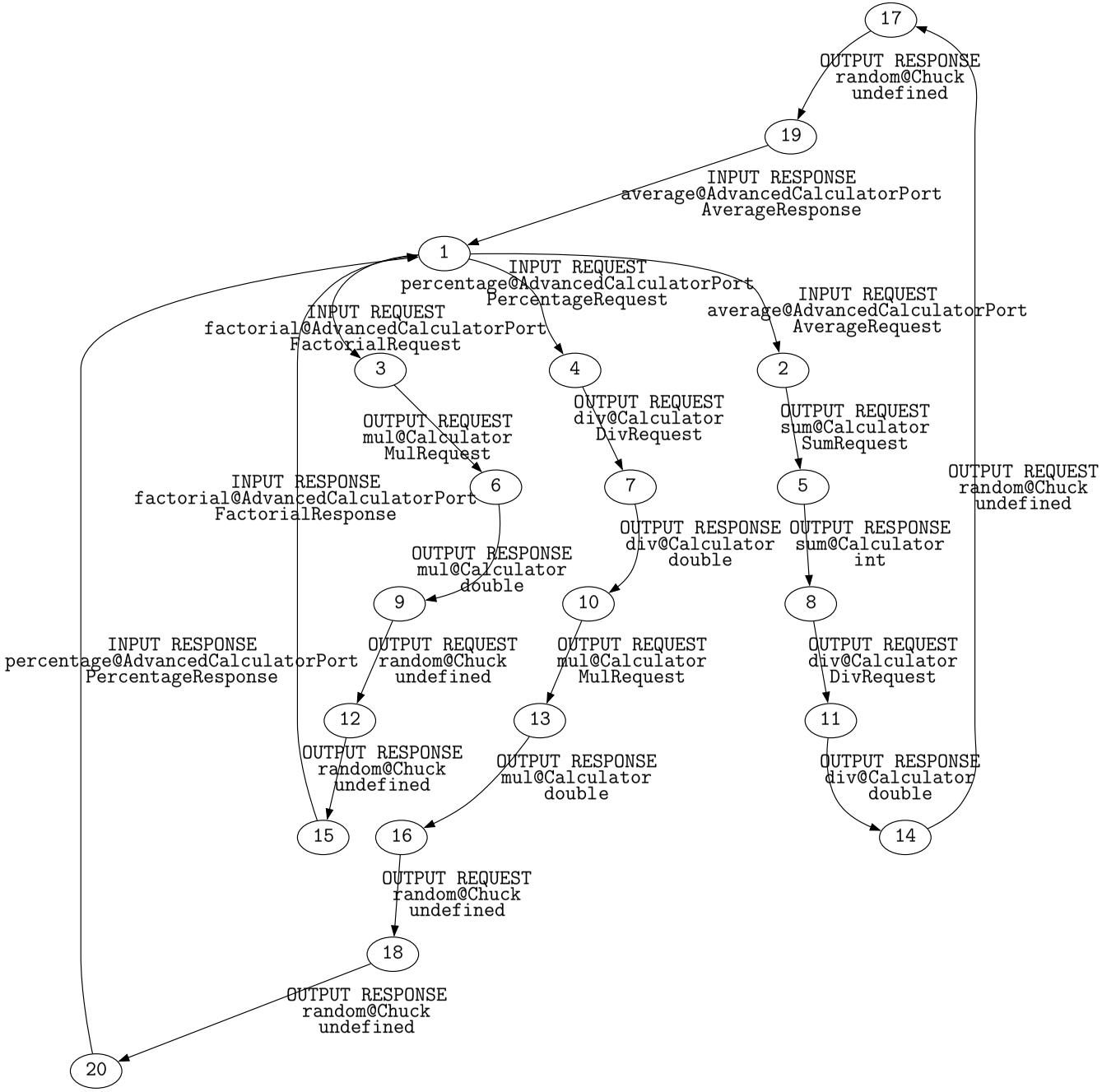


Figura 5.2: Automa del servizio **AdvancedCalculatorService**.

5.2 Roulette

Il sistema è composto da tre servizi interconnessi che collaborano per simulare una sessione di roulette. Il servizio `Table` è responsabile della gestione delle scommesse: riceve le puntate dai giocatori, calcola il numero vincente in maniera casuale e determina, in base alle scommesse registrate, chi ha vinto o perso. Il servizio `Croupier` funge da coordinatore, interagendo con la `Table` per ottenere i risultati (tramite l'operazione `rienNeVaPlus`) e comunicandoli poi ai giocatori, inviando messaggi di payout o di perdita. Il servizio `Player`, infine, simula il comportamento di un giocatore, generando scommesse casuali e inviandole al servizio `Table`, oltre a ricevere le notifiche sui risultati del gioco.

5.2.1 Croupier

Il servizio `Croupier` gestisce la logica di coordinamento del gioco. Il servizio è configurato in modalità *sequential* e dispone di porte d'output verso il servizio `Player` e la `Table` (utilizzando l'interfaccia `TableToCroupierInterface`). Nel blocco `init`, imposta una callback sull'operazione `wakeUp` tramite lo Scheduler e definisce un cron job che regola il ritmo delle puntate. Nel blocco `main`, quando viene invocata l'operazione `wakeUp`, il servizio chiama l'operazione `rienNeVaPlus` della `Table` per ottenere i risultati della puntata e, in seguito, invia messaggi specifici ai giocatori per notificare le vincite (mediante l'operazione `payout`) e le perdite (mediante l'operazione `lost`).

```
1  from scheduler import Scheduler      // imported the
   ↪ Scheduler
2  from console import Console
3  from .Table import TableToCroupierInterface
4  from .Player import PlayerGameInterface
5  type CroupierParam {
6      table_location: string
7  }
8  interface LocalInterface {
9      OneWay:
10         wakeUp( undefined )
11 }
12 service Croupier( p : CroupierParam ) {
13     embed Console as Console
14     embed Scheduler as Scheduler
15     execution: sequential
16     outputPort Player {
17         protocol: sodep
```

```
18          interfaces: PlayerGameInterface
19      }
20      outputPort Table {
21          location: p.table_location
22          protocol: sodep
23          interfaces: TableToCroupierInterface
24      }
25      inputPort Local {
26          location: "local"
27          interfaces: LocalInterface
28      }
29      init {
30          setCallbackOperation@Scheduler( { operationName =
              ↪  "wakeUp" })
31          // setting cronjob
32          setCronJob@Scheduler( {
33              jobName = "bet"
34              groupName = "roulette"
35              cronSpecs << {
36                      second = "0"
37                      minute = "0/1"
38                      hour = "*"
39                      dayOfMonth = "*"
40                      month = "*"
41                      dayOfWeek = "?"
42                      year = "*"
43              }
44          })()
45          enableTimestamp@Console( true )()
46      }
47      main {
48          [ wakeUp() ] {
49              rienNeVaPlus@Table()( response )
50              for( w in response.winners ) {
51                  Player.location = w.location_player
52                  undef( payout_req )
53                  payout_req << {
54                      payout = w.payout
55                      number = w.number
56                  }
```

```
57                  payout@Player( payout_req )
58              }
59              for( l in response.loosers ) {
60                  Player.location = l.location_player
61                  undef( lost_req )
62                  lost_req << {
63                      lost = l.lost
64                      number = l.number
65                      winnerNumber = response.winningNumber
66                  }
67                  lost@Player( lost_req )
68              }
69          }
70      }
71  }
```

Appena avviato `Croupier` si trova nello stato 1. Da qui, verranno inviate una serie di richieste per inizializzare l'ambiente (archi $(1,2)$, $(2,3)$, $(3,4)$, $(4,5)$, $(5,6)$). Queste transizioni sono dovute alle operazioni presenti nella procedura `init`. Arrivati allo stato 6 si inizia la gestione delle operazioni presenti nel `main`, partendo dalla ricezione di un'operazione `wakeUp`, inviata dal cron job definito nell'`init` (arco $(6,7)$. A seguito della ricezione di questa operazione viene inviata una richiesta `rienNeVaPlus` ($(7,8)$ e $(8,9)$). Successivamente verranno avvisati i giocatori vincenti e perdenti tramite le operazioni `payout` e `lost`. Si noti la presenza dell'arco $(9,7)$, giustificato dalla possibilità di non avere giocatori che abbiano perso.

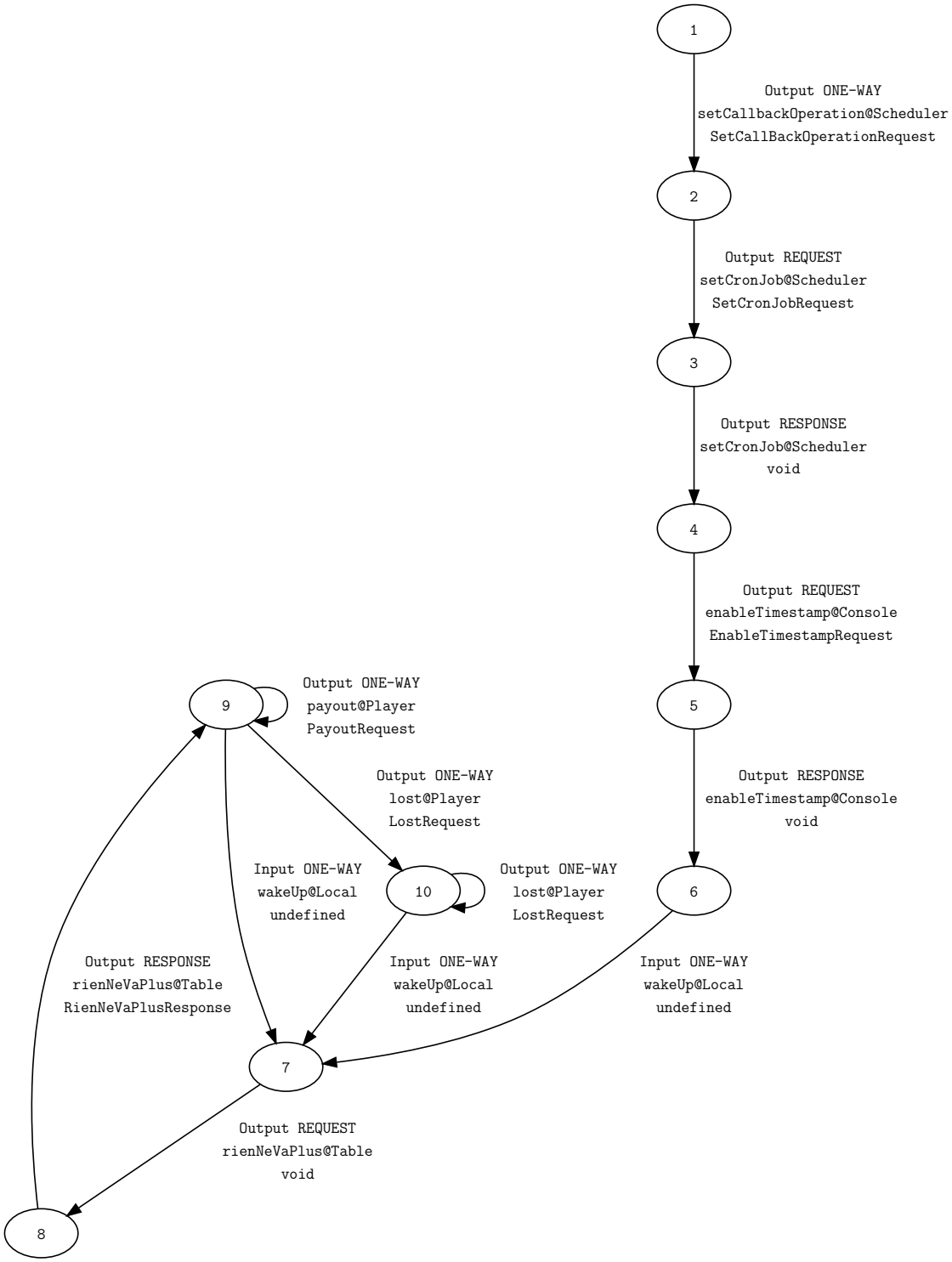


Figura 5.3: Automa del servizio **Croupier**.

5.2.2 Table

Il servizio `Table` rappresenta il cuore del gioco, occupandosi della registrazione e gestione delle scommesse inviate dai giocatori. Configurato in modalità *sequential*, il servizio dispone di due porte di input: una per ricevere le scommesse dai giocatori e l'altra per interagire con il `Croupier` tramite l'interfaccia `TableToCroupierInterface`. Inoltre, possiede un output port verso il servizio `Player`. Nel blocco `main`, l'operazione `straightUpBet` registra la scommessa di un giocatore, controllandone la validità e sincronizzando l'accesso al database delle puntate per evitare conflitti durante lo spin della ruota. L'operazione `rienNeVaPlus` calcola il numero vincente in modo casuale e analizza le scommesse registrate, determinando chi vince e chi perde, aggiornando di conseguenza il database e stampando un report del turno appena concluso.

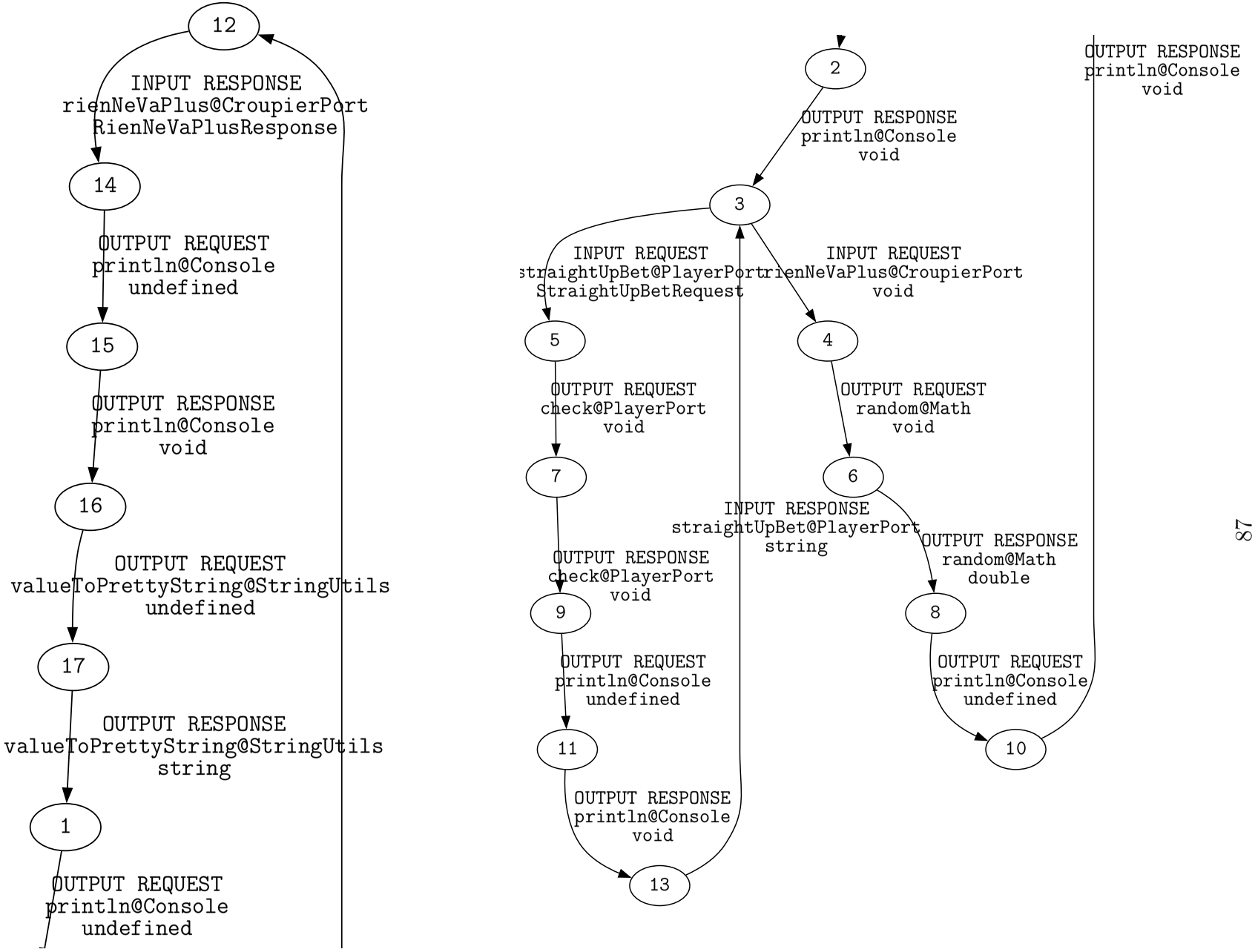
```
1  from console import Console
2  from string_utils import StringUtils
3  from json_utils import JsonUtils
4  from math import Math
5  from .Player import PlayerGameInterface
6  type StraightUpBetRequest: void {
7      player: string
8      amount: int
9      number: int
10     player_location: string
11 }
12 type RienNeVaPlusResponse {
13     winningNumber: int
14     winners* {
15         location_player: string
16         payout: int
17         number: int
18     }
19     loosers* {
20         location_player: string
21         lost: int
22         number: int
23     }
24 }
25 interface TableToPlayerInterface {
26     RequestResponse:
27         straightUpBet( StraightUpBetRequest )( string )
               ↪ throws LocatioNotValid
```

```
28 }
29 interface TableToCroupierInterface {
30     RequestResponse:
31         rienNeVaPlus( void )( RienNeVaPlusResponse )
32 }
33 type TableServiceParam {
34     locations {
35         playerPort: string
36         croupierPort: string
37     }
38 }
39 service TableService( p : TableServiceParam ) {
40     execution: sequential
41     embed Math as Math
42     embed Console as Console
43     embed JsonUtils as JsonUtils
44     embed StringUtils as StringUtils
45     inputPort PlayerPort {
46         location: p.locations.playerPort
47         protocol: sodep
48         interfaces: TableToPlayerInterface
49     }
50     inputPort CroupierPort {
51         location: p.locations.croupierPort
52         protocol: sodep
53         interfaces: TableToCroupierInterface
54     }
55     outputPort PlayerPort {
56         protocol: sodep
57         interfaces: PlayerGameInterface
58     }
59     init {
60         println@Console("Service Table is running...")()
61     }
62     main {
63         [ straightUpBet( request )( response ){
64             scope( check_player ) {
65                 install( default => throw(
                       ↪ LocatioNotValid))
66                 PlayerPort.location = request.
```

```
                    ↪ player_location
67                  check@PlayerPort()()
68                  // synchronizing the bet so that it
                    ↪ cannot be placed while the 'wheel is
                    ↪  spinning'.
69                  synchronized( spinning ) {
70                      global.db.bets.( request.player )[ #
                        ↪ global.db.bets.( request.player
                        ↪ ) ] << request
71                  }
72                  response = "Received straight up bet on
                    ↪ number " + request.number + " for
                    ↪ player " + request.player
73                  println@Console( response )()
74              }
75          }]
76          [ rienNeVaPlus()( response ) {
77              random@Math()( temp )
78              winningNumber = response.winningNumber = int(
                ↪  temp * 37 )
79              println@Console("winningNumber: ==> " +
                ↪ winningNumber )()
80              synchronized( spinning ) {
81                  global.db.spin[ #global.db.spin ] =
                    ↪ winningNumber
82                  foreach ( gioc : global.db.bets ) {
83                      for ( bet in global.db.bets.( gioc )
                        ↪ ) {
84                          if ( winningNumber == bet.number
                            ↪ ){
85                              response.winners[ #response.
                                ↪ winners ] << {
86                                  location_player = bet.
                                    ↪ player_location
87                                  payout = bet.amount * 37
88                                  number = bet.number
89                              }
90                          } else {
91                              response.loosers[ #response.
                                ↪ loosers ] << {
```

```
92                                      location_player = bet.
                                           ↪ player_location
93                                      lost = bet.amount
94                                      number = bet.number
95                                  }
96                              }
97                          }
98                      }
99                      undef( global.db.bets )
100                 }
101             }] {
102                 println@Console("Report after last run:")()
103                 valueToPrettyString@StringUtils( response )(
                        ↪ s )
104                 println@Console( s )()
105             }
106     }
107 }
```

`Table`, appena avviato, si trova nello stato 1, dal quale esce subito grazie all'operazione `println` (archi $(1,2)$, $(2,3)$). Arrivati allo stato 3, il servizio viene messo in attesa fino alla ricezione di una delle operazioni tra `straightUpBet` effettuata da un giocatore, oppure `rienNeVaPlus`, effettuata dal croupier. Nel caso si riceva la prima richiesta ($(3,5)$), viene inviata una richiesta `check` al giocatore (archi $(5,7)$, $(7,9)$), si calcola la response dell'operazione ricevuta; la si stampa (archi $(9,11)$, $(11,13)$) e la si invia al giocatore (arco $(13,3)$). Altrimenti, nel caso avvenga la ricezione dell'operazione `rienNeVaPlus` (arco $(3,4)$) si simula il giro della roulette attraverso l'operazione `random` ($(4,6)$, $(6,8)$). Si procede alla stampa del valore ottenuto ($(8,10)$, $(10,12)$) e si invia la risposta all'utente (arco $(12,14)$). Infine, si termina la gestione della richiesta stampando su terminale, in maniera formattata, un report sul giro appena terminato. (archi $(14,15)$, $(15,16)$, $(16,17)$, $(17,1)$ e $(1,2)$). Si noti come gli stati 1 e 2 vengono utilizzati sia per indicare l'operazione di `println` dichiarata nella procedura `init`, sia per rappresentare il termine della gestione dell'operazione `rienNeVaPlus`.


12
INPUT RESPONSE
rienNeVaPlus@CroupierPort
RienNeVaPlusResponse
14
OUTPUT REQUEST
println@Console
undefined
15
OUTPUT RESPONSE
println@Console
void
16
OUTPUT REQUEST
valueToPrettyString@StringUtils
undefined
17
OUTPUT RESPONSE
valueToPrettyString@StringUtils
string
1
OUTPUT REQUEST
println@Console
undefined
2
OUTPUT RESPONSE
println@Console
void
3
INPUT REQUEST
straightUpBet@PlayerPort
StraightUpBetRequest
INPUT REQUEST
rienNeVaPlus@CroupierPort
void
5
4
OUTPUT REQUEST
check@PlayerPort
void
OUTPUT REQUEST
random@Math
void
7
6
INPUT RESPONSE
straightUpBet@PlayerPort
string
OUTPUT RESPONSE
check@PlayerPort
void
OUTPUT RESPONSE
random@Math
double
9
8
OUTPUT REQUEST
println@Console
undefined
OUTPUT REQUEST
println@Console
undefined
11
10
OUTPUT RESPONSE
println@Console
void
13
OUTPUT RESPONSE
println@Console
void

5.2.3 Player

Il servizio `Player` simula il comportamento di un giocatore nel gioco. Il servizio è configurato in modalità *sequential* e possiede due porte d'input: una locale per ricevere callback dallo Scheduler e una `PlayerGamePort` per le comunicazioni di gioco, oltre a una output port per inviare scommesse al servizio `Table`. Nel blocco `init`, viene configurato lo Scheduler per impostare la callback `playerCallBack` e per pianificare un cron job che invia scommesse periodiche. Quando viene invocata la callback, il servizio genera in maniera casuale sia il numero su cui puntare sia l'ammontare della scommessa, inviando il tutto al servizio `Table` e aggiornando il proprio portafoglio in base alle risposte ricevute, che notificano sia vincite (tramite l'operazione `payout`) sia perdite (tramite l'operazione `lost`).

```
1  from scheduler import Scheduler      // imported the
       ↪ Scheduler
2  from console import Console
3  from time import Time
4  from math import Math
5  from .Table import TableToPlayerInterface
6  type PlayerCallBackRequest: void {
7      .jobName: string
8      .groupName: string
9  }
10 type PayoutRequest: void {
11     payout: int
12     number: int
13 }
14 type LostRequest: void {
15     lost: int
16     number: int
17     winnerNumber: int
18 }
19 interface PlayerCallBackInterface {
20     OneWay:
21     playerCallBack( PlayerCallBackRequest )      //
           ↪ definition of the call-back operation
22 }
23 interface PlayerGameInterface {
24     RequestResponse:
25         check( void )( void )
26     OneWay:
27         payout( PayoutRequest ),
```

```
28          lost( LostRequest )
29  }
30  type PlayerServiceParam {
31      location {
32          this: string
33          table: string
34      }
35      player {
36          name: string
37          wallet: int
38      }
39  }
40  service Player( p: PlayerServiceParam ) {
41      embed Time as Time
42      embed Scheduler as Scheduler
43      embed Console as Console
44      embed Math as Math
45      execution: sequential
46      outputPort TablePort {
47          location: p.location.table
48          protocol: sodep
49          interfaces: TableToPlayerInterface
50      }
51      // internal input port for receiving alarms from
            ↪ Scheduler
52      inputPort MySelf {
53          location: "local"
54          interfaces: PlayerCallBackInterface
55      }
56      inputPort PlayerGamePort {
57          location: p.location.this
58          protocol: sodep
59          interfaces: PlayerGameInterface
60      }
61      init {
62          enableTimestamp@Console( true )()
63          // setting the name of the callback operation
64          setCallbackOperation@Scheduler( { operationName =
                ↪  "playerCallBack" })
65          // setting cronjob
```

```
66          setCronJob@Scheduler( {
67              jobName = "bet"
68              groupName = "roulette"
69              cronSpecs << {
70                      second = "*/5"
71                      minute = "*"
72                      hour = "*"
73                      dayOfMonth = "*"
74                      month = "*"
75                      dayOfWeek = "?"
76                      year = "*"
77              }
78          })()
79          enableTimestamp@Console( true )()
80          global.player = p.player.name
81          global.wallet = p.player.wallet
82          global.location = p.location.this
83          println@Console("Player: " + global.player + "
              ↪ enabled on " + PlayerGamePort.location)()
84      }
85      main {
86          [ playerCallBack( request ) ] {
87                  // generate a straight up bet
88                  // number
89                  random@Math()( temp )
90                  number = int( temp * 37 )
91                  // amount
92                  random@Math()( temp )
93                  amount = int( temp * 100 )
94                  // senfing bet to table
95                  bet_req << {
96                      player = global.player
97                      amount = amount
98                      number = number
99                      player_location = global.location
100                 }
101                 synchronized( wallet ) {
102                     println@Console( "Player " + global.
                          ↪ player + " is trying to send
                          ↪ straight up bet on number " +
```

```
                    ↪ number + ", amount: " + amount )
                    ↪ ()
103                 if ( ( global.wallet - amount )  > 0
                    ↪ ) {
104                   scope( bet ) {
105                       global.wallet = global.wallet
                          ↪  - amount
106                       install( default =>
107                               global.wallet =
                                  ↪ global.wallet +
                                  ↪ amount
108                               println@Console("
                                  ↪ Error received
                                  ↪ from Table " +
                                  ↪ bet.( bet.(
                                  ↪ default) ) )()
109                       )
110                       straightUpBet@TablePort(
                          ↪ bet_req )( response )
111                       println@Console( "Table reply
                          ↪  " + response )()
112                   }
113                 } else {
114                     println@Console(  "Player " +
                       ↪ global.player + " skipped
                       ↪ bet because wallet is low ("
                       ↪  + global.wallet + ")" )()
115                 }
116             }
117         }
118         [ payout( request ) ] {
119             synchronized( wallet ) {
120                 global.wallet = global.wallet + request.
                       ↪ payout
121             }
122             println@Console( "Payout " + request.payout +
                   ↪  ", for number " + request.number + "
                   ↪ =>> current wallet " + global.wallet )()
123         }
124         [ lost( request ) ] {
```

```
125                 println@Console( "Lost " + request.amount +
                        ↪ "! The winner number is " + request.
                        ↪ winnerNumber + ", your number was " +
                        ↪ request.number + " =>> current wallet "
                        ↪ + global.wallet )()
126             }
127             [ check()() { nullProcess } ]
128         }
129 }
```

Il servizio `Player` una volta avviato esegue le operazioni di inizializzazione presenti in `init` (archi $(1,2)$, $(2,3)$, $(3,4)$, $(4,5)$, $(5,6)$, $(6,7)$, $(7,8)$, $(8,9)$, $(9,10)$), al termine delle quali il servizio è disponibile alla ricezione di quattro operazioni: `playerCallBack`, `payout`, `lost` e `check`. Nel caso venga ricevuta una richiesta di tipo `check` (arco $(10,11)$) la gestione di questa terminerà immediatamente, permettendo la ricezione di nuove operazioni (arco $(11,10)$). Altrimenti, nel caso giungano operazioni di tipo `lost` o `payout` (arco $(10,8)$) si avviserà l'utente dello stato aggiornato della scommessa attraverso operazioni di `println` (archi $(8,9)$). Infine, nel caso dell'operazione `playerCallBack` (arco $(10,12)$) si procederà alla generazione casuale della prossima scommessa e del relativo importo (archi $(12,13)$, $(13,14)$, $(14,15)$, $(15,16)$); si informa l'utente via terminale della scommessa da piazzare (archi $(16,17)$ e $(17,18)$). Se l'utente ha abbastanza credito nel proprio portafogli la scommessa verrà inoltrata verso il tavolo di gioco (archi $(18,19)$ e $(19,8)$), altrimenti si informerà l'utente del credito insufficiente (archi $(18,9)$ e $(9,10)$).

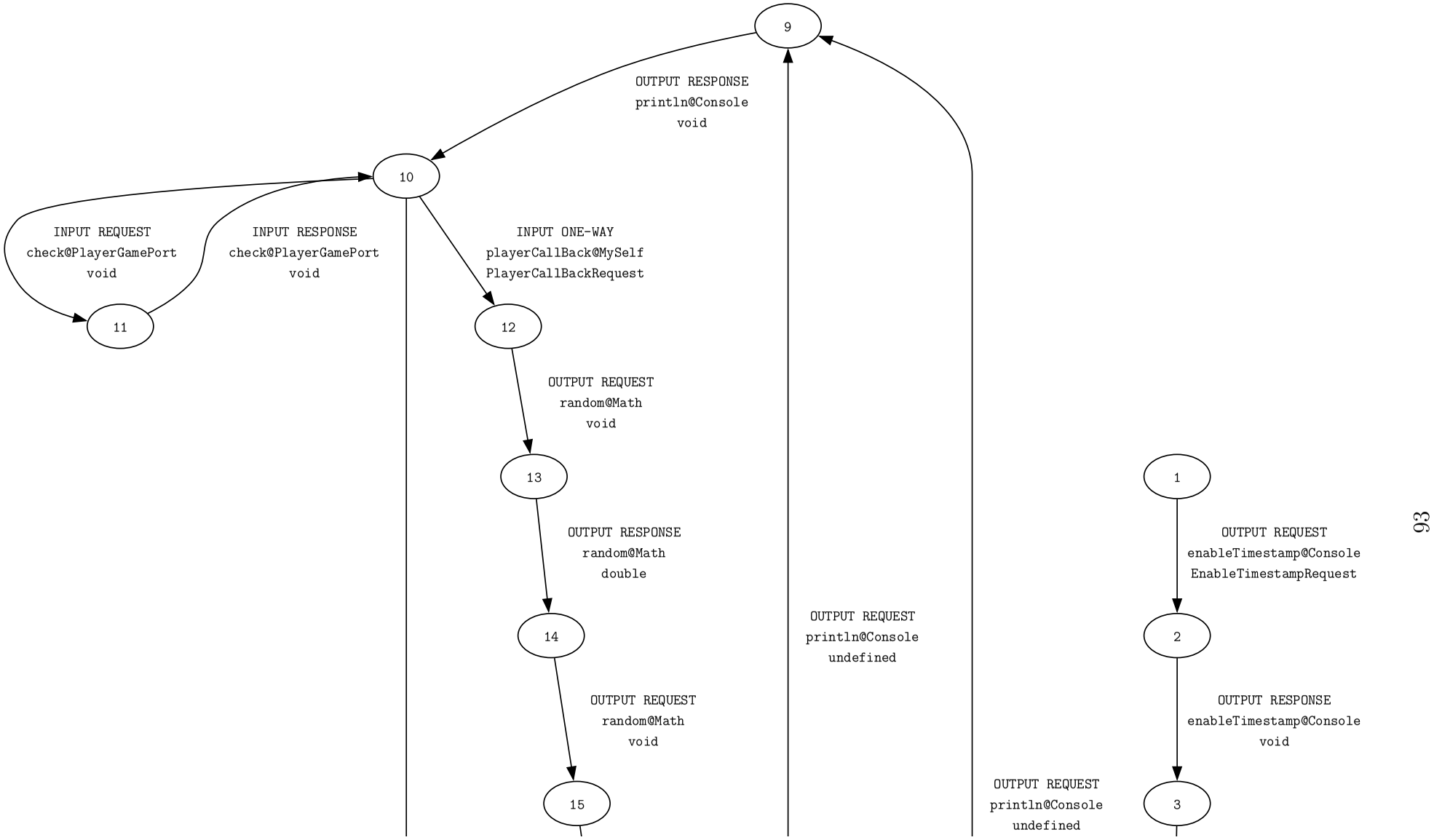


Figura 5.4: Automa del servizio **Player** - pt.1

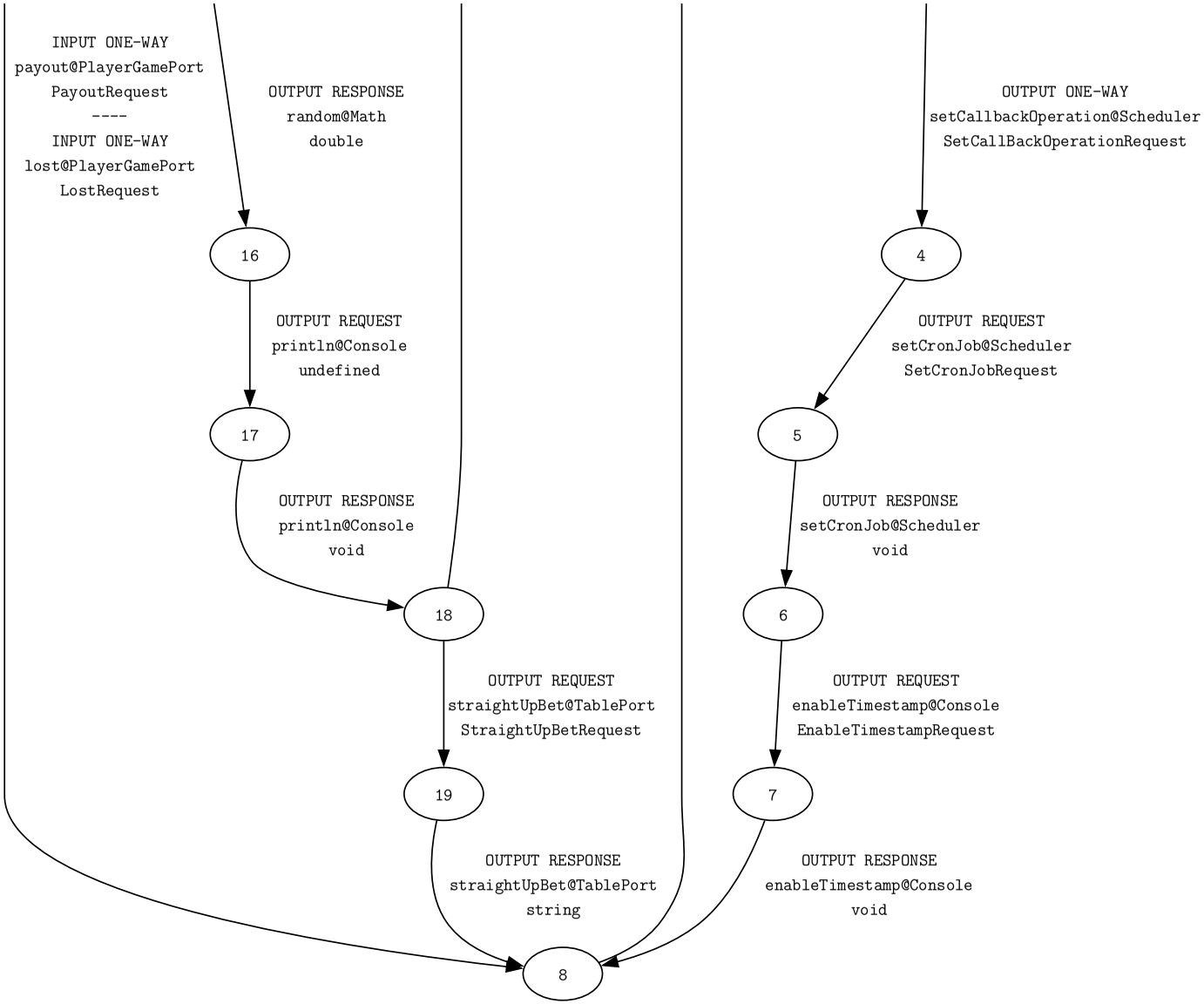


Figura 5.5: Automa del servizio **Player** - pt.2

Capitolo 6

Conclusioni

6.1 Sintesi dei risultati ottenuti

In questa tesi sono stati affrontati temi di grande rilevanza nel campo dei sistemi distribuiti e dei microservizi, con un focus particolare sull'analisi statica dei servizi scritti in Jolie. Il contributo principale è rappresentato dallo sviluppo di *JolieGraph*, un tool in grado di:

- Estrarre e analizzare l'Abstract Syntax Tree (AST) dei servizi Jolie.
- Generare automi che modellano il flusso di comunicazione tra i servizi.
- Produrre rappresentazioni grafiche mediante il formato DOT, facilitando l'interpretazione e il debugging dei flussi di comunicazione all'interno di servizi Jolie.

Il lavoro ha messo in luce come l'approccio formale, unito a una visualizzazione efficace, possa contribuire significativamente a comprendere le interazioni complesse in ambienti software eterogenei.

6.2 Limitazioni dell'approccio attuale

Nonostante i risultati ottenuti, l'approccio presenta alcune limitazioni che meritano di essere approfondite:

- **Execution Mode Concurrent:** Nella modalità di esecuzione *concurrent* vengono rese disponibili più sessioni in contemporanea. Questo comportamento non può essere rappresentata in maniera diretta attraverso grafi, evidenziando una lacuna nella modellizzazione dei sistemi concorrenti.

- **Utilizzo di DirectedGraph anziché DirectedMultiGraph:** Dal punto di vista teorico, gli automi dovrebbero essere modellati come multigrafi diretti, poiché possono esistere più transizioni (archi) tra due nodi con etichette diverse. Nell'implementazione attuale, sono stati impiegati dei semplici grafi diretti. Nei casi in cui esistano più transizioni tra due nodi, le etichette vengono concatenate su di un unico arco, così da rappresentare in modo completo le comunicazioni. L'utilizzo di questa classe non è immediato, in quanto non vengono consentiti loop tra i nodi dei multigrafi.

6.3 Prospettive e sviluppi futuri

Il lavoro svolto apre interessanti prospettive per ulteriori sviluppi:

- **Composizione in Coreografie:** Un aspetto particolarmente stimolante è la possibilità di comporre le viste locali dei servizi in coreografie. Le coreografie forniscono una rappresentazione globale e coordinata delle interazioni tra più servizi, evidenziando in modo integrato il flusso comunicativo. Il tool *Chorer*[18] permette la generazione di coreografie a partire da servizi Erlang, utilizzando DOT per la loro rappresentazione. Un prossimo obiettivo sarà quello di integrare *JolieGraph* con *Chorer*, permettendo la generazione automatica di coreografie per i servizi Jolie.
- **Parallel Statement e Diamond Pattern:** Un ulteriore sviluppo futuro riguarda l'aggiunta del supporto ai parallel statement. Ciò sarebbe realizzarbile attraverso l'impiego di *Diamond Pattern* per rappresentare le istruzioni eseguite in parallelo. Il *Diamond Pattern* è una struttura di controllo che permette di suddividere il flusso di esecuzione in più rami paralleli, i quali vengono successivamente riconciliati in un unico flusso. Questa configurazione, che assume la forma di un diamante, facilita la gestione della sincronizzazione tra i processi, garantendo che i dati elaborati in parallelo vengano allineati correttamente prima di proseguire con l'esecuzione del programma. Durante lo sviluppo bisognerà garantire che gli statement `install` abbiano una priorità maggiore, e quindi verranno eseguiti prima, dei `throw`.

In conclusione, sebbene l'approccio presentato presenti alcune limitazioni, i risultati ottenuti rappresentano un significativo passo avanti verso una migliore comprensione e analisi dei sistemi distribuiti basati su microservizi. Le prospettive di integrazione con altri tool e l'adozione di nuove tecniche di rappresentazione offrono interessanti spunti per sviluppi futuri, con l'obiettivo di migliorare ulteriormente la qualità, l'affidabilità e la visibilità dei flussi comunicativi nei sistemi software.

Riferimenti bibliografici

[1] Jolie community, "Jolie, the service-oriented programming language — jolie-lang.org." `https://www.jolie-lang.org/`. [Accessed 04-03-2025].

[2] F. Montesi, *JOLIE: a Service-oriented Programming Language*. PhD thesis, University of Bologna, 2010.

[3] E. Gansner, E. Koutsofios, and S. North, "Drawing graphs with dot," 2006.

[4] M. Gabbrielli and S. Martini, "Linguaggi di programmazione: principi e paradigmi.."

[5] Y. Gao, K. Salomaa, and S. Yu, "Transition complexity of incomplete dfas," *Fundamenta Informaticae*, vol. 110, pp. 143–158, 1 2011.

[6] A. de Luca and F. D'Alessandro, *Teoria degli Automi Finiti*. Springer Milan, 2013.

[7] J. Ellson, E. Gansner, L. Koutsofios, S. C. North, and G. Woodhull, "Graphviz— open source graph drawing tools," in *Lecture Notes in Computer Science*, pp. 483–484, Springer Berlin Heidelberg, 2002.

[8] JGraphT Project, "JGraphT: Java graph library." `https://jgrapht.org/`. Accessed March 8, 2025.

[9] JGraphT Project, "Defaultdirectedgraph (jgrapht documentation)." `https://jgrapht.org/javadoc/org.jgrapht.core/org/jgrapht/graph/DefaultDirectedGraph.html`. Accessed March 8, 2025.

[10] JGraphT Project, "Dotexporter (jgrapht documentation)." `https://jgrapht.org/javadoc-1.4.0/org/jgrapht/nio/dot/DOTExporter.html`. Accessed March 8, 2025.

[11] M. P. Papazoglou and D. Georgakopoulos, "Service-oriented computing: Introduction," *Communications of the ACM*, vol. 46, no. 10, pp. 24–28, 2003.

[12] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-oriented computing: State-of-the-art and research challenges," *IEEE Computer*, vol. 40, no. 11, pp. 38–45, 2007.

[13] N. Dragoni, S. Giallorenzo, A. Lluch-Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: Yesterday, today, and tomorrow," in *Present and Ulterior Software Engineering* (M. Mazzara and B. Meyer, eds.), pp. 195–216, Springer, 2017.

[14] N. Alshuqayran, N. Ali, and R. Evans, "A systematic mapping study in microservice architecture," in *Proceedings of the 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, pp. 44–51, 2016.

[15] Jolie community, "Introduction - Jolie Documentation — docs.jolie-lang.org." `https://docs.jolie-lang.org/v1.12.x/introduction/index.html`. [Accessed 04-03-2025].

[16] Jolie community, "libjolie 1.12.4 javadoc (org.jolie-lang) — javadoc.io." `https://javadoc.io/doc/org.jolie-lang/libjolie`, 2025. [Accessed 04-03-2025].

[17] B. Meyer and K. Arnout, "Componentization: The visitor example," *Computer*, vol. 39, pp. 23–30, 7 2006.

[18] G. Genovese and I. Lanese, *gabrielegenovese/chorer*. 2 2025.