ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

Dipartimento di Informatica – Scienza e Ingegneria (DISI)

Corso di Laurea in Informatica

Tesi di Laurea

Simulating a Racing Choreographies Language

Relatore:
Prof. Ivan Lanese

Presentata da:
Gianluca Casaburi

Sessione di Marzo 2026
Anno Accademico 2024/2025

Abstract

Il linguaggio Racing Choreographies estende il modello di programmazione coreografica introducendo il costrutto di race, che consente di rappresentare competizione tra processi concorrenti per fornire un valore a un ricevente, mantenendo traccia sia del vincitore sia del perdente e permettendo la successiva sincronizzazione tramite discharge.

Questa tesi sperimentale presenta la progettazione e l'implementazione di un tool per rendere eseguibili programmi in Racing Choreographies. Il lavoro integra la definizione di una grammatica ANTLR4, un parser in C++17 con costruzione di un AST e validazione strutturale post-parsing e un simulatore basato su una macchina astratta che implementa una semantica operazionale semplificata di Racing Choreographies direttamente eseguibile.

Alla mia famiglia, con gratitudine.

Indice

Capitolo 1

Introduzione

La progettazione di sistemi distribuiti richiede di coordinare più processi indipendenti che comunicano tra loro. Ogni processo interagisce con gli altri tramite scambio di messaggi. Il comportamento complessivo del sistema è dato dalla composizione di singoli comportamenti locali dei processi coinvolti.

La progettazione di questi sistemi presenta difficoltà strutturali. Nella letteratura sui sistemi concorrenti e distribuiti sono noti diversi problemi. I principali sono:

- mismatch nelle comunicazioni: ovvero quando un processo tenta di inviare un messaggio ma nessun processo nel programma è definito per riceverlo e quindi si ha una comunicazione nella coreografia che non è bilanciata da una corrispondente azione di ricezione.
- deadlock dovuti a specifiche incoerenti tra i partecipanti;

Nel modello tradizionale di progettazione di questi sistemi il comportamento di ciascun processo viene definito separatamente e la correttezza delle interazioni viene verificata a posteriori. Questo modello non rende esplicita la rappresentazione globale del sistema pertanto risulta complesso ragionare sul suo comportamento.

Per risolvere il limite di questo modello si usa un diverso paradigma che inverte la prospettiva: la programmazione coreografica [1]. Invece di definire prima i comportamenti locali dei singoli processi, si descrive direttamente l'interazione globale tra essi. Una coreografia viene definita come una specifica che elenca in modo esplicito le comunicazioni e le scelte che coinvolgono i partecipanti del sistema.

Questo paradigma è stato formalizzato con l'idea che una descrizione globale possa essere proiettata sui singoli processi preservando la coerenza delle interazioni e prevenendo errori di coordinamento tra i partecipanti [1, 2].

In una coreografia una comunicazione è espressa come:

```
p.e -> q.x;
```

che indica che il processo `p` valuta l'espressione `e` e invia il valore risultante al processo `q`, che lo assegna alla variabile `x`. La comunicazione è descritta una sola volta senza dover coordinare manualmente invio e ricezione nei singoli endpoint.

Si consideri ora un semplice servizio *echo*. Un client `c` invia un messaggio a un server `s` e il server restituisce lo stesso messaggio come risposta:

```
c.msg -> s.msg;
s.msg -> c.res;
```

La coreografia descrive direttamente l'interazione tra i due processi senza che vengano definiti separatamente il comportamento del client e quello del server.

Tuttavia, nei sistemi distribuiti reali non tutte le interazioni sono sequenziali. È frequente che più processi competano per fornire una risposta o per accedere a una risorsa condivisa. Questo introduce un elemento di non determinismo in quanto non è possibile conoscere a priori quale processo risponderà per primo.

Il linguaggio Racing Choreographies estende il modello coreografico introducendo un costrutto esplicito di competizione chiamato `race`.

Si consideri ora un'estensione dell'esempio precedente: oltre al server `s`, due worker `w1` e `w2` competono per fornire una risposta.

```
w1.r -> s.x;
w2.r -> s.x;
race s[k] : w1.r , w2.r -> s.ans;
```

La race rappresenta una competizione tra due processi che tentano di inviare un valore allo stesso ricevente. Il vincitore determina il valore ricevuto dal processo destinatario. Inoltre, è necessario tenere traccia anche del processo perdente, in modo da non perdere l'informazione prodotta e consentire una successiva sincronizzazione.

Il linguaggio Racing Choreographies rende quindi esplicito un fenomeno tipico dei sistemi distribuiti: la competizione tra eventi concorrenti e la necessità di gestire in modo coerente sia l'esito vincente sia quello perdente.

1.1 Scopo della tesi

Il linguaggio Racing Choreographies introduce una estensione del paradigma di programmazione coreografica tramite il costrutto di race. Tale estensione non appartiene al nucleo standard delle coreografie classiche. Il modello delle coreografie, infatti, è ampiamente studiato e formalizzato in letteratura [1, 2], ma i modelli esistenti solitamente non prevedono meccanismi espliciti di competizione a livello globale. Per questo motivo il lavoro presentato in questa tesi si colloca in un contesto sperimentale.

Il contributo è, partendo da una specifica teorica del linguaggio che integra le race nel modello coreografico, l'implementazione di un tool che permette di scrivere ed eseguire programmi secondo questo paradigma.

In particolare, il lavoro comprende:

- la formalizzazione della sintassi concreta del linguaggio;
- l'implementazione di un parser in C++ con costruzione di un Abstract Syntax Tree;
- la progettazione di una macchina astratta che realizza la semantica operazionale del linguaggio e la sua implementazione in un simulatore eseguibile.

1.2 Struttura della tesi

La tesi è organizzata in sei capitoli e segue una progressione che va dalla presentazione del modello teorico fino alla realizzazione concreta del tool sviluppato.

Capitolo 2 Presenta il background teorico. Viene descritto il linguaggio Racing Choreographies a partire dalla sua specifica formale, introducendo la sintassi astratta e la semantica operazionale su cui si basa il lavoro implementativo.

Capitolo 3 Affronta il passaggio dalla sintassi astratta alla sintassi concreta, la quale viene successivamente tradotta in una grammatica ANTLR che rappresenta la base per la generazione del lexer e del parser.

Capitolo 4 Descrive l'implementazione del front-end del tool, includendo la costruzione del parser tramite ANTLR, la generazione delle strutture sintattiche del programma e le fasi di validazione ed elaborazione effettuate prima della simulazione.

Capitolo 5 Presenta l'implementazione del simulatore, descrivendo la macchina astratta che esegue i programmi, le strutture runtime utilizzate e alcuni esempi di esecuzione del linguaggio.

Capitolo 2

Background teorico

Questo capitolo presenta il modello teorico del linguaggio Racing Choreographies seguendo la struttura della specifica formale del linguaggio, che definisce sintassi astratta, semantica operazionale e proprietà di correttezza del modello [3].

2.1 Il linguaggio Racing Choreographies

Racing Choreographies estende il modello delle coreografie classiche mantenendone i costrutti standard, in particolare le comunicazioni globali, e introducendo due costrutti che rappresentano la principale novità del linguaggio: `race` e `discharge`.

Accanto a questi, il linguaggio comprende anche gli altri costrutti tipici del modello coreografico, tra cui assegnamenti locali, selezioni di etichetta, condizionali, chiamate di procedura e ricorsione. In questa sezione il focus è posto soprattutto sulle comunicazioni, sulle `race` e sul `discharge` che costituiscono gli elementi più importanti del linguaggio.

Comunicazioni

```
p.e -> q.x;
```

Il costrutto base delle coreografie è la comunicazione globale, che indica che il processo `p` valuta localmente `e` e invia il valore risultante a `q`, che lo memorizza nella variabile locale `x`. In questo modo la comunicazione è descritta una sola volta a livello globale e determina simultaneamente il comportamento di entrambi i processi.

Race

Riprendiamo ora l'esempio citato nel Capitolo 1: si supponga che un client `c` invia una richiesta a un server `s`, e due worker `w1` e `w2` competono per fornire una risposta.

```
...
w1.r = req;
w2.r = req;
race s[k] : w1.r , w2.r -> s.ans;
...
```

Si nota come il costrutto `race` introduca una competizione tra due processi (`w1`, `w2`) che tentano di inviare un valore al medesimo ricevente (`s`). Questo costrutto rappresenta l'estensione principale introdotta da Racing Choreographies.

La definizione formale del costrutto dice che `p` e `q` competono per consegnare a `s` il valore risultante dalla valutazione di `e` ed `e'`. Il valore del vincitore viene scritto in `s.x`.

```
race s[k] : p.e , q.e' -> s.x
```

Si sottolineano tre aspetti fondamentali:

Determinazione del vincitore La race è non deterministica: entrambe le risoluzioni, vittoria del lato sinistro oppure del lato destro, sono ammissibili.

Persistenza dell'esito e del valore perdente Il valore del perdente non viene ignorato. Dopo la race, il processo perdente rimane in una situazione sospesa rispetto al ricevente e la competizione continua ad avere effetto anche nei passi successivi della computazione. In particolare ci sono due scenari che possono accadere.

Il primo è quello in cui il ricevente utilizza il costrutto `discharge` per recuperare il valore prodotto dal processo perdente. Il secondo è quello in cui tale sincronizzazione non avviene immediatamente: la computazione prosegue lungo il ramo determinato dal vincitore ma l'informazione relativa al perdente non viene eliminata. Se la coreografia contiene una chiamata ricorsiva, che riporta l'esecuzione in un contesto in cui quella race è ancora rilevante, allora il processo perdente rimane ancora in attesa rispetto a quella competizione. Questo significa che la race non produce soltanto un valore vincente ma lascia un effetto persistente sull'evoluzione successiva della coreografia.

Uso dell'esito nel flusso di controllo L'esito della race può guidare una scelta di controllo tramite un costrutto condizionale basato sul race value `s[k]`, cioè l'esito della race identificata da `k` sul processo `s`.

Discharge

```
discharge s[k] : p -> s.x;
```

Il costrutto di `discharge` realizza la sincronizzazione con il perdente. Nella specifica, il discharge è descritto come un'azione che consente al ricevente della race di ricevere anche il messaggio del processo perdente associato a quella race, dove `p` è il processo che ha perso la race `s[k]` e `s.x` è la variabile in cui il ricevente `s` memorizza il valore perdente.

Questo costrutto è importante perché permette al linguaggio di rendere esplicita una situazione in cui un evento perdente rispetto a una competizione può comunque essere rilevante e richiedere una sincronizzazione esplicita per non perdere informazione.

La specifica del linguaggio [3] discute anche l'interazione tra race e massimo parallelismo, formalizzata tramite regole di delay basate su nozioni di interferenza e indipendenza (condizioni di Bernstein [4]), oltre a condizioni di well-formedness [1] e componenti come calcolo target ed Endpoint Projection [5, 1]. Ai fini della tesi vengono presentate nelle sezioni successive solo la sintassi astratta e la semantica operazionale.

2.2 Sintassi astratta

La sintassi astratta definisce la struttura formale dei programmi e specifica le categorie sintattiche del linguaggio e le modalità di composizione dei costrutti.

Si introducono preliminarmente le metavariabili utilizzate nella definizione:

- `p,q,r,s` indicano processi;
- $\bar{p}$ indica una tupla di processi;
- `x` indica una variabile locale a un processo;
- `v` indica un valore;
- `l` indica un'etichetta;

- `k` indica un identificatore di race;
- `X` indica un nome di procedura.

Definizione completa della sintassi

La sintassi astratta del linguaggio può essere riassunta nel seguente schema:

$$
\begin{array}{lcl}
\mathbb{C} & ::= & X_{\bar{p}} = C_{i \in I} \\
C & ::= & I; C \mid \texttt{if } p.e \texttt{ then } C_1 \texttt{ else } C_2 \mid \texttt{if } s[k] \texttt{ then } C_1 \texttt{ else } C_2 \mid X_{\bar{p}} \mid 0 \mid \overline{X}_{\bar{p}} \\
I & ::= & p.e \to q.x \mid p \to q[L] \mid p.x = e \mid s[k] : p.e\ q.e' \perp s.x \mid s[k] : p \to s.x \\
e & ::= & v \mid x
\end{array}
$$

Programmi Un programma è composto da zero o più definizioni di procedure seguite da una coreografia principale:

$$P ::= D^* \ ; \ C$$

dove D^* indica una sequenza (eventualmente vuota) di definizioni di procedura e C rappresenta la coreografia principale.

Definizioni di procedura Una procedura è identificata da un nome X, da una tupla di processi partecipanti e da un corpo espresso come coreografia:

$$D ::= X(\bar{p}) = C$$

dove $\bar{p}$ rappresenta la tupla dei processi coinvolti nella procedura.

Coreografie Le coreografie descrivono il comportamento globale del sistema e sono definite dalla seguente grammatica:

$$
\begin{array}{rl}
C ::= & I; C \\
& \mid \texttt{if } p.e \texttt{ then } C \texttt{ else } C \\
& \mid \texttt{if } s[k] \texttt{ then } C \texttt{ else } C \\
& \mid X(\bar{p}) \\
& \mid 0
\end{array}
$$

Una coreografia può quindi essere:

- un'istruzione seguita da un'altra coreografia;
- un condizionale basato sulla valutazione locale di un'espressione `p.e`;
- un condizionale basato sull'esito di una race identificata da `s[k]`;
- una chiamata di procedura;
- il termine `0`, che rappresenta la terminazione.

Istruzioni Le istruzioni costituiscono le azioni elementari della coreografia:

$$\begin{aligned} I ::= {} & p.e \to q.x \\ & \mid p \to q[l] \\ & \mid p.x = e \\ & \mid \texttt{race}\ \ s[k] : p.e, q.e' \to s.x \\ & \mid \texttt{discharge}\ \ s[k] : p \Rightarrow s.x \end{aligned}$$

Le forme rappresentano rispettivamente:

- comunicazione di un valore dal processo `p` al processo `q`;
- selezione di etichetta;
- assegnamento locale;
- race tra due processi `p` e `q` con ricevente `s`;
- discharge del processo perdente della race.

Espressioni Le espressioni sono limitate a valori e variabili:

$$e ::= v \mid x$$

dove v rappresenta un valore e x una variabile.

2.3 Semantica operazionale

La semantica operazionale del linguaggio Racing Choreographies è definita nella specifica formale tramite una relazione di transizione su configurazioni globali. La versione presentata in questa sezione rappresenta una semplificazione della semantica originale che omette regole e aspetti più avanzati presenti nella specifica completa

che servono a dimostrare proprietà sul linguaggio [3]. In questo modo si ottiene un modello direttamente implementabile tramite una macchina astratta eseguibile.

La configurazione globale assume la forma:

$$C, \Sigma, M$$

dove:

- C è la coreografia residua;
- Σ è lo store delle variabili localizzate, cioè una funzione che associa a coppie costituite da un processo e da una variabile locale il valore attualmente memorizzato da quel processo;
- M è una memoria esplicita delle race ovvero una struttura che associa a ogni identificatore di race già risolto le informazioni necessarie a ricordarne l'esito, tra cui il lato vincitore, il lato perdente, il valore vincente e il valore perdente.

La relazione di transizione è:

$$C, \Sigma, M \to C', \Sigma', M'$$

Race

$$\frac{s[k] \notin dom(M) \qquad \Sigma(p, e) \downarrow v \qquad \Sigma(q, e') \downarrow v'}{C, \Sigma, M,\ s[k] : p.e\ q.e' \perp s.d; C \to C, \Sigma[s.d \mapsto v], M[s[k] \mapsto (L, R, v, v')], C} (\text{LRACE})$$

Se la race identificata da $s[k]$ non è ancora presente in M e le espressioni vengono valutate rispettivamente a v e v', allora nel caso di vittoria del lato sinistro viene assegnato a $s.d$ il valore v. In M viene memorizzata una tupla contenente l'informazione sul vincitore (L), sul perdente (R), sul valore vincente e sul valore perdente.

$$\frac{s[k] \notin dom(M) \qquad \Sigma(p, e) \downarrow v \qquad \Sigma(q, e') \downarrow v'}{C, \Sigma, M,\ s[k] : p.e\ q.e' \perp s.d; C \to C, \Sigma[s.d \mapsto v'], M[s[k] \mapsto (R, L, v', v)], C} (\text{RRACE})$$

Nel caso di vittoria del lato destro, viene assegnato a $s.d$ il valore v' e in M viene registrata la tupla corrispondente con indicazione del lato vincitore e dei valori coinvolti.

Le due regole mostrano che la race non si limita a scegliere un vincitore, ma lascia memoria dell'intera competizione. Questo è essenziale sia nel caso in cui il perdente venga immediatamente recuperato tramite `discharge`, sia nel caso in cui

la computazione prosegua e, anche attraverso chiamate ricorsive, si ritorni a una configurazione in cui l'informazione sul perdente risulta ancora rilevante.

Comunicazione

$$\frac{\Sigma(p, e) \downarrow v}{C, \Sigma, M,\ p.e \to q.x; C \to C, \Sigma[q.x \mapsto v], M, C}(\text{COM})$$

Il processo p valuta l'espressione e ottenendo v. Il valore viene assegnato alla variabile $q.x$ nello store. La memoria M non viene modificata.

Selezione

$$\frac{}{C, \Sigma, M,\ p \to q[L]; C \to C, \Sigma, M, C}(\text{SEL})$$

La selezione di etichetta non modifica né lo store né la memoria delle race; l'esecuzione prosegue con la coreografia successiva.

Assegnamento

$$\frac{\Sigma(p, e) \downarrow v}{C, \Sigma, M,\ p.x = e; C \to C, \Sigma[p.x \mapsto v], M, C}(\text{ASG})$$

Il valore dell'espressione viene assegnato alla variabile locale $p.x$ nello store.

Condizionali locali

$$\frac{\Sigma(p, e) \downarrow true}{C, \Sigma, M,\ if\ p.e\ then\ C_1\ else\ C_2 \to C, \Sigma, M, C_1}(\text{CNDT})$$

Se l'espressione valutata localmente è vera, viene selezionato il ramo then.

$$\frac{\Sigma(p, e) \downarrow false}{C, \Sigma, M,\ if\ p.e\ then\ C_1\ else\ C_2 \to C, \Sigma, M, C_2}(\text{CNDF})$$

Se l'espressione è falsa, viene selezionato il ramo else.

Condizionali su race

$$\frac{s[k] \in dom(M) \qquad M[s[k]] = (L, \_, \_, \_)}{C, \Sigma, M,\ if\ s[k]\ then\ C_1\ else\ C_2 \to C, \Sigma, M, C_1}(\text{RCNDT})$$

Se la memoria indica che il lato sinistro ha vinto, viene eseguito il ramo then.

$$\frac{s[k] \in dom(M) \qquad M[s[k]] = (R, \_, \_, \_)}{C, \Sigma, M,\ if\ s[k]\ then\ C_1\ else\ C_2 \to C, \Sigma, M, C_2}(\text{RCNDF})$$

Se la memoria indica che ha vinto il lato destro, viene eseguito il ramo else.

Discharge

$$\frac{s[k] \in dom(M) \qquad M[s[k]] = (w, \ell, v_{win}, v_{lose})}{C, \Sigma, M,\ discharge\ s[k] : \ell \to s.x; C \to C, \Sigma[s.x \mapsto v_{lose}], M, C}(\text{DIS})$$

Il valore del processo perdente, memorizzato in M, viene assegnato alla variabile $s.x$. La memoria M non viene rimossa, ma può essere marcata come già utilizzata a livello implementativo.

Chiamata di procedura

$$\frac{\Delta(X) = (\bar{p}_{form}, C_{body})}{C, \Sigma, M,\ call\ X(\bar{p}_{act}); C \to C, \Sigma, M,\ subst(C_{body}, \bar{p}_{form} \mapsto \bar{p}_{act}); C}(\text{CALL})$$

La chiamata di procedura viene realizzata tramite sostituzione dei parametri formali con quelli attuali nel corpo della procedura.

Terminazione

Il termine 0 rappresenta una configurazione terminale in quanto non ci sono ulteriori regole applicabili e la computazione termina.

Capitolo 3

Dalla sintassi astratta alla grammatica

Nel capitolo precedente è stata introdotta la sintassi astratta del linguaggio Racing Choreographies insieme alla semantica operazionale.

Tuttavia, la sintassi astratta descrive la struttura logica dei programmi indipendentemente dalla loro rappresentazione testuale e per poter scrivere ed eseguire programmi è necessario definire una sintassi concreta e una grammatica per il parser.

Il processo seguito è stato quello di progettare una sintassi concreta poi formalizzata in una grammatica espressa in Extended Backus–Naur Form (EBNF) e tradotta infine in una grammatica ANTLR [6].

Ai fini della comprensione del linguaggio viene illustrato il passo più rilevante ovvero la progettazione della sintassi concreta, che stabilisce come i costrutti teorici presentati nel capitolo precedente vengono effettivamente scritti nei programmi.

La grammatica completa utilizzata per l'implementazione del parser è disponibile nella repository del progetto nel file `RacingChoreo.g4` nella cartella `/grammar` [7].

3.1 Principi e requisiti della sintassi concreta

La progettazione della sintassi concreta di Racing Choreographies è stata guidata dalle seguenti scelte progettuali secondo uno stile C-like.

Separazione tra definizioni e coreografia principale

Un programma è composto da zero o più definizioni di procedura seguite da una coreografia principale identificata dalla keyword `main`. Questa scelta rende esplicito il punto di ingresso dell'esecuzione del programma.

Uso di keyword esplicite

I costrutti principali del linguaggio sono introdotti tramite keyword riservate, tra cui `proc`, `main`, `call`, `race`, `discharge`, `if` ed `else`.

L'uso di parole chiave esplicite facilita la distinzione tra identificatori del programma e costrutti del linguaggio.

Struttura a blocchi

Le coreografie sono racchiuse tra parentesi graffe `{ ... }`, come accade in molti linguaggi di programmazione imperativi in quanto questo consente di organizzare in modo chiaro sequenze di istruzioni e rami condizionali.

Terminazione implicita della coreografia

Il termine di terminazione `0`, presente nella sintassi astratta, non viene rappresentato esplicitamente nella sintassi concreta. La chiusura di un blocco corrisponde implicitamente alla terminazione della coreografia contenuta.

Sequenzialità esplicita tramite punto e virgola

Le istruzioni e le chiamate di procedura sono terminate dal punto e virgola `;`, rendendo esplicita la sequenza delle azioni. I costrutti condizionali, invece, sono strutturati come blocchi e non richiedono il punto e virgola finale.

Distinzione tra condizionali locali e su race

Le due forme di condizionale previste dalla sintassi astratta sono rese sintatticamente distinte:

```
if (p.e) { ... } else { ... }

if (s[k]) { ... } else { ... }
```

Nel primo caso la condizione dipende dalla valutazione locale di un'espressione appartenente a un processo. Nel secondo caso la condizione dipende dall'esito di una race identificata da `s[k]`.

Espressioni semplici

Le espressioni del linguaggio sono volutamente limitate a valori letterali o identificatori. Non sono presenti operatori aritmetici poiché il focus del linguaggio è descrivere interazioni tra processi piuttosto che calcoli locali.

3.2 Sintassi concreta attraverso un esempio

Per illustrare la sintassi concreta del linguaggio Racing Choreographies si riprende l'esempio completo di programma introdotto precedentemente che utilizza i costrutti del linguaggio.

Il programma descrive una interazione tra quattro processi: un client `c`, due worker `w1` e `w2`, e un server `s`. Il server avvia i due worker tramite un messaggio di attivazione. I worker producono quindi due possibili contributi e competono per fornire una risposta tramite una `race`. Il valore prodotto dal worker vincitore viene inviato al client, mentre il valore del worker perdente viene successivamente recuperato tramite il costrutto di `discharge`.

Poiché il linguaggio non prevede operatori aritmetici né funzioni di calcolo, i valori prodotti dai worker sono stati rappresentati per comodità come costanti numeriche assegnate localmente.

Da notare come in questo esempio le select non sono strettamente necessarie poiché il comportamento del client è identico nei due rami ma sono state inserite solo per rendere esplicito quale worker abbia prodotto il risultato.

Così appare il programma completo secondo la sintassi concreta:

```
proc Request(c, w1, w2, s) {
    s.start -> w1.start;
    s.start -> w2.start;
    w1.r = 7;
    w2.r = 5;

    race s[k] : w1.r , w2.r -> s.ans;

    if (s[k]) {
        s.ans -> c.res;
        s -> c[FromW1];
        discharge s[k] : w2 -> s.lost;
    } else {
        s.ans -> c.res;
        s -> c[FromW2];
        discharge s[k] : w1 -> s.lost;
    }
}

main {
    call Request(c, w1, w2, s);
}
```

Il programma è composto da una definizione di procedura e da una coreografia principale.

La procedura `Request` descrive l'interazione tra i quattro processi coinvolti, mentre il blocco `main` rappresenta il punto di ingresso dell'esecuzione del programma e invoca la procedura tramite la keyword `call`.

Le prime istruzioni

```
s.start -> w1.start;
s.start -> w2.start;
```

rappresentano comunicazioni globali: il processo `s` invia un messaggio di attivazione ai due worker `w1` e `w2`. Il valore trasmesso viene memorizzato nelle rispettive variabili `start` dei worker.

Le istruzioni successive

```
w1.r = 7;
w2.r = 5;
```

sono assegnamenti locali, che permettono ai worker di produrre i valori che verranno utilizzati nella competizione.

La competizione è espressa dal costrutto di `race`:

```
race s[k] : w1.r , w2.r -> s.ans;
```

In questa istruzione i processi `w1` e `w2` competono per fornire un valore al processo `s`. Il valore del processo vincitore viene assegnato alla variabile `s.ans`. L'identificatore `s[k]` rappresenta la race con id `k` e consente di riferirsi successivamente al suo esito.

L'esito della competizione viene utilizzato nel costrutto condizionale

```
if (s[k]) { ... } else { ... }
```

che seleziona il ramo di esecuzione in base al vincitore della race.

Nel ramo selezionato il valore vincente viene inviato al client tramite una comunicazione:

```
s.ans -> c.res;
```

Successivamente il server invia al client una selezione di etichetta (`FromW1` oppure `FromW2`) per indicare quale worker ha prodotto il risultato.

Infine, il costrutto

```
discharge s[k] : w1 -> s.lost
discharge s[k] : w2 -> s.lost
```

consente di recuperare il valore prodotto dal processo perdente della race. Questo valore viene memorizzato nella variabile `s.lost`, rendendo esplicita la sincronizzazione con il processo che non ha vinto la competizione.

Capitolo 4

Implementazione del parser

Il contributo di questa tesi è l'implementazione di un tool per l'esecuzione di programmi scritti nel linguaggio *Racing Choreographies*. Il sistema realizzato comprende due componenti principali: un front-end del linguaggio, responsabile dell'analisi lessicale e grammaticale dei programmi e della costruzione della loro rappresentazione interna (AST), e un simulatore che esegue i programmi secondo la semantica operazionale presentata nel Capitolo 2.

L'architettura del tool è organizzata come una pipeline di trasformazioni successive che partono dal codice sorgente del programma e producono rappresentazioni via via più strutturate.

Nella prima fase il codice sorgente del programma viene passato in input a un *lexer*, che produce una sequenza di tokens. Questa sequenza viene poi fornita in input a un *parser*, il quale costruisce un *Concrete Syntax Tree* (CST), ovvero una rappresentazione ad albero che riflette fedelmente la struttura sintattica del programma secondo la grammatica del linguaggio, includendo tutti i dettagli derivanti dalle regole grammaticali.

Successivamente, il CST viene trasformato in un *Abstract Syntax Tree* (AST). L'AST rappresenta una versione più compatta e astratta della struttura sintattica del programma: elimina gli elementi sintattici non necessari (ad esempio nodi intermedi della grammatica o simboli ausiliari) e mantiene solo le informazioni rilevanti per il modello del linguaggio come descritto nel Capitolo 2.

Una volta costruito l'AST, il tool esegue una fase di validazione che verifica alcune proprietà strutturali dei programmi. Se il programma risulta valido, l'AST può essere utilizzato dal simulatore (Capitolo 5) per eseguire la coreografia secondo la semantica operazionale vista nel Capitolo 2.

In questo capitolo viene descritto il front-end del tool.

Il flusso generale del sistema è rappresentato nella Figura 4.1.

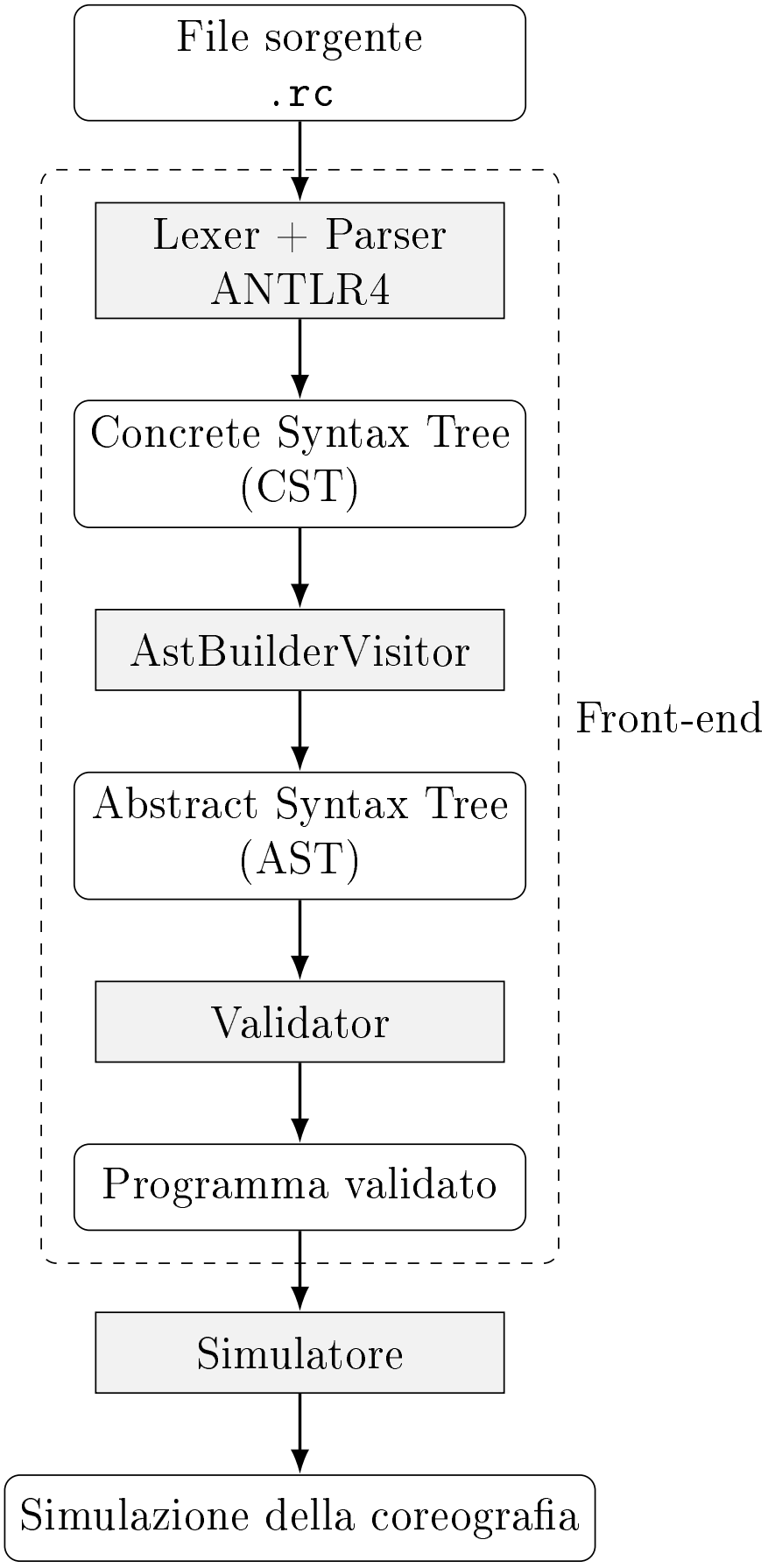


Figura 4.1: Pipeline generale del tool

4.1 Architettura del front-end

Il front-end è costruito utilizzando il framework ANTLR4 [6], che permette di generare automaticamente lexer e parser in diversi linguaggi di programmazione; per questo tool è stato utilizzato C++. A partire dalla grammatica definita nel Capitolo 3, ANTLR genera il codice necessario per riconoscere la struttura sintattica dei programmi e costruire il loro albero sintattico. L'implementazione completa del tool è disponibile nel repository del progetto [7]. In particolare, lexer e parser si trovano nella directory `/generated`, mentre la grammatica è definita nel file `RacingChoreo.g4` all'interno della directory `/grammar`.

Il flusso del front-end generato da ANTLR è descritto nel seguente schema:

File sorgente (`.rc`)
↓
Lexer (ANTLR)
↓
Parser (ANTLR)
↓
Concrete Syntax Tree (CST)

Supponiamo ora di avere un *file.rc* scritto secondo la sintassi concreta vista nel Capitolo 2 che rappresenta il programma dell'esempio introdotto nei capitoli precedenti, in cui due worker $w1$ e $w2$ competono tramite una race per inviare un valore al processo s. Consideriamo ora questo frammento di codice del programma:

```
race s[k] : w1.r , w2.r -> s.ans;
```

Quando questo programma viene dato in input al tool, la prima trasformazione consiste nella conversione del codice sorgente in una sequenza di tokens. Il tool permette di visualizzare questo risultato tramite il comando:

```
> rc_parser tokens <file.rc>
```

L'output del lexer mostra i token riconosciuti nel programma insieme alla loro posizione nel file sorgente. Ad esempio, per l'istruzione precedente si ottiene:

```
6:4  RACE   "race"
6:9  ID     "s"
6:10 LBRACK "["
6:11 ID     "k"
6:12 RBRACK "]"
6:14 COLON  ":"
6:16 ID     "w1"
6:18 DOT    "."
6:19 ID     "r"
6:21 COMMA  ","
6:23 ID     "w2"
6:25 DOT    "."
6:26 ID     "r"
6:28 ARROW  "->"
6:31 ID     "s"
6:32 DOT    "."
6:33 ID     "ans"
6:36 SEMI   ";"
```

Ogni riga dell'output indica la posizione del token nel file sorgente (linea e colonna), il tipo di token riconosciuto e il testo corrispondente nel programma.

La sequenza di token prodotta dal lexer costituisce l'input per il parser, che costruisce il *Concrete Syntax Tree*. Il CST rappresenta la struttura sintattica del programma e contiene nodi che riflettono direttamente le produzioni grammaticali.

Nel caso dell'istruzione di race il parser costruisce un sottoalbero del CST che segue direttamente le produzioni della grammatica utilizzate per riconoscere tale istruzione. I nodi dell'albero corrispondono quindi sia ai token del programma (ad esempio `RACE`, `ID`, `COLON`, `ARROW`) sia alle regole grammaticali che strutturano l'istruzione.

Di conseguenza il CST contiene nodi per ciascun elemento sintattico della frase inclusi simboli puramente strutturali come parentesi quadre, virgole e separatori. Per l'istruzione di race l'albero include, ad esempio, nodi distinti per il costrutto `race`, per l'identificatore del processo ricevente `s`, per la variabile `k`, per le due espressioni concorrenti `w1.r` e `w2.r`, per il simbolo `->` e per la destinazione `s.ans`.

Questa rappresentazione è utile per verificare che il programma rispetti la sintassi del linguaggio, ma risulta troppo dettagliata per le fasi successive del sistema. Molti nodi dell'albero rappresentano infatti solo elementi della sintassi concreta e non

corrispondono ai costrutti del modello del linguaggio. Per questo motivo il CST viene trasformato in un *Abstract Syntax Tree* (AST).

4.2 Abstract Syntax Tree

Il front-end del tool introduce una seconda rappresentazione interna del programma ovvero l'*Abstract Syntax Tree* (AST) che rappresenta il programma in una forma più compatta e più vicina alla struttura del linguaggio. In particolare, mentre il CST riflette il modo in cui il programma è stato riconosciuto dalla grammatica, l'AST mette direttamente in evidenza i costrutti del linguaggio, come procedure, blocchi di istruzioni, interazioni e costrutti di controllo.

Consideriamo ora il *file.rc* del programma dell'esempio introdotto in precedenza per vedere come viene rappresentato. Il tool permette di stampare una rappresentazione testuale sul terminale dell'albero tramite il comando:

```
> rc_parser ast file.rc
```

L'output relativo al programma di esempio è il seguente:

```
Program
  Procedures (1)
    ProcDef Request(c,w1,w2,s)
      Block (6 stmt)
        InteractionStmt
          Comm s.start -> w1.start
        InteractionStmt
          Comm s.start -> w2.start
        InteractionStmt
          Assign w1.r = 7
        InteractionStmt
          Assign w2.r = 5
        InteractionStmt
          Race s[k] : w1.r , w2.r -> s.ans
        IfRace (s[k])
        Then:
          Block (3 stmt)
            InteractionStmt
              Comm s.ans -> c.res
            InteractionStmt
              Select s -> c [FromW1]
            InteractionStmt
              Discharge s[k] : w2 -> s.lost
        Else:
          Block (3 stmt)
            InteractionStmt
              Comm s.ans -> c.res
            InteractionStmt
              Select s -> c [FromW2]
            InteractionStmt
              Discharge s[k] : w1 -> s.lost
  Main
    Block (1 stmt)
      Call Request(c,w1,w2,s)
```

L'implementazione completa dell'AST si trova nella directory `/src` nel repository del progetto [7]. Di seguito vengono mostrate sono le principali parti di codice per descrivere le scelte progettuali che sono state fatte.

Alla radice dell'albero si trova il nodo `Program`, che contiene l'insieme delle de-

finizioni di procedura e il blocco principale `Main`. Questa organizzazione riflette direttamente la struttura dei programmi del linguaggio.

In tutte le strutture dell'AST compare inoltre il campo `SourceRange loc`. Questo campo rappresenta la posizione del costrutto nel programma sorgente e viene utilizzato per associare ai nodi dell'albero le informazioni di posizione (file, riga e colonna). In questo modo il tool può produrre messaggi di errore che fanno riferimento direttamente al codice sorgente.

```
struct Program {
    std::vector<std::unique_ptr<ProcDef>> procedures;
    std::unique_ptr<Main> main;
    SourceRange loc;
};
```

Nell'implementazione il nodo radice è definito come una struttura che contiene una lista di procedure e il nodo principale.

```
struct Block {
    std::vector<std::unique_ptr<Stmt>> statements;
    SourceRange loc;
};
```

Si osserva che il corpo della procedura `Request` è rappresentato come un nodo `Block` contenente una sequenza di statement questo perchè, invece di mantenere nell'albero la struttura ricorsiva della grammatica, l'AST rappresenta esplicitamente i blocchi come sequenze di istruzioni.

```
using Stmt = std::variant<
    InteractionStmt,
    CallStmt,
    IfLocalStmt,
    IfRaceStmt
>;
```

Gli statement del linguaggio comprendono diverse forme sintattiche, tra cui interazioni, chiamate di procedura e condizionali. Nell'implementazione queste alternative sono modellate tramite un tipo somma.

Questa soluzione consente di rappresentare in modo tipizzato le diverse forme che uno statement può assumere evitando di utilizzare gerarchie di classi.

L'AST non viene generato direttamente dal parser. Il parser di ANTLR produce infatti un CST che riflette la grammatica del linguaggio ma la costruzione dell'AST consiste in una trasformazione a partire dal CST.

Questa trasformazione è implementata tramite il meccanismo dei *visitor* fornito da ANTLR. In particolare, la classe `AstBuilderVisitor` è un'estensione della classe `RacingChoreoBaseVisitor` generata automaticamente da ANTLR, nella quale vengono ridefiniti i metodi di visita necessari per costruire i nodi dell'AST.

Il tool definisce una classe `AstBuilderVisitor` che attraversa ricorsivamente il CST e costruisce i nodi dell'AST.

```
class AstBuilderVisitor : public RacingChoreoBaseVisitor {
public:
    std::any visitProgram(
        RacingChoreoParser::ProgramContext* ctx) override;
};
```

Durante la visita dei nodi del CST, il visitor estrae le informazioni rilevanti e costruisce i nodi corrispondenti dell'AST.

Consideriamo nuovamente l'istruzione di `race` presente nel programma come esempio per capire il funzionamento:

```
race s[k] : w1.r , w2.r -> s.ans;
```

Nel CST, come abbiamo visto in precedenza, questa istruzione è rappresentata da diversi nodi sintattici derivanti dalla grammatica. Il visitor raccoglie invece solo le informazioni di nostro interesse e costruisce il nodo dell'AST:

```
struct Race {
    RaceId id;
    ProcExpr left;
    ProcExpr right;
    ProcVar target;
    SourceRange loc;
};
```

Questa struttura rappresenta esplicitamente i componenti semantici della `race`: l'identificatore della competizione, le due espressioni concorrenti e la variabile del processo ricevente.

4.3 Validazione del programma

Una volta costruito l'*Abstract Syntax Tree*, il front-end del tool esegue una fase di validazione del programma. Questa fase ha lo scopo di verificare alcune proprietà strutturali del programma.

Come mostrato nella pipeline descritta all'inizio del capitolo, la validazione si colloca immediatamente dopo la costruzione dell'AST e prima dell'esecuzione della simulazione. Il validatore opera quindi su una rappresentazione del programma già indipendente dalla grammatica.

Il compito principale di questa fase è individuare errori che non riguardano la sintassi del programma, ma la sua struttura o l'uso scorretto dei costrutti del linguaggio. In particolare, il validatore verifica le seguenti proprietà:

- **Corretta definizione e invocazione delle procedure.** Il validatore verifica che ogni chiamata di procedura faccia riferimento a una procedura effettivamente definita nel programma e che non ci siano riferimenti a nomi di procedura inesistenti.

- **Coerenza dei parametri nelle chiamate di procedura.** Per ogni istruzione `call`, viene controllato che il numero di processi passati come argomenti corrisponda al numero di parametri formali dichiarati nella definizione della procedura.

- **Uso coerente dei processi nelle interazioni.** Il validatore verifica che i processi utilizzati nelle comunicazioni, nelle selezioni e negli altri costrutti del linguaggio siano tra quelli dichiarati nel contesto della procedura corrente.

- **Uso consistente degli identificatori di race.** Per i costrutti `race`, `if (s[k])` e `discharge`, viene controllata la coerenza dell'identificatore della race e del processo associato, in modo da garantire che tali riferimenti siano utilizzati in modo consistente all'interno della coreografia.

- **Presenza delle informazioni necessarie per l'esecuzione.** Il validatore verifica che il programma contenga gli elementi minimi necessari per l'esecuzione. In particolare controlla la presenza del blocco `main`, che il programma

sia composto correttamente dal punto di vista strutturale (ovvero zero o più definizioni di procedura seguite dalla coreografia principale) e che le definizioni di procedura siano coerenti evitando duplicazioni di nomi tra procedure.

Questi controlli sono implementati nel modulo `Validation` del front-end, che attraversa l'AST e verifica le proprietà richieste. In caso di errore, il validatore produce un messaggio associato alla posizione corrispondente nel file sorgente sfruttando le informazioni di localizzazione propagate durante la costruzione dell'AST.

4.4 Gestione degli errori

Gli errori possono emergere in tre momenti distinti del processo:

- durante l'analisi lessicale e sintattica;
- durante la validazione strutturale del programma;
- durante l'esecuzione della simulazione.

Nel front-end della pipeline gli errori lessicali e sintattici vengono intercettati tramite un *error listener* che estende il meccanismo di gestione degli errori fornito da ANTLR. Esso permette di intercettare gli errori sintattici tramite l'interfaccia `BaseErrorListener`. Nel tool è stata definita una classe `ErrorListener` che estende questo meccanismo per raccogliere in modo strutturato le informazioni sugli errori rilevati durante il parsing. L'implementazione si trova nei file `ErrorListener` nella directory `/src` [7]

```
struct SyntaxError {
    std::string file;
    size_t line;
    size_t column;
    std::string message;
    std::string offendingText;
};
```

La classe ha una struttura dati che registra per ciascun errore:

- il file sorgente;
- la posizione dell'errore (linea e colonna);

- il messaggio generato dal parser;
- il token che ha causato l'errore.

Quando il parser rileva un errore, ANTLR invoca automaticamente il metodo `syntaxError`. Il metodo salva le informazioni relative all'errore all'interno di una lista mantenuta dall'oggetto listener.

Errori lessicali

Gli errori lessicali vengono individuati dal lexer quando incontra sequenze di caratteri che non corrispondono ad alcun token definito nella grammatica. Consideriamo un programma banale per mostrare il funzionamento:

```
main { @ }
```

Il simbolo `@` non è definito nella grammatica del linguaggio e il lexer produce quindi un errore di riconoscimento del token e si ottiene:

```
err_lex_01.rc:1:7: error: token recognition error at: '@'
  main { @ }
         ^
```

L'errore indica la posizione esatta del carattere non riconosciuto nel file sorgente.

Errori sintattici

Gli errori sintattici vengono rilevati dal parser quando la sequenza di token prodotta dal lexer non soddisfa le regole della grammatica del linguaggio. Consideriamo una istruzione di `race` in cui manca la virgola tra le due espressioni concorrenti:

```
race p[k] : p.x q.y -> p.z;
```

Il parser segnala:

```
err_01.rc:4:18: error: missing ',' at 'q'
    race p[k] : p.x q.y -> p.z;
                     ^
```

In modo analogo vengono segnalati errori dovuti a simboli mancanti o strutture incomplete, come ad esempio:

- espressioni mancanti in un assegnamento;
- variabili incomplete in una comunicazione;
- blocchi mancanti nei costrutti condizionali.

Errori strutturali

Come abbiamo visto nella sezione precedente una volta completato il parsing e costruito l'AST il tool esegue una fase di validazione del programma. In questa fase vengono individuati errori che non riguardano la sintassi del programma ma la sua struttura logica. Consideriamo ad esempio il seguente frammento di codice nel blocco per capire come vengono gestiti questi tipi di errori`main`:

```
main {
    call Request(c, w1, w2, s);
}
```

In questo caso il programma tenta di invocare la procedura `Request`, che non è stata definita tra le procedure del programma. Durante la fase di validazione il tool rileva questa inconsistenza e produce il seguente messaggio di errore:

```
ok_01.rc:20:4: error: call to undefined procedure 'Request'
      call Request(c, w1, w2, s);
      ^
```

Errori di esecuzione

Un ultima categoria di errori è quella degli errori che possono emergere durante l'esecuzione della coreografia nel simulatore. La gestione di questi errori è implementata all'interno della macchina di simulazione e viene descritta nel Capitolo 5.

Capitolo 5

Implementazione del simulatore

Nel Capitolo 4 è stato descritto il front-end del tool grazie al quale si ottiene un programma validato che può essere passato al simulatore.

Il simulatore ha il compito di eseguire il programma rappresentato nell'AST secondo la semantica operazionale introdotta nel Capitolo 2. In particolare il suo scopo è quello di interpretare tale programma e produrre un'evoluzione esecutiva osservabile.

Il simulatore riceve in input un AST già costruito e validato e mantiene durante l'esecuzione uno stato globale della simulazione. Tale stato comprende lo store delle variabili dei processi, una memoria dedicata alle race e una traccia degli eventi eseguiti. A questi elementi si aggiunge una struttura di supporto per gestire l'avanzamento nei blocchi di istruzioni e le chiamate di procedura.

L'output della simulazione produce una traccia passo-passo delle azioni svolte e lo stato finale delle componenti runtime.

Il flusso ad alto livello del funzionamento del tool è illustrato in Figura 5.1.

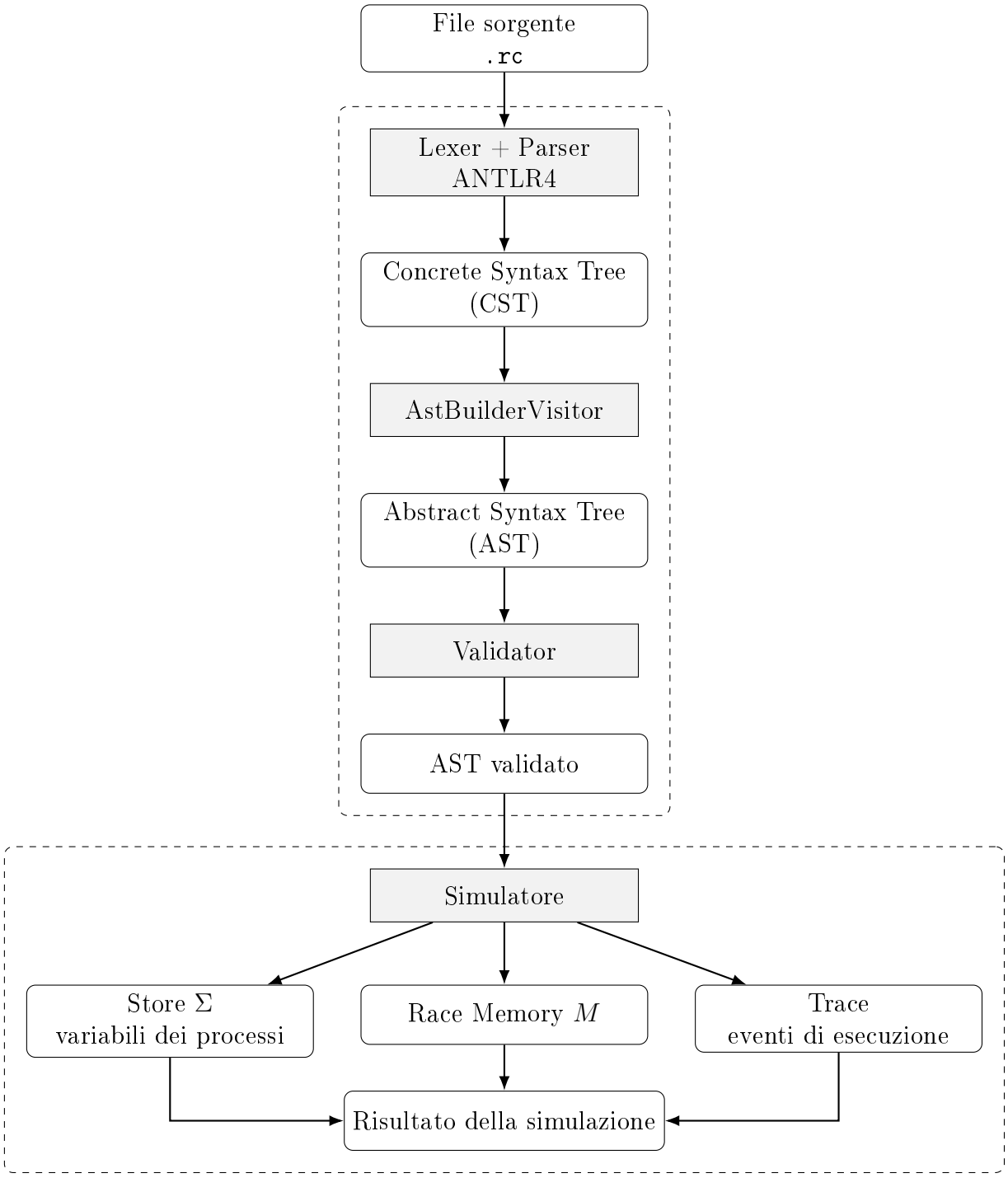
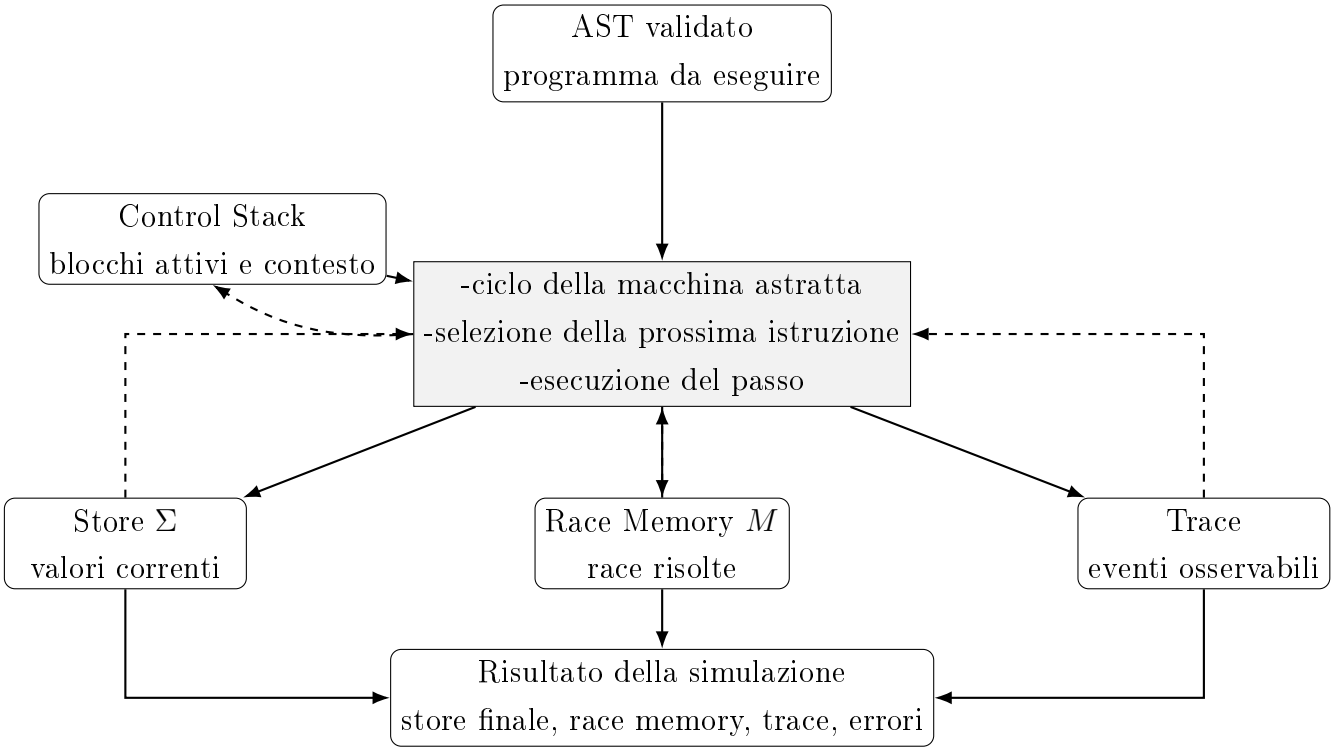


Figura 5.1: Flusso del funzionamento completo del tool dal file sorgente al risultato della simulazione.

5.1 Architettura del simulatore

In questa sezione viene descritta in maggiore dettaglio l'architettura del simulatore. Il simulatore è una macchina astratta che esegue il programma mantenendo esplicitamente lo stato della simulazione. L'esecuzione è modellata attraverso un insieme di strutture runtime che rappresentano la configurazione corrente della computazione.

L'architettura è descritta dal seguente schema:



L'elemento di partenza è l'AST validato prodotto dal front-end. Esso costituisce l'input della simulazione. Il simulatore attraversa progressivamente l'AST per determinare quale istruzione debba essere eseguita in ogni passo.

Per governare l'avanzamento nell'AST il simulatore utilizza una struttura di controllo esplicita, rappresentata da una pila di frame di esecuzione. Questa componente, indicata nello schema come *Control Stack*, mantiene il contesto corrente della simulazione: il blocco attivo, la posizione della prossima istruzione da eseguire e le informazioni per gestire le chiamate di procedura.

Il nucleo del simulatore è costituito dal ciclo della macchina astratta che, ad ogni iterazione, individua la prossima istruzione da eseguire, ne determina l'effetto e aggiorna le strutture runtime. Questo ciclo realizza la transizione da una configurazione di esecuzione alla successiva traducendo in termini operativi il comportamento

del linguaggio descritto formalmente nel Capitolo 2 ovvero:

$$C, \Sigma, M \to C', \Sigma', M'$$

Le strutture runtime aggiornate dal ciclo di esecuzione sono tre:

Store Σ. Lo store rappresenta la memoria delle variabili dei processi. Nell'implementazione esso è modellato come una mappa che associa a ogni coppia processo-variabile il valore corrente corrispondente. Tutte le operazioni che modificano i valori del programma come assegnamenti, comunicazioni o scritture prodotte da una race hanno come effetto un aggiornamento dello store.

Race Memory M. La memoria delle race è la struttura dedicata che per ogni race risolta conserva le informazioni necessarie a descriverne l'esito: i partecipanti, il vincitore, il perdente, i valori associati e lo stato di eventuale `discharge`. La scelta di una struttura dedicata serve per separare logicamente queste informazioni dallo store in quanto l'esito di una race non è solo un valore scritto in una variabile ma anche un'informazione di controllo che viene riutilizzata successivamente ad esempio nei costrutti `if (s[k])` e `discharge`.

Trace. La trace serve a raccogliere la sequenza degli eventi osservabili generati durante la simulazione. Ogni passo esecutivo può produrre un evento che descrive l'azione svolta, come un assegnamento, una comunicazione, la risoluzione di una race o una selezione. Questa componente ha il ruolo di mostrare l'output della simulazione permettendo di seguire l'evoluzione della coreografia in modo esplicito.

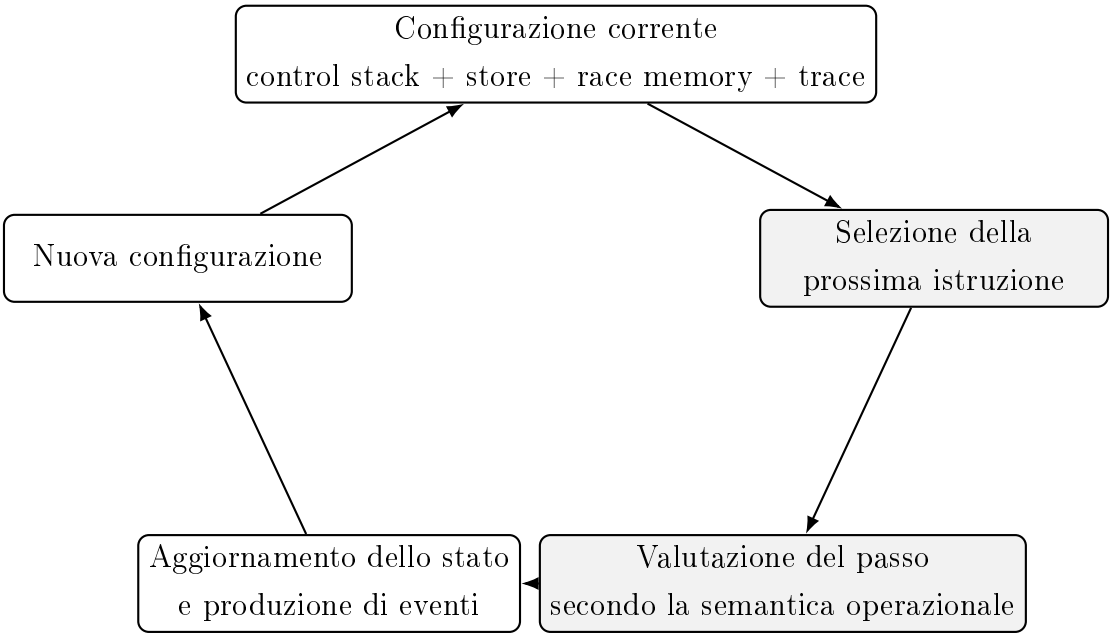
Oltre a queste tre componenti principali il simulatore utilizza alcuni elementi ausiliari necessari al controllo dell'esecuzione, come il contatore dei passi, la profondità delle chiamate di procedura, la policy di risoluzione delle race e il generatore pseudo-casuale usato nei casi nondeterministici. L'implementazione completa di tali elementi e delle strutture principali descritte in questa sezione è disponibile nei file delle directory `/src/sim` e `/src/runtime` nel repository del progetto [7].

5.2 Modello di esecuzione

Dal punto di vista operativo il simulatore esegue un programma come una successione di passi elementari. Ad ogni passo la macchina astratta considera la configurazione corrente della simulazione, individua la prossima istruzione da eseguire e ne

applica l'effetto aggiornando lo stato runtime. Nel simulatore per rendere eseguibile la semantica del linguaggio, descritta come una relazione di transizione tra configurazioni, si utilizza un ciclo di esecuzione che legge una configurazione corrente, produce una nuova configurazione e ripete il processo sul nuovo stato ottenuto.

Il modello di esecuzione adottato è rappresentato nel seguente schema:



Per descrivere il funzionamento del modello si consideri il seguente frammento di programma dell'esempio introdotto nei capitoli precedenti. In questo esempio il processo `s` invia un messaggio ai due worker per avviarne l'elaborazione. I worker producono quindi due possibili contributi che partecipano alla race. Poiché nel linguaggio considerato non sono ancora presenti operazioni aritmetiche o funzioni, i contributi dei worker sono rappresentati da costanti.

```
s.start -> w1.start;
s.start -> w2.start;

w1.r = 7;
w2.r = 5;

race s[k] : w1.r , w2.r -> s.ans;
```

Assumiamo inoltre che la simulazione si trovi all'inizio del blocco che contiene queste istruzioni e che la memoria delle race e la trace siano inizialmente vuote. L'evoluzione del modello può allora essere rappresentata come una sequenza di configurazioni.

Configurazione iniziale. All'inizio dell'esecuzione lo store non contiene ancora valori associati alle variabili del programma, mentre la race memory e la trace risultano vuote.

```
Control stack:
  ip -> s.start -> w1.start;

Store:
  <empty>

Race Memory:
  <empty>

Trace:
  <empty>
```

Dopo il primo passo. La macchina seleziona l'istruzione `s.start -> w1.start`. Il valore del messaggio viene trasferito dal server al primo worker e lo store viene aggiornato con la variabile `w1.start`. Il control stack avanza alla successiva istruzione e la trace registra l'evento di comunicazione.

```
Istruzione eseguita:
  s.start -> w1.start

Nuova configurazione:

Control stack:
  ip -> s.start -> w2.start;

Store:
  w1.start = start

Race Memory:
  <empty>

Trace:
  com s.start -> w1.start
```

Dopo il secondo passo. Viene ora eseguita la comunicazione verso il secondo worker. Anche in questo caso il simulatore aggiorna lo store e registra l'evento nella trace.

```
Istruzione eseguita:
  s.start -> w2.start

Nuova configurazione:

Control stack:
  ip -> w1.r = 7;

Store:
  w1.start = start
  w2.start = start

Race Memory:
  <empty>

Trace:
  com s.start -> w1.start
  com s.start -> w2.start
```

Dopo il terzo passo. Il primo worker produce ora il proprio contributo assegnando il valore costante `7` alla variabile `w1.r`. Lo store viene aggiornato e la trace registra l'evento di assegnamento.

```
Istruzione eseguita:
  w1.r = 7

Nuova configurazione:

Control stack:
  ip -> w2.r = 5;

Store:
  w1.start = start
  w2.start = start
  w1.r = 7

Race Memory:
  <empty>

Trace:
  com s.start -> w1.start
  com s.start -> w2.start
  asg w1.r = 7
```

Dopo il quarto passo. Anche il secondo worker produce il proprio contributo, assegnando la costante `5` alla variabile `w2.r`. A questo punto lo store contiene entrambi i valori che parteciperanno alla race.

```
Istruzione eseguita:
  w2.r = 5

Nuova configurazione:

Control stack:
  ip -> race s[k] : w1.r , w2.r -> s.ans;

Store:
  w1.start = start
  w2.start = start
  w1.r = 7
  w2.r = 5

Race Memory:
  <empty>

Trace:
  com s.start -> w1.start
  com s.start -> w2.start
  asg w1.r = 7
  asg w2.r = 5
```

Dopo il quinto passo. Viene infine eseguita l'istruzione di race. Il simulatore valuta i due contributi concorrenti `w1.r` e `w2.r` e aggiorna di conseguenza sia lo store sia la race memory. Supponiamo che il vincitore risulti `w1`: il valore `7` viene quindi scritto nella variabile target `s.ans`, mentre nella memoria delle race vengono registrate le informazioni sulla competizione appena risolta.

```
Istruzione eseguita:
  race s[k] : w1.r , w2.r -> s.ans

Nuova configurazione:

Control stack:
  ip -> fine del frammento

Store:
  w1.start = start
  w2.start = start
  w1.r = 7
  w2.r = 5
  s.ans = 7

Race Memory:
  s[k] -> winner=w1, loser=w2, vWinner=7, vLoser=5

Trace:
  com  s.start -> w1.start
  com  s.start -> w2.start
  asg  w1.r = 7
  asg  w2.r = 5
  race s[k] winner=w1 loser=w2 write s.ans=7
```

Con questo esempio è stato mostrato il principio generale del modello di esecuzione: ad ogni passo la macchina astratta legge la configurazione corrente, seleziona l'istruzione puntata dal control stack, applica l'effetto operativo corrispondente e produce una nuova configurazione. L'esecuzione completa della coreografia è dunque semplicemente la successione di queste transizioni.

Il modello di esecuzione descritto nella sezione precedente richiede una struttura che permetta al simulatore di determinare quale istruzione del programma debba

essere eseguita ad ogni passo. Nel simulatore questo compito è svolto da una struttura esplicita di controllo citata in precedenza chiamata *Control Stack*. Essa serve a rappresentare il contesto corrente dell'esecuzione.

L'AST costruito dal front-end rappresenta il programma come una gerarchia di blocchi di istruzioni (`Block`) contenenti sequenze di `Stmt`. Per poter attraversare questa struttura durante la simulazione, il simulatore mantiene una pila di frame di esecuzione. Ogni frame descrive il blocco attualmente attivo e la posizione della prossima istruzione da eseguire al suo interno.

La struttura utilizzata per rappresentare un frame è la seguente:

```
struct BlockFrame {
    const ast::Block* block = nullptr;
    size_t ip = 0;
    std::unordered_map<std::string, std::string> subst;

    std::string procName;
    ast::SourceRange callLoc;
};
```

Il campo `block` identifica il blocco di istruzioni attualmente in esecuzione, mentre `ip` (*instruction pointer*) indica la posizione della prossima istruzione all'interno del blocco. In questo modo il simulatore può accedere direttamente allo statement corrente e avanzare progressivamente all'interno della sequenza di istruzioni.

Il campo `subst` rappresenta invece la sostituzione tra parametri formali e processi effettivi nelle chiamate di procedura. Quando una procedura viene invocata tramite un'istruzione `call`, i processi passati come argomenti vengono associati ai parametri della procedura e la sostituzione risultante viene memorizzata nel frame corrispondente. Questo permette al simulatore di risolvere correttamente i nomi dei processi all'interno del corpo della procedura.

Gli ultimi due campi vengono utilizzati per gestire il ritorno dalla procedura e per registrare correttamente l'evento corrispondente nella trace.

Durante la simulazione il control stack evolve dinamicamente in base alla struttura del programma. All'inizio dell'esecuzione viene inserito nella pila un frame relativo al blocco `main`.

Quando il simulatore incontra una chiamata di procedura, viene creato un nuovo frame corrispondente al corpo della procedura invocata e questo frame viene inserito in cima alla pila. L'esecuzione prosegue quindi all'interno della procedura. Una volta

terminato il blocco della procedura, il frame viene rimosso dalla pila e il controllo ritorna al frame precedente.

La scelta di gestire le chiamate di procedura tramite una pila di frame, invece di affidare l'esecuzione alla ricorsione del linguaggio e quindi delegare parte dello stato della computazione allo stack del runtime, è stata fatta per rendere la configurazione della macchina completamente esplicita. In questo modo il simulatore mantiene il controllo diretto dello stato dell'esecuzione e può gestire in modo più semplice la produzione della trace degli eventi.

Si è mostrato quindi come ad ogni passo la macchina astratta seleziona l'istruzione puntata dal *control stack*, ne valuta l'effetto e aggiorna lo stato runtime della simulazione. L'effetto prodotto dipende ovviamente dal tipo costrutto: alcune istruzioni aggiornano lo store, altre modificano il flusso di controllo e altre, come i costrutti legati alle *race*, coinvolgono anche la race memory. L'implementazione completa è disponibile nei file della directory `/src/sim` del repository del progetto [7].

5.3 Errori runtime e policy di risoluzione delle race

Come anticipato nel Capitolo 4, oltre agli errori lessicali, sintattici e strutturali, il tool produce anche errori che emergono solo durante l'esecuzione della simulazione.

Nel simulatore questi errori vengono rappresentati tramite un'eccezione che associa al messaggio di errore anche la posizione corrispondente nel programma sorgente. L'implementazione di questo meccanismo è disponibile nel modulo runtime del progetto [7].

Un caso interessante riguarda di errori sono quelli dei i costrutti legati alle race. Poiché `if (s[k])` e `discharge` dipendono dalle informazioni registrate nella *race memory*, il simulatore verifica che tali informazioni siano presenti e coerenti con l'istruzione eseguita.

Ad esempio, nel seguente frammento il costrutto `discharge` tenta di recuperare il contributo del processo `w1`, anche se questo processo risulta vincitore della race:

```
race s[k] : w1.r , w2.r -> s.ans;

if (s[k]) {
    discharge s[k] : w1 -> s.lost;
}
```

In questo caso il simulatore rileva l'incoerenza e produce il seguente errore a runtime:

```
.\file.rc:11:8: runtime error:
discharge expects loser 'w2', got 'w1'
```

Un altro esempio si verifica quando un costrutto tenta di accedere a una race che non è stata ancora risolta. Ad esempio:

```
discharge s[k] : w2 -> s.lost;
race s[k] : w1.r , w2.r -> s.ans;
```

Poiché la race `s[k]` non è ancora stata eseguita, l'informazione richiesta non è disponibile e il simulatore segnala:

```
.\file.rc:5:4: runtime error: race 's[k]' not resolved
```

Un ulteriore aspetto importante del simulatore riguarda la gestione delle policy di risoluzione delle race. Il linguaggio infatti tramite le race introduce un elemento di nondeterminismo. Per rendere questo comportamento eseguibile il simulatore supporta tre diverse policy:

- `left`, che seleziona sempre il contributo di sinistra;
- `right`, che seleziona sempre il contributo di destra;
- `random`, che sceglie il vincitore in modo pseudo-casuale.

Questa scelta è stata fatta perché permette di usare il simulatore sia in modo deterministico, permettendo di analizzare in modo controllato specifici scenari di esecuzione, sia in modo pseudo-casuale permettendo di osservare possibili evoluzioni alternative della stessa coreografia.

Nel caso si usi utilizzi come policy quella random il generatore pseudo-casuale può essere inizializzato con un seed esplicito. Questo è stato fatto per permettere di ottenere esecuzioni riproducibili. Inoltre variando il seed è possibile esplorare diversi possibili esiti della race.

L'implementazione delle policy di risoluzione e dei controlli runtime è disponibile nei file del simulatore e del runtime del repository del progetto, in particolare nelle directory `/src/sim` e `/src/runtime` [7].

5.4 Esempio completo di simulazione

Per concludere la descrizione del simulatore in questa sezione viene considerato un esempio completo di esecuzione.

Consideriamo il programma introdotto nei capitoli precedenti in cui un client `c` invia una richiesta a un server `s`. Due worker, `w1` e `w2`, producono due possibili risultati in modo concorrente. Il server utilizza una `race` per determinare quale contributo debba essere considerato vincente. Il valore vincente viene inviato al client mentre il contributo perdente viene recuperato successivamente tramite il costrutto `discharge`. La simulazione produce il seguente output:

```
init @<init>:0:0 w1.req = 7
init @<init>:0:0 w2.req = 5

call @file.rc:20:4 Request(c,w1,w2,s)

asg @file.rc:3:4 w1.r = 7
asg @file.rc:4:4 w2.r = 5

race @file.rc:6:4 s[k] winner=w1 loser=w2 write s.ans=7
ifRace @file.rc:8:4 s[k] winner=w1 -> then

com @file.rc:9:8 s.ans = 7 -> c.res
sel @file.rc:10:8 s -> c [FromW1]

dis @file.rc:11:8 s[k] loser=w2 write s.lost=5
ret @file.rc:20:4 Request


Final Store Sigma:
  c.res = 7
  s.ans = 7
  s.lost = 5

Final Races M:
  s[k]: left=w1, right=w2, winner=w1, loser=w2,
        vWin=7, vLose=5, discharged=true
```

Supponiamo che i due worker dispongano dei valori iniziali `w1.req = 7` e `w2.req = 5`. Nel simulatore questi valori vengono forniti esplicitamente come configurazione iniziale dello store tramite le opzioni `-init` della riga di comando.

Questo esempio mostra come il simulatore renda osservabile l'evoluzione della coreografia: la trace descrive la sequenza delle azioni eseguite, lo store finale riassume l'effetto della computazione sui valori del programma e la memoria delle race conserva l'esito delle competizioni.

La trace descrive l'evoluzione della simulazione. I primi eventi `init` non corrispondono a istruzioni del programma ma sono generati dal simulatore quando applica allo store iniziale i valori passati tramite le opzioni `-init` della riga di comando. In questo caso essi introducono nello store i valori iniziali disponibili per i due worker.

L'esecuzione inizia con l'evento che indica l'ingresso nella procedura `Request`:

```
call Request(c,w1,w2,s)
```

Dal punto di vista del modello di esecuzione questo evento corrisponde alla creazione di un nuovo frame nel *control stack*, che rappresenta il contesto della procedura invocata.

I due eventi successivi costruiscono i contributi concorrenti prodotti dai worker. Questi passi aggiornano lo store assegnando alle variabili dei worker i valori che verranno confrontati nella `race`:

```
asg w1.r = 7
asg w2.r = 5
```

Successivamente si verifica l'evento di `race`:

```
race s[k] winner=w1 loser=w2 write s.ans=7
```

Qui il simulatore valuta i due contributi concorrenti e determina il vincitore secondo la policy di risoluzione configurata. In questo caso il valore prodotto da `w1` risulta vincente. Il simulatore scrive quindi il valore `7` nella variabile `s.ans` e registra l'esito della competizione nella *race memory*. Quest'ultima conserva informazioni strutturate sulla race tra cui: i partecipanti, il vincitore, il perdente e i valori associati ai due contributi.

Il passo successivo mostra come il costrutto `if (s[k])` venga valutato consultando la *race memory*.

```
ifRace s[k] winner=w1 -> then
```

Nel ramo selezionato il server invia al client il valore vincente:

```
com s.ans = 7 -> c.res
```

Subito dopo viene eseguita una selezione

```
sel s -> c [FromW1]
```

Infine il costrutto `discharge` recupera il contributo perdente dalla *race memory*:

```
dis s[k] loser=w2 write s.lost=5
```

Lo store finale riassume l'effetto complessivo dell'esecuzione:

- `c.res = 7` rappresenta il risultato inviato al client;
- `s.ans = 7` contiene il valore vincente della race;
- `s.lost = 5` contiene il contributo perdente recuperato tramite `discharge`.

Lo stato finale della *race memory* conserva una descrizione strutturata della competizione:

```
s[k]: left=w1, right=w2, winner=w1, loser=w2,
      vWin=7, vLose=5, discharged=true
```

Capitolo 6

Conclusioni e sviluppi futuri

Questa tesi ha presentato la progettazione e l'implementazione di un tool per la scrittura e l'esecuzione di programmi nel linguaggio *Racing Choreographies*.

Come discusso nel Capitolo 2, la macchina astratta implementa una versione semplificata della semantica operazionale rispetto a quella descritta nel modello teorico del linguaggio *Racing Choreographies*.

Il contributo principale di questa tesi consiste nell'aver realizzato uno strumento che costituisce una prima base operativa di un linguaggio ancora in fase sperimentale nonostante non copra ancora tutti gli aspetti teorici e semantici presenti nella specifica completa.

Proprio per questo il lavoro svolto apre a diverse direzioni di sviluppo futuro. Un primo sviluppo naturale consiste nell'estendere il simulatore in modo da che realizzi la semantica completa definita nel modello teorico del linguaggio.

Inoltre il modello teorico del linguaggio considera ulteriori proprietà che non sono state implementate in questo lavoro tra cui: le condizioni di well-formedness delle coreografie, il supporto al massimo parallelismo tramite regole di delay basate sulle condizioni di Bernstein e le proprietà strutturali necessarie alla corretta endpoint projection.

Un'ulteriore direzione di lavoro, particolarmente importante nel contesto della programmazione coreografica, è l'introduzione della *Endpoint Projection*. Il tool sviluppato in questa tesi opera interamente a livello di coreografia globale ma un passo successivo sarà proiettare tale descrizione sui singoli endpoint ottenendo comportamenti locali coerenti con la specifica globale.

In conclusione, il risultato principale di questa tesi è aver trasformato una specifica formale sperimentale in un primo artefatto eseguibile. Il tool realizzato offre una base utile per futuri studi sul linguaggio *Racing Choreographies* e per successive evoluzioni della sua implementazione.

Bibliografia

[1] Fabrizio Montesi. *Introduction to Choreographies*. Cambridge: Cambridge University Press, 2023. ISBN: 978-1-108-83376-9. DOI: 10.1017/9781108981491.

[2] Kohei Honda, Nobuko Yoshida e Marco Carbone. «Foundations of Session Types and Behavioural Contracts». In: *ACM Computing Surveys* 49.4 (2016), pp. 1–36. DOI: 10.1145/2873052.

[3] Ivan Lanese e Saverio Giallorenzo. *Racing Choreographies – Draft Specification*. Unpublished manuscript. 2025.

[4] A. J. Bernstein. «Analysis of Programs for Parallel Processing». In: *IEEE Transactions on Electronic Computers* EC-15.5 (1966), pp. 757–763. DOI: 10.1109/PGEC.1966.264565.

[5] Kohei Honda, Nobuko Yoshida e Marco Carbone. «Multiparty Asynchronous Session Types». In: *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 2008, pp. 273–284. DOI: 10.1145/1328438.1328472.

[6] ANTLR Project. *ANTLR 4 Documentation*. 2024. URL: https://www.antlr.org/.

[7] Janji03. *Racing-Choreographies-parser: A C++ parser using ANTLR for a language Racing-Choreographies*. GitHub repository. 2026. URL: https://github.com/Janji03/Racing-Choreographies-Simulator.