UNIVERSITÀ DEGLI STUDI DI ROMA
TOR VERGATA

FACOLTÀ DI INGEGNERIA
Corso di Laurea Magistrale in Ingegneria Informatica

TESI DI LAUREA IN
Ingegneria Informatica

Una metodologia Model-Driven Engineering per lo sviluppo di Domain-Specific Languages

Relatore:
Prof. **A. Pellegrini**
Correlatori:
Dr. **Romolo Marotta**
Prof. **Guglielmo De Angelis**

Laureando:
Simone Bauco
Matr. **0324495**

ANNO ACCADEMICO 2023/2024

A Maria Chiara,
per il suo infinito sostegno

Indice

Abstract

L'obiettivo di questa tesi è esplorare l'applicazione dell'approccio Model-Driven Engineering (MDE) allo sviluppo di Domain-Specific Languages (DSL), concentrandosi sulla definizione di una rappresentazione intermedia basata sul modello ad attori. Ciò semplifica in modo significativo lo sviluppo dei DSL, rendendo accessibile l'uso di piattaforme di calcolo ad alte prestazioni, comprese quelle eterogenee, a esperti di dominio che non dispongono delle competenze tecniche necessarie per gestire le specificità di tali ambienti d'esecuzione.

A sostegno di questo approccio, la tesi include un caso di studio sullo Stream Processing speculativo, in cui un DSL ispirato a SQL per la specifica di query sugli stream viene tradotto in codice C ottimizzato per un ambiente di esecuzione speculativa. Questo caso di studio dimostra l'efficacia dell'approccio MDE nel facilitare la parallelizzazione e la distribuzione del carico computazionale.

1. Introduzione

In questo capitolo, verrà introdotto il sistema progettato e realizzato nell'ambito di questa tesi, per quanto riguarda lo scenario applicativo, gli obiettivi preposti, ed una rapida panoramica del lavoro svolto e dei risultati ottenuti.

1.1 Scenario Applicativo

Durante gli ultimi anni, la crescente esigenza computazionale ha portato gli sviluppatori di software a dover affrontare le complessità di piattaforme di calcolo ad alte prestazioni. Di conseguenza, lo sviluppo di applicazioni ottimizzate ed efficienti ha come requisito fondamentale la conoscenza di paradigmi complessi, che è spesso limitata ai soli professionisti nel mondo dell'ICT: ciò porta esperti di domini diversi da quello informatico a dover affrontare aspetti di basso livello che non sono di loro competenza, rallentando - o spesso impedendo - lo sviluppo di applicazioni efficienti che possano supportare le loro attività.

Ad esempio, la realizzazione di Domain-Specific Language (DSL), cioè linguaggi di programmazione progettati e ottimizzati per risolvere problemi specifici in un determinato dominio applicativo, richiede una duplice competenza: lo sviluppatore, oltre a conoscere l'ambito in cui si pone il DSL, deve padroneggiare gli strumenti di programmazione disponibili per la specifica architettura hardware.

Un caso particolare è dato dall'impiego di architetture eterogenee, divenute ormai uno standard nell'High Performance Computing. Queste architetture combinano diversi tipi di unità di elaborazione, ad esempio CPU, GPU e FP-

GA, all'interno di un singolo sistema, al fine di ottimizzare le prestazioni e l'efficienza energetica. Esse offrono la possibilità di sfruttare i punti di forza di ciascun componente hardware, diversificando l'esecuzione in base alle caratteristiche del carico di lavoro. D'altra parte, sviluppare applicazioni ottimizzate per architetture eterogenee richiede una padronanza approfondita dei metodi e degli strumenti di programmazione relativi ad ogni singolo componente hardware coinvolto, per cui tali prodotti restano, ad oggi, di diffusione limitata nei domini diversi dall'ICT, escludendo quei casi in cui siano disponibili dei supporti capaci di nascondere la loro intrinseca complessità.

1.2 Obiettivi

Lo scopo di questa tesi è di esplorare una direzione di ricerca che è stata aperta negli ultimi anni, che prevede una rivalutazione completa dello stack software, allo scopo di astrarre allo sviluppatore gli aspetti di basso livello che caratterizzano le architetture hardware: in cima al nuovo stack, sono presenti i Domain Specific Language (DSL), che sono realizzati da professionisti del dominio di riferimento; nella proposta presentata in questa tesi, questi linguaggi vengono tradotti in un modello intermedio, basato sull'Actor Model, cioè un paradigma computazionale ideato per l'elaborazione parallela. Il modello intermedio è infine tradotto in codice di basso livello, ottimizzato per l'architettura hardware di riferimento, eventualmente eterogenea, che può essere compilato utilizzando toolchain di compilazione standard per le diverse architetture.

1.3 Panoramica del Lavoro Svolto

Entrando nei dettagli, il lavoro di tesi ha riguardato l'implementazione, secondo i principi appena esposti, di un Domain-Specific Language per lo stream processing speculativo, eseguibile sull'ambiente di esecuzione speculativa ROOT-Sim. La catena di compilazione è supportata da una trasformazione

Model-to-Model del DSL in una rappresentazione intermedia, basata sul livello degli attori.

Lo stream di dati è generato a partire dal dataset statico «TLC Trip Record Data», contenente dati relativi a tutte le corse di taxi che avvengono sull'intero territorio della città di New York.

Il DSL di alto livello è un linguaggio, ispirato da SQL, che consente di specificare query sul flusso di dati. Questo linguaggio viene tradotto nel modello ad attori intermedio, che viene infine compilato in codice C eseguibile su un ambiente di esecuzione parallelo e distribuito preso dalla letteratura.

Un aspetto fondamentale, che verrà meglio affrontato in seguito, riguarda la traduzione dal DSL verso il modello ad attori: questa fase prevede l'individuazione di una serie di attori, derivati dai singoli operatori del DSL, che comunicano tra di loro in base ad una topologia, anch'essa sintetizzata a partire dall'insieme di query.

Il sistema sviluppato in questa tesi sarà confrontato con Apache Spark, un motore di elaborazione dati ampiamente utilizzato per l'analisi su larga scala, che fornisce API per l'esecuzione di query SQL su flussi di dati.

L'analisi si concentrerà sulle prestazioni del sistema, valutando sia la capacità di elaborazione che l'efficienza nell'utilizzo delle risorse, e confrontando i risultati con quelli ottenuti da esecuzioni analoghe su Apache Spark.

1.4 Risultati

I risultati delle analisi di prestazione, presentati in dettaglio in una sezione successiva, evidenziano come lo stack software proposto in questa tesi possa offrire, in alcuni casi, performance superiori rispetto a un sistema consolidato e specializzato come Apache Spark.

Questo dimostra l'efficacia dell'approccio basato sul Model-Driven Engineering presentato in questa tesi, il quale semplifica significativamente lo sviluppo di linguaggi domain-specific ottimizzati per architetture di basso livello. Grazie a questo approccio, è possibile sfruttare appieno le potenzialità delle

architetture hardware target, raggiungendo un elevato grado di ottimizzazione sia in termini di prestazioni che di efficienza delle risorse.

2. Contesto

Nel corso di questo capitolo, verrà contestualizzato l'ambito di riferimento in cui questo lavoro ed i suoi obiettivi si pongono. Verrà descritto l'impiego delle architetture eterogenee e dei Domain-Specific Language, introducendo l'approccio Model-Driven Engineering come una possibile soluzione per semplificare l'implementazione di applicazioni domain-specific efficienti aventi come target piattaforme eterogenee.

2.1 Architetture Eterogenee

Per processamento eterogeneo, si intende l'impiego di sistemi complessi, composti da diverse tipologie di unità di elaborazione, che utilizzano paradigmi di elaborazione differenti e sono progettate per task di natura diversa, ma che collaborano per offrire le migliori prestazioni per esigenze computazionali diversificate. Dunque, in un'architettura hardware che supporta il processamento eterogeneo, detta architettura eterogenea, le unità di calcolo si basano su instruction set differenti: ciò richiede, al fine di sviluppare un'applicazione che possa essere eseguita su tali architetture, nella sua interezza o in parte, la realizzazione di diverse copie dello stesso programma, per ogni distinto instruction set.

Tipicamente, le architetture eterogenee integrano alla CPU altre unità di elaborazione come le General Purpose GPU (GPGPU) e dispositivi hardware riprogrammabili, come i Field Programmable Gate Array (FPGA). Ciò presenta la possibilità fondamentale di diversificare l'esecuzione in base alla natura di questa: operazioni complesse di gestione del flusso di controllo possono esse-

re riservate alla CPU, mentre per elaborazioni massivamente parallele di dati omogenei, è possibile effettuare l'offloading dell'esecuzione sulla GPU. La flessibilità e la riconfigurabilità dei dispositivi hardware riprogrammabili, unite alla possibilità di personalizzazione che questi offrono, consentono di modulare l'hardware su misura, in base alle necessità delle applicazioni: ciò presenta l'ulteriore vantaggio di ridurre l'impatto energetico dei sistemi informatici.

Inoltre, le architetture eterogenee hanno consentito di superare le inevitabili limitazioni fisiche nello sviluppo di processori sempre più performanti. La legge di Moore afferma come la densità dei chip, cioè il numero di transistor presenti all'interno di un circuito integrato, sia destinato a raddoppiare circa ogni due anni [51]. Alcuni studi [42, 58] hanno evidenziato vincoli fisici per cui la legge di Moore è destinata a raggiungere un limite intrinseco. Come vedremo meglio in seguito, l'adozione di architetture eterogenee rappresenta un'opzione efficace per affrontare le crescenti esigenze computazionali, evitando i limiti esistenti nell'estensione dei singoli componenti hardware.

2.1.1 Architetture eterogenee nell'exascale computing

Nel contesto dell'High Performance Computing (HPC), uno degli obiettivi primari degli ultimi anni è stata la transizione dal calcolo petascale, caratterizzato da una capacità computazionale dell'ordine delle 10^{15} operazioni in virgola mobile al secondo (FLOPS), al calcolo exascale, che mira a raggiungere prestazioni dell'ordine dei 10^{18} FLOPS. Questo passaggio è cruciale per affrontare sfide computazionali sempre più complesse in campi come la simulazione scientifica, l'intelligenza artificiale e l'analisi di grandi quantità di dati.

Nel raggiungimento di tale obiettivo, avvenuto nel maggio 2022, con il supercomputer Frontier, l'impiego di architetture eterogenee è stato cruciale: già nel 2011, infatti, l'International Exascale Software Program (IESP) ha individuato le architetture eterogenee come necessarie per la transizione dal petascale all'exascale computing [19].

Lo studio in [35] analizza i trend architetturali nella progettazione dei supercalcolatori fin dall'istituzione del progetto Top500, che redige la classifica

dei più potenti supercomputer del pianeta. Uno degli aspetti affrontati da questo studio riguarda l'andamento nel tempo del parametro R_{max}, che valorizza, in FLOPs, le prestazioni di ogni supercomputer sul benchmark LINPACK, che prevede la risoluzione di un sistema denso di equazioni lineari. Graficando l'andamento di R_{max} di ogni supercomputer che ha raggiunto la prima posizione, e confrontandolo con il valore R_{max} ideale basato sulla legge di Moore, cioè ottenibili da una calcolatore avente le caratteristiche determinate dalla legge di Moore, si osserva il seguente risultato: ogni supercomputer ha sempre prestazioni superiori, rispetto alla previsione data dalla legge di Moore, e ciò costituisce una prova sperimentale dell'efficacia delle architetture eterogenee nell'affrontare i limiti presenti e futuri della legge di Moore.

Il grafico in Figura 2.1 mostra la variazione nel tempo del numero totale di supercomputer basati su architettura eterogenea nella classifica Top500, mentre il grafico in Figura 2.2 mostra l'andamento della percentuale di macchine eterogenee tra i primi 10 supercalcolatori al mondo. In entrambi i casi, il trend crescente dimostra come le architetture eterogenee siano diventate lo standard de facto nel mondo dell'HPC, e come siano state cruciali nella sfida dell'exascale computing: gli unici due supercalcolatori ad aver raggiunto questo traguardo, Frontier e Aurora, sono entrambi basati su una combinazione di CPU e GPU core.

Pertanto, è evidente che l'adozione di architetture eterogenee costituisca lo stato dell'arte nell'ambito dell'High Performance Computing (HPC).

2.1.2 Architetture eterogenee nell'utilizzo general-purpose

Nonostante i vantaggi che le architetture eterogenee introducono e il successo che hanno avuto nel mondo dell'HPC, il loro utilizzo in contesti aziendali non è affatto priva di ostacoli.

Lo studio in [7], mediante interviste a professionisti di varie aziende, esplora le principali sfide a cui queste sono sottoposte, in vari stadi nell'adozione delle architetture eterogenee, dalla migrazione iniziale verso hardware diversificato fino all'ottimizzazione delle risorse. La difficoltà principale riguarda

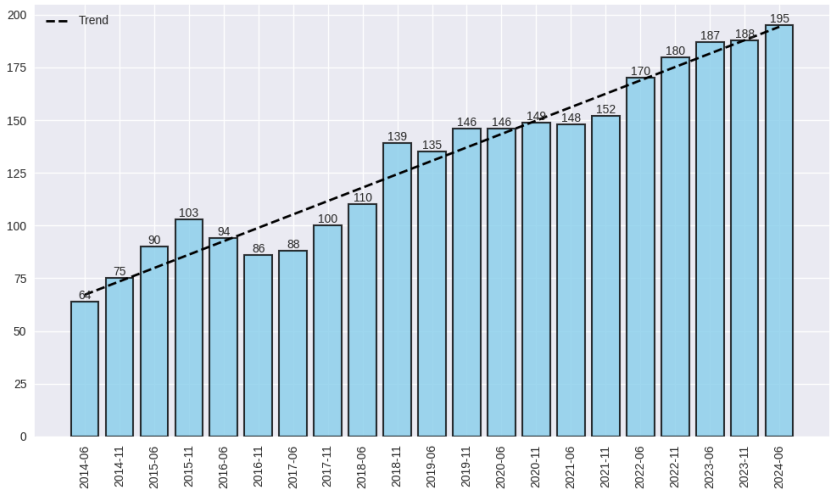


Figura 2.1: Numero totale di supercomputer basati su architetture eterogenee nei Top500

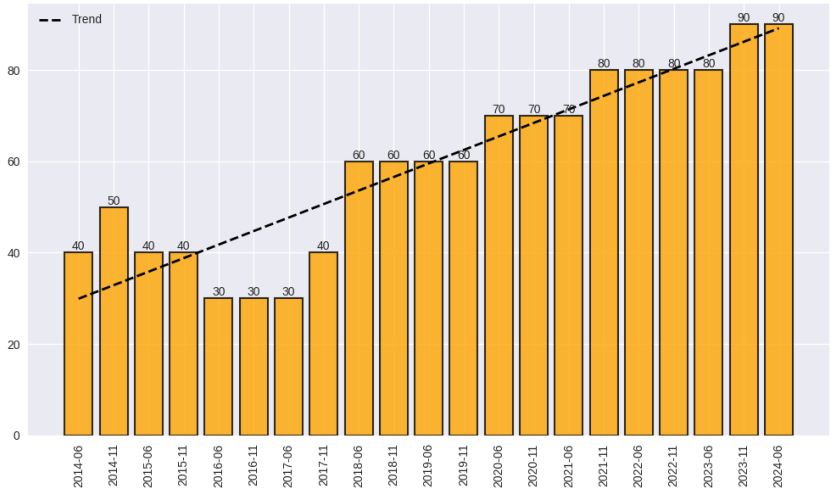


Figura 2.2: Percentuale dei top 10 supercomputer basati su architettura eterogenea

la mancanza di esperienza degli sviluppatori nelle tecnologie specifiche per ogni componente hardware che si vuole introdurre nell'architettura: la curva di apprendimento rischia di essere un problema non banale specialmente per quei componenti hardware per cui, ad oggi, non esistono supporti software consolidati, come ad esempio le FPGA.

In generale, le toolchain esistenti offrono un livello di astrazione ridotto, rispetto al caso delle architetture CPU-based, e ciò comporta un effort significativo non solo nella migrazione verso approcci ibridi, ma anche in fasi più avanzate di ottimizzazione dell'utilizzo delle risorse.

Nel tempo, sono state sviluppate varie soluzioni per la programmazione eterogenea: il primo framework è stato OpenCL [52], seguito da altri layer di astrazione come SyCL [30] e Kokkos [20]. Questi offrono la possibiltà di astrarre l'architettura eterogenea dal punto di vista del programmatore, non risolvendo tuttavia alcune problematiche centrali: tra queste, la facilità di programmazione, specialmente per esperti di dominio, e la presenza di una modalità di esecuzione che preveda uno scheduling autonomo ed ottimale dell'esecuzione sui vari componenti hardware.

2.2 Domain Specific Languages

Un Domain Specific Language (DSL) è un linguaggio di programmazione progettato per risolvere un problema specifico in un dominio ben definito, in contrapposizione ai linguaggi general-purpose (GPL), che sono costruiti per risolvere problemi software di qualsiasi natura.

Un DSL può essere interno o esterno: nel primo caso, il linguaggio è implementato all'interno di un GPL, cioè usando la sintassi e i costrutti offerti dal GPL; nel secondo caso, il linguaggio prevede una sintassi propria e personalizzata.

Rispetto ai GPL, i DSL offrono i seguenti vantaggi [49]:

- Notazioni domain-specific, che aumentano l'espressività rispetto ai costrutti offerti dai GPL, portando ad un aumento della produttività: mag-

giore è la chiarezza offerta dal linguaggio, più diventa immediato trovare eventuali errori nel codice e modificare il sistema;

- Mapping diretto tra costrutti del DSL e codice ottimizzato di basso livello, che accresce l'efficienza del software finale;
- I DSL, a differenza dei GPL, non hanno la necessità di essere direttamente eseguibili; di conseguenza, offrono allo sviluppatore un livello di astrazione più alto e, inoltre, possono essere indipendenti dalla piattaforma hardware sottostante.
- Fornendo un linguaggio chiaro e preciso, possono contribuire a migliorare la comunicazione con gli esperti di dominio, creando un'interfaccia verso il codice che possano comprendere più facilmente.

Dunque, uno dei principali punti a favore dei DSL consiste nel nascondere al programmatore gli aspetti architetturali dell'ambiente di runtime. Ciò consente di sfruttare questi linguaggi come livello di astrazione nello sviluppo di applicazioni per architetture eterogenee: in questo caso, il front-end per lo sviluppatore è costituito soltanto dal DSL, che viene automaticamente compilato in codice di basso livello, ottimizzato per i particolari dispositivi hardware. In questo modo, l'accessibilità ad architetture hardware complesse, come quelle eterogenee, viene estesa anche ad esperti di dominio che, altrimenti, non avrebbero le competenze necessarie per valorizzarle.

2.2.1 Difficoltà nello sviluppo di DSL

Lo sviluppo dei DSL, sia interni che esterni, prevede una serie di difficoltà, dovute principalmente al fatto che lo sviluppatore deve avere profonde competenze non solo nel dominio di riferimento, ma anche nell'ambito del language development. Le tecniche e i pattern di sviluppo dei DSL sono diversi rispetto a quelli relativi alla programmazione con i GPL, e talvolta richiedono una maggiore e più profonda conoscenza di aspetti legati all'architettura hardware impiegata.

La decisione di sviluppare un DSL, inizialmente, può non essere evidente e, in generale, segue un investimento temporale importante impiegato nella programmazione con GPL. Perciò, molte aziende ed esperti di dominio ritardano, o addirittura evitano, la scelta di impiegare i DSL a supporto delle loro attività [49].

Una delle obiezioni principali mosse verso l'impiego dei DSL riguarda la preoccupazione che l'aggiunta di un nuovo linguaggio nella toolchain di sviluppo del software possa comportare un effort non minimale nell'assimilarlo. Inoltre, introdurre un nuovo livello di indirezione può allontanare lo sviluppatore da ciò che accade a livelli di astrazione inferiori [23]. Tuttavia, l'effort nell'apprendimento di un DSL è senza dubbio minimale rispetto al caso dei GPL, dal momento che sono più semplici e orientati verso uno specifico problema; inoltre, l'adozione dei DSL non deve occultare completamente una parte del progetto, ma deve soltanto renderla più facilmente accessibile e manipolabile.

Il rischio maggiore nell'implementazione e nell'utilizzo dei DSL riguarda il problema della *Blinkered Abstraction* [23]: l'astrazione offerta dal DSL, sebbene utile, può restringere il campo di percezione o comprensione. Con un'astrazione limitante, si spende più energia nel cercare di adattare il mondo alla propria astrazione piuttosto che il contrario: quando si incontra un aspetto che non si integra con l'astrazione, il primo tentativo è quello di farlo adattare in base a ciò che l'astrazione offre, piuttosto che estendere questa per assorbire facilmente nuovi elementi significativi. Questo problema porta ad una riluttanza nello sviluppo iterativo di un DSL, che può essere esacerbata nell'interazione con esperti di dominio, i quali possono rivelarsi ancora più reticenti davanti alla modifica di un linguaggio con cui hanno acquisito familiarità. Di conseguenza, come nel caso di ogni altra astrazione, si deve pensare un DSL come un prodotto in continua evoluzione.

2.3 Approccio Model-Driven Engineering

Nella sezione precedente sono state evidenziate le sfide associate allo sviluppo dei DSL. Tra queste, una delle più rilevanti riguarda la necessità per lo sviluppatore di possedere una duplice competenza: da un lato, una profonda conoscenza del dominio applicativo e, dall'altro lato, una padronanza dei dettagli implementativi, anche di basso livello, richiesti per l'implementazione del linguaggio. Nel caso di architetture eterogenee, le competenze degli sviluppatori dei DSL devono anche comprendere la conoscenza dei dettagli architetturali e degli schemi di sincronizzazione necessari al supporto dell'esecuzione delle applicazioni su tali sistemi.

Uno degli scopi di questa tesi è illustrare come l'adozione di un diverso approccio possa rendere lo sviluppo dei DSL nettamente più agevole. Questo approccio alternativo è dato dal Model-Driven Engineering.

In passato, i modelli erano visti soltanto come artefatti documentativi, a supporto della progettazione e dello sviluppo di sistemi software complessi con lo scopo, ad esempio, di facilitare la comunicazione con gli stakeholder. L'approccio Model-Driven Engineering, d'altra parte, considera i modelli come artefatti centrali nel processo di ingegnerizzazione del software: tramite alcune tecniche, come il meta-modeling, la trasformazione di modelli e la generazione del codice, è possibile arrivare alla creazione automatica di sistemi software partendo proprio dai modelli.

2.3.1 Modello e meta-modello

Con il termine modello, si intende un sistema che aiuta a definire un sistema reale in esame, detto *System Under Study* (SUS), e che può fornire predizioni o inferenze sul SUS senza la necessità di considerare quest'ultimo direttamente [16]. In particolare, un modello si distingue da altri artefatti perché rispetta i seguenti criteri [64]:

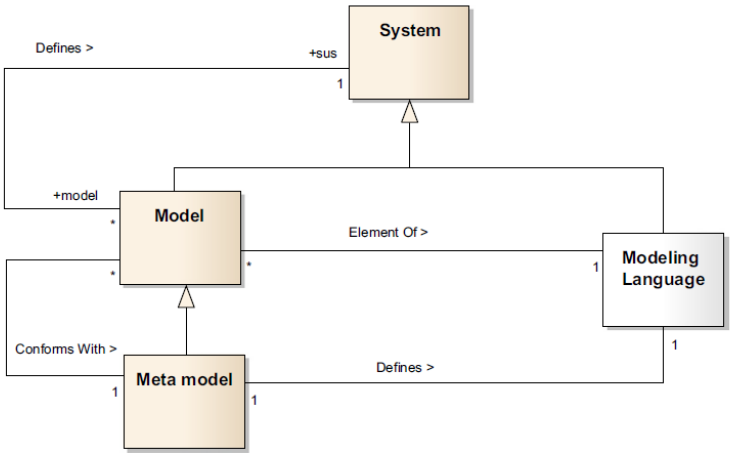


Figura 2.3: Definizione di metamodello [16]

- Criterio di *mapping*: esiste un SUS reale che è rappresentato, o mappato, dal modello;
- Criterio di *riduzione*: il modello rappresenta una versione ridotta del SUS, quindi non tutte le proprietà di quest'ultimo devono essere presenti nel modello;
- Criterio di *praticità*: il modello deve essere utile, cioè deve sostituire il SUS per determinati scopi.

Un altro concetto centrale è quello di meta-modello, definito come un ulteriore modello che definisce un linguaggio per esprimere un modello. In altre parole, un meta-modello definisce un linguaggio di modellazione: quest'ultimo è dato dall'insieme di tutti i possibili modelli conformi al meta-modello. Per definire la nozione di conformità al meta-modello, bisogna descrivere alcuni aspetti che caratterizzano quest'ultimo, tra cui la sintassi astratta del meta-modello, che è rappresentata da una o più sintassi concrete, la semantica e la pragmatica, cioè le linee guida che definiscono l'uso pratico del linguaggio.

La definizione di un linguaggio di modellazione parte da un'analisi approfondita dei concetti, astrazioni e relazioni all'interno del dominio di riferimento. Il risultato di questa fase, detta analisi del dominio, produce la sintassi astratta del linguaggio di modellazione. Inoltre, in questa fase sono specificati anche eventuali vincoli di well-formedness o che sussistono nelle relazioni tra i con-

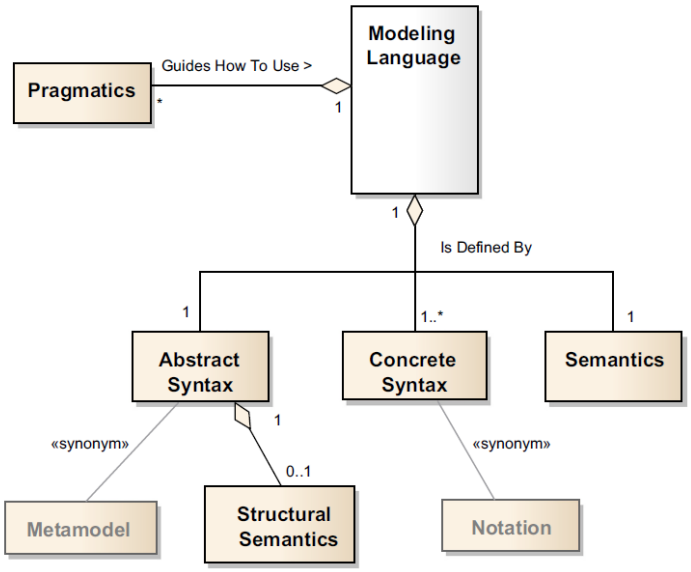


Figura 2.4: Definizione di linguaggio di modellazione [16]

cetti: questi possono essere rappresentati sia attraverso il linguaggio naturale che in maniera più strutturata, attraverso linguaggi dichiarativi come Object Constraint Language (OCL).

La sintassi concreta rappresenta la notazione del linguaggio, cioè il modo in cui gli utenti utilizzano il linguaggio. Questa può essere di varie tipologie, ad esempio grafica, testuale o tabulare, e rappresenta un aspetto fondamentale del linguaggio in quanto corrisponde all'interfaccia verso l'utente; di conseguenza, si deve trovare un tradeoff equilibrato tra semplicità ed espressività.

In generale, la semantica determina il significato di un'espressione sintatticamente valida di un dato linguaggio. Per i linguaggi di modellazione, esistono due tipologie di semantiche: eseguibili e non eseguibili. Le semantiche eseguibili si riferiscono a concetti collegati ai linguaggi di programmazione, ad esempio all'ordine di esecuzione dei programmi. Le semantiche non eseguibili, invece, non sono collegate al concetto di esecuzione, e possono riferirsi ad aspetti quali, ad esempio, la specifica dei requisiti, come nel caso degli use case diagrams. Inoltre, le semantiche si dividono in formali ed informali: nel primo caso, esse sono definite formalmente attraverso appositi framework, come Structural Operational Semantics (SOS) [3], mentre nel secondo caso sono descritte in maniera informale, ad esempio attraverso specifiche in linguaggio naturale.

Dunque, l'approccio MDE prevede l'utilizzo dei linguaggi di modellazione per costruire modelli, ad un certo livello di astrazione, che supportino lo sviluppo di applicazioni software [72]. In [16], viene definita un'applicazione software come un "*sistema ottenuto dall'integrazione di piattaforme software, artefatti generati tramite trasformazioni model-to-text, artefatti prodotti direttamente dagli sviluppatori, e modelli direttamente eseguibili nel contesto di una particolare piattaforma software*".

In questa definizione, è cruciale il concetto di generazione: in un sistema software possono coesistere artefatti generati automaticamente, tramite trasformazioni model-to-text, e artefatti prodotti direttamente dagli sviluppatori. Inoltre, è possibile avere artefatti intermedi che non sono di interesse per l'esecuzione dell'applicazione, ma che sono prodotti, o generati, allo scopo di agevolare il processo di produzione, o generazione, degli artefatti finali. Riguardo la generazione, l'approccio MDE prevede due tipologie di trasformazione:

- Trasformazione model-to-text (M2T): prevede la generazione automatica di artefatti software, tipicamente codice sorgente, a partire da un modello.
- Trasformazione model-to-model (M2M): consiste nella conversione di un modello in un secondo modello, tipicamente più vicino al dominio della soluzione.

In generale, gli artefatti intermedi che supportano lo sviluppo sono modelli, prodotti a partire da una trasformazione M2M.

2.3.2 MDE per lo sviluppo di DSL

Lo scopo principale di questa tesi, come già accennato, riguarda l'esplorazione dell'utilizzo di approcci MDE per supportare uno sviluppo di DSL più snello e semplice. L'utilizzo di un approccio MDE, in questo contesto, permette anche di massimizzare il riuso del software: differenti DSL potranno sfruttare le stesse funzionalità offerte da librerie di esecuzione di più basso livello, consentendo un'esecuzione efficiente di applicazioni provenienti da domini disparati.

Nell'ottica di un dispiegamento delle applicazioni su architetture eterogenee, il vantaggio nel riuso di componenti software di basso livello è evidente—per quanto, in questa tesi, questo specifico aspetto non verrà esplorato.

Per valutare la bontà dell'approccio, si considererà come use case lo sviluppo di un DSL per supportare l'esecuzione di pipeline di stream processing su ambienti di esecuzione speculativi, secondo i principi appena esposti. In particolare, vengono individuati due meta-modelli: il DSL, che specifica notazione e sintassi per la definizione di query sullo stream, ed un ulteriore linguaggio intermedio, basato sul modello degli attori; il DSL viene convertito, mediante una trasformazione M2M, in un modello intermedio; infine, quest'ultimo viene compilato, mediante una trasformazione M2T, in codice eseguibile.

2.3.3 Necessità di una rappresentazione intermedia

L'adozione di un secondo meta-modello come rappresentazione intermedia è una scelta strategica per ridurre il costo e la complessità della creazione di nuovi linguaggi. Separando le trasformazioni tra il DSL e il codice di basso livello, si può concentrare lo sviluppo sulle specificità del linguaggio, senza preoccuparsi della parte finale della compilazione, che rimane invariata e riutilizzabile.

In assenza della rappresentazione intermedia, la fase di sviluppo di un nuovo DSL richiederebbe, oltre alla realizzazione del linguaggio stesso, anche l'implementazione della trasformazione M2T verso il codice di basso livello. Di conseguenza, lo sviluppatore dovrebbe conoscere appieno le caratteristiche dell'architettura hardware, per poter progettare e realizzare una trasformazione verso codice efficiente. Nel caso di architetture eterogenee, le M2T da realizzare sarebbero molteplici. Inoltre, sarebbe necessario anche prevedere l'esistenza di un software di orchestrazione a runtime del workload. Utilizzando una rappresentazione intermedia, tutti questi aspetti possono essere tralasciati dallo sviluppatore del DSL che potrà concentrarsi unicamente sulla definizione del linguaggio di interesse.

Introducendo la rappresentazione intermedia, la fase di sviluppo non richiede la realizzazione della trasformazione M2T, ma soltanto di una trasfor-

mazione M2M verso il modello intermedio. La trasformazione M2T può essere trattata una volta sola per tutti i linguaggi che si appoggiano a questo approccio, riducendo significativamente il carico di lavoro e disaccoppiando lo sviluppatore dagli aspetti più complessi e tecnici che caratterizzano l'architettura hardware sottostante.

Il meta-modello scelto, nell'ambito di questa tesi, è ispirato al modello degli attori, un paradigma computazionale consolidato [29, 27, 5] per la programmazione concorrente, in cui gli elementi costitutivi di base sono gli *attori*, cioè entità autonome che comunicano esclusivamente attraverso lo scambio di messaggi. In risposta ad un messaggio, ogni attore può eseguire dei calcoli, creare nuovi attori e inviare messaggi ad altri attori, senza la necessità di condividere memoria con essi. Perciò, questo modello semplifica la gestione della concorrenza, eliminando la necessità di sincronizzazione esplicita e riducendo i problemi legati a race condition e deadlock.

Inoltre, la scelta di un paradigma ben consolidato consente di semplificare la costruzione della trasformazione M2T verso il codice di basso livello, in quanto la letteratura ha prodotto negli anni diverse librerie e toolkit aventi come base proprio il modello degli attori, come Akka[12] e Microsoft Orleans[53]. Nel caso specifico di questa tesi, non vengono utilizzati framework o librerie basati sul modello degli attori, ma l'esecuzione ha luogo sull'ambiente di runtime ROOT-Sim [57], che offre supporto per l'esecuzione (speculativa) parallela e distribuita di applicazioni concorrenti.

3. Stato dell'arte

In questo capitolo, verrà presentata un'analisi dettagliata dello stato dell'arte, considerando diversi aspetti legati allo sviluppo del lavoro di tesi.

In particolare, verrà descritto il panorama di ricerca negli ambiti introdotti nel precedente capitolo. Oltre all'impiego dei DSL, citando alcuni esempi rilevanti che hanno come scopo la semplificazione dello sviluppo del software, verranno analizzati i framework esistenti che facilitano la loro progettazione e realizzazione, fino ad approfondire l'adozione del paradigma MDE nell'ingegneria del software.

3.1 Diffusione dei DSL

Negli ultimi anni, uno dei trend di ricerca più diffusi nell'ambito dell'ingegneria del software ha riguardato lo sviluppo di DSL. Questo approccio ha suscitato un crescente interesse grazie alla sua capacità di offrire soluzioni su misura per contesti particolari, migliorando l'efficienza dello sviluppo e riducendo la complessità. L'adozione dei DSL ha portato a numerosi studi e applicazioni pratiche, dimostrando come questi linguaggi siano diventati strumenti fondamentali per la progettazione di software altamente specializzato.

In particolare, la letteratura evidenzia una vasta gamma di domini in cui i DSL hanno dimostrato la loro utilità, non limitandosi solo alla semplificazione dello sviluppo software, ma anche per altri obiettivi specifici e diversificati. Ad esempio, in ambito matematico, sono stati sviluppati diversi DSL per la rappresentazione e risoluzione di equazioni alle derivate parziali (PDE) [6, 18, 46]. Un altro dominio rilevante è quello dell'IoT: il lavoro in [24] descrive

lo sviluppo di un DSL incentrato sulla costruzione di un'interfaccia comune per dispositivi IoT eterogenei, mentre in [9] viene presentato un DSL per la simulazione di sistemi IoT.

Ulteriori esempi di domini applicativi includono la biologia [11, 47], il Machine Learning [26, 67], la programmazione mobile [40], la modellazione e analisi delle prestazioni [10, 61], la simulazione [15, 22, 74], la modellazione atmosferica in ambito meteorologico [1], fino all'image [48] e volume [14] processing.

Dunque, la considerevole quantità di ricerca nello sviluppo di DSL dimostra come questi strumenti siano cruciali per ridurre e gestire la complessità delle infrastrutture tecnologiche, offrendo al contempo agli esperti di dominio interfacce più intuitive e accessibili rispetto a quelle fornite dai linguaggi di programmazione generali [8, 37, 38, 39].

Inoltre, diversi studi citati in precedenza [14, 74] adottano architetture eterogenee come riferimento hardware, evidenziando come l'uso dei DSL possa facilitare l'accesso a questo tipo di piattaforme.

3.2 Strumenti a supporto dello sviluppo di DSL

Lo sviluppo di un nuovo DSL senza l'ausilio di strumenti specializzati è un compito tutt'altro che semplice: per costruire un compilatore o un interprete per un nuovo linguaggio, sono necessari componenti fondamentali come un parser, strutture dati per la gestione degli alberi sintattici astratti (AST), trasformazioni, controlli di tipo e molti altri elementi [45].

Per questo motivo, nel corso del tempo sono stati introdotti diversi pattern, metodologie di sviluppo, toolkit, ambienti di sviluppo integrati (IDE) e framework che supportano la creazione di DSL o offrono strumenti pratici per semplificarne la realizzazione.

Lo studio in [31] presenta un Systematic Mapping Study (SMS) [36] che presenta una panoramica dei tool, Language Workbench (LW) o framework per lo sviluppo di DSL proposti in letteratura tra il 2012 e il 2019.

I tool individuati nello studio comprendono strumenti per la costruzione di DSL, ossia tool individuali utilizzati direttamente nella creazione di un DSL, Language Workbench (LW), ovvero ambienti integrati, anche detti toolbox, dedicati allo sviluppo di DSL e, infine, strumenti di utilità, impiegati per fornire funzionalità di supporto allo sviluppo dei linguaggi.

Dall'analisi di 230 paper selezionati, tra le tecnologie più citate ci sono framework e tool appartenenti all'ecosistema Eclipse, in particolare facenti parte di Eclipse Modeling Framework (EMF) [65] e di Graphical Modeling Framework. Tra questi, il framework Xtext [21], il tool Papyrus [25], cioè uno strumento di editing grafico per UML2, che offre supporto i profili UML [2], e il framework Sirius [70], che consente di creare workbench di modellazione personalizzate, composte da diagrammi, tabelle e alberi. Al di fuori dell'ecosistema Eclipse, altre LW che hanno riscontrato un discreto utilizzo sono MetaEdit+ [68], JetBrains Meta Programming System (MPS) [55] e Spoofax [34].

Un ulteriore esempio è dato da Delite [67], un framework per la compilazione di DSL, che introduce una serie di componenti riutilizzabili per la costruzione di DSL in domini differenti, che possono essere compilati in codice eseguibile su piattaforme eterogenee.

In generale, il debugging rappresenta un aspetto cruciale nello sviluppo software. A differenza della programmazione con linguaggi GPL, nella creazione di DSL lo sviluppatore non ha a disposizione debugger o strumenti equivalenti, e la realizzazione di questi è paragonabile, in termini di effort, allo sviluppo di debugger per GPL. Per ovviare a questa mancanza, Lindeman et al. [45] hanno proposto un framework per l'implementazione di debugger indipendenti dal linguaggio target, composto da un DSL di alto livello per la specifica di operazioni di debug, un tool intermedio che aggiunge queste al modello del DSL target e, infine, un'infrastruttura di runtime per l'esecuzione delle attività di debugging.

Accanto alle tecnologie, nel corso degli anni sono stati introdotte metodologie strutturate per standardizzare la progettazione e realizzazione dei linguaggi specifici di dominio [49, 66]. Ad esempio, il lavoro in [66] definisce un approccio

per lo sviluppo sistematico dei DSL. In sintesi, vengono individuate quattro attività principali:

(1) Definizione del modello linguistico centrale del DSL, che produce la sintassi astratta del linguaggio;

(2) Definizione del comportamento degli elementi del linguaggio, cioè come questi interagiscono tra loro;

(3) Definizione di una o più sintassi concrete del DSL;

(4) Integrazione degli artefatti prodotti dal DSL con la piattaforma o infrastruttura di esecuzione.

Oltre alle metodologie, sono stati introdotti anche design pattern [62] e linee guida [33] per facilitare le decisioni ricorrenti nella progettazione e nello sviluppo dei DSL.

3.3 Model-Driven Engineering

I concetti fondamentali del paradigma MDE e i benefici che esso apporta all'ingegneria del software sono stati esaminati nel capitolo precedente. In questa sezione, si analizzerà l'adozione dell'approccio MDE nel settore industriale e le sfide attualmente irrisolte.

Whittle et al. [73] hanno esaminato l'impatto del MDE in contesti aziendali, mediante interviste a professionisti in vari settori. Dallo studio risulta una vasta diffusione del MDE nel settore industriale e, inoltre, emergono alcuni benefici collaterali associati a questo approccio. Ad esempio, in vari casi, l'MDE ha generato implicitamente una documentazione utile per l'architettura del software, contribuendo a migliorare la consapevolezza degli sviluppatori riguardo al sistema di destinazione.

L'intervista condotta in [50] è rivolta a quattro grandi aziende, dove alcune delle quali presentano sviluppatori esperti nell'approccio MDE, mentre

altre ne stanno sperimentando l'introduzione. Oltre a confermare alcuni aspetti evidenziati dal precedente studio, i risultati dimostrano come alcune delle criticità riscontrate dagli sviluppatori riguardino la scarsa maturità dei tool e delle metodologie a supporto del paradigma MDE.

Purtroppo, la letteratura recente non offre studi sull'utilizzo a livello industriale dell'approccio MDE, se non in alcuni domini specifici, per cui non è immediato determinare se gli strumenti e le metodologie abbiano raggiunto un grado di maturità adeguato.

Nel 2017 e nel 2018 si sono svolti due eventi per esaminare lo stato dell'arte del MDE, sia dal punto di vista della ricerca che della pratica. Questi incontri hanno unito esperti del settore industriale, accademico e della comunità open-source per analizzare i progressi realizzati nel decennio precedente e le sfide ancora irrisolte nella ricerca e nell'applicazione dell'approccio MDE.

Il lavoro in [13] riassume le principali difficoltà che continuano a caratterizzare il panorama del MDE. In particolare, sono evidenziate sia sfide di natura tecnica, che aspetti sociali e relativi alla comunità.

Tra le sfide tecniche, particolare attenzione è riservata all'usabilità dei tool disponibili, per cui uno degli obiettivi fissati è quello di creare strumenti di modellazione più intuitivi ed accessibili. Un altro aspetto fondamentale è la scalabilità: una sfida fondamentale consiste nell'affrontare la crescente domanda di supporto per modelli, metamodelli e trasformazioni molto grandi, e ciò richiede miglioramenti nelle prestazioni delle infrastrutture di modellazione.

Tra le sfide sociali, si annovera la necessità di facilitare la modellazione collaborativa tra diversi stakeholder, inclusi utenti non tecnici, per garantire che le loro esigenze siano adeguatamente rappresentate. Una delle sfide comunitarie principali, infine, riguarda la creazione e manutenzione di repository di modelli, dotati di misure di qualità e curati dalla comunità, per facilitare l'analisi automatizzata e l'evoluzione dei modelli.

3.4 Il Modello ad Attori

Il modello ad attori, inizialmente presentato da Hewitt negli anni '70 [29], rappresenta un paradigma computazionale consolidato per la programmazione concorrente [4, 5, 17, 27].

Negli ultimi anni, diversi framework hanno dimostrato l'efficacia dell'Actor Model in diversi contesti. Un esempio è CANTO [63], un framework per il Machine Learning distribuito direttamente sui dispositivi della rete fog in infrastrutture IoT. Un ulteriore esempio molto recente, in un dominio applicativo diverso, cioè quello della neuroingegneria, consiste in un framework per il *sensory encoding* visuale, basato anch'esso sul modello degli attori [44].

Il lavoro in [43] evidenzia i vantaggi di una modellazione orientata agli attori di sistemi complessi, a supporto del progetto Ptolemy II [60].

In ambito MDE, il sistema descritto in [41] prevede un ambiente di modellazione denominato ACTRESS che offre la possibilità, tramite un DSL basato sul modello degli attori, di introdurre meccanismi di *self-adaptation* a sistemi software, astraendo il concetto di Feedback Control Loop (FCL).

Tutto ciò dimostra come il modello degli attori sia, ancora oggi, assolutamente attuale ed efficace per modellare esecuzioni concorrenti complesse, offrendo inoltre una versatilità che consente la sua applicazione in contesti eterogenei.

4. Il modello ad attori

Il modello ad attori è un paradigma matematico per la computazione concorrente, in cui l'unità fondamentale è l'attore. Gli attori sono entità autonome ed auto-contenute che eseguono operazioni in parallelo e comunicano tra loro esclusivamente tramite lo scambio di messaggi asincroni.

Secondo la definizione fornita da Agha [4], un attore è un agente computazionale che mappa ogni comunicazione ricevuta in una terna composta da:

- Un insieme finito di comunicazioni inviate ad altre attori;
- Un nuovo comportamento, che determina le azioni eseguite in risposta alla comunicazione successiva;
- Un insieme finito di nuovi attori creati.

Secondo l'astrazione del *mail system* [4], ad ogni attore è associato un *mail address*, cioè un indirizzo che viene utilizzato dagli altri attori per inviare comunicazioni verso l'attore.

4.1 Asincronia nel modello ad attori

L'interazione tra attori consente la sincronizzazione e la coordinazione dei processi senza la necessità di uno stato condiviso.

Il concetto di clock globale unico non è significativo nel contesto del modello degli attori: l'informazione in ogni agente computazionale, i.e. attore, è localizzata all'interno dell'agente stesso, e viene resa nota ad altri attori solo attraverso una comunicazione; di conseguenza, ogni agente computazionale

possiede un clock locale, in base al quale gli stati locali dell'agente, cioè gli eventi che si verificano all'interno dell'agente, sono ordinati. Gli ordinamenti locali, relativi ad attori differenti, sono collegati l'un l'altro attraverso l'ordine di attivazione, che rappresenta una relazione causale tra eventi che occorrono in agenti diversi. Quindi, l'ordinamento globale degli eventi è un ordine parziale dove non è definito un ordinamento tra eventi relativi ad agenti diversi a meno che non esistano relazioni causali che li legano, direttamente o indirettamente. Di conseguenza, un sistema distribuito realizzato secondo questo paradigma è intrinsecamente asincrono.

Chiaramente, resta possibile costruire sistemi sincroni, introducendo agenti che si occupano della sincronizzazione, cioè che orchestrano le esecuzioni dei singoli attori.

4.2 Modalità di comunicazione tra attori

In generale, la comunicazione tra elementi computazionali diversi può essere di due tipologie: sincrona, quando il mittente e il destinatario devono essere attivi e pronti a comunicare nello stesso momento, e asincrona, se il destinatario non deve essere pronto ad accettare il messaggio al momento dell'invio.

La comunicazione sincrona, nell'ambito di un sistema concettualmente distribuito, introduce una serie di problematiche. Anzitutto, se un agente A_1 sta inviando informazioni verso l'agente A_2, c'è necessità di un meccanismo che impedisca ad un ulteriore agente A_3 di stabilire una comunicazione verso A_1. Un altro problema si presenta nel caso in cui la velocità di trasmissione delle informazioni, lato mittente, sia maggiore rispetto alla velocità di ricezione, lato destinatario. Una soluzione a questi due problemi consiste nell'introduzione di un meccanismo di bufferizzazione delle comunicazioni, secondo cui esiste uno spazio, per ogni agente, in cui possono essere memorizzate, temporaneamente, comunicazioni in entrata.

Il meccanismo di comunicazione, nel modello degli attori, è basato sulla *comunicazione asincrona bufferizzata*: poiché sono disponibili buffer che con-

servano i messaggi ricevuti, il mittente non deve attendere che il destinatario riceva la comunicazione per continuare con le sue attività.

Un vantaggio fondamentale che questa tipologia di comunicazione offre riguarda la possibilità, per un agente, di comunicare con sé stesso. La comunicazione ricorsiva, tale per cui mittente e destinatario coincidono, è impossibile se il destinatario deve essere libero nel momento in cui il mittente avvia la comunicazione: ciò conduce ad una situazione di deadlock, in quanto il mittente resta indefinitamente in *busy waiting* di sé stesso.

Ogni attore, in un determinato momento, ha la possibilità di comunicare con un certo gruppo di altri attori: l'insieme delle comunicazioni possibili definisce una topologia. Se i gruppi di attori con cui ciascun attore può interagire rimangono costanti nel tempo, la topologia è considerata statica; altrimenti, è definita dinamica.

La riconfigurabilità, nei sistemi ad attori, è ottenuta tramite lo scambio di indirizzi: nell'astrazione del *mail system*, secondo cui ogni attore possiede un indirizzo, gli attori possono comunicare solamente con altri agenti di cui conoscono l'indirizzo; di conseguenza, ogni attore può riconfigurare parte della topologia semplicemente comunicando il proprio indirizzo verso altri attori.

Il modello ad attori, infine, prevede che, per ogni attore, l'ordine di arrivo delle comunicazioni inviate verso quell'attore sia un ordine lineare.

4.3 Task

Nell'Actor Model, il concetto di task è espresso mediante una terna composta da:

- Un'*etichetta* (tag), che identifica il task univocamente nel sistema;
- Un *destinatario* (target), cioè l'indirizzo dell'attore a cui è rivolta la comunicazione;
- Una *comunicazione*, che contiene le informazioni dirette all'attore target, che processerà il task.

Una comunicazione può essere una tupla di valori: ad esempio, può includere una *envelope*, che contiene informazioni come il tag e l'indirizzo target, ed un payload, cioè i dati veri e propri associati alla comunicazione.

Il target deve essere un indirizzo valido. Di conseguenza un attore A_1, per inviare una comunicazione verso un attore target A_2, deve conoscerne l'indirizzo. In particolare, A_1 può conoscere l'indirizzo di A_2 se ha ricevuto questa informazione da una particolare comunicazione, oppure se A_2 è stato creato da A_1, come effetto del processamento di qualche task.

4.4 Comportamento degli attori

Un ulteriore concetto che caratterizza l'attore è il suo comportamento. Quando un attore riceve una comunicazione, esso la accetta se processa il task che contiene la comunicazione. Le azioni che l'attore compie, nel processamento del task, definiscono il suo comportamento.

In dettaglio, quando un attore accetta una comunicazione, può creare nuovi attori oppure nuovi task: in quest'ultimo caso, l'attore crea una comunicazione verso un secondo attore, che processerà il task secondo il suo specifico comportamento.

Inoltre, in corrispondenza del processamento di un task, ogni attore deve specificare un comportamento sostitutivo, che corrisponde alla sequenza di azioni che verranno compiute al processamento del task successivo, quindi alla ricezione di una nuova comunicazione.

4.5 Il modello ad attori come rappresentazione intermedia

Nell'ambito di questa tesi, viene definito un DSL basato sul modello degli attori, chiamato *ActorLanguage*: questo linguaggio viene utilizzato come rappresentazione intermedia all'interno dell'architettura software, come mostrato in Figura 4.1.

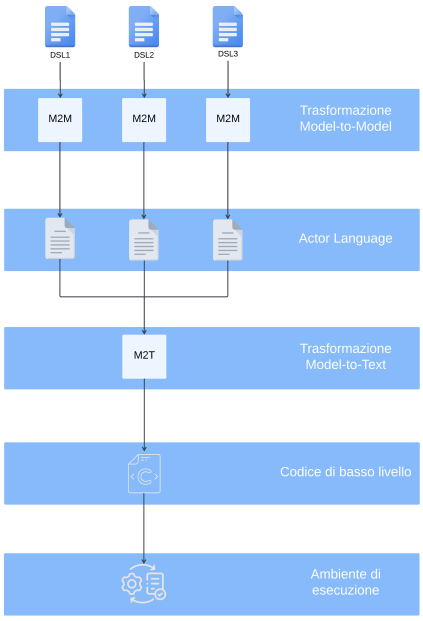


Figura 4.1: Architettura Software di riferimento

Come già detto nel capitolo introduttivo, l'obiettivo finale di questo lavoro è la realizzazione di un sistema per la creazione di DSL basato sui concetti del Model-Driven Engineering. L'architettura di riferimento di questo sistema è composta, in ordine decrescente di astrazione, da:

- Un numero arbitrario di Domain Specific Languages;
- Trasformazioni Model-to-Model (M2M), cioè le traduzioni di ogni DSL di alto livello nel linguaggio intermedio;
- DSL intermedio, i.e. *ActorLanguage*, basato sul modello degli attori;
- Trasformazione Model-to-Text (M2T), cioè la traduzione univoca del DSL *ActorLanguage* verso il codice di basso livello;
- Codice di basso livello, ottimizzato per una specifica piattaforma di esecuzione;
- Ambiente di esecuzione, su cui il codice di basso livello viene eseguito.

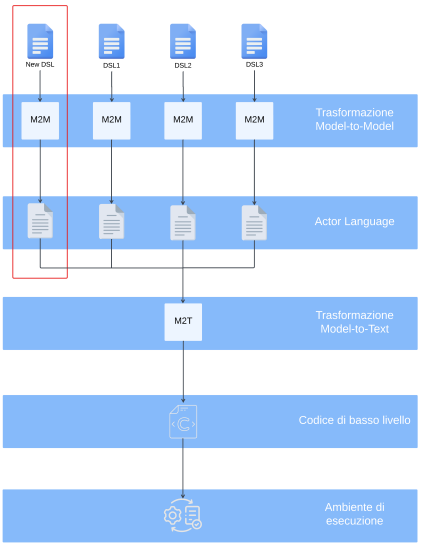


Figura 4.2: Creazione di un nuovo DSL

Dall'architettura software di riferimento, diventa ancora più evidente un aspetto che è già stato discusso precedentemente, cioè la possibilità, garantita dalla presenza del modello degli attori intermedio, di sviluppare un insieme arbitrario di DSL senza dover affrontare le caratteristiche di basso livello della piattaforma di esecuzione target.

Come si vede in Figura 4.2, la creazione di un nuovo DSL prevede soltanto la realizzazione di una nuova trasformazione M2M verso il modello intermedio, in quanto questa ha natura domain-specific. Il successivo deployment sull'ambiente di esecuzione, invece, è garantito dalla trasformazione M2T successiva.

5. Definizione dei Domain-Specific Language

In questo capitolo, verrà descritta la progettazione e lo sviluppo dei due DSL realizzati in questa tesi: il primo è l'ActorLanguage introdotto nel capitolo precedente, ossia la rappresentazione intermedia a supporto della costruzione di ulteriori DSL. Il secondo è un DSL di alto livello, basato sul caso di studio di Stream Processing scelto, utilizzato per valutare la semplicità di realizzazione di un DSL e relativo dispiegamento delle applicazioni con esso prodotte su un ambiente parallelo (o distribuito).

Nella descrizione della progettazione e dello sviluppo dei DSL verrà seguita come riferimento una procedura, sintetizzata nella Tabella 5.1, ispirata all'approccio sistematico proposto da Strembeck et al. [66].

Attività principale	**Sotto-attività**	**Output**
Definizione del modello di linguaggio di base del DSL	Identificare l'astrazione del dominio, aggiungere astrazione del dominio al modello di linguaggio, definire i vincoli del modello di linguaggio	Sintassi astratta del linguaggio
Definizione del comportamento degli elementi linguistici del DSL	Selezionare il/i modello/i degli elementi linguistici, definire il comportamento degli elementi linguistici del modello DSL	Uno o più diagrammi del flusso di controllo
Definizione della/e sintassi/e concreta/e del DSL	Definire i simboli per gli elementi del modello di linguaggio, definire regole di produzione/composizione del DSL, definire la sintassi concreta del DSL	Sintassi concreta del linguaggio
Integrazione degli artefatti del DSL con la piattaforma/infrastruttura	Mappare gli artefatti del DSL sulle funzionalità della piattaforma, estendere/adattare la piattaforma, definire le trasformazioni DSL-to-platform, testare il DSL, verificare il DSL integrato	Trasformazione M2T oppure M2M

Tabella 5.1: Panoramica delle principali attività e sotto-attività nella progettazione e sviluppo di un DSL.

5.1 JetBrains MPS

Dal punto di vista pratico, per lo sviluppo di entrambi i DSL è stata scelta la Language Workbench MPS, prodotta da JetBrains. La decisione di adottare questa LW è stata motivata da un aspetto chiave: rispetto ad altre soluzioni, JetBrains MPS è risultato più intuitivo e facile da utilizzare, anche grazie alla disponibilità di una vasta documentazione/tutorial e al supporto di una community attiva e collaborativa—l'interazione diretta con il team di sviluppo di JetBrains MPS ha consentito di affrontare alcune problematiche tecniche incontrate durante la progettazione e lo sviluppo dell'ActorLanguage.

Questa facilità d'uso è resa possibile, ad esempio, dalla presenza di un *projectional editor*, un editor per linguaggi che permette di modificare direttamente l'AST (Abstract Syntax Tree), eliminando la necessità di un parser.

In particolare, i *projectional editor*, rispetto ai *parser-based editor*, presentano diversi vantaggi: tra questi, vi è la possibilità di avere un *code-completion menu* che supporta lo sviluppatore ed evita, ad esempio, di creare riferimenti verso nodi inesistenti. Una panoramica delle caratteristiche dei *projectional editor*, e una valutazione dell'utilizzo di questi all'interno di MPS, è presente in [71].

Dunque, un linguaggio all'interno di MPS è formato da tutti i nodi che compongono l'AST, che possono essere di tipi diversi, e dalle relazioni tra questi. Ogni nodo che forma l'AST possiede un nodo *parent* (se il nodo non è *root*, cioè non corrisponde alla radice dell'AST), dei nodi *child*, delle proprietà e dei riferimenti verso altri nodi dell'AST.

Ogni nodo, in un linguaggio, può essere di tipo diverso. Ogni nodo mantiene un riferimento al suo *concept*: esso definisce una classe di nodi e la struttura, in termini di proprietà, nodi *children* e riferimenti, di tutte le istanze di nodi appartenenti alla classe.

Un linguaggio, all'interno di MPS, è definito da un insieme di concetti e da informazioni supplementari. Queste informazioni supplementari prendono il nome di aspetti del linguaggio. Alcuni dei principali aspetti definiti da MPS

sono:

- *Structure*: contiene le definizioni di tutti i *concept* del linguaggio;
- *Constraints*: definisce dei vincoli, che restringono le possibili relazioni tra i nodi e i valori che le proprietà di questi possono assumere;
- *Behavior*: contiene eventuali metodi, definiti per ogni *concept*, che i nodi possono invocare;
- *Editor*: definisce i *projectional editor* di ogni *concept*, cioè le interfacce utente per l'editing dei nodi;
- *Actions*: definisce delle azioni da eseguire in momenti specifici del ciclo di vita di ogni nodo;
- *Intentions*: consentono la definizione di semplici manipolazioni del codice che supportano l'utente del linguaggio;
- *Generator*: definisce una trasformazione M2M verso un altro linguaggio, anch'esso implementato in MPS;
- *TextGen*: definisce una trasformazione M2T;
- *Typesystem*: consente la definizione di *checking rules* che possono essere verificate durante l'utilizzo del linguaggio.

5.2 Definizione di *ActorLanguage*

Il linguaggio *ActorLanguage* è il DSL basato sul modello degli attori ed utilizzato, nell'architettura software, come rappresentazione intermedia.

In seguito, verranno descritte in dettaglio le attività svolte in ogni fase della realizzazione del linguaggio.

5.2.1 Definizione del modello di linguaggio di base

La prima attività da svolgere, in questa fase iniziale, riguarda l'identificazione delle astrazioni di dominio: questo linguaggio vuole consentire, ad alto livello, la creazione di un insieme di attori che comunicano tra loro, rispettando le caratteristiche dell'Actor Model, introdotto nel capitolo precedente.

L'astrazione alla base del linguaggio è quella dell'*attore*. Il secondo concetto fondamentale è quello di comunicazione: affinché possa essere eseguita una qualsiasi computazione distribuita tra gli attori, questi hanno bisogno di comunicare tra loro. La comunicazione avviene attraverso scambi di *messaggi*. Per poter inviare un messaggio verso un altro attore, bisogna conoscere il suo *indirizzo*. Di conseguenza, il messaggio è formato da una *envelope*, che contiene l'indirizzo di destinazione e la *priorità* del messaggio, e dal *payload*, cioè i dati in esso contenuti.

Come già detto, la comunicazione nel modello ad attori è asincrona e bufferizzata, perciò ogni attore ha una *coda di messaggi* associata, che consente la bufferizzazione delle comunicazioni.

La ricezione del messaggio avviene tramite l'azione di *fetch* dalla coda di messaggi. Quando un attore riceve ed accetta una comunicazione, esegue una sequenza di *azioni*, che prende il nome di *comportamento*. Nel capitolo precedente è stato evidenziato che, in risposta al processamento di un task, quindi all'accettazione di una comunicazione, un attore può creare nuovi attori oppure creare un nuovo task, i.e. una nuova comunicazione. Quindi, le azioni disponibili, che definiscono il comportamento, devono prevedere la *creazione di un nuovo attore*, la *creazione di un nuovo messaggio*, ed il conseguente *invio* di quest'ultimo.

Ulteriori azioni che l'attore può eseguire, in risposta ad una comunicazione, includono la *creazione di una nuova envelope* e la *creazione di un nuovo payload*, da associare successivamente ad un messaggio, e l'esecuzione di una *funzione esterna*, che prende come parametro un messaggio e produce, come risultato, un nuovo payload.

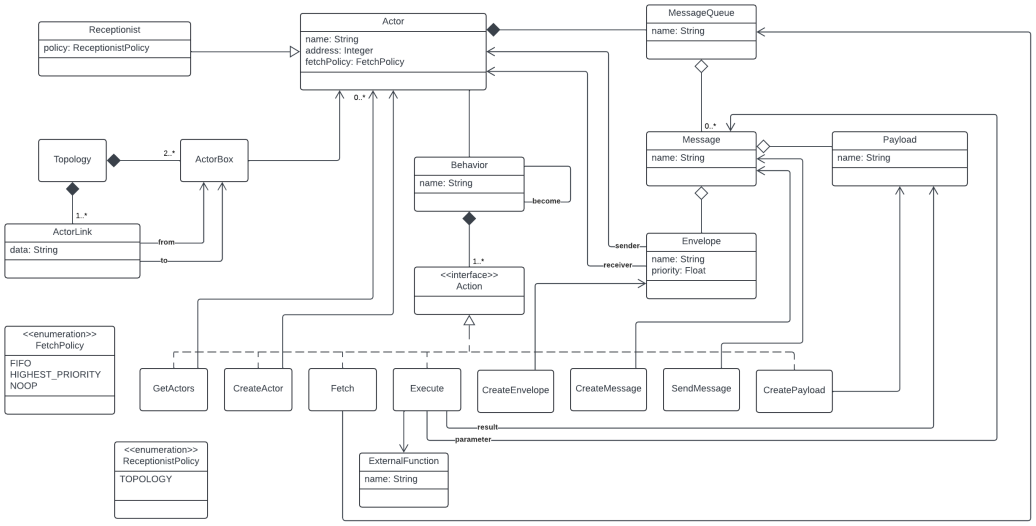


Figura 5.1: Sintassi astratta di *ActorLanguage*

In corrispondenza del processamento di una comunicazione, ogni attore deve specificare un comportamento sostitutivo: questo, detto *become*, è associato ad ogni comportamento.

Il linguaggio deve prevedere il concetto di *topologia*, cioè un grafo che interconnette gli attori: l'attore A_1 può comunicare con l'attore A_2 se e solo se la topologia contiene un arco che connette il nodo relativo ad A_1 e il nodo relativo ad A_2. I nodi sono modellati dal concetto *ActorBox*, mentre gli archi da *ActorLink*. Ad ogni *ActorLink* possono essere associati dei dati significativi per il dominio dello specifico DSL che è stato realizzato.

Nel modello ad attori, gli attori non dispongono di una visione completa della topologia. Perciò, esiste una tipologia particolare di attore, detto *Receptionist*, il quale ha la responsabilità esclusiva di fornire a ciascun attore le informazioni sugli attori a lui interconnessi. L'azione *get_actors* consente ad ogni attore di ottenere i suoi vicini, i.e. i nodi ad esso interconnessi nella topologia, dall'attore receptionist.

Il modello del linguaggio, composto dalle astrazioni chiave e dai collegamenti tra queste, è riassunto in Figura 5.1.

Per quanto riguarda i vincoli, questi sono stati formulati in linguaggio naturale, e successivamente implementati nell'aspetto *constraints* di MPS. I vincoli

in linguaggio naturale sono i seguenti:

- Il nome di ogni attore deve essere globalmente unico;
- Il nome di ogni comportamento deve essere globalmente unico;
- Il nome di ogni envelope deve essere localmente unico, i.e. all'interno del behavior in cui ne viene definita la creazione;
- Se p è il valore - di tipo *float* - che rappresenta la priorità associata ad una envelope, allora p deve appartenere ad un intervallo predeterminato a livello di linguaggio;
- L'attore destinatario, associato ad una envelope, può essere l'attore mittente oppure un attore ottenuto tramite la primitiva *get_actors*;
- Il nome di ogni payload deve essere localmente unico;
- Il parametro di una funzione esterna deve essere un messaggio definito all'interno dello stesso behavior, oppure il messaggio ricevuto all'attivazione del behavior;
- Il nome di ogni messaggio deve essere localmente unico;
- L'azione di invio di un messaggio può accettare, come parametro, soltanto un messaggio definito all'interno dello stesso behavior.

5.2.2 Definizione del comportamento degli elementi linguistici

Nella definizione del comportamento degli elementi linguistici, sono state individuate le principali funzionalità non banali offerte dal linguaggio e, per ognuna di queste, è stata creata una rete di Petri che le descrive.

Le funzionalità individuate sono state: l'invio di un messaggio, che include ulteriori attività, come la creazione del messaggio e la creazione o selezione di una envelope e di un payload; la creazione di un nuovo attore, che include la creazione o selezione di un comportamento (Figura 5.3).

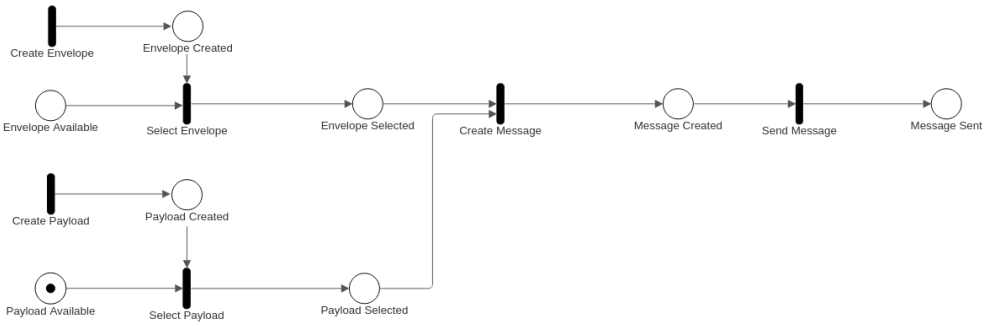


Figura 5.2: Rete di Petri che descrive l'invio di un messaggio

La rete di Petri che descrive l'invio di un messaggio è mostrata in Figura 5.2. La rete mostra due transazioni che possono essere attivate liberamente, che rappresentano la creazione di una nuova *envelope* e di un nuovo *payload*; l'attivazione di queste produce un token all'interno dei posti *Envelope Created* e *Payload Created*, che rappresentano la condizione in cui questi elementi sono stati creati. Un'ulteriore condizione riguarda la disponibilità, all'interno del *behavior*, di *payload* ed *envelope* già esistenti: questo aspetto è modellato dai posti *Payload Available* ed *Envelope Available*. Si può notare che *Payload Availabe* presenta un token, dal momento che in ogni *behavior* associato ad ogni attore, se la politica di *fetch* è diversa da `NOOP`, è sempre possibile assegnare ad un nuovo messaggio il *payload* del messaggio ricevuto. Attraverso le transazioni *Select Envelope* e *Select Payload*, è possibile associare al messaggio rispettivamente una *envelope* ed un *payload*, sia appena creati che già disponibili all'interno del *behavior*. Una volta selezionati entrambi questi elementi, è possibile creare il messaggio, tramite la transazione *Create Message* e, finalmente, inviarlo, attraverso la transazione *Send Message*.

La descrizione della funzionalità di creazione di un nuovo attore è presente nella rete di Petri in Figura 5.3.

Come si vede nella rete, la creazione di un attore presuppone la definizione di un nome e la selezione di una politica di *fetch* e di un *behavior* esistenti.

Figura 5.3: Rete di Petri che descrive la creazione di un attore

5.2.3 Definizione della sintassi concreta

La sintassi concreta di *ActorLanguage* è unica e prevalentemente testuale; soltanto il concetto di topologia prevede due sintassi, una testuale ed una grafica.

Il *root concept* del linguaggio è dato dal *concept ActorScript*, che modella il file su cui viene specificato l'insieme di attori, l'insieme di behavior e tutti gli altri concetti che caratterizzano il linguaggio, che ora vedremo in dettaglio.

La parte iniziale di *ActorScript* contiene la dichiarazione di uno o più tipi: il concetto di tipo non è presente nella sintassi astratta del linguaggio; infatti, è stato introdotto successivamente per arricchire il concetto di *payload*, consentendo di caratterizzarlo per tipo.

Successivamente alla dichiarazione dei tipi, *ActorScript* prevede la definizione delle funzioni esterne. Queste sono dichiarate per nome, senza specificarne ulteriori caratteristiche: l'idea alla base è quella di fornire supporto per l'esecuzione di funzioni di libreria, di carattere domain-specific.

La terza sezione è dedicata alla definizione dei *behavior*: qui si nota la composizione di diverse notazioni, all'interno della dichiarazione dei behavior, che avviene tramite la primitiva *create_behavior*. In particolare, il linguaggio prevede il concetto astratto *ActorAction*: ognuna delle azioni interne al *behavior* estende questo concetto.

Nello specifico delle azioni, queste sono:

- Esecuzione di una funzione esterna, attraverso la primitiva *execute*, che richiede il nome della funzione esterna, il messaggio da passare come parametro, il nome e il tipo del payload risultante, che potrà successivamente essere associato ad un messaggio.

- Richiesta degli attori "vicini" al *receptionist*, tramite la primitiva *get_actors*, che richiede la policy di ricerca dei vicini[1]. Inoltre, questa primitiva prevede che venga specificato il nome della lista dei riferimenti agli attori ottenuti dal receptionist.

- Creazione di un messaggio, tramite la primitiva *create_message*, che richiede come parametri una *envelope* ed un *payload*. Entrambi questi parametri possono essere selezionati tra le istanze associate al behavior, i.e. create precedentemente tramite le apposite primitive *create_envelope* e *create_payload*, oppure create direttamente dentro la definizione del messaggio, grazie alla *composabilità* delle sovracitate primitive con *create_message*.

- Creazione di una envelope, tramite la appena citata *create_envelope*, che prevede come parametri una *ActorReference* relativa all'attore destinatario ed un valore *float* di priorità, oltre al nome della *envelope*;

- Creazione di un payload, tramite la primitiva *create_payload*, che richiede il tipo - tra i tipi dichiarati a inizio script -, il contenuto, cioè una stringa, ed il nome del *payload*.

- Invio di un messaggio, per cui sono previste due primitive: la prima, *send_message*, invia un messaggio soltanto al destinatario specificato nella envelope, mentre la seconda, *send_message_to_group*, invia n messaggi ad n attori specificati in una lista di *ActorReference*, di solito ottenuta da *get_actors*. La seconda primitiva, inoltre, richiede il *payload*

[1]Attualmente, l'unica policy supportata è TOPOLOGY, che prevede la restituzione di tutti i nodi della topologia che, nell'albero, rappresentano i *children* dell'attore richiedente.

e la priorità dei messaggi in uscita, mentre si occupa automaticamente della creazione della *envelope*.

- Ulteriori azioni previste dal linguaggio, e componibili nella definizione dei behavior, riguardano costrutti tipici dei linguaggi di programmazione general-purpose, tra cui gli statement *for*, *if*, *switch-case*, e *for each actor reference*, che consente di iterare su una lista di *ActorReference*.

La quarta sezione di *ActorScript* contiene la definizione degli attori. A tale scopo, esistono due primitive: la prima è *create_actor*, che crea un singolo attore, caratterizzato da un nome, un comportamento ed una *FetchPolicy*, cioè la modalità con cui effettua l'operazione di *fetching* dalla coda di messaggi ad esso associata; la seconda è *create_actors*, che crea un numero specifico di attori con un certo *baseName*, che viene suffissato da un valore intero incrementale, un comportamento ed una *fetchPolicy*, che sono comuni a tutti gli attori.

La quinta ed ultima sezione di *ActorScript* riguarda la definizione della topologia. Questa sezione è l'unica ad essere caratterizzata da una doppia sintassi, una testuale ed una grafica. Entrambe le sintassi consentono di creare un certo numero di *ActorBox*, in genere una per attore, che modellano i nodi della topologia, ed un certo numero di *ActorLink*, che invece modellano gli archi. A questi ultimi, è possibile assegnare dei dati associati, sotto forma di stringa.

Un esempio di *ActorScript* è mostrato in Figura 5.4.

5.2.4 Integrazione con la piattaforma

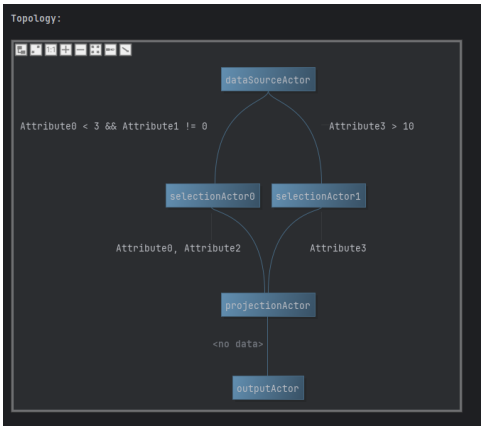
ROOT-Sim

La trasformazione M2T ha l'obiettivo di generare, a partire da un modello di *ActorLanguage*, del codice eseguibile sull'ambiente di esecuzione (speculativa) ROOT-Sim.

The ROme OpTimistic Simulator (ROOT-Sim) [57] è un runtime per l'esecuzione di applicazioni task-based parallelo e distribuito nato per supportare

```
ActorScript :

  Types:

  declare type type1
  declare type type2

  ------------------------------------------------------------------------------------------

  External Functions:

  include selection;
  include projection;
  include output;

  ------------------------------------------------------------------------------------------
  Behaviors:

  create_behavior ( projection , receivedMessage ) {
    projectionResult = execute ( projection , receivedMessage , type1 );
    List<ActorReference> actorReferences = get_actors ( Policy.TOPOLOGY );
    foreach ( ActorReference reference in actorReferences ) {
      Message message = create_message (
        projectionResult ,
        Envelope envelope = create_envelope ( reference , 5.0 );
      );
      send_message ( message );
    }
    No Become Statement
  }

  create_behavior ( selection , receivedMessage ) {
    selectionResult = execute ( selection , receivedMessage , type1 );
    List<ActorReference> actorReferences = get_actors ( Policy.TOPOLOGY );
    send_message_to_group ( actorReferences , selectionResult , 5.0 );
    No Become Statement
  }

  create_behavior ( output , receivedMessage ) {
    result = execute ( output , receivedMessage , type2 );
    No Become Statement
  }

  create_behavior ( dataSource , receivedMessage ) {
    List<ActorReference> actorReferences = get_actors ( Policy.TOPOLOGY );
    send_message_to_group ( actorReferences , receivedMessage.payload , 5.0 );
    No Become Statement
  }

  ------------------------------------------------------------------------------------------

  Actors:

  create_actor ( dataSourceActor , dataSource , FIFO );
  create_actors ( 2 , selectionActor , selection , FIFO );
  create_actor ( projectionActor , projection , FIFO );
  create_actor ( outputActor , output , FIFO );

  ------------------------------------------------------------------------------------------

  Topology:

  actors graph {

    actors :
      actor box actor : dataSourceActor
      actor box actor : selectionActor0
      actor box actor : selectionActor1
      actor box actor : projectionActor
      actor box actor : outputActor
    links :
      actor link actor from : dataSourceActor actor to : selectionActor0 {

        data :
          Attribute0 < 3 && Attribute1 != 0
      }
      actor link actor from : dataSourceActor actor to : selectionActor1 {

        data :
          Attribute3 > 10
      }
      actor link actor from : selectionActor0 actor to : projectionActor {

        data :
          Attribute0, Attribute2
      }
      actor link actor from : selectionActor1 actor to : projectionActor {

        data :
          Attribute3
      }
      actor link actor from : projectionActor actor to : outputActor {

        data :
          <no data>
      }
  }
```

(a) Script completo, con topologia testuale



(b) Topologia a grafo

Figura 5.4: Esempio di utilizzo del linguaggio *ActorLanguage*

la simulazione speculativa parallela e distribuita, basato su un paradigma di sincronizzazione ottimistica, cioè basata su *rollback*. Esso offre un modello di programmazione, basato sullo standard ANSI-C, che supporta in maniera trasparente tutti i servizi necessari per l'esecuzione parallela.

Le API principali offerte da ROOT-Sim, che sono alla base della trasformazione M2T, sono:

- `int ProcessEvent(int me, time_type now, int event_type, void *content, int size, void *state)`: callback, che deve essere implementata a livello applicativo, che consente all'applicazione di definire il processamento degli eventi nella simulazione;
- `int ScheduleNewEvent(int where, time_type timestamp, int event_type, void *content, int size)`: consente l'injection di un nuovo evento nel sistema, destinato all'oggetto identificato da `where`.

Inoltre, nella trasformazione M2T, verrà utilizzata una libreria accessoria a ROOT-Sim, cioè ROOT-Sim Topology [56], che consente di definire una topologia configurabile ed accessibile durante l'esecuzione di una simulazione in ROOT-Sim.

Trasformazione Model-to-Text

La trasformazione M2T si fonda sulla corrispondenza tra il concetto di attore in *ActorLanguage* e quello di processo logico (LP) in ROOT-Sim: ogni attore è associato a un LP distinto. In particolare, il mapping avviene tramite un indirizzo, ovvero un valore intero attribuito a ciascun attore, che rappresenta l'ID dell'LP corrispondente.

La topologia di *ActorLanguage* viene tradotta direttamente in una topologia a grafo, sfruttando la libreria ROOT-Sim Topology: questa consente di specificare interconnessioni tra LP, tramite le seguenti API: (A) `bool AddTopologyLink(struct topology *topology, lp_id_t from, lp_id_t to, double probability)`, che aggiunge un collegamento tra due

LP, aventi ID dati da `from` e `to`; (B) `bool SetTopologyLinkData(struct topology *topology, lp_id_t from, lp_id_t to, void *data)` che consente, invece, di associare dati arbitrari - nel nostro caso stringhe - ad ogni arco della topologia.

Come già detto, l'API che consente di specificare le azioni da compiere nel processamento degli eventi, durante la simulazione, è `ProcessEvent`: in base al valore del parametro `me`, è possibile diversificare le elaborazioni degli eventi in base al particolare LP (o attore). ROOT-Sim prevede due tipologie di eventi standard: `LP_INIT` rappresenta l'evento di inizializzazione dell'LP; `LP_FINI` è la tipologia di evento che comunica la terminazione dell'esecuzione. A questi due eventi standard è stato aggiunto un ulteriore evento generico `EVENT` che rappresenta la trasmissione di una comunicazione tra due attori.

Entrando nel dettaglio della trasformazione, i concetti principali di *Actor-Language* vengono compilati in codice per ROOT-Sim nel seguente modo:

- *Behavior*: può essere visto come la sequenza di azioni che un LP deve compiere in corrispondenza di un evento di tipologia `EVENT`. Di conseguenza, si traduce in una definizione di funzione, che può essere invocata durante l'esecuzione di `ProcessEvent`.

- *Creazione di un attore*: si traduce nell'aggiunta, al corpo della funzione `ProcessEvent`, dell'invocazione del *behavior*, i.e. funzione, corrispondente, se il valore di `me` equivale all'indirizzo dell'attore.

- *ActorReference*: si traduce semplicemente nell'ID dell'LP corrispondente all'attore. Allo stesso modo, una lista di *ActorReference* equivale ad una lista di ID degli LP corrispondenti.

- *Funzione esterna*: l'invocazione di una funzione esterna corrisponde all'invocazione effettiva dell'implementazione della funzione che viene fornita, in genere, da una libreria esterna.

- L'operazione di *fetch* non ha bisogno di una controparte nel codice di basso livello, in quanto è implicita: l'attivazione del *behavior*, cioè della funzione corrispondente, è scatenata dalla ricezione di un messaggio.

- Analogamente al punto precedente, il concetto di coda di messaggi è inglobato nel funzionamento interno di ROOT-Sim, che garantisce la comunicazione asincrona bufferizzata.

- Il *receptionist* non viene modellato direttamente come un attore, quindi come un LP distinto. Ogni LP possiede un puntatore alla **struct** topology che mantiene informazioni sulla topologia: la primitiva *get_actors* si traduce nell'invocazione della funzione **void** GetAllReceivers(**struct** topology *topology, lp_id_t from, lp_id_t *receivers), offerta dalla libreria ROOT-Sim topology, che popola l'array receivers con gli indirizzi degli LP collegati a quello corrente, in accordo alla topologia.

- Il concetto di *payload* ha carattere domain-specific, dato dalla presenza dell'attributo *tipo*: nel caso di studio, che verrà dettagliatamente descritto in seguito, questo concetto non sarà tradotto direttamente, ma sarà inglobato nella traduzione del *messaggio*, in funzione del valore dell'attributo *tipo*.

- Analogamente al punto precedente, il *messaggio* è anch'esso di natura domain-specific. Per questo motivo, la sua creazione è demandata ad una funzione di libreria, che verrà descritta successivamente, che popola delle strutture differenti in base all'attributo *tipo* del *payload* associato. L'invio del messaggio, d'altra parte, avviene attraverso la già citata API ScheduleNewEvent.

- Il concetto di *envelope* viene implementato tramite la seguente struttura:

```
1    typedef struct {
2        lp_id_t sender;
3        float priority;
4    } Envelope;
```

Listing 5.1: Struttura dell'Envelope

Si osserva che nella definizione della struttura non è presente l'attributo *receiver*, ovvero l'ID dell'LP destinatario. Durante lo sviluppo della trasformazione M2T e delle librerie di supporto, questo dato è stato ritenuto superfluo e, pertanto, è stato rimosso per garantire un maggiore risparmio nell'uso della memoria.

5.3 Definizione di *QueryLanguage*

Il linguaggio *QueryLanguage* è un DSL che, nell'architettura in Figura 4.1, si pone al livello di astrazione più alto. Tale linguaggio è ispirato a Streaming SQL [32], di cui è di fatto un sottoinsieme. Il dominio di interesse del linguaggio è lo stream processing, facendo riferimento ad un caso di studio specifico, cioè l'analisi del dataset *TLC Trip Record Data* [54], diventato ormai un benchmark di riferimento nel settore della data analysis.

In sintesi, *QueryLanguage* è un linguaggio ispirato da SQL, che consente di specificare query general-purpose. Esso offre costrutti di base per creare, popolare, modificare e interrogare tabelle.

5.3.1 Definizione del modello di linguaggio di base

Differentemente dal caso di *ActorLanguage*, per *QueryLanguage* l'individuazione delle astrazioni chiave risulta essere più agevole, in quanto esso è fondato su uno dei DSL più noti e approfonditamente studiati, ossia SQL.

Il concetto chiave è quello di *Statement*, che fornisce un'interfaccia per tutti i costrutti del linguaggio, che sono:

- *CreateTable*: consente di creare una nuova tabella, oppure una sorgente di flusso di dati;
- *InsertInto*: consente di inserire dati all'interno di una tabella esistente;

- *Delete*: consente di eliminare dati da una tabella esistente, eventualmente in funzione di una condizione;
- *Update*: consente di modificare dati all'interno di una tabella esistente, eventualmente in funzione di una condizione;
- *Select*: consente di interrogare una tabella esistente, cioè di estrarre dati, eventualmente secondo una condizione.

Gli statement *InsertInto*, *Delete* e *Update*, seppure presenti nel linguaggio, non sono di interesse in un contesto di *stream processing*, dove è presente una sorgente di dati dinamica e non modificabile dall'applicazione. Tuttavia, questi sono stati implementati per questioni di riusabilità del linguaggio in contesti differenti, legati al processamento di dati statici.

Tutti gli statement fanno riferimento ad un insieme di colonne, di dimensione non predefinita, e ad una tabella a cui queste colonne appartengono. Differentemente dagli altri statement, il costrutto *Select* può fare riferimento a più di una tabella, nel caso di operazione di *Join*.

Gli statement *Delete*, *Update* e *Select* possono fare riferimento ad una condizione per, rispettivamente, eliminare, modificare o selezionare soltanto i dati che la rispettano. Questo è modellato dal costrutto *Where*, che consente di specificare una condizione. Una condizione può essere *semplice* o *multipla*: nel primo caso, essa è composta da un riferimento verso la colonna di interesse, da un operatore di confronto e da un valore che deve essere confrontato con il valore associato alla colonna; nel secondo caso, una condizione multipla corrisponde ad un albero binario, dove ogni nodo può essere una foglia, dunque una condizione semplice, oppure avere due nodi figli, che possono essere foglie dell'albero, oppure radici di un ulteriore sottoalbero. Un esempio di costruzione di un albero binario a partire da una condizione multipla è mostrato in Figura 5.5.

L'unico statement ad avere ulteriori concetti collegati è *Select*, per cui è possibile specificare i costrutti *GroupBy*, *OrderBy* e *Window*.

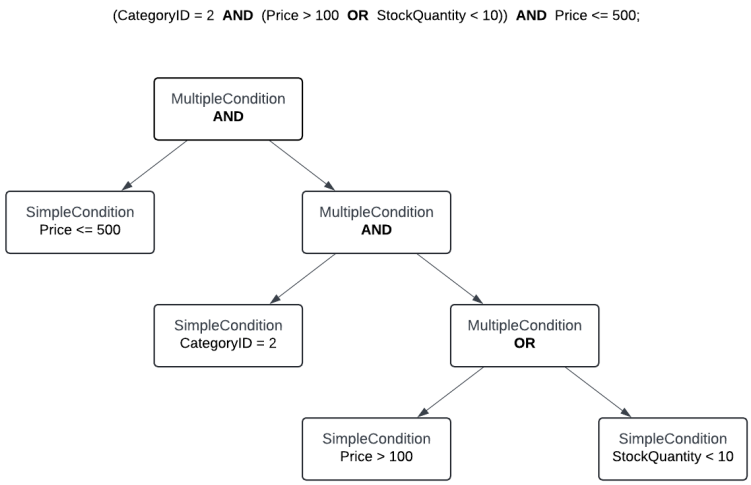


Figura 5.5: Costruzione di un albero binario da una condizione multipla

GroupBy consente di effettuare il raggruppamento dei risultati dell'operazione di selezione in funzione del valore di una particolare colonna.

OrderBy prevede l'ordinamento dei risultati dell'operazione di selezione in base al valore di una specifica colonna.

Window, infine, è utilizzato per definire finestre temporali di dati su cui, tipicamente, effettuare aggregazioni. Le finestre vengono definite in base al valore di una colonna, di tipo *DATETIME*.

Infine, quando uno statement *Select* ha un collegamento verso una colonna, a questa può essere associata una *funzione di aggregazione*. Tra queste, dato un insieme di valori, sono definite le seguenti funzioni: *Min*, che estrae il valore minimo dall'insieme; *Max*, che estrae il valore massimo dall'insieme; *Average*, che calcola il valore medio dell'insieme; *Sum*, che calcola la somma dei valori nell'insieme; *Count*, che calcola la cardinalità dell'insieme.

La sintassi astratta di *QueryLanguage* è riassunta in Figura 5.6.

Analogamente a ciò che è stato fatto per *ActorLanguage*, i vincoli sono stati formulati in linguaggio naturale e, successivamente, implementati nell'aspetto *constraints* di MPS. I vincoli in linguaggio naturale, alcuni dei quali derivati da quelli previsti dalla sintassi di SQL, sono i seguenti:

- Lo *scope* di visibilità delle colonne, negli statement, è ristretto alle co-

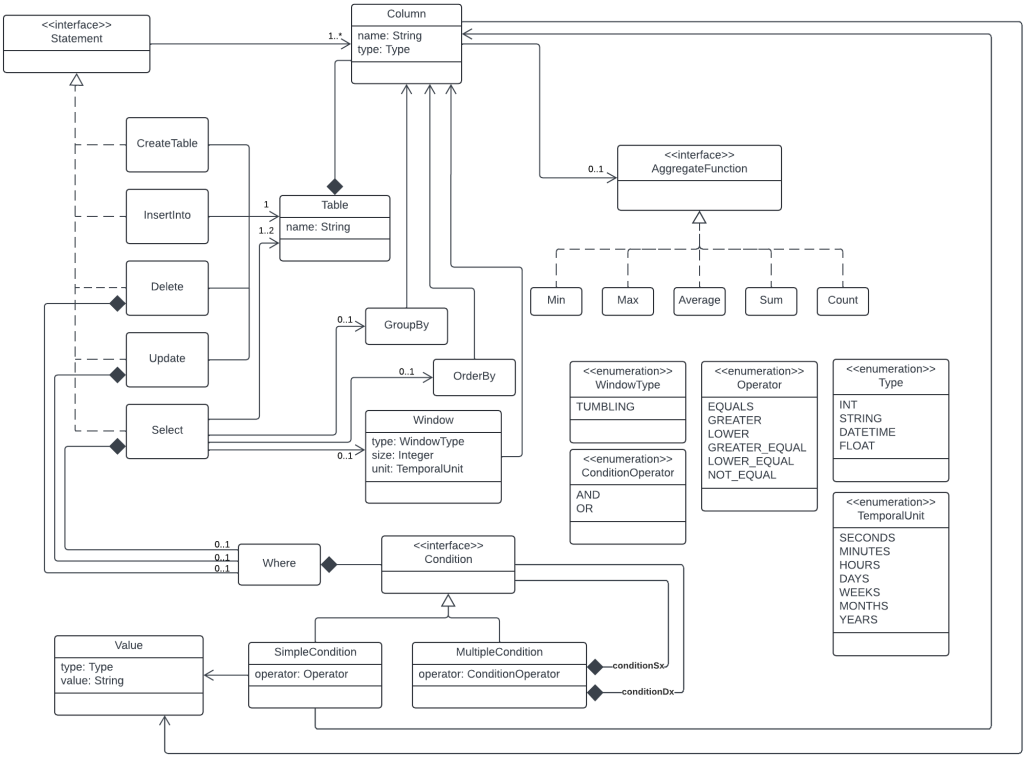


Figura 5.6: Sintassi astratta di *QueryLanguage*

lonne della tabella associata.

- Una colonna può avere un riferimento ad una funzione di aggregazione se e solo se esiste un collegamento tra un'istanza di *Select* e l'istanza della colonna.

- Se uno statement *Select* prevede collegamenti verso almeno uno dei concetti *GroupBy* e *OrderBy*, le colonne associate a questi ultimi devono essere le colonne associate alla particolare istanza di *Select*.

- Se lo statement *Select* presenta un collegamento verso il concetto *GroupBy*, le colonne associate a quest'ultimo devono essere tutte le colonne che non prevedono una funzione di aggregazione associata.

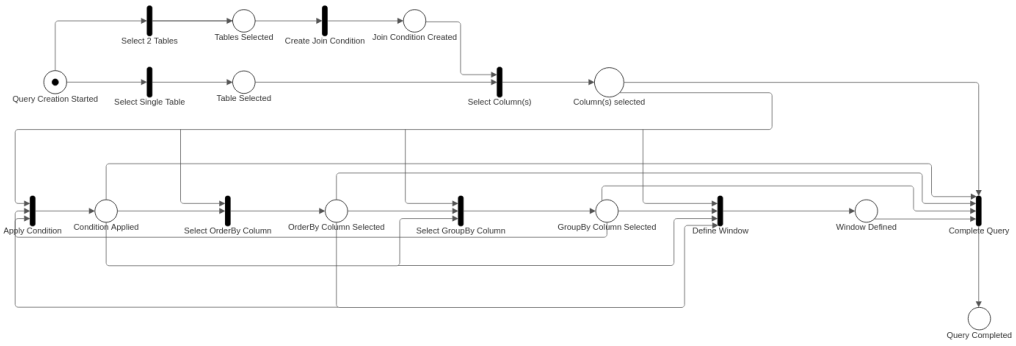


Figura 5.7: Rete di Petri per la creazione di una query di selezione

5.3.2 Definizione del comportamento degli elementi linguistici

Per la descrizione del comportamento degli elementi linguistici, è stata scelta la funzionalità di creazione di una query di selezione, in quanto questa consente di descrivere molte delle funzionalità accessorie offerte dal linguaggio.

La rete di Petri corrispondente è presente in Figura 5.7. In corrispondenza della creazione della query (posto *Query Creation Started*) è possibile associare a questa una o due tabelle: nel secondo caso (*Select 2 Tables*), diventa necessario specificare una condizione di join. Una volta selezionate una o due tabelle, bisogna individuare una o più colonne da associare alla query, in base alle quali sarà eseguita una proiezione. Successivamente, diventa già possibile completare la creazione della query, oppure specificare, senza un particolare ordine, una condizione da applicare nella selezione (*Apply Condition*), una colonna in base alla quale ordinare la tabella risultante (*Select OrderBy Column*), una colonna in base alla quale raggruppare il risultato (*Select GroupBy Column*), oppure una finestra temporale in funzione della quale suddividere i dati (*Define Window*). La presenza di ognuno di questi aspetti non è obbligatoria.

5.3.3 Definizione della sintassi concreta

La sintassi concreta di *QueryLanguage* è unica e completamente testuale. Il *root concept* è il concetto *Script*, che contiene una lista di statement. Tutti gli

```
Script

CREATE TABLE Person (
  ID Integer
  fullName String
  birthDate Date
  age Integer
  universityId Integer
)

CREATE TABLE University (
  ID Integer
  name String
  city String
)

INSERT INTO University ( ID, name, city )
VALUES ( 1, University of Rome Tor Vergata, Rome )

INSERT INTO Person ( ID, age, birthDate, fullName, universityId )
VALUES ( 1, 23, 2000-01-01, Mario Rossi, 1 )

INSERT INTO Person ( ID, age, birthDate, fullName, universityId )
VALUES ( 2, 30, 1994-01-01, John Smith, 1 )

UPDATE Person
SET ( age ) = ( 24 )
WHERE ID == 1

DELETE FROM Person
WHERE ID == 2

SELECT universityId, AVG(age)
FROM Person
WHERE age <= 35
GROUP BY universityId
ORDER BY age
<no window>

SELECT *
FROM Person JOIN University ON Person.universityId = University.ID
WHERE ( Person.age >= 18 ) AND ( ( University.city == Rome ) OR ( University.city == Milan ) )
<no groupBy>
<no orderBy>
<no window>
```

```
Script

CREATE TABLE Taxis (
  VendorID Integer
  tpep_pickup_datetime String
  tpep_dropoff_datetime String
  passenger_count Integer
  trip_distance Float
  RatecodeID Integer
  PULocationID Integer
  DOLocationID Integer
  payment_type Integer
  fare_amount Float
  extra Float
  mta_tax Float
  tip_amount Float
  tolls_amount Float
  improvement_surcharge Float
  total_amount Float
  congestion_surcharge Float
  Airport_fee Float
)

CREATE VIEW HighTipsRides AS
SELECT *
FROM Taxis
WHERE tip_amount >= 10.0
<no groupBy>
<no orderBy>
WINDOW ( tpep_pickup_datetime , TUMBLING , 60 SECONDS )

SELECT PULocationID, DOLocationID
FROM HighTipsRides
<no where>
<no groupBy>
<no orderBy>
<no window>

SELECT PULocationID, SUM(total_amount)
FROM Taxis
WHERE ( passenger_count > 3 ) OR ( extra != 0.0 )
GROUP BY PULocationID
ORDER BY total_amount
WINDOW ( tpep_pickup_datetime , TUMBLING , 3 HOURS )
```

Figura 5.8: Esempi di utilizzo del linguaggio *QueryLanguage*

statement introdotti in precedenza sono stati realizzati come implementazione di un'unica interfaccia, di nome *DBOperation*.

La sintassi testuale degli statement è del tutto analoga alla notazione SQL. Inoltre, nello sviluppo del linguaggio è stato aggiunto il concetto di *vista*, cioè una nuova tabella, creata attraverso una selezione, che può essere interrogata separatamente.

In Figura 5.8 sono presenti due esempi di *Script*, uno dei quali dimostra l'utilizzo delle viste.

5.3.4 Integrazione con la piattaforma: Trasformazione Model-to-Model

Nell'architettura software rappresentata in Figura 4.1, *QueryLanguage* si pone al livello dei DSL. Perciò, in questo caso, l'integrazione con la piattaforma consiste nella trasformazione M2M verso la rappresentazione intermedia, cioè *ActorLanguage*.

La trasformazione ha come scopo quello di supportare unicamente il caso di studio relativo allo stream processing sul dataset citato in precedenza: di conseguenza, non viene supportata la traduzione di statement diversi dalla *Select*. Infatti, nel caso di studio considerato, che verrà descritto in dettaglio in una sezione successiva, vi è un'unica tabella logica, che contiene i dati prodotti dallo stream; di conseguenza, non è stato ritenuto necessario offrire supporto alle operazioni di creazione e gestione di ulteriori tabelle. Nella trattazione successiva, con il termine query verranno indicate univocamente le query di selezione, prodotte attraverso lo statement *Select*.

L'idea alla base della trasformazione M2M è quella di individuare una topologia a partire dall'insieme di query e delegare le azioni domain-specific, in questo caso le operazioni SQL modellate dal linguaggio, a librerie esterne, sfruttando il concetto di "esecuzione di funzioni esterne" offerto da *ActorLanguage*.

Per supportare il caso di studio, è stata necessaria la creazione di un modulo di *DataInjection* ad hoc, che verrà dettagliato in una sezione successiva. Questo modulo è presente nell'algoritmo di trasformazione, come un attore, il cui comportamento è delegato all'implementazione, che riveste il ruolo di radice nell'albero della topologia.

Le query vengono suddivise in due partizioni: il primo gruppo contiene soltanto le query che fanno riferimento a tabelle standard, cioè definite tramite lo statement *CreateTable*; le query appartenenti al secondo gruppo, invece, hanno come tabella di riferimento una *vista*, cioè una tabella prodotta attraverso una selezione. In seguito, le query appartenenti al primo gruppo saranno indicate come "query standard", mentre quelle appartenenti al secondo gruppo come "query sulle viste".

La trasformazione M2M verrà descritta in seguito, suddividendo la trattazione in base alle le fasi di creazione dell'*ActorScript*.

Dichiarazione dei tipi personalizzati e delle funzioni esterne Sono previsti due tipi, cioè *ROWS* e *GROUPS*, che rappresentano rispettivamente

liste di righe e di gruppi di righe. Questi sono i tipi associati ai *payload* che popolano i messaggi che vengono scambiati tra gli attori.

Per quanto riguarda le funzioni esterne, ne viene dichiarata una per ogni operazione SQL prevista dal linguaggio, che sono: selezione, proiezione, join, raggruppamento, ordinamento, suddivisione per finestre temporali (*window*) e tutte le funzioni di aggregazione precedentemente enumerate, cioè *min*, *max*, *average*, *sum*, *count*.

Definizione dei *behavior* Come già detto, l'esecuzione delle operazioni domain-specific è demandata alle funzioni esterne. In particolare, viene individuato un behavior per ogni funzione esterna, che rispetta una struttura standard, cioè è formato dall'invocazione di *execute*, per l'esecuzione della funzione, dall'invocazione di *get_actors*, per ottenere gli attori vicini e dall'invio del risultato della funzione ai vicini. Inoltre, allo scopo di modellare la presenza dell'attore di *DataInjection*, viene incluso un ulteriore behavior vuoto, che verrà assegnato a questo attore.

Creazione degli attori Il primo attore ad essere creato è quello che simula la presenza del modulo di *DataInjection*, chiamato *dataSource*. Questo attore ha come *fetchPolicy* NOOP, cioè ignora eventuali messaggi in arrivo.

La creazione degli attori successivi presuppone un'analisi dell'insieme delle query. In particolare, da ogni query vengono estratti un certo numero di attori come segue:

- Se la query prevede la clausola *Where* e si riferisce ad una sola tabella, deve essere individuato un attore di selezione;
- Se la query prevede la clausola *Where* e si riferisce a due tabelle (*join*):
  - Se la condizione c si può scomporre in due sotto-condizioni single-table c_1 e c_2 tali che $c = c_1 \wedge c_2$, allora vanno individuati due attori di selezione;

  - Se la condizione c si può scomporre in due sotto-condizioni single-table c_1 e c_2 tali che $c = c_1 \vee c_2$, allora va individuato un solo attore di selezione, relativo a c;
  - Se la condizione c è già single-table, va individuato un solo attore di selezione, relativo a c.

- Se la query seleziona un sottoinsieme proprio degli attributi della tabella, deve essere individuato un attore di proiezione;
- Se la query prevede la clausola *GroupBy*, deve essere individuato un attore di raggruppamento;
- Se la query prevede la clausola *OrderBy*, deve essere individuato un attore di ordinamento;
- Se la query si riferisce a più di una tabella, deve essere individuato un attore di *join*;
- Se almeno una colonna ha una funzione di aggregazione associata, allora deve essere individuato un attore corrispondente alla specifica funzione di aggregazione;
- Se la query prevede la clausola *Window*, deve essere individuato un attore *window*.

Ad ognuno di questi attori, è associato il *behavior* corrispondente, e la *fetchPolicy* FIFO. L'indirizzo degli attori è determinato automaticamente, in maniera incrementale.

Inoltre, al fine di ridurre il numero di attori nel sistema, in particolare per quanto concerne gli attori di selezione e di *window*, non viene creato un attore per query ma, rispettivamente, un attore unico per condizione e per dimensione della finestra. In altre parole, se due query presentano la stessa condizione, oppure clausole *Window* con la stessa dimensione, allora verrà creato un unico attore che rappresenta entrambe le query.

Dal punto di vista pratico, MPS offre la possibilità di definire delle *mapping label*, che mappano un nodo, in genere appartenente al modello sorgente, verso un secondo nodo, generalmente appartenente al modello di destinazione. Nella trasformazione, ogni attore è associato ad una *mapping label*: in questo modo, gli attori di selezione possono essere acceduti, nell'intero sistema, a partire dalla condizione associata alla query, mentre gli attori *window* a partire dalla dimensione della finestra. Tutti gli altri attori, invece, possono essere acceduti a partire dalla query stessa. Nella fase di costruzione della topologia, queste etichette permetteranno di accedere direttamente agli attori specifici, partendo da ogni query.

La procedura di creazione degli attori è riassunta nell'Algoritmo 1.

Creazione della topologia Come già detto, il punto cruciale della trasformazione M2M è la definizione della topologia, in quanto determina le interazioni fra gli attori, quindi tra i singoli operatori SQL.

Il primo step della creazione della topologia consiste nella definizione delle *ActorBox*, che rappresentano i nodi del grafo risultante: viene creata una *ActorBox* per ogni attore creato precedentemente.

Successivamente, si passa alla creazione degli *ActorLink* (o link), cioè degli archi. Inizialmente, vengono considerate soltanto le query standard, in quanto le query sulle viste presuppongono la creazione della vista per cui, nella topologia, la catena di attori relativa ad ognuna di queste query deve essere agganciata alla catena di attori che definiscono la vista.

L'ordine di precedenza tra operatori, che viene presupposto nella creazione dei link per le query standard, è il seguente:

$$dataSource \rightarrow window \rightarrow selezione \rightarrow proiezione \rightarrow raggruppamento \rightarrow aggregazione \rightarrow ordinamento$$

Invece, nel caso delle query sulle viste, la catena di attori segue la query di

Algorithm 1: Algoritmo di creazione degli attori

Input: Insieme delle query Q
Output: Insieme di attori $\mathcal{A}$

1. Crea l'attore *dataSource*, con *fetchPolicy* NOOP e con il corrispondente comportamento associato.

2. **Per ogni** query $q \in Q$:

   (a) IF q si riferisce ad una sola tabella e contiene la clausola *Where* con condizione c AND non esiste un attore associato a c THEN
   - Crea un attore di selezione s e la mapping label $c \rightarrow s$

   (b) IF q si riferisce ad due tabelle e contiene la clausola *Where* con condizione c AND non esiste un attore associato a c THEN
   - Se la condizione c si può scomporre in due sotto-condizioni single-table c_1 e c_2 tali che $c = c_1 \wedge c_2$, crea due attori di selezione s_1 e s_2 e le mapping label $c_1 \rightarrow s_1$ e $c_2 \rightarrow s_2$;
   - Se la condizione c si può scomporre in due sotto-condizioni single-table c_1 e c_2 tali che $c = c_1 \vee c_2$, crea un attore di selezione s e la mapping label $c \rightarrow s$;
   - Se la condizione c è già single-table, crea un attore di selezione s e la mapping label $c \rightarrow s$.

   (c) IF q contiene la clausola *Window* con dimensione d AND non esiste un attore associato a d THEN
   - Crea un attore *window* w e la mapping label $d \rightarrow w$

   (d) IF q seleziona un sottoinsieme proprio degli attributi della tabella THEN
   - Crea un attore di proiezione p e la mapping label $q \rightarrow p$

   (e) IF q contiene la clausola *GroupBy* THEN
   - Crea un attore di raggruppamento g e la mapping label $q \rightarrow g$

   (f) IF q contiene la clausola *OrderBy* THEN
   - Crea un attore di ordinamento o e la mapping label $q \rightarrow o$

   (g) IF q fa riferimento a più di una tabella THEN
   - Crea un attore di *join* j e la mapping label $q \rightarrow j$

   (h) IF almeno una colonna in q ha una funzione di aggregazione associata THEN
   - Crea un attore per la funzione di aggregazione a e la mapping label $q \rightarrow a$

3. Assegna il *behavior* corrispondente e la *fetchPolicy* FIFO a ciascun attore.

4. Genera automaticamente l'indirizzo per ogni attore in modo incrementale.

Return: $\mathcal{A}$

definizione della vista, in base al seguente ordinamento:

$$\overbrace{dataSource \rightarrow \cdots \rightarrow ultimo\ nodo}^{\text{query di definizione della vista}} \rightarrow selezione \rightarrow proiezione \rightarrow raggruppamento \rightarrow aggregazione \rightarrow ordinamento$$

dove l'ultimo nodo dipende dalla natura della query di definizione della vista.

In entrambi i casi, la validità degli ordinamenti è garantita da alcuni lavori presenti nella letteratura [28, 69].

L'idea alla base della creazione dei link consiste nell'analizzare tutte le query e aggiungere i link corrispondenti alla struttura di ognuna di esse, rispettando gli ordinamenti visti. La procedura completa è descritta nell'algoritmo 2.

Osservazioni Si può notare, ad esempio nella formulazione dell'Algoritmo 2, che il join viene supportato soltanto nel caso di query su due viste, oppure su una vista e sulla tabella originale, in quanto il linguaggio è stato progettato in un contesto in cui si ha una sola tabella, corrispondente ai dati prodotti dallo stream in oggetto. Di conseguenza, non ha senso effettuare il join, se non definendo delle viste.

Inoltre, per questioni di tempo e di praticità, il caso di join con condizione scomponibile in due sotto-condizioni single-table in OR tra loro non è stato considerato nell'implementazione, seppur presente negli algoritmi descritti.

Algorithm 2: Algoritmo di creazione degli *ActorLink*

Input: Insieme delle query Q
Output: Insieme di *ActorLink* $\mathcal{L}$

1. Itera su tutte le query $q \in Q$:
   (a) Inizializza $lastActor \leftarrow$ NULL.
   (b) Se q è una query standard, allora $lastActor \leftarrow dataSource$; altrimenti, inizializza $lastActor$ all'ultimo nodo della query di definizione della vista.
   (c) Se q non prevede il *join*:
      i. Se q è una query standard:
         - Se q prevede la clausola *Window* con dimensione d, allora collega *dataSource* all'attore dato dalla *mapping label* $d \rightarrow w$, associando all'arco la dimensione d. $lastActor \leftarrow w$.
      ii. Se q prevede la clausola *Where* con condizione c, allora collega $lastActor$ all'attore dato dalla *mapping label* $c \rightarrow s$, associando all'arco la condizione c. $lastActor \leftarrow s$.
      iii. Se q seleziona un sottoinsieme proprio degli attributi, collega $lastActor$ all'attore dato dalla *mapping label* $q \rightarrow p$, associando all'arco i nomi degli attributi selezionati. $lastActor \leftarrow p$.
      iv. Se q prevede la clausola *GroupBy*, allora collega $lastActor$ all'attore dato dalla *mapping label* $q \rightarrow g$, associando all'arco il nome dell'attributo di raggruppamento. $lastActor \leftarrow g$.
      v. Se q prevede una colonna con una funzione aggregata associata, collega $lastActor$ all'attore dato dalla *mapping label* $q \rightarrow a$, associando all'arco il nome della colonna. $lastActor \leftarrow a$.
      vi. Se q prevede la clausola *OrderBy*, allora collega $lastActor$ all'attore dato dalla *mapping label* $q \rightarrow o$.
   (d) Altrimenti, se q prevede il *join*:
      i. Inizializza $lastActor1$ e $lastActor2$, rispettivamente, agli ultimi attori nelle catene di definizione delle viste utilizzate dalla tabella.
      ii. Se q contiene la clausola *Where*:
         - Se la condizione c associata a $Where$ è decomponibile in due sottocondizioni single-table c_1, c_2 in AND:
            A. Crea due link tra $lastActor1$ e $lastActor2$ e, rispettivamente, gli attori dati dalle mapping label $c_1 \rightarrow s_1$ e $c_2 \rightarrow s_2$, etichettandoli con le condizioni c_1 e c_2.
            B. Crea due link da s_1 e s_2 all'attore dato dalla mapping label $q \rightarrow j$, etichettati con i nomi degli attributi di join delle due tabelle.
            C. Se q seleziona un sottoinsieme proprio degli attributi della tabella, collega j all'attore dato dalla mapping label $q \rightarrow p$, etichettato con i nomi degli attributi di proiezione. $lastActor \leftarrow p$.
            D. Segui i punti iv, v e vi di (c).
         - Se la condizione c associata a $Where$ è decomponibile in due sottocondizioni single-table c_1, c_2 in OR:
            A. Crea due link da $lastActor1$ e $lastActor2$ verso l'attore dato dalla mapping label $q \rightarrow j$, etichettati con gli attributi di join delle due tabelle. $lastActor \leftarrow j$.
            B. segui i punti ii, iii, iv, v, vi di (c).
         - Se c è già una condizione single-table:
            A. In base alla vista a cui c si riferisce, crea un link tra $lastActor1$ ($lastActor2$) verso l'attore dato dalla mapping label $c \rightarrow s$, ed un link da $lastActor2$ ($lastActor1$) verso l'attore dato da $q \rightarrow j$. $lastActor \leftarrow j$.
            B. Crea un link dall'attore dato da $c \rightarrow s$ verso l'attore dato da $q \rightarrow j$.
            C. Segui i punti iii, iv, v, vi di (c).
      iii. Altrimenti, se q non contiene la clausola $Where$:
         A. Crea due link da $lastActor1$ e $lastActor2$ verso l'attore dato dalla mapping label $q \rightarrow j$, etichettati con gli attributi di join delle due tabelle. $lastActor \leftarrow j$.
         B. Segui i punti iii, iv, v e vi di (c).

Return: $\mathcal{L}$

6. Caso di studio: Stream Processing

In questa sezione verrà descritto il caso di studio introdotto per la valutazione del sistema a cui si è accennato nei capitoli precedenti. Tale caso di studio riguarda lo *stream processing* di dati relativi alle corse di taxi sull'intero territorio della città di New York.

Il contesto scelto riguarda lo *stream processing*, una tecnica che consente di elaborare in tempo reale flussi continui di dati. A differenza dell'elaborazione batch, che gestisce grandi quantità di dati statici, lo stream processing lavora su dati che vengono generati costantemente, garantendo risposte immediate e aggiornamenti in tempo reale. Questo è particolarmente utile in scenari come il monitoraggio di eventi o transazioni che richiedono elaborazioni tempestive.

Nel nostro caso, i dati utilizzati per lo stream processing provengono dal dataset *TLC Trip Record Data*, che contiene informazioni dettagliate sulle corse di taxi nella città di New York.

6.1 TLC Trip Record Data

Il dataset *TLC Trip Record Data* raccoglie, dal 2009, dati dettagliati su ogni corsa di taxi che avviene sull'intero territorio della città di New York. Le corse incluse nel dataset riguardano i *yellow taxi*, i *green taxi* e i *for-hire vehicle*: ai fini di questo caso di studio, sono considerate solo le corse relative ai *yellow taxi*.

Un riassunto dei campi presenti all'interno del dataset è visibile nella Tabella 6.1.

Nel caso di studio è stato analizzato il sotto-dataset relativo al mese di gennaio 2024, che include un totale di 2.824.448 corse. Il file originale, in formato `parquet`, è stato convertito in formato `csv` per facilitarne la gestione nella programmazione in `C`.

6.2 Modulo di *Data Injection*

In precedenza, è stato evidenziato come lo stream processing, a differenza del batch processing, gestisca l'elaborazione di dati dinamici. Perciò, è necessario individuare un modo per costruire, a partire dal dataset statico *TLC Trip Record Data*, uno stream di dati.

A tale scopo è stato realizzato un modulo, nel codice di basso livello, che si occupa di trasformare la sorgente statica di dati, i.e. il file `csv`, nello stream utilizzato dall'ambiente di runtime.

Questo modulo è implementato come un processo logico (`LP`) che legge il file `csv` di input una riga per volta, associando alla tupla corrispondente il timestamp dato dal valore del campo `tpep_pickup_datetime`. In base al valore di questo campo - e al valore del campo nella riga successiva - l'`LP` rischedula la sua esecuzione al "tempo di produzione" della riga successiva.

L'Algoritmo 3 presenta una descrizione informale del comportamento dell'LP che realizza il modulo di *Data Injection*.

6.3 Libreria `PseudoSQL`

Il modulo appena visto è parte della libreria `PseudoSQL` [59], che è stata realizzata per offrire supporto all'esecuzione degli operatori `SQL` nell'ambiente di runtime `ROOT-Sim`.

Questa libreria offre API per eseguire tutte le operazioni `SQL` di base, cioè selezione, proiezione, raggruppamento, ordinamento, join e funzioni aggregate.

Nome Campo	**Descrizione**
`VendorID`	Codice che indica il fornitore TPEP che ha fornito il record. (1 = Creative Mobile Technologies, LLC; 2 = VeriFone Inc.)
`tpep_pickup_datetime`	Data e ora in cui il tassametro è stato attivato.
`tpep_dropoff_datetime`	Data e ora in cui il tassametro è stato disattivato.
`Passenger_count`	Numero di passeggeri nel veicolo. (Valore inserito dal conducente)
`Trip_distance`	Distanza del viaggio in miglia, riportata dal tassametro.
`PULocationID`	Zona TLC Taxi in cui il tassametro è stato attivato.
`DOLocationID`	Zona TLC Taxi in cui il tassametro è stato disattivato.
`RateCodeID`	Codice tariffario finale in vigore alla fine del viaggio. (1 = tariffa standard; 2 = JFK; 3 = Newark; 4 = Nassau o Westchester; 5 = tariffa negoziata; 6 = corsa di gruppo)
`Store_and_fwd_flag`	Indica se il record del viaggio è stato memorizzato nel veicolo prima di essere inviato al fornitore (Y = sì; N = no).
`Payment_type`	Codice numerico che indica come il passeggero ha pagato il viaggio. (1 = carta di credito; 2 = contanti; 3 = senza addebito; 4 = contestazione; 5 = sconosciuto; 6 = viaggio annullato)
`Fare_amount`	Tariffa calcolata in base a tempo e distanza dal tassametro.
`Extra`	Extra e sovrattasse varie, correntemente include solo $0.50 e $1 per le ore di punta o notturne.
`MTA_tax`	Tassa MTA di $0.50, applicata automaticamente in base alla tariffa del tassametro.
`Improvement_surcharge`	Sovrattassa di miglioramento di $0.30 per viaggio, introdotta nel 2015.
`Tip_amount`	Importo della mancia. Viene popolato automaticamente per mance su carta di credito, mentre sono esclusi i contanti.
`Tolls_amount`	Totale dei pedaggi pagati durante il viaggio.
`Total_amount`	Totale addebitato ai passeggeri, escluse le mance in contanti.
`Congestion_Surcharge`	Importo totale per la sovrattassa di congestione dello Stato di New York.
`Airport_fee`	Tassa di $1.25 per il prelievo nei soli aeroporti LaGuardia e John F. Kennedy.

Tabella 6.1: Campi e descrizione dei dati del *TLC Trip Record Data*

Algorithm 3: Modulo di *Data Injection*

Input: `data.csv`: file contenente il dataset, `now`: istante corrente di simulazione

1. Inizializza `rows_list` come lista vuota.
2. Leggi la riga successiva da `data.csv` e assegnala a `current_line`.
3. Imposta `current_time` al valore del campo `tpep_pickup_datetime` di `current_line`.
4. Crea una tupla `t`, a partire da `current_line`, mediante la funzione `PopulateRow()`.
5. Aggiungi `t` a `rows_list`.
6. Leggi la riga successiva e assegna a `next_line`.
7. Se `next_line` è disponibile:
   (a) Imposta `next_time` al valore del campo `tpep_pickup_datetime` di `next_line`.
   (b) Se `next_time` è uguale a `current_time`:
      i. Crea una tupla `t_next`, a partire da `current_line`, mediante la funzione `PopulateRow()`.
      ii. Aggiungi `t_next` a `rows_list`.
      iii. Vai a 2.
   (c) Altrimenti:
      i. Calcola la differenza tra `next_time` e `current_time`.
      ii. Invia tutte le tuple in `rows_list` tramite un messaggio agli LP successivi.
      iii. Rischedula l'esecuzione dello stesso LP al tempo `now + differenza`.

L'astrazione alla base della libreria è il concetto di `Row`, che modella una riga - o tupla - della tabella che viene popolata a partire dallo stream di dati.

```
typedef struct {
    unsigned long timestamp;
    int num_elements;
    RowValue elements[19];
    char table_name[32];
} Row;
```

Listing 6.1: Definizione della struttura Row

Per quanto riguarda la comunicazione tra `LP`, questa avviene attraverso due tipologie di messaggi, `RowsMessage` e `GroupsMessage`, rispettivamente se il payload della comunicazione è composto da una lista di righe, oppure da una lista di gruppi, cioè liste di righe. Le strutture C corrispondenti sono presenti nei Listing 6.2 e 6.3.

```
typedef struct {
    Envelope e;
    Schema schema;
    size_t size;
    Row rows[];
```

```
6 } RowsMessage;
```

Listing 6.2: Definizione della struttura RowsMessage

```
1 typedef struct {
2     Envelope e;
3     Schema schema;
4     size_t size;
5     unsigned char serialized_array[];
6 } GroupsMessage;
```

Listing 6.3: Definizione della stuttura GroupsMessage

La presenza della struttura `Row`, così come delle strutture di supporto alla comunicazione, dimostra come il codice di basso livello non sia completamente agnostico rispetto al dominio di riferimento del DSL di alto livello.

Di conseguenza, neanche la trasformazione M2T può ignorare il dominio di interesse: questa è una delle principali sfide aperte nella realizzazione dello stack software proposto da questa tesi, che ha l'obiettivo di riutilizzare la trasformazione M2T per l'implementazione di DSL differenti.

Tornando alla descrizione della libreria, le principali API offerte sono:

- Funzioni di inizializzazione (Tabella 6.2), invocate in risposta ad un evento di tipo `LP_INIT`, che inizializzano lo stato dell'`LP` con strutture e variabili di supporto alla loro esecuzione.
- Wrapper per gli operatori `SQL` (Tabella 6.3), invocati all'interno dei behavior degli `LP` in risposta ad un evento di tipo `EVENT`, contenenti come parametro lo stato associato agli `LP`.
- Modulo di *Data Injection*, già descritto precedentemente, implementato dalla funzione `void DataIngestion(struct topology *topology, lp_id_t me, simtime_t now, DataSourceData *data, FILE *file, Schema *schema)`.
- Funzioni per la terminazione e per il cleanup (Tabella 6.4), invocate alla ricezione dell'evento `LP_FINI`, oppure in risposta all'evento `TERMINATE`, che comunica la terminazione della simulazione.

- Funzioni per l'invio e la ricezione di messaggi (Tabella 6.5).

Funzione	**Descrizione**
`void DataIngestionInit(lp_id_t me, simtime_t now, FILE **file, char *filename, Schema *schema)`	Inizializza lo stato di un LP di *Data Injection*.
`void WindowInit(struct topology *topology, lp_id_t from, lp_id_t me)`	Inizializza lo stato di un LP *Window*, associandovi la dimensione della finestra e preallocando la lista di Row associata.
`void SelectionInit(struct topology *topology, lp_id_t from, lp_id_t me)`	Inizializza lo stato di un LP di selezione, effettuando il parsing della condizione nella struttura Condition, che viene associata allo stato.
`void ProjectionInit(struct topology *topology, lp_id_t from, lp_id_t me)`	Inizializza lo stato di un LP di proiezione e associandovi la lista degli attributi da selezionare.
`void OrderByInit(struct topology *topology, lp_id_t from, lp_id_t me)`	Inizializza lo stato di un LP di ordinamento associandovi l'attributo di interesse.
`void GroupByInit(struct topology *topology, lp_id_t from, lp_id_t me)`	Inizializza lo stato di un LP di raggruppamento, associandovi l'attributo di interesse.
`void AggregateFunctionInit(struct topology *topology, lp_id_t from, lp_id_t me)`	Inizializza lo stato di un LP per una funzione aggregata, associandovi il nome della colonna di interesse.
`void JoinInit(struct topology *topology, lp_id_t from1, lp_id_t from2, lp_id_t me)`	Inizializza lo stato di un LP di join, preallocando due liste di Row per le due tabelle di interesse.

Tabella 6.2: API di inizializzazione degli LP

Funzione	**Descrizione**
`RowsLinkedList *wSelection(RowsMessage *rcv_msg, void *data)`	Wrapper per l'operatore di selezione, ritorna una lista collegata di `Row` che rispettano la condizione specificata nello stato.
`RowsLinkedList *wProjection(RowsMessage *rcv_msg, void *data)`	Wrapper per l'operatore di proiezione, ritorna una lista collegata di `Row`, mantenendo solo gli attributi presenti nello stato.
`RowsLinkedList *wOrderBy(RowsMessage *rcv_msg, void *data)`	Wrapper per l'operatore di ordinamento, ritorna una lista collegata di `Row`, ordinate secondo il valore dell'attributo specificato nello stato.
`GroupsLinkedList *wGroupBy(RowsMessage *rcv_msg, void *data)`	Wrapper per l'operatore di raggruppamento, ritorna una lista collegata di liste collegate di `Row`, raggruppate per valore dell'attributo specificato nello stato.
`RowsLinkedList *wJoin(RowsMessage *msg, void *data)`	Wrapper per l'operatore di join, ritorna una lista collegata di `Row`, unendo due stream differenti di `Row`.
`RowsLinkedList *ExecuteWindow(RowsMessage *rcv_msg, WindowData *data)`	Wrapper per l'operatore *Window*, ritorna una lista collegata di `Row`, appartenenti alla finestra temporale, oppure `NULL` se la finestra non è terminata.
`void AggregateFunctionRowsInput(RowsMessage *msg, AggregateFunctionData *data, AggregateFunctionType type)`	Wrapper per una funzione aggregata, nel caso di input formato da una lista di `Row`, calcola il valore della funzione aggregata sulla lista di `Row` in input.
`RowsLinkedList *AggregateFunctionGroupedInput(GroupsLinkedList *groups, AggregateFunctionData *data, AggregateFunctionType type, Schema schema)`	Wrapper per una funzione aggregata, nel caso di input formato da una lista di liste di `Row`, calcola il valore della funzione aggregata su ogni di `Row` in input, e crea delle nuove `Row` con i risultati.

Tabella 6.3: Wrapper per gli operatori `SQL`

Funzione	**Descrizione**
`void DataIngestionCleanUp(FILE *file, DataSourceData *data, Schema *schema)`	Effettua il cleanup delle strutture associate allo stato del modulo di *Data Injection* chiamante; va invocato in risposta all'evento `LP_FINI`.
`void WindowCleanUp(WindowData *data)`	Effettua il cleanup delle strutture associate allo stato dell'`LP` *Window* chiamante; va invocato in risposta all'evento `LP_FINI`.
`void SelectionCleanUp(SelectionData *data)`	Effettua il cleanup delle strutture associate allo stato dell'`LP` di selezione chiamante; va invocato in risposta all'evento `LP_FINI`.
`void ProjectionCleanUp(ProjectionData *data)`	Effettua il cleanup delle strutture associate allo stato dell'`LP` di proiezione chiamante; va invocato in risposta all'evento `LP_FINI`.
`void GroupByCleanUp(GroupByData *data)`	Effettua il cleanup delle strutture associate allo stato dell'`LP` di raggruppamento chiamante; va invocato in risposta all'evento `LP_FINI`.
`void AggFunctionCleanUp(AggregateFunctionData *data)`	Effettua il cleanup delle strutture associate allo stato dell'`LP` per funzione aggregata chiamante; va invocato in risposta all'evento `LP_FINI`.
`void OrderByCleanUp(OrderByData *data)`	Effettua il cleanup delle strutture associate allo stato dell'`LP` di ordinamento chiamante; va invocato in risposta all'evento `LP_FINI`.
`void JoinCleanUp(JoinData *data)`	Effettua il cleanup delle strutture associate allo stato dell'`LP` di join chiamante; va invocato in risposta all'evento `LP_FINI`.
`void TerminateWindow(struct topology *topology, WindowData *window_data, lp_id_t me, simtime_t now, Schema schema)`	Effettua il flush della lista di `Row` accumulate nella finestra corrente verso gli `LP` successivi, nella topologia; va invocato in risposta dell'evento `TERMINATE` da ogni `LP` *Window*.
`void ForwardTerminationMessage(struct topology *topology, lp_id_t me, simtime_t now)`	Effettua l'inoltro del messaggio di tipo `TERMINATE` ricevuto, e imposta i flag di terminazione nello stato; va invocato in risposta dell'evento `TERMINATE` da ogni `LP` diverso da quelli *Window*.

Tabella 6.4: API di cleanup e terminazione

Funzione	**Descrizione**
`void CreateAndSendMessageFromList(lp_id_t sender_id, float priority, RowsLinkedList *list, simtime_t now, lp_id_t *receivers, unsigned long num_receivers)`	Consente di creare un messaggio a partire da una lista collegata di `Row` ed inviarlo ad un insieme di `LP`.
`void CreateAndSendMessageFromGroupsList(lp_id_t sender_id, float priority, GroupsLinkedList *list, simtime_t now, lp_id_t *receivers, unsigned long num_receivers)`	Consente di creare un messaggio a partire da una lista collegata di liste collegate di `Row`, ed inviarlo ad un insieme di `LP`; prevede la serializzazione della lista di liste.
`GroupsLinkedList *DeserializeGroupsMessage(GroupsMessage *msg)`	Deserializza un messaggio inviato a partire da una lista collegata di liste collegate di `Row`, restituendo quest'ultima.

Tabella 6.5: API per lo scambio di messaggi

7. Analisi delle prestazioni

In questo capitolo, si procederà all'analisi delle prestazioni dell'applicazione generata automaticamente a partire da una definizione ad alto livello mediante il DSL proposto per il caso di studio. Infatti, un aspetto essenziale per appurare la validità della soluzione proposta riguarda la competitività dell'applicazione generata a partire da un DSL realizzato sfruttando l'approccio delineato in questa tesi rispetto a soluzioni off-the-shelf specificatamente ottimizzate per supportare applicazioni di stream processing. Un'analisi della facilità di utilizzo del DSL proposto o della semplicità di realizzazione di un DSL utilizzando il framework architetturale presentato in questa tesi sono certamente interessanti, ma per motivi temporali sono demandate a lavori futuri.

Si inizierà definendo un insieme di query sullo stream tramite il linguaggio *QueryLanguage*, per generare, attraverso la catena di traduzioni, codice `C` eseguibile nell'ambiente `ROOT-Sim`. Le misurazioni di performance verranno ottenute dall'esecuzione di questo codice, utilizzando come baseline di confronto misurazioni analoghe, estratte dall'esecuzione delle stesse query all'interno del framework Apache Spark.

7.1 Progettazione degli esperimenti

L'insieme di query scelte, per la generazione del codice su cui effettuare la valutazione delle prestazioni del sistema, è raffigurato nel Listing 7.1.

Sebbene non fosse nell'interesse dello studio definire query significative, è stato ritenuto opportuno adottare un insieme di query che simulassero dei

```
1  CREATE VIEW AirportRides AS
2  SELECT *
3  FROM Taxis
4  WHERE Airport_fee != 0.0
5  WINDOW(TUMBLING, 30 MINUTES)
6
7  SELECT payment_type, AVG(total_amount)
8  FROM AirportRides
9  GROUP BY payment_type
10
11 SELECT PULocationID, SUM(passenger_count)
12 FROM Taxis
13 GROUP BY PULocationID
14 ORDER BY passenger_count
15 WINDOW (TUMBLING , 30 MINUTES)
16
17 CREATE VIEW MadisonSquareGardenRides AS
18 SELECT *
19 FROM Taxis
20 WHERE ( PULocationID == 161 ) OR ( DOLocationID == 161 )
21 WINDOW (TUMBLING , 30 MINUTES)
22
23 SELECT PULocationID, AVG(congestion_surcharge)
24 FROM MadisonSquareGardenRides
25 GROUP BY PULocationID
26 ORDER BY congestion_surcharge
27
28 SELECT PULocationID, SUM(trip_distance)
29 FROM Taxis
30 GROUP BY PULocationID
31 ORDER BY trip_distance
32 WINDOW (TUMBLING , 30 MINUTES)
33
34 CREATE VIEW HighTipsRides AS
35 SELECT *
36 FROM Taxis
37 WHERE tip_amount > 10.0
38 WINDOW (TUMBLING , 30 MINUTES)
39
40 SELECT PULocationID, DOLocationID
41 FROM HighTipsRides
42
43 CREATE VIEW GroupRides AS
44 SELECT *
45 FROM Taxis
46 WHERE passenger_count > 4
47 WINDOW (TUMBLING , 30 MINUTES)
48
49 SELECT *
50 FROM Taxis
51 WHERE ( payment_type == 1 ) AND ( total_amount > 100.0 )
52 WINDOW (TUMBLING , 30 MINUTES)
53
54
```

Listing 7.1: Query utilizzate negli esperimenti

Caratteristica	Valore
Architettura CPU	x86_64
Modello CPU	11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz
Core	4
Thread per Core	2
Numero Totale di CPU	8
Frequenza Max CPU	4.2 GHz
Memoria RAM Totale	16 GiB

Tabella 7.1: Caratteristiche hardware della macchina utilizzata negli esperimenti

casi reali di interrogazione dello stream. Per questo motivo, alcune query si riferiscono a luoghi cruciali della città di New York, come gli aeroporti e luoghi di aggregazione (e.g., Madison Square Garden), mentre altre ad aspetti di potenziale interesse, ad esempio la distanza media percorsa raggruppata per luogo di partenza, oppure l'analisi delle corse di gruppo e di quelle che presentano mance elevate.

La topologia generata dalle query appena introdotte, nel modello ad attori intermedio, è raffigurata in Figura 7.1.

Il codice prodotto dalla trasformazione M2M viene infine compilato, attraverso la trasformazione M2T, in codice `C`. L'esecuzione di questo codice avviene su un calcolatore CPU-only general-purpose, le cui caratteristiche sono riassunte nella Tabella 7.1.

Allo stesso tempo, è stata realizzata una soluzione analoga per l'esecuzione dello stesso insieme di query sulla stessa macchina, utilizzando il framework Apache Spark, un potente motore di elaborazione dati open-source progettato per il trattamento rapido e distribuito di grandi volumi di dati, che fornisce la possibilità di eseguire query SQL su dati in streaming.

Ai fini delle misurazioni, l'esecuzione di Spark avviene all'interno di un *Docker* container. Questo approccio permette di raccogliere dati accurati sul consumo di memoria medio e di picco per ciascuna esecuzione all'interno del container, utilizzando le *Docker Engine API*.

Le metriche di interesse per lo studio delle prestazioni del sistema sono le seguenti:

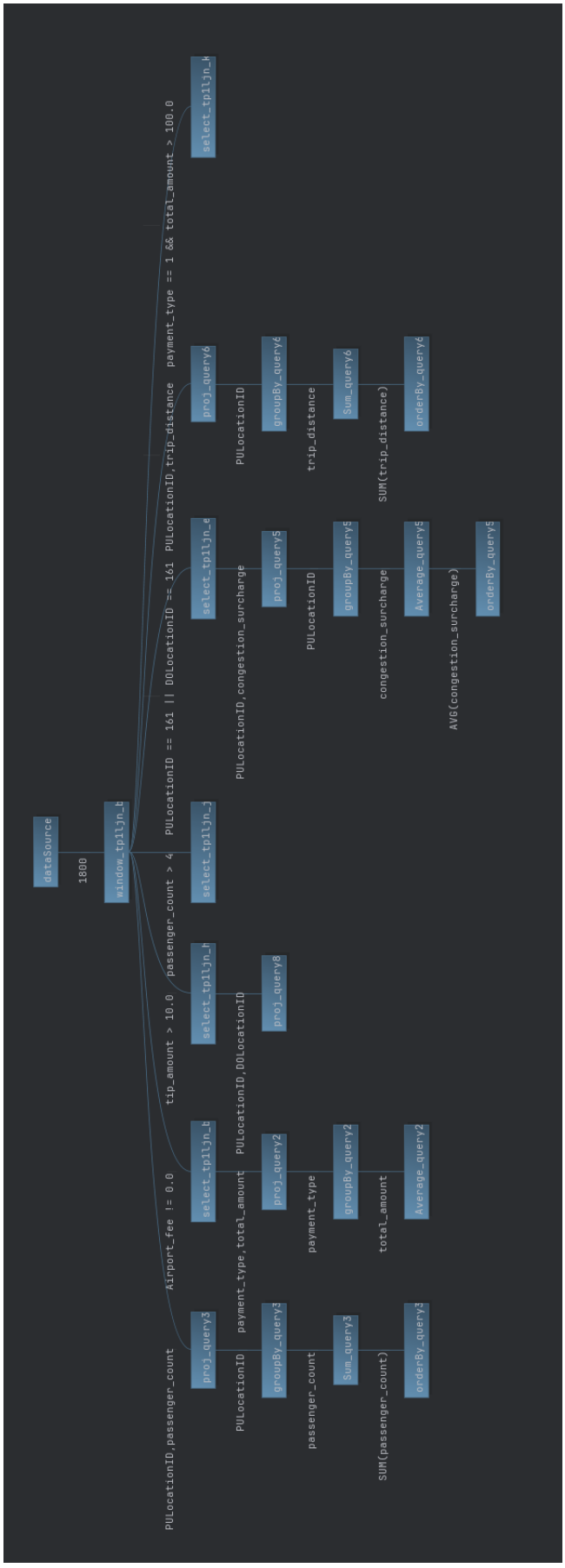


Figura 7.1: Topologia di attori generata dalle query in Figura ??

- Speedup rispetto all'esecuzione seriale su ROOT-Sim: esso è definito, per $c = \{1, 2, ..., 8\}$, come $T_{p,c}/T_s$, dove $T_{p,c}$ indica il tempo di esecuzione in parallelo, su c core, mentre T_s il tempo di esecuzione seriale sul runtime ROOT-Sim;
- Tempo di esecuzione delle query;
- Utilizzo medio della memoria RAM;
- Picco di utilizzo della memoria RAM.

Tutte queste metriche vengono analizzate in funzione del numero di *core* assegnati all'esecuzione, mentre le ultime tre vengono valutate anche in corrispondenza di variazioni della dimensione della finestra. Il numero di core varia da 1 a 8, mentre l'insieme di dimensioni della finestra è il seguente:

$$\{1 \text{ minuto}, 30 \text{ minuti}, 1 \text{ ora}, 2 \text{ ore}, 3 \text{ ore}, 6 \text{ ore}, 12 \text{ ore}, 18 \text{ ore}, 24 \text{ ore}\}$$

Nel caso di variazione della dimensione della finestra, il numero di core è stato fissato a 4, pari al numero di core fisici del calcolatore di riferimento.

7.2 Risultati e discussione

Nei risultati, il sistema descritto in questa tesi è identificato con il nome ActorStream. Ogni configurazione è stata eseguita 10 volte per ridurre l'influenza di eventuali outlier e ottenere risultati più stabili e rappresentativi.

7.2.1 Tempi di esecuzione al variare del numero di core

Il grafico che descrive l'andamento dei tempi di esecuzione al variare del numero di core, in entrambi i sistemi, è presente nella Figura 7.2.

In dettaglio, ActorStream presenta tempi migliori rispetto a Spark in corrispondenza dell'utilizzo dei core fisici, mentre con l'introduzione dell'*Hyper-Threading* le prestazioni di ActorStream tendono a peggiorare, mentre Spark si stabilizza.

Per *Hyper-Threading* si intende una tecnologia, offerta dai processori Intel, che prevede l'esposizione, da parte della CPU, di due contesti di esecuzione per ogni core fisico, trasformando ogni unità di esecuzione in due core logici, che consentono il processamento concorrente di due thread software differenti. In generale, sfruttando i tempi di inattività in cui il core avrebbe altrimenti atteso il completamento di altri processi, questa tecnologia aumenta la capacità di elaborazione della CPU, consentendo l'esecuzione di un secondo flusso logico.

Nel caso specifico di ActorStream, le performance introdotte dall'*Hyper-Threading* sono inferiori rispetto a quelle che si raggiungono eseguendo soltanto su core fisici.

Per evidenziare ulteriormente questo punto, è stato analizzato il tempo di processamento delle query associando all'esecuzione soltanto due core, in un caso solamente core fisici, mentre nel secondo caso un core fisico ed uno virtuale. Dai risultati, è stato possibile osservare un aumento di circa il 50% del tempo di esecuzione nel secondo caso, rispetto al primo. Un comportamento analogo si verifica paragonando una configurazione con 4 core fisici ad una con tre core fisici ed uno virtuale. Ciò evidenzia come la scalabilità sia limitata dall'utilizzo dell'*Hyper-Threading*.

Le ragioni per cui si presenta questo fenomeno non sono immediate. In generale, esso è imputabile ad effetti secondari sulle dinamiche *off-core*, come ad esempio sull'architettura di cache. Un'analisi approfondita sia del sistema che della piattaforma di riferimento, allo scopo di verificare le cause dietro questo fenomeno, sono rimandate a lavori successivi.

7.2.2 Consumo della memoria al variare del numero di core

Dal grafico in Figura 7.3, si può notare come i consumi di memoria RAM, sia medi che di picco, siano stabili in entrambi i sistemi. Nel caso specifico della configurazione di query adottata, che prevede una finestra da 1800 secondi, ActorStream offre prestazioni migliori, in termini di consumo di memoria.

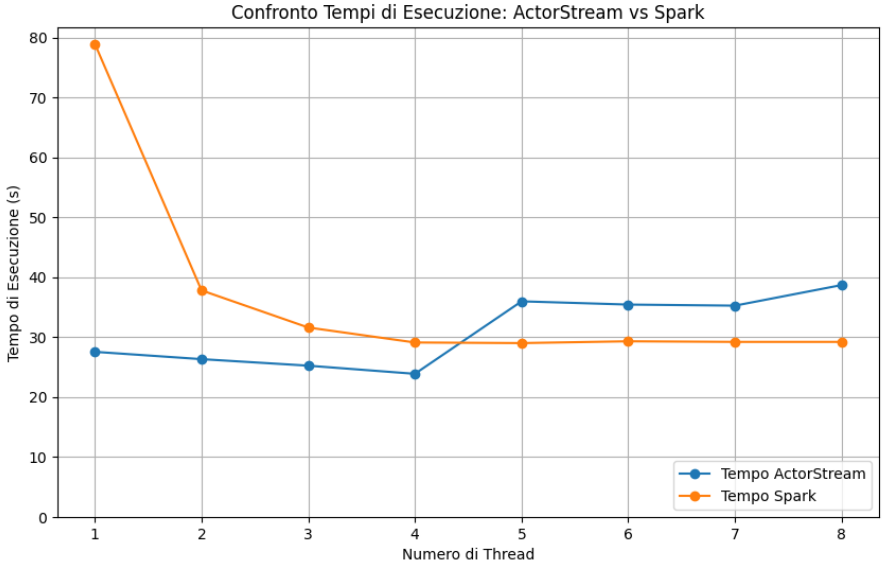


Figura 7.2: Tempi di esecuzione al variare del numero di core

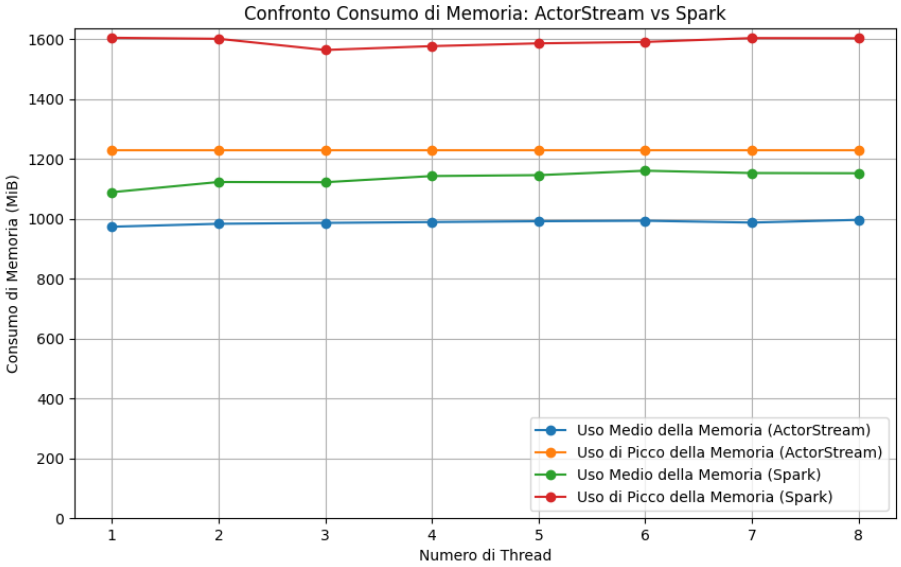


Figura 7.3: Consumo di memoria medio e di picco al variare del numero di core

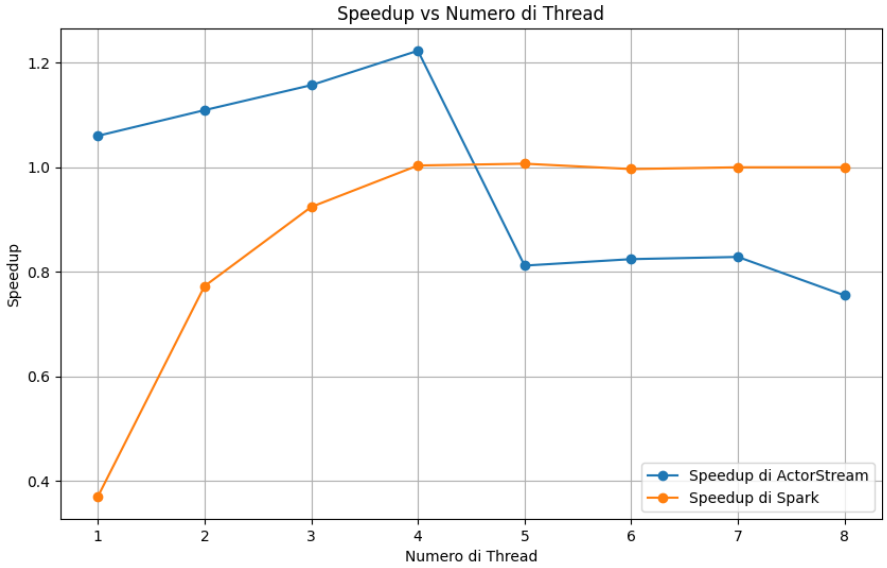


Figura 7.4: Speedup al variare del numero di core

7.2.3 Speedup al variare del numero di core

La Figura 7.4 mostra l'andamento dello speedup in funzione del numero di core. Il tempo di esecuzione seriale T_s è pari a 29.2 secondi. Dal grafico, si può notare ActorStream presenti uno speedup maggiore di 1 solamente nelle esecuzioni su core fisici, per gli stessi motivi descritti precedentemente.

D'altra parte, Spark non riesce ad ottenere, in questo caso, prestazioni migliori rispetto al caso seriale.

7.2.4 Tempi di esecuzione al variare della dimensione della finestra

In Figura 7.5 sono rappresentati i tempi di esecuzione ottenuti da entrambi i sistemi variando la dimensione della finestra.

In generale, ActorStream offre prestazioni migliori rispetto a Spark e, in entrambi i casi, i tempi tendono ad essere stabili anche variando la dimensione della finestra. Il picco presente nei tempi di Spark, relativamente alla finestra di dimensione pari a un minuto, deriva dalla gestione interna delle finestre prevista da Spark, che porta il tempo di esecuzione ad assumere, per finestre

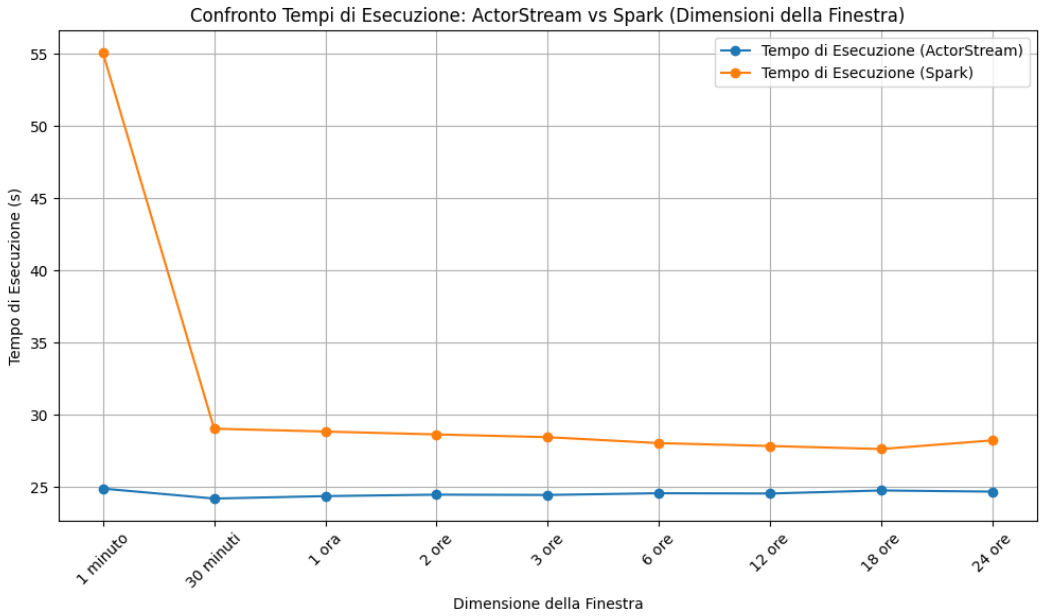


Figura 7.5: Tempi di esecuzione al variare della dimensione della finestra

che vanno da un minuto a 10 minuti, un andamento come quello mostrato in Figura 7.6.

7.2.5 Utilizzo della memoria al variare della dimensione della finestra

I risultati, mostrati in Figura 7.7, dimostrano come l'utilizzo della memoria relativo a Spark, sia medio che di picco, siano stabili ed indipendenti dalla dimensione della finestra. Ciò non vale nel caso di ActorStream, che presenta un trend crescente in entrambe le metriche di utilizzo della memoria.

Per comprendere meglio le differenze nell'utilizzo della memoria tra i due sistemi, è necessario introdurre alcuni concetti relativi al modo in cui Spark gestisce i flussi di dati. Ciò permetterà di confrontare il suo funzionamento con quello di ActorStream in modo più approfondito.

Il modello di *Structured Streaming* offerto da Spark prevede l'astrazione dello stream di dati come una tabella illimitata, detta *Input Table*, a cui viene aggiunta una riga per ogni nuovo elemento ricevuto dallo stream. Una query su questa tabella, dunque una query sullo stream, si traduce nella generazione

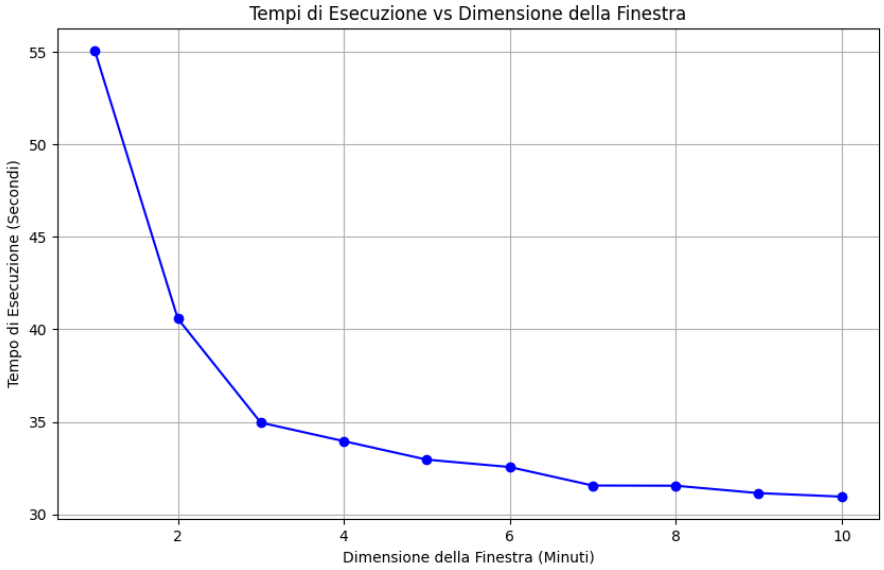


Figura 7.6: Tempi di esecuzione al variare della dimensione della finestra in Spark

di una *Result Table*: l'aggiunta di una o più righe alla *Input Table* innesca l'aggiornamento di una o più *Result Table*, in base ad eventuali query interessate ai dati ricevuti.

L'aspetto fondamentale dello *Structured Streaming*, che differenzia Spark da ActorStream, riguarda la materializzazione dell'intera tabella di input: nel caso di Spark, i dati sono processati in maniera incrementale: la lettura di un nuovo dato dallo stream porta, come già detto, all'aggiornamento della *Result Table*; in corrispondenza di questo aggiornamento, il dato ricevuto viene subito scartato. Nel caso specifico di un'aggregazione, la ricezione di un nuovo dato comporta soltanto l'aggiornamento di uno *stato intermedio*, evitando la necessità di mantenere tutte le tuple ricevute all'interno della *Input Table*.

La gestione delle finestre, in Spark, è vista come un caso speciale di aggregazione: le finestre temporali sono dei gruppi ed ogni riga, quindi i dati associati ad essa, appartiene ad uno di questi gruppi. Ciò consente di effettuare aggregazioni, anche basate su finestre temporali, in maniera incrementale.

In ActorStream, d'altra parte, ogni finestra di dati è processata in blocco: il modulo di *Data Injection* inietta dati nella topologia; gli attori - o `LP` -

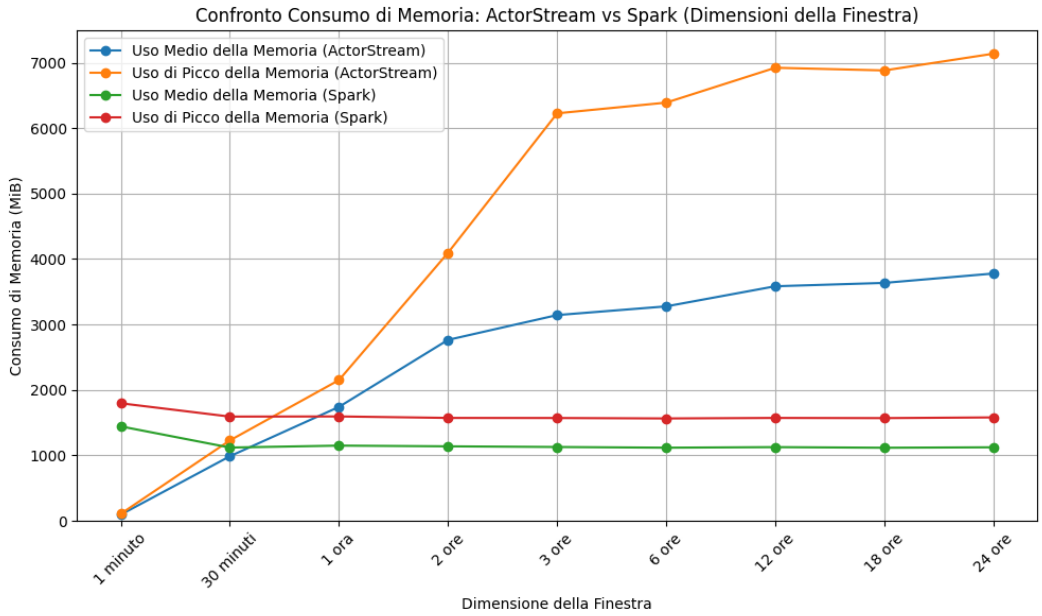


Figura 7.7: Consumo di memoria medio e di picco al variare della dimensione della finestra

che hanno la responsabilità di definire delle finestre di dati si limitano ad accumulare tutti i dati che hanno timestamp compatibile ai limiti della finestra corrente; quando il limite temporale associato alla finestra scade, tutte le righe accumulate sono inviate in blocco agli attori/LP successivi; in base alla natura di questi, essi possono effettuare aggregazioni, quindi ridurre la dimensione della lista di righe, oppure altre operazioni che non diminuiscono la cardinalità di questa, inviando agli attori/LP successivi il risultato.

Di conseguenza, è evidente come il consumo medio di memoria, nel caso di ActorStream, debba essere direttamente proporzionale alla dimensione della finestra, in quanto questo determina la dimensione dello "stato" intermedio. Al contrario, lo "stato" intermedio di Spark non dipende dalla dimensione di eventuali finestre associate alle query.

8. Conclusioni

In questa tesi, è stata descritta la progettazione e la realizzazione di uno stack software composto da un DSL di alto livello, *QueryLanguage*, costruito per definire query in stile `SQL` su uno stream di dati; una rappresentazione intermedia, costituita da un DSL basato sull'Actor Model, cioè *ActorLanguage*; una libreria per l'esecuzione dei principali operatori `SQL` sull'ambiente di esecuzione ROOT-Sim.

La compilazione del DSL di alto livello verso il codice `C` ha richiesto l'implementazione di una trasformazione Model-to-Model da *QueryLanguage* ad *ActorLanguage*, seguita da una trasformazione Model-to-Text da *ActorLanguage* verso il codice di basso livello.

Il confronto tra la soluzione proposta in questo lavoro e un framework che rappresenta lo stato dell'arte nel processamento di grandi volumi di dati, come Apache Spark, evidenzia un netto vantaggio prestazionale della prima in termini di tempi di elaborazione, mentre il secondo dimostra un impatto sulla memoria più scalabile in relazione alla dimensione della finestra temporale associata alle query.

Ciò dimostra il raggiungimento dell'obiettivo primario della tesi, cioè quello di dimostrare l'efficacia dell'approccio Model-Driven Engineering per la costruzione di Domain-Specific Language ottimizzati per un'architettura di runtime specifica.

8.1 Sviluppi futuri

In seguito ai risultati ottenuti nel presente lavoro, è possibile delineare una serie di sviluppi futuri con lo scopo di ampliare e potenziare ulteriormente le capacità del sistema proposto e, inoltre, proseguire e consolidare la linea di ricerca già aperta che pone al centro l'approccio Model-Driven Engineering per la semplificazione dello sviluppo software.

La ricerca ha messo in evidenza non solo le prestazioni competitive rispetto a framework affermati, come Apache Spark, ma anche opportunità significative per ottimizzare ulteriormente le risorse e migliorare l'efficienza del processamento di grandi volumi di dati.

Un primo passo in questa direzione consiste nell'ottimizzare l'utilizzo della memoria, il che contribuirebbe certamente a una maggiore scalabilità. Ad esempio, l'adozione di una gestione differente dei dati raccolti in finestre temporali, che rappresenta uno dei principali limiti del sistema, potrebbe costituire una valida soluzione in questa direzione.

Successivamente, il principale contributo futuro per l'adozione su larga scala dell'approccio Model-Driven Engineering nello sviluppo software concerne la realizzazione di una trasformazione Model-to-Text completamente svincolata dagli aspetti specifici del dominio dei DSL sovrastanti. Questo aspetto non rappresenta soltanto una difficoltà concettuale affrontata in questa ricerca, ma una delle principali sfide aperte nell'ambito del Model-Driven Engineering.

La creazione di una trasformazione M2T generale e riutilizzabile, in grado di tradurre diversi DSL in codice di basso livello, ridurrebbe significativamente le attività necessarie per lo sviluppo di applicazioni arbitrarie. In tal modo, il focus sarebbe posto unicamente sulla definizione del DSL e sulla trasformazione M2M verso il modello intermedio basato sugli attori.

Un ulteriore passo in avanti, rispetto al presente lavoro, consiste nell'adozione di architetture hardware più complesse, eventualmente eterogenee. L'adozione di un'architettura eterogenea implica la realizzazione di differenti copie di artefatti software eseguibili sui vari componenti hardware, richiedendo di

conseguenza altrettante trasformazioni M2T. Ulteriormente, il processamento su una piattaforma eterogenea richiede dei meccanismi di schedulazione del flusso di esecuzione sui differenti componenti hardware. Questi obiettivi sono stati considerati fuori dalla portata di questo lavoro, pur consistendo in direzioni di ricerca fondamentali.

9. Ringraziamenti

Ringrazio mamma e papà, per il loro costante sostegno.

Ringrazio la mia famiglia e i miei amici, per il loro interesse e incoraggiamento.

Ringrazio i miei amici e colleghi universitari, per aver condiviso con me e reso speciale questo percorso.

Ringrazio il mio relatore, prof. Alessandro Pellegrini, per la sua infinita dedizione, competenza e disponibilità, e i miei correlatori, dott. Romolo Marotta e prof. Guglielmo de Angelis, per il loro supporto e la loro conoscenza. Il vostro sostegno è stato fondamentale.

Elenco delle figure

List of listings

List of Algorithm

Bibliografia

[1] Robert A van Engelen. 2001. Atmol: A domain-specific language for atmospheric modeling. *Journal of computing and information technology* 9, 4 (2001), 289–303.

[2] Anas Abouzahra, Jean Bézivin, Marcos Didonet Del Fabro, and Frédéric Jouault. 2005. A practical approach to bridging domain specific languages with UML profiles. In *Proceedings of the Best Practices for Model Driven Software Development at OOPSLA*, Vol. 5. 2005.

[3] Luca Aceto, Wan Fokkink, and Chris Verhoef. 2001. Structural operational semantics. In *Handbook of process algebra*. Elsevier, 197–292.

[4] Gul Agha. 1986. An overview of actor languages. *ACM Sigplan Notices* 21, 10 (1986), 58–67.

[5] Gul A Agha, Ian A Mason, Scott F Smith, and Carolyn L Talcott. 1997. A foundation for actor computation. *Journal of functional programming* 7, 1 (1997), 1–72.

[6] Martin S Alnæs, Anders Logg, Kristian B Ølgaard, Marie E Rognes, and Garth N Wells. 2014. Unified form language: A domain-specific language for weak formulations of partial differential equations. *ACM Transactions on Mathematical Software (TOMS)* 40, 2 (2014), 1–37.

[7] Hugo Andrade, Lucy Ellen Lwakatare, Ivica Crnkovic, and Jan Bosch. 2019. Software challenges in heterogeneous computing: A multiple case study in industry. In *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 148–155.

[8] Ankica Barišić, Vasco Amaral, Miguel Goulao, and Bruno Barroca. 2011. Quality in use of domain-specific languages: a case study. In *Proceedings of the 3rd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools*. 65–72.

[9] José A Barriga, Pedro J Clemente, Encarna Sosa-Sánchez, and Álvaro E Prieto. 2021. SimulateIoT: Domain Specific Language to design, code generation and execute IoT simulation environments. *IEEE Access* 9 (2021), 92531–92552.

[10] Maicon Bernardino, Avelino F Zorzo, and Elder M Rodrigues. 2016. Canopus: A domain-specific language for modeling performance testing. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 157–167.

[11] Lesia Bilitchenko, Adam Liu, Sherine Cheung, Emma Weeding, Bing Xia, Mariana Leguia, J Christopher Anderson, and Douglas Densmore. 2011. Eugene–a domain specific language for specifying and constraining synthetic biological parts, devices, and systems. *PloS one* 6, 4 (2011), e18882.

[12] Jonas Bonér. 2010. Introducing Akka - Simpler Scalability, Fault-Tolerance, Concurrency & Remoting through Actors. `https://jonasboner.com/introducing-akka/` Accessed: 2024-10-17.

[13] Antonio Bucchiarone, Jordi Cabot, Richard F Paige, and Alfonso Pierantonio. 2020. Grand challenges in model-driven engineering: an analysis of the state of the research. *Software and Systems Modeling* 19 (2020), 5–13.

[14] Hyungsuk Choi, Woohyuk Choi, Tran Minh Quan, David GC Hildebrand, Hanspeter Pfister, and Won-Ki Jeong. 2014. Vivaldi: A domain-specific language for volume processing and visualization on distributed heterogeneous systems. *IEEE transactions on visualization and computer graphics* 20, 12 (2014), 2407–2416.

[15] Biagio Cosenza, Nikita Popov, Ben Juurlink, Paul Richmond, Mozhgan Kabiri Chimeh, Carmine Spagnuolo, Gennaro Cordasco, and Vittorio Scarano. 2018. OpenABL: a domain-specific language for parallel and distributed agent-based

simulations. In *Euro-Par 2018: Parallel Processing: 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27-31, 2018, Proceedings 24*. Springer, 505–518.

[16] Alberto Rodrigues Da Silva. 2015. Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures* 43 (2015), 139–155.

[17] Joeri De Koster, Tom Van Cutsem, and Wolfgang De Meuter. 2016. 43 years of actors: a taxonomy of actor models and their key properties. In *Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*. 31–40.

[18] Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, et al. 2011. Liszt: a domain specific language for building portable mesh-based PDE solvers. In *Proceedings of 2011 international conference for high performance computing, networking, storage and analysis*. 1–12.

[19] Jack Dongarra, Pete Beckman, Terry Moore, Patrick Aerts, Giovanni Aloisio, Jean-Claude Andre, David Barkai, Jean-Yves Berthou, Taisuke Boku, Bertrand Braunschweig, et al. 2011. The international exascale software project roadmap. *The international journal of high performance computing applications* 25, 1 (2011), 3–60.

[20] H Carter Edwards and Christian R Trott. 2013. Kokkos: Enabling performance portability across manycore architectures. In *2013 Extreme Scaling Workshop (xsw 2013)*. IEEE, 18–24.

[21] Sven Efftinge and Markus Völter. 2006. oAW xText: A framework for textual DSLs. In *Workshop on Modeling Symposium at Eclipse Summit*, Vol. 32.

[22] Roland Ewald and Adelinde M Uhrmacher. 2014. SESSL: A domain-specific language for simulation experiments. *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 24, 2 (2014), 1–25.

[23] Martin Fowler. 2010. *Domain-specific languages*. Pearson Education.

[24] Cristian González García, Jordán Pascual Espada, Edward Rolando Núñez Valdez, and Vicente García Díaz. 2014. Midgar: Domain-specific language to generate smart objects for an internet of things platform. In *2014 Eighth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*. IEEE, 352–357.

[25] Sébastien Gérard, Cédric Dumoulin, Patrick Tessier, and Bran Selic. 2007. 19 Papyrus: A UML2 tool for domain-specific language modeling. In *Dagstuhl workshop on model-based engineering of embedded real-time systems*. Springer, 361–368.

[26] Joan Giner-Miguelez, Abel Gómez, and Jordi Cabot. 2023. A domain-specific language for describing machine learning datasets. *Journal of Computer Languages* 76 (2023), 101209.

[27] Irene Greif. 1975. *Semantics of communicating parallel processes.* Ph. D. Dissertation. Massachusetts Institute of Technology.

[28] Philipp Marian Grulich. 2023. *Query Compilation for Modern Data Processing Environments*. Technische Universitaet Berlin (Germany).

[29] Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence. In *Advance papers of the conference*, Vol. 3. Stanford Research Institute Menlo Park, CA, 235.

[30] The Khronos Group Inc. 2020. SYCL™ 2020 Specification (revision 9). `https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html#acknowledgements` Accessed: 2024-10-17.

[31] Aníbal Iung, João Carbonell, Luciano Marchezan, Elder Rodrigues, Maicon Bernardino, Fabio Paulo Basso, and Bruno Medeiros. 2020. Systematic mapping study on domain-specific language development tools. *Empirical Software Engineering* 25 (2020), 4205–4249.

[32] Namit Jain, Shailendra Mishra, Anand Srinivasan, Johannes Gehrke, Jennifer Widom, Hari Balakrishnan, Uğur Çetintemel, Mitch Cherniack, Richard Tibbetts, and Stan Zdonik. 2008. Towards a streaming SQL standard. *Proceedings*

of the VLDB Endowment International Conference on Very Large Data Bases 1, 2 (Aug. 2008), 1379–1390. https://doi.org/10.14778/1454159.1454179

[33] Gabor Karsai, Holger Krahn, Claas Pinkernell, Bernhard Rumpe, Martin Schindler, and Steven Völkel. 2014. Design guidelines for domain specific languages. *arXiv preprint arXiv:1409.2378* (2014).

[34] Lennart CL Kats and Eelco Visser. 2010. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*. 444–463.

[35] Awais Khan, Hyogi Sim, Sudharshan S Vazhkudai, Ali R Butt, and Youngjae Kim. 2021. An analysis of system balance and architectural trends based on top500 supercomputers. In *The International Conference on High Performance Computing in Asia-Pacific Region*. 11–22.

[36] Barbara A Kitchenham, David Budgen, and O Pearl Brereton. 2011. Using mapping studies as the basis for further research–a participant-observer case study. *Information and Software Technology* 53, 6 (2011), 638–651.

[37] Tomaž Kosar, Sašo Gaberc, Jeffrey C Carver, and Marjan Mernik. 2018. Program comprehension of domain-specific and general-purpose languages: replication of a family of experiments using integrated development environments. *Empirical Software Engineering* 23 (2018), 2734–2763.

[38] Tomaž Kosar, Marjan Mernik, and Jeffrey C Carver. 2012. Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments. *Empirical software engineering* 17 (2012), 276–304.

[39] Tomaž Kosar, Nuno Oliveira, Marjan Mernik, Varanda João Maria Pereira, Matej Črepinšek, Cruz Daniela Da, and Rangel Pedro Henriques. 2010. Comparing general-purpose and domain-specific languages: An empirical study. *Computer Science and Information Systems* 7, 2 (2010), 247–264.

[40] Dean Kramer, Tony Clark, and Samia Oussena. 2010. MobDSL: A Domain Specific Language for multiple mobile platform deployment. In *2010 IEEE Interna-*

tional Conference on Networked Embedded Systems for Enterprise Applications. IEEE, 1–7.

[41] Filip Křikava, Philippe Collet, and Robert B France. 2014. Actress: Domain-specific modeling of self-adaptive software architectures. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. 391–398.

[42] Suhas Kumar. 2015. Fundamental limits to Moore's law. *arXiv preprint arXiv:1511.05956* (2015).

[43] Edward A Lee. 2011. Heterogeneous actor modeling. In *Proceedings of the ninth ACM international conference on Embedded software*. 3–12.

[44] Franklin Leong, Babak Rahmani, Demetri Psaltis, Christophe Moser, and Diego Ghezzi. 2024. An actor-model framework for visual sensory encoding. *Nature Communications* 15, 1 (2024), 808.

[45] Ricky T Lindeman, Lennart CL Kats, and Eelco Visser. 2011. Declaratively defining domain-specific language debuggers. *ACM SIGPLAN Notices* 47, 3 (2011), 127–136.

[46] Mathias Louboutin, Michael Lange, Fabio Luporini, Navjot Kukreja, Philipp A Witte, Felix J Herrmann, Paulius Velesko, and Gerard J Gorman. 2019. Devito (v3. 1.0): an embedded domain-specific language for finite differences and geophysical exploration. *Geoscientific Model Development* 12, 3 (2019), 1165–1187.

[47] Kanak Mahadik, Christopher Wright, Jinyi Zhang, Milind Kulkarni, Saurabh Bagchi, and Somali Chaterji. 2016. Sarvavid: a domain specific language for developing scalable computational genomics applications. In *Proceedings of the 2016 international conference on supercomputing*. 1–12.

[48] Richard Membarth, Oliver Reiche, Frank Hannig, Jürgen Teich, Mario Körner, and Wieland Eckert. 2015. Hipa cc: A domain-specific language and compiler for image processing. *IEEE Transactions on Parallel and Distributed Systems* 27, 1 (2015), 210–224.

[49] Marjan Mernik, Jan Heering, and Anthony M Sloane. 2005. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)* 37, 4 (2005), 316–344.

[50] Parastoo Mohagheghi, Wasif Gilani, Alin Stefanescu, and Miguel A Fernandez. 2013. An empirical study of the state of the practice and acceptance of model-driven engineering in four industrial cases. *Empirical software engineering* 18, 1 (2013), 89–116.

[51] Gordon E Moore et al. 2006. Moore's law at 40.

[52] Aaftab Munshi. 2009. The opencl specification. In *2009 IEEE Hot Chips 21 Symposium (HCS)*. IEEE, 1–314.

[53] Thomas Nelson. 2022. Introducing Microsoft Orleans. In *Introducing Microsoft Orleans: Implementing Cloud-Native Services with a Virtual Actor Framework*. Springer, 17–27.

[54] NYC TLC Trip Record Data 2024. `https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page` Accessed: 2024-10-17.

[55] Vaclav Pech, Alex Shatalin, and Markus Voelter. 2013. JetBrains MPS as a tool for extending Java. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. 165–168.

[56] Alessandro Pellegrini and et al. 2024. ROOT-Sim Topology. `https://github.com/ROOT-Sim/topology`. Accessed: 2024-10-10.

[57] Alessandro Pellegrini, Roberto Vitali, and Francesco Quaglia. 2012. The rome optimistic simulator: Core internals and programming model. In *4th International ICST Conference on Simulation Tools and Techniques*.

[58] James R Powell. 2008. The quantum limit to Moore's law. *Proc. IEEE* 96, 8 (2008), 1247–1248.

[59] PseudoSQL C Library for ROOT-Sim 2024. `https://github.com/simoneb00/Actors/tree/master/pseudosql` Accessed: 2024-10-17.

[60] II Ptolemy. 2013. Ptolemy II.

[61] Kyle L Spafford and Jeffrey S Vetter. 2012. Aspen: A domain specific language for performance modeling. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–11.

[62] Diomidis Spinellis. 2001. Notable design patterns for domain-specific languages. *Journal of systems and software* 56, 1 (2001), 91–99.

[63] Satish Narayana Srirama and Deepika Vemuri. 2023. CANTO: An actor model-based distributed fog framework supporting neural networks training in IoT applications. *Computer Communications* 199 (2023), 1–9.

[64] Herbert Stachowiak. 1973. Allgemeine modelltheorie. *(No Title)* (1973).

[65] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. 2008. *EMF: eclipse modeling framework*. Pearson Education.

[66] Mark Strembeck and Uwe Zdun. 2009. An approach for the systematic development of domain-specific languages. *Software: Practice and Experience* 39, 15 (2009), 1253–1292.

[67] Arvind Sujeeth, HyoukJoong Lee, Kevin Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand Atreya, Martin Odersky, and Kunle Olukotun. 2011. OptiML: an implicitly parallel domain-specific language for machine learning. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*. 609–616.

[68] Juha-Pekka Tolvanen and Matti Rossi. 2003. Metaedit+ defining and using domain-specific modeling languages and code generators. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 92–93.

[69] Jonas Traub, Philipp M Grulich, Alejandro Rodríguez Cuéllar, Sebastian Breß, Asterios Katsifodimos, Tilmann Rabl, and Volker Markl. 2019. Efficient Window Aggregation with General Stream Slicing.. In *EDBT*, Vol. 19. 97–108.

[70] Vladimir Viyović, Mirjam Maksimović, and Branko Perisić. 2014. Sirius: A rapid development of DSM graphical editor. In *IEEE 18th International Conference on Intelligent Engineering Systems INES 2014*. IEEE, 233–238.

[71] Markus Voelter, Janet Siegmund, Thorsten Berger, and Bernd Kolb. 2014. Towards user-friendly projectional editors. In *International Conference on Software Language Engineering*. Springer, 41–61.

[72] Markus Völter, Thomas Stahl, Jorn Bettin, Arno Haase, and Simon Helsen. 2013. *Model-driven software development: technology, engineering, management*. John Wiley & Sons.

[73] Jon Whittle, John Hutchinson, and Mark Rouncefield. 2013. The state of practice in model-driven engineering. *IEEE software* 31, 3 (2013), 79–85.

[74] Jiajian Xiao, Philipp Andelfinger, Wentong Cai, Paul Richmond, Alois Knoll, and David Eckhoff. 2020. OpenABLext: An automatic code generation framework for agent-based simulations on CPU-GPU-FPGA heterogeneous platforms. *Concurrency and Computation: Practice and Experience* 32, 21 (2020), e5807.