ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

Progettazione e sviluppo di un prototipo di simulatore ad eventi discreti reattivo

Tesi di laurea in:
Laboratorio di Sistemi Software

Relatore
Prof. Danilo Pianini

Candidato
Giacomo Accursi

Correlatore
Dott. Gianluca Aguzzi

IV Sessione di Laurea
Anno Accademico 2022-2023

Sommario

Le simulazioni rappresentano una delle metodologie più importanti per l'analisi, la comprensione e la previsione del comportamento di sistemi complessi presenti in una vasta gamma di discipline scientifiche, ingegneristiche, socio-economiche e naturali, consentendo di esplorare le dinamiche di sistemi reali o teorici. Tra le diverse metodologie, la simulazione ad eventi discreti occupa un ruolo fondamentale, offrendo un potente strumento per modellare ed analizzare sistemi dinamici che evolvono al verificarsi di eventi.

La programmazione reattiva, d'altro canto, sta guadagnando sempre più popolarità nell'industria del software grazie alla sua capacità di affrontare le sfide poste dalle applicazioni moderne, concentrandosi sulla gestione efficiente di flussi di dati asincroni.

In un simulatore ad eventi discreti, la costruzione e l'aggiornamento del grafo delle dipendenze fra eventi rappresentano attività critiche, soggette a potenziali problemi di performance in presenza di un numero elevato di nodi. Utilizzando la programmazione reattiva è possibile definire implicitamente le dipendenze fra eventi. Ciascun evento osserva gli eventi di interesse e reagisce in maniera asincrona alla loro occorrenza, eliminando la necessità del grafo.

La tesi ha come obiettivo la progettazione e l'implementazione di un prototipo di simulatore ad eventi discreti reattivo basato sul simulatore Alchemist, ovvero un simulatore ad eventi discreti che utilizzi tecniche di programmazione reattiva per eliminare la necessità di un grafo per la gestione delle dipendenze fra eventi, attualmente presente in Alchemist.

Alla mia famiglia.

Ringraziamenti

Il mio percorso universitario giunge ora alla sua conclusione dopo 5 anni intensi, contraddistinti da momenti difficili ma anche tanti altri ricchi di gioie e soddisfazioni. Credo sia doveroso ringraziare chi in questi anni mi è stato vicino e mi ha accompagnato in questo percorso.

Parto con il ringraziare Sonia, Danilo e Diego, la mia famiglia. Sono state le persone che più mi hanno supportato, ma anche sopportato. Nessuno come loro ha dovuto affrontare i miei sbalzi di umore. Apprezzo immensamente ogni loro sacrificio.

È doveroso ringraziare il prof. Danilo Pianini e il dott. Gianluca Aguzzi, i quali mi hanno accompagnato in questo progetto di tesi. Grazie per la professionalità e la disponibilità che avete dimostrato nei miei confronti.

Grazie ai miei migliori amici, Mattia Me., Mattia Mi., Augusto, Nicolò, Mirko, Gianmarco, Filippo C., Filippo B., Simone, Vittorio, Brando e Elia. Il bene che vi voglio è indescrivibile.

Grazie ad Andrea A., il mio compagno di progetti, la persona con cui ho condiviso ogni singolo aspetto di questo percorso. Per me non sei più un semplice collega, ma un amico.

Grazie ad Andrea G. e a Davide, due persone splendide che mi hanno accompagnato in questo percorso. Ho imparato tanto da voi.

Ci tengo a ringraziare anche Carlotta. Anche se le nostre strade si sono divise da un po' di tempo, mi sei sempre stata vicino, mi hai sopportato e mi hai fatto diventare una persona migliore.

Grazie infine a Sarah. Sebbene tu sia entrata a far parte della mia vita solo recentemente, mi hai spronato a non mollare e mi hai sollevato il morale nei momenti difficili.

Indice

Elenco delle figure

List of Listings

Capitolo 1

Introduzione

1.1 Contesto e motivazioni

Le simulazioni computerizzate (o semplicemente *simulazioni*) rappresentano un pilastro fondamentale nell'ambito della ricerca scientifica, dell'ingegneria, della progettazione e dell'apprendimento, permettendo di esplorare e comprendere fenomeni complessi, simulare scenari reali e testare ipotesi in un ambiente controllato e riproducibile. Paul Humphreys [Hum90] definisce le simulazioni come "qualsiasi metodo implementato al computer per esplorare le proprietà di modelli matematici dove i metodi analitici non sono disponibili". Le simulazioni si riferiscono alla creazione di modelli digitali che riproducono il comportamento di sistemi reali o immaginari nel tempo. Questi modelli possono variare dalla semplice rappresentazione grafica alla simulazione di processi dinamici complessi. Un esempio di sistema che è possibile simulare è il sistema climatico terrestre, il quale è estremamente dinamico e complesso e influenzato da una vasta gamma di fattori, tra cui temperatura, umidità, correnti oceaniche, venti e attività solare. Le simulazioni sono diventate indispensabili in diverse aree, incluse la modellazione dei sistemi naturali, economici e sociali.

Ad alto livello, è possibile considerare una simulazione come un metodo per lo studio di sistemi. In questa accezione più ampia del termine, ci si riferisce all'intero processo, il quale include la scelta di un modello, la ricerca di un modo per implementare il modello in una forma che possa essere eseguita su un computer,

il calcolo dell'output dell'algoritmo e la visualizzazione e lo studio dei dati risultati. Secondo Winsberg [Win03]:

> "Gli studi di simulazione di successo fanno più che calcolare numeri. Fanno uso di una varietà di tecniche per trarre inferenze da questi numeri. Le simulazioni fanno uso creativo di tecniche di calcolo che possono essere motivate solo extra-matematicamente ed extra-teoricamente. Come tale, a differenza delle semplici computazioni che possono essere eseguite su un computer, i risultati delle simulazioni non sono automaticamente affidabili. Molto impegno e competenza vanno nel decidere quali risultati delle simulazioni sono affidabili e quali no."

La definizione sopracitata interpreta la simulazione come l'uso di un computer per risolvere o approssimare la soluzione delle equazioni matematiche di un modello che si intende rappresentare, sia esso reale o ipotetico.

Un altro approccio è cercare di definire una simulazione come indipendente dalla nozione di simulazione al computer e poi definire "simulazione al computer" come una simulazione eseguita da un computer digitale programmato. Utilizzando questo approccio, una simulazione è qualsiasi sistema che si crede abbia un comportamento dinamico sufficientemente simile ad un altro sistema tale che il primo possa essere studiato per apprendere il secondo. Questo è in linea con la definizione data da Hartmann [Har96], secondo il quale "una simulazione imita un processo con un altro processo", dove in questo caso il termine processo si riferisce unicamente a qualche oggetto o sistema il cui stato cambia nel tempo. Questa definizione verrà poi corretta da Huges [Hug99], il quale obiettò che la definizione di Hartmann escludeva simulazioni che imitano la struttura di un sistema piuttosto che le sue dinamiche. Humphreys ha revisionato [Hum04] la sua definizione per accordarsi con le osservazioni di Hartmann e Hughes come segue:

> "Il sistema S fornisce una simulazione di base di un oggetto o processo B nel caso in cui S sia un dispositivo computazionale concreto che produce, attraverso un processo temporale, soluzioni a un modello computazionale che rappresenta correttamente B, sia dinamicamente che staticamente. Se inoltre il modello computazionale utilizzato da

S rappresenta correttamente la struttura del sistema reale R, allora S fornisce una simulazione di base del sistema R rispetto a B."

1.2 Obiettivi della tesi

Negli ultimi vent'anni è stato compiuto un lavoro significativo sul tema dei metodi e degli approcci per ottimizzare i modelli di simulazione a eventi discreti. Allora, come oggi, una delle sfide più grandi nell'ottimizzazione delle simulazioni a eventi discreti è l'incapacità di identificare precisamente la soluzione ottimale per un determinato modello di sistema. Questo è particolarmente vero quando lo spazio delle soluzioni possibili si espande. Negli ultimi vent'anni la velocità di calcolo è aumentata, i costi di calcolo e modellazione sono diminuiti e si sono verificati sviluppi teorici nel campo dell'ottimizzazione della simulazione [Ril13]. Gli approcci di ottimizzazione algoritmica si sono evoluti nel tempo man mano che la simulazione è diventata più diffusa. Le tecniche di ottimizzazione coinvolgono numerose valutazioni dinamiche dello spazio delle soluzioni multidimensionale di una simulazione nella ricerca di una soluzione ottimale. Il lavoro nell'ambito dell'ottimizzazione della simulazione si è concentrato principalmente sulla progettazione e valutazione di vari approcci algoritmici ed euristici nella ricerca dello spazio delle soluzioni di essa.

Per quanto riguarda i simulatori ad eventi discreti, uno degli aspetti critici, quando si parla di performance, è la gestione e l'aggiornamento delle dipendenze fra eventi duranti la simulazione. Il simulatore Alchemist utilizza attualmente un grafo per gestire le dipendenze fra eventi. La creazione e l'aggiornamento del grafo possono portare a problemi di performance in presenza di un elevato numero di nodi. Mediante l'uso della programmazione reattiva è possibile eliminare la necessità di un grafo, descrivendo le dipendenze come relazione osservante/osservato, in cui ciascun evento osserva gli eventi di interesse e reagisce in maniera asincrona alla loro occorrenza. Un evento che non dipende più da un altro evento ne termina semplicemente l'osservazione. L'obiettivo della tesi è lo studio delle simulazioni ad eventi discreti e dei principi della programmazione reattiva, aventi come fine la progettazione e lo sviluppo di un simulatore ad eventi discreti che utilizzi le

tecniche di programmazione reattiva per eliminare il grafo delle dipendenze fra eventi utilizzato attualmente nel simulatore Alchemist.

1.3 Struttura della tesi

Nel capitolo 2 si intende effettuare una panoramica sulle simulazioni ad eventi discreti, introducendone il funzionamento e spiegando le differenze fra simulazioni ad eventi discrete e continue. Si intende inoltre fornire un confronto fra simulazioni ad eventi discrete event-driven e time-driven. Al termine del capitolo verranno descritte le caratteristiche principali del simulatore Alchemist e le peculiarità sul suo funzionamento. Il capitolo 3 mira a descrivere la filosofia della programmazione reattiva, analizzandone vantaggi e criticità, astraendo però dalla loro implementazione. Il capitolo 4 descrive l'implementazione reattiva nel linguaggio Kotlin, parlando di come sono implementate le Kotlin Coroutines e i Kotlin Flow. Nel capitolo 5 si potrà trovare la progettazione e l'implementazione del prototipo di simulatore sviluppato.

Capitolo 2

Simulazioni ad Eventi Discreti

Un **sistema ad eventi discreti** è un sistema a stati discreti guidato da eventi, il cui il progresso del tempo è determinato dall'occorrenza di eventi grazie ai quali il sistema evolve. Nel contesto di una simulazione ad eventi discreti, un **evento** rappresenta un cambiamento nello stato del sistema, mentre lo **stato** riguarda la rappresentazione del sistema, in termini di proprietà, variabili e relazioni fra le entità, in un dato istante di tempo. Gli esempi di eventi possono includere l'arrivo di un messaggio in un sistema di comunicazione, il completamento di un processo, o qualsiasi altro avvento che influenzi il comportamento del sistema. La simulazione avanza di passi discreti, con ogni passo che corrisponde all'accadimento di un nuovo evento. Questo approccio è spesso utilizzato per modellare e analizzare sistemi complessi, come reti di computer, catene di approvvigionamento, code di attesa o altri scenari in cui gli eventi guidano il comportamento del sistema.

2.1 Terminologia e componenti

I concetti alla base di un sistema ad eventi discreti sono i seguenti:

- **Sistema**: collezione di entità che interagiscono fra loro nel tempo per raggiungere determinati obiettivi.
- **Modello**: una rappresentazione astratta del sistema, spesso contenente relazioni strutturali, logiche o matematiche che descrivono il sistema stesso.

- **Stato del sistema**: una collezione di variabili che contengono tutte le informazioni necessarie per descrivere il sistema in ogni momento.
- **Entità**: un oggetto nel sistema che richiede una rappresentazione esplicita nel modello. Le entità possono essere create in fase di inizializzazione della simulazione o durante essa.
- **Attributi**: sono le proprietà di una determinata entità.
- **Evento**: una occorrenza istantanea che cambia lo stato del sistema.
- **Future Event List (FEL)**: contiene tutti gli eventi futuri ordinati cronologicamente in base al tempo di accadimento. Deve essere implementata in modo che gestisca in maniera efficiente l'aggiunta, la rimozione e la ricerca di eventi.
- **Clock**: una variabile rappresentante il tempo simulato. In generale non esiste nessuna correlazione fra il tempo simulato e il tempo necessario per eseguire la simulazione.

Una simulazione ad eventi discreti procede acquisendo una sequenza di *snapshot*, i quali rappresentano l'evoluzione del sistema nel tempo. Uno snapshot ad un determinato istante t, include non solo lo stato del sistema al tempo t, ma anche la lista degli eventi futuri e lo stato di tutte le entità del sistema.

2.2 Algoritmo Event-scheduling Time-advance

La dinamicità intrinseca delle simulazioni ad eventi discreti impone la necessità di gestire attentamente il flusso temporale attraverso il monitoraggio costante del clock. Un meccanismo fondamentale per garantire l'evoluzione accurata della simulazione, rispettando l'ordine cronologico degli eventi, è rappresentato dall'algoritmo "*event-scheduling time-advance*", basato sull'utilizzo della future event list.

L'algoritmo in questione, il cui flusso di controllo è illustrato in Figura 2.1, è suddiviso in tre sezioni chiave: *inizializzazione*, *ciclo di elaborazione degli eventi* e *fase di output*. La simulazione inizia al tempo 0, quando il *main program* dà

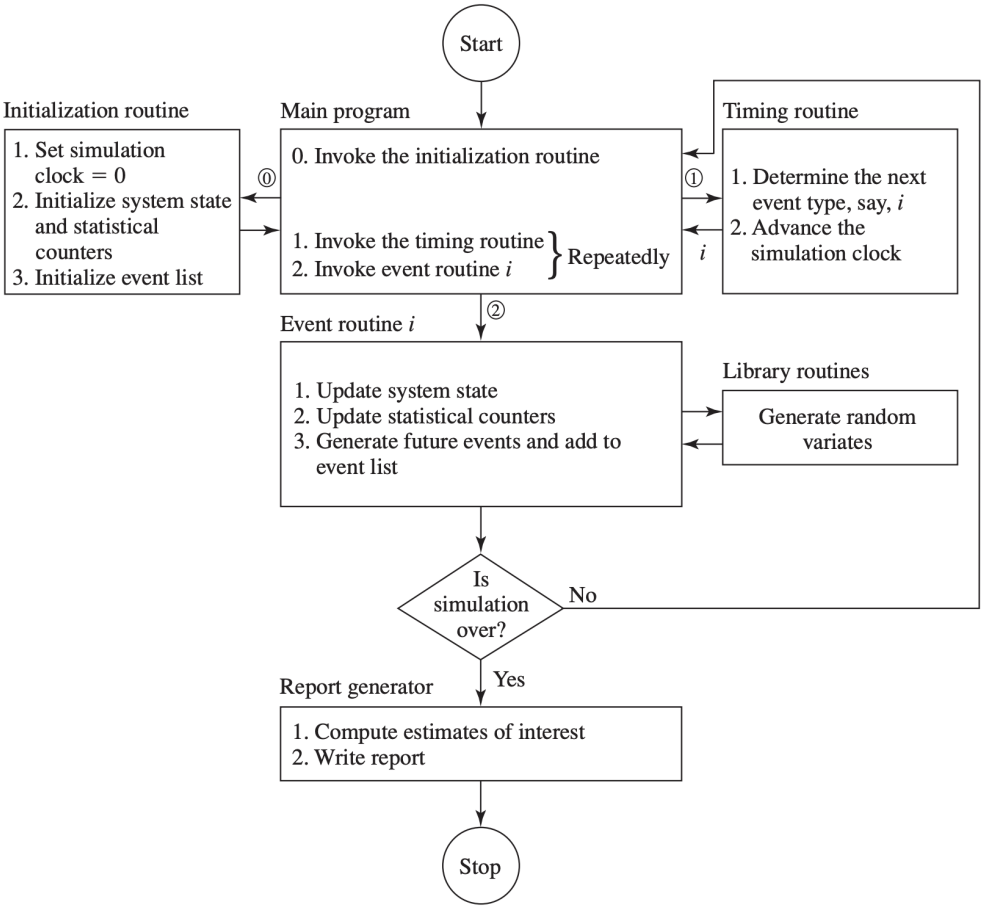

Figura 2.1: Flusso di controllo dell'algoritmo event-scheduling time-advance [Law15].

avvio alla simulazione invocando la *initialization routine*, nella quale il clock viene settato a 0 e le entità e le variabili di stato sono inizializzate. Il controllo torna poi al main program, il quale invoca una *timing routine* per identificare il prossimo evento da eseguire. La lista degli eventi futuri contiene tutti gli eventi ordinati per tempo di accadimento: $FEL = [t_1, t_2, \dots t_n], t_1 \leq t_2 \leq \cdots \leq t_n$. Il tempo t è il valore corrente del clock, mentre l'evento associato al tempo t_1 è chiamato evento imminente, ovvero il prossimo evento che dovrà verificarsi [BINN10]. Il controllo torna nuovamente al *main program*, il quale invoca l'*event routine* grazie al quale l'evento imminente viene eseguito effettivamente. L'esecuzione dell'evento imminente, seguita dalla generazione di uno snapshot correlato, comporta l'avanzamento del clock al successivo istante temporale. L'evento imminente è rimosso quindi dalla FEL ed eseguito, generando un nuovo snapshot al tempo t_1 basato sulla natura dell'evento e lo snapshot precedente. In questa fase nuovi eventi futuri possono essere generati e aggiunti opportunamente alla FEL. Il processo si ripete ciclicamente fino alla conclusione della simulazione. Durante la fase di output vengono invece elaborate e registrate le statistiche di interesse.

2.2.1 Gestione Future Event List

Una FEL, come anticipato, è la struttura dati che contiene l'elenco degli eventi programmati per verificarsi in futuro. Tradizionalmente l'elenco è ordinato in base al tempo di esecuzione al quale l'evento è stato schedulato, ma questo non è un requisito. La gestione efficiente della FEL è di fondamentale importanza, alcuni modelli di simulazione impiegano più tempo CPU nella gestione della lista piuttosto che qualsiasi altro aspetto, come per esempio la generazione di numeri casuali, l'elaborazione di eventi, l'esecuzione di operazioni aritmetiche varie ecc...

Nella gestione degli eventi che compongono la FEL, ci sono due operazioni critiche: l'operazione di inserimento (*enqueue*) e l'operazione di cancellazione (*dequeue*). Un'operazione di cancellazione viene eseguita per elaborare l'evento o perché un evento, precedentemente pianificato, deve essere annullato per qualche motivo. L'inserimento e la cancellazione possono verificarsi in una posizione precisa nella lista degli eventi, oppure potrebbe essere necessario avviare una ricerca per determinare la giusta posizione. Importante è anche l'operazione di modifica,

in cui una ricerca di evento esistente presente in lista è seguita da una modifica di qualche aspetto dell'evento stesso, come ad esempio la modifica del tempo di accadimento al quale è schedulato.

Esistono tre criteri principali per determinare l'efficacia della struttura dati e dell'algoritmo per la gestione della FEL:

- **velocità**: la struttura dati e l'algoritmo per inserire ed eliminare un evento devono essere ottimizzate per minimizzare il tempo di esecuzione. Per ottenere tempi di esecuzione rapidi è fondamentale una ricerca degli eventi efficiente. Un algoritmo efficiente tende a valutare il minor numero di eventi possibile per l'inserimento o la cancellazione;

- **robustezza**: è necessaria una gestione efficace degli errori per affrontare scenari imprevisti. Occorre una validazione dei dati durante l'inserimento per garantire l'integrità della struttura dati;

- **adattabilità**: la FEL dovrebbe consentire la configurazione di parametri chiave per adattarsi a diverse modalità di gestione degli eventi o politiche temporali. Dovrebbe inoltre essere progettata in modo che nuove funzionalità o tipi di eventi possano essere facilmente integrati senza modificare drasticamente il codice esistente.

2.3 Partenza e terminazione

Nella Sezione 2.2 è stato descritto come la simulazione avanza durante la sua esecuzione. Tipicamente per far partire una simulazione, in fase di inizializzazione viene generato un evento specifico che viene poi inserito nella lista di eventi futuri. La terminazione può invece essere raggiunta tramite diversi meccanismi:

- tempo di simulazione massimo: al tempo 0 viene schedulato un evento di stop ad un tempo futuro T_e. In questo caso si è certi che la simulazione verrà eseguita nell'intervallo $[0, T_e]$;

- eventi speciali di terminazione: il tempo di esecuzione T_e è determinato dalla simulazione stessa. In genere T_e dipende dal numero di occorrenze di un de-

terminato evento. Per esempio, potrebbe essere il tempo del completamento del centesimo servizio presso un determinato centro di assistenza;

- terminazione quando la FEL è vuota.

Nel secondo e nel terzo caso il tempo per eseguire la simulazione non è conosciuto a priori e può variare da una simulazione all'altra.

Le esecuzioni di una simulazione possono essere classificate in **transient** e **steady state**. La selezione del tipo di simulazione è particolarmente importante e la decisione deve essere presa in base all'output che si intende analizzare. Con il termine transient ci si riferisce ad una simulazione che termina, per causa di un evento speciale o grazie ad un tempo prefissato. Una simulazione steady state, al contrario, non termina mai. L'obiettivo in questo caso è studiare il comportamento a lungo termine del sistema.

2.4 Distribuzioni di probabilità

Il successo di una simulazione richiede un approccio completo che va oltre la semplice creazione di un diagramma di flusso del sistema in esame, la sua traduzione in un "programma" per computer e la replicazione di alcune configurazioni proposte del sistema. L'uso della probabilità e della statistica è un aspetto fondamentale di uno studio di simulazione, tanto che ogni team di modellazione dovrebbe includere almeno un esperto con una solida formazione in queste tecniche [Law15]. Per effettuare una simulazione, utilizzando input casuali come i tempi di arrivo o le dimensioni della domanda, occorre specificare le loro distribuzioni di probabilità. La scelta delle corrette distribuzioni determina in maniera importante l'accuratezza del modello.

Nelle simulazioni ad eventi discreti è comune modellare il tempo trascorso tra gli arrivi successivi degli eventi utilizzando distribuzioni di probabilità. Ad esempio, nel caso di una coda di attesa in un sistema di servizio, i tempi tra gli arrivi dei clienti possono essere modellati utilizzando una **distribuzione esponenziale** o di **Poisson**. Queste distribuzioni consentono di rappresentare realisticamente il processo di arrivo degli eventi e di stimare la frequenza con cui si verificano. I tempi

di servizio potrebbero essere invece costanti o probabilistici. Se i tempi sono completamente casuali, potrebbe essere necessario dover utilizzare una **distribuzione esponenziale**. Potrebbe anche accadere che i tempi di servizio siano costanti, ma una variabilità casuale causi fluttuazioni in modo negativo o positivo. Ad esempio, il tempo impiegato da un tornio per attraversare un albero di 10 centimetri dovrebbe essere costante. Tuttavia, il materiale potrebbe presentare lievi differenze di durezza o lo strumento potrebbe usurarsi. Qualsiasi evento potrebbe causare tempi di lavorazione diversi. In questi casi la **distribuzione normale** potrebbe descrivere il tempo di servizio [BINN10]. Oltre ai tempi di arrivo e di servizio, le simulazioni possono coinvolgere altre variabili casuali che influenzeranno il comportamento del sistema: i ritardi tra l'arrivo di un evento e la sua elaborazione possono essere modellati utilizzando distribuzioni di probabilità per rappresentare il tempo impiegato per completare determinate attività o processi.

2.5 Simulazioni ad eventi discreti vs continue

Ogni modello di simulazione è la specifica di un sistema fisico (o una parte di esso) in termini di un insieme di stati ed eventi. Effettuare una simulazione significa imitare l'occorrenza degli eventi che evolvono nel tempo e riconoscerne gli effetti, rappresentati come stati [FT01]. Gli eventi futuri indotti dagli stati devono essere pianificati. Come è possibile notare in Figura 2.2, in una simulazione continua i cambiamenti di stato avvengono continuamente nel tempo, mentre in una simulazione discreta il verificarsi di un evento è istantaneo e limitato a un punto selezionato nel tempo. Il concetto di tempo continuo modella il tempo come un continuum di punti temporali successivi. Questo implica che i dati forniti per un certo arco di tempo sono specificati come una funzione continua nel tempo. In genere si assume che questa funzione sia anche differenziabile, in modo che i cambiamenti della variabile di stato nel tempo possano essere modellate come la derivata prima della variabile stessa. Gli intervalli di tempo non svolgono un ruolo specifico come principio organizzativo. Naturalmente si può considerare qualsiasi intervallo di tempo, ma non intervalli di tempo predefiniti di lunghezza finita che strutturano l'intero modello [OM08].

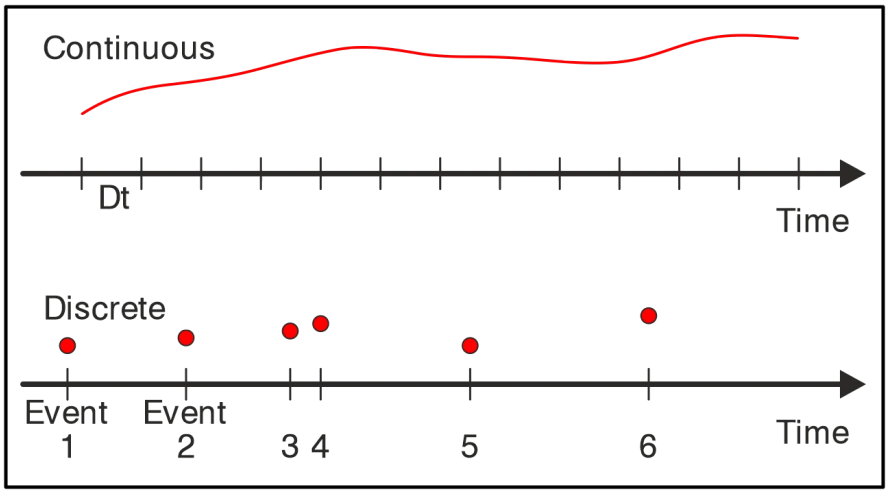

Figura 2.2: Cambiamento di stato nel tempo in simulazioni continue vs simulazioni ad eventi discreti [Hel08].

Ad esempio, il livello dell'acqua in un serbatoio (la quale è una variabile di stato) può cambiare in modo continuo (ovvero con incrementi e decrementi infinitamente piccoli) e in qualsiasi punto del tempo continuo. Spesso questi modelli sono descritti da equazioni differenziali che specificano i tassi di variazione degli stati nel tempo. Le simulazioni a tempo continuo non sono veramente continue nel senso matematico del termine, poiché per ragioni computazionali è impossibile rappresentare il tempo in modo continuo su un computer. Ci sono diverse ragioni per cui le simulazioni a tempo continuo non possono essere veramente continue:

- limitazioni computazionali: i computer hanno una quantità finita di memoria e risorse di calcolo. Per simulare un processo continuo, il tempo deve essere discretizzato in intervalli finiti. Questo porta a una rappresentazione approssimata del tempo continuo, con una risoluzione determinata dalla precisione temporale della simulazione;

- errori numerici: anche con un elevato numero di punti discreti nel tempo, l'approssimazione del tempo continuo introduce errori numerici. Questi errori possono accumularsi nel tempo e influenzare l'accuratezza complessiva della simulazione;

- algoritmi di integrazione numerica: nelle simulazioni a tempo continuo, le equazioni differenziali che descrivono il sistema devono essere risolte numericamente utilizzando algoritmi di integrazione numerica, come ad esempio il metodo di Runge-Kutta. Questi algoritmi discretizzano implicitamente il tempo per calcolare l'evoluzione del sistema nel tempo.

In generale, poiché la simulazione in tempo continuo tiene traccia dello stato del sistema in modo continuo, è più granulare e in certe situazioni (soprattutto nelle scienze naturali) più accurata. D'altra parte, la simulazione in tempo continuo può richiedere più risorse computazionali, perché il tracciamento continuo degli stati del sistema può essere costoso.

Un'altra differenza tra le simulazioni a tempo continuo e quelle ad eventi discreti è la loro adattabilità a problemi sia continui che discreti [ÖB09]. Più precisamente:

- la simulazione in tempo continuo può essere utilizzata per modellare processi continui e discreti. Sebbene i processi discreti non siano continui per natura, i modelli in tempo continuo possono modellarli con successo e generare risultati validi. Ad esempio, può succedere che fenomeni come il flusso del traffico di veicoli, siano approssimati da un modello a tempo continuo considerando il traffico come un fluido. Tuttavia, un modello in tempo continuo richiederebbe più tempo e calcoli per lo stesso problema discreto. Inoltre, per alcuni sistemi discreti, l'output di un modello di simulazione in tempo continuo potrebbe non avere senso;

- i modelli discreti potrebbero, in teoria, essere applicati anche ai sistemi a tempo continuo. Tuttavia, un modello a eventi discreti ne fornirebbe una visione meno raffinata, a causa della sua natura.

In conclusione, è preferibile utilizzare un modello ad eventi discreti quando:

- il sistema da modellare è caratterizzato da eventi distinti che influenzano il comportamento complessivo, come code di attesa, processi di produzione e reti di computer.

- il sistema mostra comportamenti che possono essere modellati efficacemente attraverso una serie di eventi distinti e continui;

- le risorse computazionali sono limitate o gli eventi significativi sono rari nel tempo.

Mentre è da preferire la simulazione continua quando:

- il sistema è meglio descritto da equazioni differenziali o processi che si evolvono in modo continuo nel tempo, come sistemi meccanici o processi chimici;
- è necessaria una rappresentazione accurata dei cambiamenti nel tempo, specialmente quando i dettagli più sottili del comportamento sono cruciali per l'analisi;
- si dispone di risorse computazionali sufficienti per gestire la complessità richiesta.

2.6 Simulazioni Time-Driven vs Event-Driven

Esistono due tipi di simulazione discreta che possono essere distinti per quanto riguarda le politiche di avanzamento nel tempo all'interno della simulazione. In una simulazione discreta *time-driven* il tempo simulato avanza in passaggi temporali (o *ticks*) di dimensioni constanti Δ, ovvero, l'osservazione del sistema dinamico simulato è discretizzata in intervalli di tempo unitari. La scelta di Δ determina la precisione della simulazione e il tempo di simulazione trascorso: i *ticks* abbastanza brevi da garantire la precisione richiesta generalmente implicano un tempo di simulazione più lungo. Intuitivamente, per strutture event-driven irregolarmente distribuite nel tempo, l'approccio *time-driven* genera algoritmi di simulazione inefficienti.

La simulazione discreta con approccio *event-driven* discretizza l'osservazione del sistema simulato negli istanti in cui si verificano gli eventi. Una simulazione ad eventi discreti, quando eseguita sequenzialmente, elabora ripetutamente l'occorrenza degli eventi nel tempo simulato, mantenendo una lista di eventi ordinata per tempo che tiene traccia degli eventi temporizzati programmati per accadere in futuro, un orologio globale che indica il tempo corrente e la variabili di stato $S = (s_1, s_2, \ldots, s_n)$, che definiscono lo stato attuale del sistema. Come già descritto nella Capitolo 2, un motore di simulazione guida la simulazione prendendo

continuamente il primo evento dalla lista degli eventi futuri, simulando l'effetto dell'evento, cambiando le variabili di stato e eventualmente rimuovendo anche gli eventi obsoleti. Questo viene eseguito fino a quando viene raggiunto un tempo di fine predefinito, o non ci sono più eventi che devono verificarsi.

L'approccio *event-driven* si concentra sulla relazione causale tra gli eventi esterni e le azioni eseguite da un sistema, ovvero sul **perché** accade qualcosa, mentre il modello *time-driven* si concentra sul tempismo delle azioni, ovvero sul **quando** accade qualcosa. Nel primo caso, le azioni vengono eseguite **il prima possibile** all'arrivo degli eventi, e il tempo può essere gestito mediante segnali temporali trattati come eventi esterni asincroni. Al contrario, nel secondo caso le azioni vengono eseguite **al momento giusto** secondo un programma, e gli eventi esterni possono essere gestiti mediante meccanismi di polling [TD95]. La scelta tra i due modelli dipende dal dominio di applicazione. Nei sistemi complessi, i due modelli dovrebbero coesistere per soddisfare requisiti contrastanti. Di conseguenza, un linguaggio di programmazione dovrebbe consentire al progettista di scegliere e mescolare astrazioni per catturare nel modo più espressivo possibile i concetti chiave legati a un problema specifico o a un sotto-problema.

2.7 Simulazione parallela

La **simulazione parallela a eventi discreti** (PDES) riguarda l'esecuzione di un singolo programma di simulazione a eventi discreti su un computer sfruttando il calcolo parallelo [Fuj90]. Distribuendo l'esecuzione di una simulazione su più processori, si cerca di ridurre notevolmente il tempo di esecuzione del modello, fino a un fattore pari al numero di processori. Questo concetto è valido in teoria, in realtà la **legge di Amdahl** stabilisce che il miglioramento della velocità di esecuzione di un programma parallelizzabile su un sistema multi-core è limitato dalla frazione sequenziale del programma [Amd67]. In altra parole, anche se una parte del programma può essere eseguita in modo parallelo, ci sarà sempre una parte sequenziale che non può essere parallelizzata. La legge di Amdahl può essere

espressa con la seguente formula:

$$S_{max} = \frac{1}{(1-P) + \frac{P}{N}}$$

Dove:

- S_{max} è il miglioramento massimo della velocità di esecuzione ottenibile parallelizzando il programma.
- P è la frazione di codice che può essere parallelizzata.
- N è il numero di core disponibili.

Se si intende simulare un modello militare di grandi dimensioni o una rete di comunicazione contenente migliaia di nodi, il tempo di esecuzione potrebbe essere eccessivo e si potrebbe prendere in considerazione la simulazione parallela. Un altro uso della simulazione parallela potrebbe essere il processo decisionale in tempo reale. Ad esempio, in un sistema di controllo del traffico aereo, potrebbe risultare utile simulare diverse ore di traffico aereo per decidere come reindirizzarlo al meglio [Wie98].

Lo sviluppo di una simulazione parallela richiede la scomposizione del modello in **processi logici** (LP). I singoli LP sono assegnati a processori diversi, ognuno dei quali si occupa di simulare la propria parte di modello. Gli LP comunicano fra loro inviandosi messaggi o eventi con data e ora. Quando si progetta una simulazione parallela è di fondamentale importanza garantire che gli eventi del modello, indipendentemente dal loro LP, vengano elaborati nella corretta sequenza temporale, mantenendone in questo modo l'ordine causale. Se ogni LP elabora tutti i suoi eventi (generati da se stesso o da un altro LP) in ordine di tempo crescente, garantendo che gli eventi successivi siano eseguiti solo dopo quelli precedenti, e garantendo che gli eventi siano correttamente sincronizzati durante l'esecuzione in parallelo, allora l'ordine causale viene automaticamente conservato nella simulazione distribuita. Questo principio è cruciale per evitare inconsistenze e risultati non realistici. Ogni LP può essere visto come un modello di simulazione sequenziale a eventi discreti, con le proprie variabili di stato locali, i propri eventi e il proprio clock.

Storicamente sono stati adottati due meccanismi di sincronizzazione differente: **conservativo** e **ottimistico** [Fuj95]. Quando si utilizza un meccanismo conservativo l'obiettivo è evitare di violare il vincolo di causalità temporale. Ad esempio, si supponga che un particolare LP si trovi attualmente al tempo di simulazione 25 e sia pronto a elaborare il prossimo evento, che ha un tempo di 30. Il meccanismo di sincronizzazione deve assicurarsi che questo LP non riceva in seguito un evento da un altro LP con tempo di accadimento inferiore a 30. Pertanto, l'obiettivo è determinare quando è effettivamente sicuro elaborare un particolare evento. La sincronizzazione conservativa presenta due principali svantaggi:

- non può sfruttare appieno il parallelismo disponibile: se l'evento A influenza in qualche modo l'evento B, allora A e B devono essere eseguiti in sequenza. Se il modello è tale per cui A influisce raramente B, allora A e B potrebbero essere elaborati in modo concorrente per la maggior parte del tempo;
- non è robusta: una modifica apparentemente piccola del modello può inficiare le prestazioni.

Nella sincronizzazione di tipo ottimistico, le violazioni del vincolo di causalità locale possono verificarsi, ma il meccanismo di sincronizzazione rileva le violazioni e le recupera. Anche in questo caso ogni LP simula la propria porzione di modello in avanti nel tempo, ma non aspetta di ricevere messaggi da altri processi. Il meccanismo *time-warp* [Jef85] è il più noto approccio ottimistico. Se un LP riceve un messaggio che avrebbe dovuto ricevere nel suo passato (e che quindi potrebbe influenzare le sue azioni da quel momento in poi), viene effettuato un *rollback* nell'LP ricevente che porta il suo clock all'ora del messaggio in arrivo. Parte del lavoro annullato potrebbe consistere nell'invio di messaggi ad altri LP. L'invio di questi messaggi viene anch'esso annullato grazie all'invio del corrispondente anti-messaggio. L'invio di anti-messaggi può generare a sua volta rollback secondari negli LP di destinazione. I meccanismi di sincronizzazione ottimistici possono sfruttare il parallelismo in modo migliore rispetto agli approcci conservativi, poiché non sono limitati dallo scenario peggiore. Tuttavia presentano comunque alcuni svantaggi:

- comportano costi computazionali maggiori legati all'esecuzione dei rollback [ZS19];

- per poter effettuare i rollback lo stato di ogni LP deve essere salvato periodicamente, comportando un maggiore utilizzo di memoria.

2.8 Cenni Storici

La storia delle simulazioni ad eventi discreti si estende per oltre mezzo secolo, risalendo ai primi anni dell'era informatica. La necessità di modellare e comprendere il comportamento dinamico dei sistemi complessi spinse i ricercatori a sviluppare metodologie e strumenti per simularne il funzionamento. Le radici concettuali delle simulazioni ad eventi discreti possono essere rintracciate nella teoria dei processi stocastici, che ha trovato applicazioni pratiche nell'ambito dell'ingegneria, della gestione delle operazioni aziendali e della logistica.

Nel corso degli anni '50 e '60, con l'avvento dei primi computer e l'espansione dei linguaggi di programmazione, emersero i primi tentativi di simulare sistemi complessi utilizzando modelli basati su eventi discreti. Questi primi sforzi furono spesso limitati dalle risorse computazionali disponibili e dalle limitazioni dei linguaggi di programmazione dell'epoca.

Negli anni '60 e '70, con l'avanzamento della tecnologia informatica e l'introduzione di linguaggi di programmazione più potenti, come FORTRAN e ALGOL, la simulazione ad eventi discreti divenne sempre più praticabile e diffusa. Durante questo periodo furono sviluppati i primi linguaggi di programmazione specificamente progettati per la simulazione, come SIMSCRIPT [DM64] e GPSS [HM68], i quali permisero ai ricercatori di modellare e analizzare sistemi complessi in modo più efficace e efficiente.

Negli anni successivi, con il continuo avanzamento della tecnologia informatica e l'introduzione di nuovi concetti e metodologie, le simulazioni ad eventi discreti continuarono ad evolversi e a trovare sempre più applicazioni in una vasta gamma di settori. Molte aziende, in particolare nel settore manifatturiero, iniziarono ad utilizzare le simulazioni come strumento di supporto alle decisioni.

A livello informatico il nuovo millennio è stato caratterizzato dalla potenza sempre crescente dei personal computer e dalla diminuzione del prezzo di questi ultimi. La disponibilità crescente di capacità di calcolo ha permesso di simulare scenari sempre più complessi. Le simulazioni hanno cominciato a integrare intelli-

genza artificiale e machine learning per migliorare la previsione e l'ottimizzazione dei sistemi. Inoltre, la crescente disponibilità di dati in tempo reale ha portato all'adozione di approcci di simulazione ibrida che combinano modelli ad eventi discreti con dati reali per migliorare la precisione delle previsioni e delle decisioni.

2.9 Aree di impiego

L'adozione delle simulazioni ad eventi discreti riveste un ruolo cruciale in numerose aree di studio e applicazioni pratiche, offrendo un metodo efficace per esplorare e comprendere il comportamento dei sistemi complessi. Attraverso l'utilizzo di modelli matematici e algoritmi appropriati, le simulazioni ad eventi discreti consentono di esaminare scenari variabili, prendendo in considerazione la casualità e la complessità intrinseche dei processi reali. In vari ambiti l'adozione di simulazioni ad eventi discreti offre numerosi vantaggi. Queste simulazioni consentono di valutare strategie, ottimizzare processi, identificare aree di miglioramento e prevedere l'andamento futuro dei sistemi analizzati. Grazie alla possibilità di eseguire esperimenti virtuali ripetibili e controllati, le simulazioni ad eventi discreti permettono agli operatori e agli studiosi di testare diverse ipotesi e strategie senza dover intervenire direttamente sui sistemi reali, riducendo così i costi e i rischi associati all'implementazione di nuove soluzioni.

2.9.1 Sistemi di Produzione e Movimentazione dei Materiali

I sistemi di produzione e movimentazione dei materiali rappresentano una delle applicazioni più importanti della simulazione. La simulazione è stata utilizzata con successo come supporto nella progettazione di nuove strutture produttive, magazzini e centri di distribuzione [KMO98] [IBH02]. Le simulazioni ad eventi discreti sono utilizzate per condurre analisi sulla capacità di produzione degli impianti, considerando le specifiche delle risorse disponibili come macchinari, manodopera e materiali. Questo aiuta ad individuare sovraccarichi o sottoutilizzazioni delle risorse e a pianificare la capacità in modo efficiente [SM]. Oltre alla produzione, viene impiegata anche per ottimizzare la logistica, comprese le operazioni di movi-

mentazione dei materiali, lo stoccaggio, il prelievo, il trasporto e la distribuzione. Questo comprende l'analisi della disposizione degli impianti, dei percorsi di movimentazione e delle politiche di gestione degli inventari per massimare l'efficienza complessiva e ridurre i costi operativi. I simulatori ad eventi discreti consentono di modellare anche i sistemi di trasporto e distribuzione dei materiali sia all'interno che all'esterno dell'impianto manifatturiero. Ciò include la pianificazione delle rotte di trasporto, l'ottimizzazione dei tempi di consegna e l'analisi dei flussi di traffico per garantire una distribuzione efficiente dei materiali. Infine, è possibile valutare le performance attuali del sistema di produzione e movimentazione dei materiali e testare strategie di miglioramento. Questo può includere l'implementazione di nuove tecnologie, l'ottimizzazione dei processi o la riduzione degli sprechi, contribuendo a migliorare complessivamente l'efficienza e la produttività del sistema. Un esempio di simulatore attualmente utilizzato per gli scopi appena descritti è **AnyLogic** [1], il quale viene utilizzato [KTZP21] per modellare sistemi di produzione e logistica, comprese le catene di approvvigionamento, le reti di distribuzione e gli impianti di produzione. Un altro simulatore presente sul mercato è **Witness**, il quale è stato utilizzato per l'ottimizzazione di processi di produzione [Chr13].

2.9.2 Sistemi informatici

La simulazione dei sistemi informatici consente di modellare e analizzare il comportamento di reti di computer, server e sistemi distribuiti. I simulatori possono essere utilizzati per valutare le prestazioni del sistema, analizzare il throughput e identificare eventuali bottleneck [LP79]. Inoltre, consentono di simulare il carico di lavoro sui sistemi, aiutando a ridimensionare correttamente l'infrastruttura e a pianificare l'allocazione di risorse. È possibile simulare anche l'utilizzo di nuovi algoritmi o politiche di gestione delle risorse, come strategie di scheduling dei processi e politiche di gestione della memoria. Modellare scenari di fallimento e valutare l'impatto dei guasti, valutando l'impatto di questi ultimi sulle prestazioni del sistema e sulla disponibilità dei servizi, consente di valutare la tolleranza ai guasti di sistemi informatici complessi. Anche la sicurezza del sistema può essere

[1] `https://www.anylogic.com`

valutata, simulando scenari di minaccia e attacchi informatici, così da identificare eventuali vulnerabilità.

2.9.3 Wireless Sensor Network

Le Wireless Sensor Network (WSN) sono sistemi distribuiti di sensori interconnessi tramite comunicazione wireless, utilizzati in una vasta gamma di settori, tra cui il monitoraggio ambientale, l'agricoltura di precisione, sanità e sorveglianza. La simulazione ad eventi discreti è importante per la valutazione dei protocolli di comunicazione e di routing specifici per le WSN. I protocolli utilizzati devono gestire alcuni aspetti peculiari delle reti wireless, come la limitata capacità energetica dei nodi sensori e le comunicazioni intermittenti a causa dell'ambiente circostante. La simulazione in questo caso è utile anche per valutare la distribuzione dei nodi sensori nello spazio fisico e per analizzare la copertura della rete. Il simulatore **OMNeT++** [VH08] è stato utilizzato per localizzare i sensori wireless all'interno della rete [WLH05] e per stimare la corretta allocazione di sensori di qualità dell'acqua in aree rurali estese [TCGATC21].

2.9.4 Traffico stradale

Le simulazioni sono ampiamente utilizzate nella simulazione di traffico stradale. Esse permettono di modellare il comportamento dei veicoli in una rete stradale. Ogni veicolo viene rappresentato come un'entità che si muove lungo la strada in base a regole specifiche di accelerazione, decelerazione, cambio di corsia e altre dinamiche di guida. È possibile modellare anche la rete stradale, includendo strade, incroci, semafori, segnaletica stradale e definirne la topologia e le relazioni di connessione tra le varie strade e incroci. Una simulazione può essere utilizzata per simulare il flusso di traffico [CLK95], tenendo conto di variabili come il volume di quest'ultimo, la velocità media e il comportamento degli automobilisti, e per simulare le performance di strade con incroci a più corsie e intersezioni multiple [SMKY13].

2.10 Simulatori presenti sul mercato

Di seguito verranno descritti brevemente alcuni tra i più famosi ed utilizzati simulatori ad eventi discreti presenti sul mercato:

- **Arena**: sviluppato da Rockwell Automation [BS03], offre tre tipologie di simulatore: "basic", "standard" e "professional". L'edizione basic è utilizzata principalmente per modellare processi aziendali e sistemi che richiedono analisi di alto livello. Per modellare sistemi più complessi è necessaria l'edizione standard, la quale utilizza un approccio basato sugli oggetti per modellare la logiche di sistema e le componenti fisiche. La versione professional consente la creazione di oggetti di simulazione personalizzati, consentendo di riflettere fedelmente i componenti e i processi reali del sistema. Gli utilizzi più frequenti di questo simulatore sono la modellazione dei processi aziendali, l'analisi delle catene di approvvigionamento, l'ottimizzazione della produzione e l'analisi del traffico.

- **SIMUL8**: sviluppato da SIMUL8 Corporation [2], consente la creazione di modelli attraverso interfaccia grafica. SIMUL8 mette a disposizione *templates* e *componenti*. I template si concentrano su particolari tipi di decisioni ricorrenti che possono essere rapidamente parametrizzate per adattarsi ad un problema specifico. I componenti invece sono icone definite dall'utente per essere riutilizzate e condivise tra simulazioni differenti. Viene utilizzato in diversi settori, tra cui il settore manifatturiero, sanitario e farmaceutico [3]. Il simulatore è disponibile in due versioni: "standard" e "professional". Entrambe mettono a disposizione le stesse features per la simulazione, ma la versione professional aggiunge la possibilità della visualizzazione 3D.

- **Witness**: sviluppato da Lanner Group[4], offre un interfaccia grafica per la creazione di modelli in modo intuitivo. Gli oggetti grafici rappresentano elementi del sistema come macchine, risorse, code e trasportatori, e possono essere collegati per modellare il flusso di lavoro. Witness fornisce diversi

[2] urlhttps://www.simul8.com
[3] `https://www.simul8.com/applications`
[4] `https://www.lanner.com/en-gb/technology/witness-simulation-software.html`

strumenti per elaborare i risultati mediante grafici e report dettagliati per valutare le prestazioni del sistema e identificare le aree di criticità. Witness è utilizzato in settori come l'aviazione, la difesa, la sanità, la logistica e il settore manifatturiero [5].

2.11 Simulatore Alchemist

Alchemist [PMV13] è un simulatore ad eventi discreti open-source sviluppato all'interno dell'Università di Bologna, che permette la simulazione di scenari inerenti la computazione pervasiva ed ispirata alla natura. Alchemist si basa sull'algoritmo *Next Reaction Method* [GB00], un algoritmo di simulazione stocastica più efficiente dell'algoritmo di Gillespie [Gil77], su cui quest'ultimo si basa.

2.11.1 Modello del dominio

Il dominio di Alchemist, rappresentato in Figura 2.4, è descritto dalle seguenti entità:

- **Molecola**: se Alchemist fosse un linguaggio imperativo, una molecola potrebbe rappresentare il nome di una variabile.
- **Concentrazione**: il valore associato a una particolare molecola. In un linguaggio imperativo la concentrazione sarebbe il valore associato alla variabile.
- **Nodo**: un contenitore di molecole e reazioni, che vivono all'interno di un environment.
- **Environment**: è l'astrazione di Alchemist per lo spazio. È un contenitore di nodi con le seguenti funzionalità:
  - è capace di dirci dov'è un nodo nello spazio;
  - è capace di dirci la distanza fra due nodi;
  - opzionalmente ha il supporto per rimuovere i nodi.

[5] `https://www.lanner.com/en-gb/sectors/`

- **Regola di collegamento**: una funzione dello stato corrente dell'environment che associa ad ogni nodo un vicinato.
- **Vicinato**: un'entità composta da un nodo e un insieme di nodi vicini. Il concetto di vicino deve essere il più generale e flessibile possibile: dal concetto fisico di agenti all'interno di un raggio, al concetto sociale di due agenti collegati da una relazione sociale di qualche tipo.
- **Reazione**: il concetto di reazione è più elaborato di quello usato nella chimica: nei modelli classici, una reazione elenca un certo numero di molecole reagenti che, combinate, producono un insieme di molecole prodotto. In Alchemist una reazione è un evento capace di cambiare lo stato dell'environment. Ogni nodo ha un possibile insieme (anche vuoto) di reazioni. Ogni reazione è definita da una lista (anche vuota) di condizioni, una o più azioni ad essa associate e una distribuzione temporale. La frequenza con la quale accade dipende da:
  - un parametro a tasso statico;
  - il valore di ciascuna condizione;
  - una distribuzione temporale;
  - un'equazione che a partire dal tasso statico e dai valori delle condizioni restituisce un tasso istantaneo.

  Il suo funzionamento può essere osservato in Figura 2.3
- **Condizione**: una funzione che prende come input l'environment corrente e restituisce in output un booleano. Per poter essere innescata, una reazione necessita che tutte le condizioni ad essa associate siano soddisfatte. Le condizioni sono tipicamente espresse come un elenco di annotazioni che devono essere disponibili in una località per l'esecuzione della reazione, ma possono anche includere considerazioni sulla forma del vicinato.
- **Azione**: modella i cambiamenti dell'environment. Le azioni possono essere la trasformazione, la rimozione o la produzione di annotazioni, lo spostamento di agenti, la duplicazione di essi e così via.

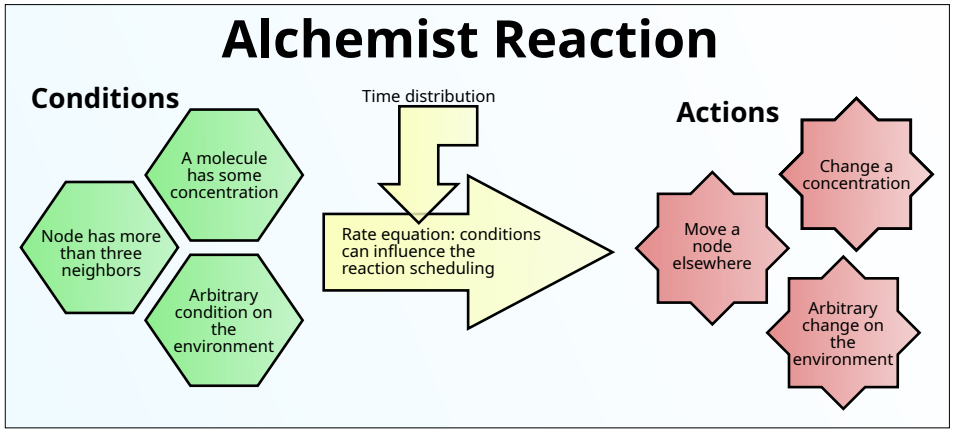

Figura 2.3: Rappresentazione grafica del funzionamento di una reazione.

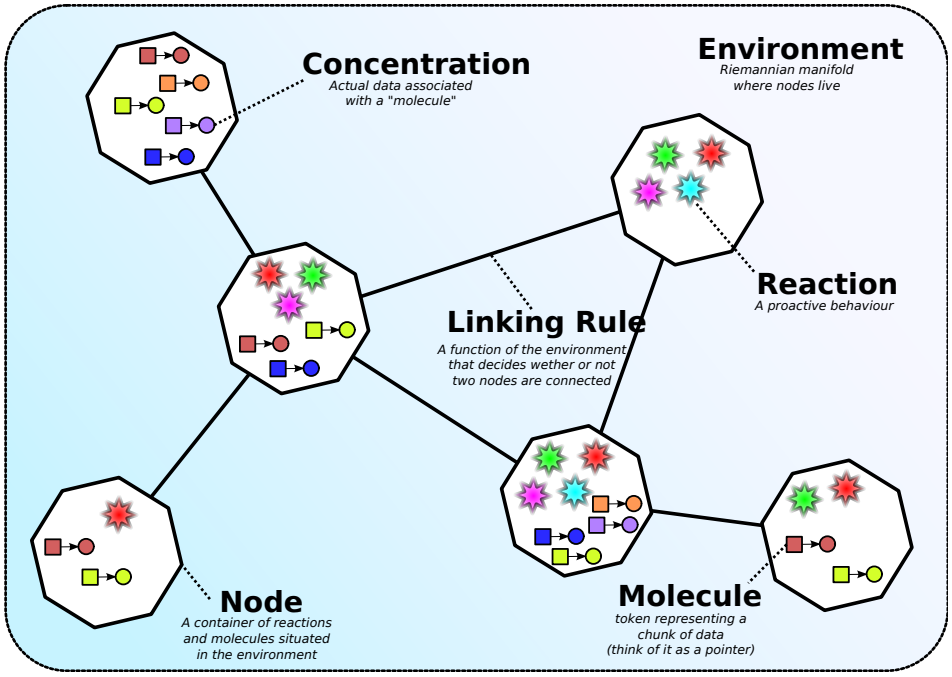

Figura 2.4: Meta-modello alla base di Alchemist.

Una delle caratteristiche principali del simulatore è la sua generalità. Essa viene raggiunta attraverso l'uso delle **incarnazioni**, ovvero le istanze concrete del meta-modello in cui ogni entità viene mappata ad un corrispettivo concetto dell'universo di interesse. Un'incarnazione di Alchemist include una definizione del tipo di concentrazione ed eventualmente un insieme di condizioni specifiche, azioni e (raramente) ambienti e reazioni che operano su tali tipi. Incarnazioni diverse possono modellare universi completamente diversi. Per esempio, come nel caso di **Biochemistry**, se la concentrazione viene definita come un numero intero positivo e vengono fornite azioni e condizioni adeguate, Alchemist diventa un simulatore stocastico di chimica con compartimenti interconnessi e mobili. Nell'incarnazione **SAPERE** le concentrazioni sono un insieme di tuple Linda-Like, ovvero sequenze ordinate di valori utilizzate dal linguaggio di programmazione Linda per la comunicazione asincrona tra processi o thread. È stata utilizzata per simulare evacuazione di folle, l'adattamento anticipato e l'esplorazione di risorse. **Scafi** è dedicata alla programmazione aggregata ed è basata sul framework Scafi [CVAP22]. Anche **Protelis** è utilizzata per la programmazione aggregata. La concentrazione è definita come Java Object. In questa incarnazione i nodi eseguono specifiche scritte nel linguaggio Protelis [PVB15].

2.11.2 Funzionalità e utilizzo del simulatore

Per configurare una simulazione in Alchemist, è necessario fornire un file in formato YAML[6]. Nel file di configurazione occorre specificare quali classi e parametri utilizzare. Le entità vengono identificate tramite chiavi. Al livello zero è possibile trovare:

- **incarnation**: è l'unica chiave obbligatoria. Serve a specificare quale incarnazione si intende utilizzare;
- **seed**: specifica il seme da utilizzare per la generazione di numeri casuali. Questo serve a garantire la riproducibilità degli esperimenti scientifici. È essenziale, a fini di debug, riuscire a garantire simulazioni sempre uguali partendo dalla stessa configurazione nel file YAML;

[6] `https://yaml.org/`

- **variables**: i valori che si intendono riutilizzare all'interno del file di configurazione;
- **environment**: la tipologia di ambiente dentro al quale si intende eseguire la simulazione. Specificando come tipo `ImageEnvironment`, è possibile specificare l'immagine da caricare. Il simulatore caricherà l'immagine convertendo i pixel di un colore selezionato in ostacoli fisici.
- **network-model**: permette di decidere come devono essere collegati fra loro i nodi. È possibile ad esempio utilizzare `connectWithinDistance` per creare i vicinati in base alla posizione dei nodi nell'environment.
- **export**: permette la configurazione e la scelta dei dati della simulazione che si intende esportare. È possibile esportare i dati in un database **mongoDB** definendo `MongoDBExporter` come tipo.
- **deployment**: permette di definire quali sono i nodi del sistema e dove si trovano. Ad esempio è possibile scegliere di definire i nodi come `point` per poi definirne la posizione all'interno dell'environment.

Il simulatore può essere eseguito in due modalità: *interattiva* o *batch*. Nella prima viene eseguita una sola simulazione per volta mentre nella seconda vengono eseguite molteplici simulazioni che dipendono dai valori delle variabili impostati dall'utente. Impostando multipli valori per ciascuna variabile, il simulatore eseguirà una simulazione per ogni possibile combinazione di esse.

Alchemist supporta il caricamento delle planimetrie e delle mappe geografiche. Il caricamento di una planimetria avviene a partire da un'immagine, convertendo i pixel adiacenti del colore specificato in ostacoli. Il supporto agli standard GPX e GPS rende invece possibile il caricamento di mappe.

2.11.3 Architettura

Engine

Di seguito si intende descrivere brevemente il funzionamento dell'algoritmo *Next Reaction Method*, utilizzato da Alchemist per selezionare il prossimo evento da

eseguire e il *Dynamic Dependency Graph*, utilizzato per gestire le dipendenze fra le reazioni presenti all'interno dei nodi.

Next Reaction Method Come anticipato, Alchemist si basa sull'algoritmo di simulazione stocastica *Next Reaction Method*. Ad ogni step della simulazione è in grado di selezionare la prossima reazione da eseguire in tempo costante. L'algoritmo richiede tempo logaritmico per aggiornare la struttura dati interna. L'algoritmo non sceglie la prossima reazione da eseguire valutando la sua velocità, ma generando un tempo putativo e ordinando le reazioni per decidere quale sia la prossima da eseguire. L'algoritmo originale è stato modificato per consentire di aggiungere, rimuovere o spostare le reazioni in modo dinamico. Le reazioni vengono memorizzate in una *Dynamic Indexed Priority Queue*. Quest'ultima è un albero binario di reazioni la cui proprietà principale è che ogni nodo memorizza una reazione il cui tempo di accadimento presunto è inferiore a quello di ciascuno dei suoi figli. Ciò significa che la prossima reazione da eseguire sarà sempre quella nella radice dell'albero e ci si potrà quindi accedere in tempo costante. Un'altra proprietà importante della *priority queue* è che in assenza dell'aggiunta di nuovi nodi all'albero, lo swap fra due nodi non ne modifica il bilanciamento. In Alchemist i nodi vengono aggiunti e quindi non si riesce a sfruttare direttamente questa proprietà. L'idea è quindi di tenere traccia, per ogni nodo, del numero di discendenti per ogni ramo, avendo così la possibilità di mantenere bilanciato l'albero a seguito dell'aggiunta di nodi.

Dynamic Dependency Graph Consiste in un grafo diretto, nel quale i nodi sono le reazioni e gli archi connettono una reazione r a tutti i nodi che dipendono da essa, ovvero quelli il cui tempo di attivazione deve essere aggiornato a seguito dell'esecuzione di r. Mantenere il grafo delle dipendenze aggiornato durante la simulazione è uno dei task più critici. Dato che si vuole supportare nativamente l'interazione fra nodi, che diventano dipendenze fra le reazioni che avvengono in questi nodi, sono definiti tre *contesti* (chiamati anche *scopes*):

- **locale**: la reazione influenza solo il nodo in cui avviene;
- **vicinato**: la reazione influenza il suo nodo e tutto il vicinato;

- **globale**: la reazione influenza tutte le altre reazioni.

Ogni reazione ha un **contesto di input**, ovvero il contesto più piccolo in cui una reazione può leggere informazioni, e un **contesto di output**, ovvero il contesto più piccolo nel quale una reazione può effettuare modifiche. La scelta del contesto giusto è cruciale: se è troppo ristretto la simulazione sarà invalida, se viene scelto troppo grande, questo impatterà pesantemente sulle performance.

L'aggiunta di una reazione implica la verifica delle sue dipendenze con tutte le reazioni del sistema. Se esiste, questa dipendenza viene aggiunta al grafo delle dipendenze. La rimozione di una reazione richiede l'eliminazione di tutte le dipendenze in cui quest'ultima è coinvolta, sia come influenzante che come influenzata. In caso di cambiamento della topologia del sistema è necessario un ulteriore controllo delle dipendenze tra le reazioni appartenenti ai nodi che hanno subito un cambiamento del proprio vicinato. Occorre eseguire una scansione in questi nodi, calcolando le nuove dipendenze con le reazioni appartenenti ai nuovi vicini e cancellando quelle con i nodi che non appartengono più al vicinato.

Capitolo 3

Programmazione reattiva

3.1 Concetti chiave

L'uso del termine **programmazione reattiva** nella letteratura scientifica risale alla metà degli anni '60. Una definizione rilevante è stata data da G. Berry nel 1991 [BB91]. Berry descrive i *programmi reattivi* in relazione alla loro controparte duale, i *programmi interattivi*:

> "I programmi interattivi interagiscono alla propria velocità con gli utenti o con altri programmi; dal punto di vista dell'utente, un sistema time-sharing è interattivo. Anche i programmi reattivi mantengono una continua interazione con l'ambiente, ma a una velocità che è determinata dall'ambiente e non dal programma stesso."

I programmi interattivi concretizzano l'idea di un modello di calcolo "*pull-based*", in cui il programma - in questo caso il consumatore - ha il controllo sulla velocità con cui i dai vengono richiesti e gestiti. Un esempio perfetto di programma interattivo è una struttura di flusso di controllo come un *ciclo for* che itera su un insieme di dati: il programma ha il controllo della velocità con cui i dati vengono recuperati dalla collezione e richiederà l'elemento successivo solo aver terminato la gestione di quello attuale.

I programmi reattivi, al contrario, incarnano l'idea di un modello di calcolo "*push-based*", in cui la velocità con la quale il programma interagisce con l'ambiente è determinata dall'ambiente stesso piuttosto che dal programma. In altre

parole, ora è il produttore dei dati a determinare la velocità con cui si verificheranno gli eventi, mentre il ruolo del programma si riduce a quello di un osservatore silenzioso che reagisce alla ricezione di essi. Esempi standard di tali sistemi sono le applicazioni con *graphical user interface* (GUI) che si occupano di vari eventi originati dall'input dell'utente, come per esempio il click del mouse. Queste applicazioni sono difficili da programmare con i tradizionali approcci di programmazione sequenziale, perché è impossibile prevedere o controllare l'ordine di arrivo degli eventi esterni. Inoltre quando si verifica un cambiamento di stato in un calcolo o in un dato, il programmatore deve aggiornare tutti gli altri che dipendono da esso. Questa gestione manuale dei cambiamenti di stato e delle dipendenze dei dati è complessa e soggetta ad errori.

Utilizzando soluzioni di programmazione tradizionali, le applicazioni interattive sono tipicamente costruite attorno alla nozione di *callback* asincrone. Coordinare le callback può essere un compito arduo, poiché numerosi frammenti di codice isolati possono manipolare gli stessi dati e il loro ordine di esecuzione non è prevedibile. Inoltre, le callback solitamente non hanno un valore di ritorno, quindi sono richiesti side-effect per modificare lo stato dell'applicazione [Coo08].

Il paradigma della programmazione reattiva è stato recentemente proposto come soluzione adatta allo sviluppo di applicazioni *event-driven*. La programmazione reattiva affronta i problemi posti dalla applicazioni *event-driven* fornendo astrazioni per esprimere i programmi come reazioni a eventi esterni e facendo in modo che il linguaggio gestisca automaticamente il flusso del tempo e le dipendenze di dati e calcoli. Ciò porta un vantaggio per i programmatori, i quali non devono preoccuparsi dell'ordine degli eventi e delle dipendenze di calcolo. I linguaggi di programmazione reattivi astraggono dalla gestione del tempo, proprio come i *garbage collector* astraggono dalla gestione della memoria.

3.2 Propagazione del cambiamento

La programmazione reattiva è un paradigma che si basa sulla nozione di valori variabili e continui nel tempo e sulla **propagazione del cambiamento**. In questo paradigma i cambiamenti di stato vengono automaticamente ed efficientemente

propagati attraverso la rete di computazioni dipendenti dal modello di esecuzione sottostante.

Si consideri un semplice esempio di calcolo della somma di due variabili.

```
var1 = 1
var2 = 2
var3 = var1 + var2
```

Nella programmazione sequenziale imperativa convenzionale il valore della variabile `var3` conterrà sempre il valore 3, ovvero la somma dei valori iniziali delle variabili `var1` e `var2`. Nella programmazione reattiva il valore di `var3` è sempre aggiornato perché viene ricalcolato ogni volta che il valore di `var1` o `var2` cambia. Questa è la nozione chiave della programmazione reattiva. I valori cambiano nel tempo e tutti i calcoli dipendenti vengono automaticamente rieseguiti. Nella terminologia della programmazione reattiva la variabile `var3` è detta dipendente dalle variabili `var1` e `var2`. La dipendenza in questione può essere osservata in Figura 3.1.

La logica della propagazione del cambiamento può essere implementata per mezzo di un grafo i cui nodi rappresentano i valori da tenere aggiornati e gli archi rappresentano le relazioni di dipendenza che coinvolgono i nodi. Utilizzando questo tipo di architettura si viene quindi a creare una rete di dipendenze computazioni ed il cambiamento di un nodo (quindi di un valore) scatena una reazione che implica l'attraversamento degli eventuali archi ad esso collegati e il conseguente aggiornamento di tutti i nodi raggiunti.

3.2.1 Strategie di propagazione

Esistono diverse strategie adottabili per la propagazione del cambiamento nella programmazione reattiva, di seguito verranno descritte le principali:

- **complete propagation**: ogni nodo, durante la propagazione, invia il suo stato attuale completo ai nodi dipendenti. In questo caso tutto lo stato precedente del nodo viene perso durante l'aggiornamento del nuovo stato. Questa strategia non è adatta nei casi in cui il grafo delle dipendenze sia particolarmente complesso o nel caso in cui gli stati necessitino di grandi quantità di dati per essere descritti, a causa del suo alto carico computazionale;

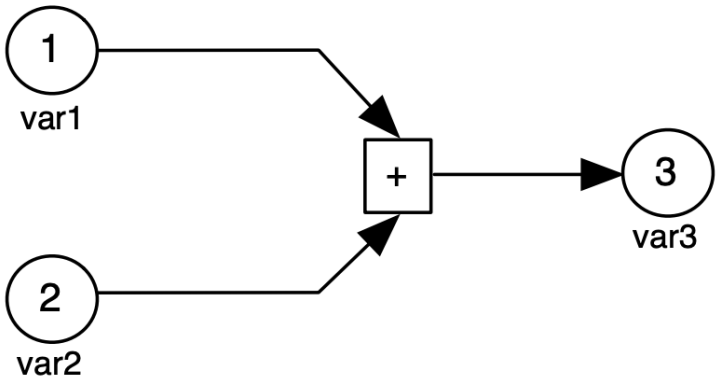

Figura 3.1: Rappresentazione grafica della dipendenza fra espressioni nella programmazione reattiva [BCC+13].

- **delta propagation**: al momento della propagazione i nodi interessati inviano solo una porzione (un *delta*) del loro stato ai nodi dipendenti, cioè solo le frazioni di informazione che sono state effettivamente coinvolte da modifiche. Questo approccio tende a minimizzare la quantità di dati da trasportare lungo la catena di dipendenze durante la propagazione dei cambiamenti, dunque risulta efficiente anche nei casi in cui si ha un grafo delle dipendenze complesso e una grande quantità di dati per la rappresentazione degli stati;

- **batch propagation**: prevede un ritardo nella propagazione dei cambiamenti, cioè la rappresentazione ritardata nel tempo. Consente l'ottimizzazione di quelle situazioni in cui abbiamo più cambiamenti vicini nel tempo che si annullano a vicenda e fanno sì che si ritorni allo stato di partenza. Nel caso in cui due o più cambiamenti vicini nel tempo generino un'effettiva mutazione di stato nel grafo viene propagato solo l'ultimo cambiamento, scartando i precedenti, ormai obsoleti. Questa strategia tende a minimizzare il numero di propagazioni e di conseguenza il carico computazionale complessivo. Tuttavia, occorre adottare un tempo di ritardo corretto al fine di ridurre il carico computazionale, ma al tempo stesso garantire una buona reattività;

- **invalidity notification propagation**: potrebbe essere definita come una "sotto-strategia" piuttosto che una vera e propria strategia implementativa.

Consiste nella richiesta di aggiornamento da parte di un nodo che riscontra di possedere uno stato non valido, oppure che riceve un messaggio di propagazione non valido. In questo caso il nodo interessato scarta l'anomalia e richiede ai nodi da cui dipende un nuovo aggiornamento.

3.3 Modelli di valutazione

Nella programmazione reattiva, per **modello di valutazione** (*evaluation model*) si intende la dinamica direzionale che viene adottata per gestire la propagazione dei cambiamenti nel grafo delle dipendenze computazionali. Il modello viene applicato a livello di linguaggio, rimanendo quindi trasparente all'utilizzatore finale. I modelli presenti in letteratura sono due, ai quali si aggiunge un terzo, ibrido fra i primi due:

- **push-based**: nel modello *push-based* la reazione inizia quando un nodo produttore cambia stato, aggiornandosi. Il nodo in questione "spinge" l'informazione attraverso i nodi consumatori, ovvero i nodi dipendenti. Questo approccio, guidato dalla disponibilità di nuove informazioni, ha il vantaggio di rendere il sistema altamente reattivo, in quanto le reazioni vengono provocate e propagate immediatamente dopo la disponibilità di nuovi dati. Questo approccio comporta, d'altra parte, a computazioni superflue e un alto carico di calcolo, a causa delle elaborazioni necessarie ad ogni cambiamento di stato del produttore.

- **pull-based**: nel modello *pull-based* sono i nodi consumatori a richiedere i nuovi valori nel momento in cui ne hanno bisogno. L'approccio in questo caso è guidato dalla domanda di nuovi dati da parte dei nodi dipendenti. Questo porta ad un sistema più flessibile, ma introduce latenze fra il momento in cui un cambiamento si verifica e il momento in cui avviene la reazione.

- **hybrid push-pull**: il modello ibrido combina i due modelli *push* e *pull*, cercando di trarre i vantaggi di entrambe le soluzioni. Il modello *push-based* funziona bene quando è richiesta una reazione istantanea al cambiamento, mentre il modello *pull-based* porta a prestazioni migliori quando nel sistema

si presentano continui cambiamenti di valori nel tempo. Questo approccio riesce quindi a trarre i benefici del modello *push-based* (efficienza e latenza bassa) e del modello *pull-based* (flessibilità nella richiesta dei valori) [Ell09].

3.4 Glitch

In programmazione reattiva un **glitch** indica un malfunzionamento dovuto ad una inconsistenza dei dati presenti nel grafo delle dipendenze computazionali. La prevenzione dei glitch è un'altra proprietà che deve essere presa in considerazione in un linguaggio reattivo [BCC+13]. Un glitch può essere provocato da una anomalia durante la propagazione di un cambiamento, oppure dal calcolo di un'espressione presente in un nodo prima della valutazione di tutte le sue dipendenze. Il risultato è una situazione nella quale un nodo esegue la propria computazione utilizzando alcuni dati aggiornati ed altri ormai obsoleti, provocando quasi certamente errori di calcolo ed inconsistenze. Si consideri il seguente esempio:

```
var1 = 1
var2 = var1 * 1
var3 = var1 + var2
```

In questo esempio, il valore della variabile `var2` dovrebbe essere sempre lo stesso di `var1`, e il valore di `var3` dovrebbe invece essere il doppio di `var1`. Inizialmente quando il valore di `var1` è 1, il valore di `var2` è 1, e quello di `var3` è 2. Se il valore di `var1` cambia, per esempio diventa 2, ci si aspetta che il valore di `var2` diventi 2 e di conseguenza quello di `var3` diventi 4. Tuttavia, in una implementazione reattiva non corretta, il cambiamento del valore di `var1` potrebbe causare il ricalcolo di `var1 + var2` prima di `var1 * 1` . In questo caso il valore di `var3` sarebbe momentaneamente 3, ovvero un valore non corretto. Prima o poi l'espressione `var1 * 1` sarà computata, in quel momento il valore di `var2` diventerà 2 e quello di `var3` sarà ricalcolato, riflettendo il valore corretto 4. Il comportamento appena descritto è osservabile in Figura 3.2.

La maggior parte dei linguaggi di programmazione reattiva elimina i glitch organizzando le espressioni in un grafo ordinato topologicamente, garantendo così

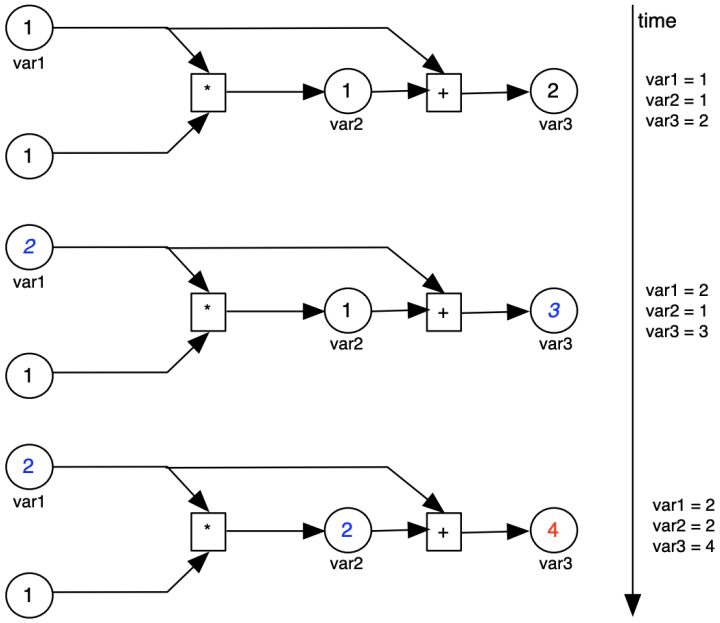

Figura 3.2: Rappresentazione di un possibile glitch nella programmazione reattiva [BCC+13].

che un'espressione venga sempre valutata dopo che tutte le dipendenze sono state valutate. Nella teoria dei grafi l'ordinamento topologico di un grafo consiste nell'ordinare linearmente i suoi nodi in modo che preso un qualunque arco xy dal nodo x al nodo y, x venga prima di y nell'ordinamento. In altre parole, i nodi sono disposti in modo tale che ognuno di essi preceda tutti i nodi collegati ai suoi archi uscenti seguendo l'ordinamento scelto. Le implementazioni reattive più recenti riescono ad evitare i glitch nei programmi reattivi eseguiti su un singolo computer, ma non nei programmi reattivi distribuiti. Evitare problemi in un ambiente distribuito non è un'attività banale a causa dei guasti di rete, ritardi e mancanza di un orologio globale.

3.5 Lifting

Quando la programmazione reattiva è incorporata nei linguaggi host (come libreria o come estensione), gli operatori esistenti (ad esempio + o *) e le funzioni o i metodi definiti dall'utente devono essere convertiti per operare con le astrazioni di base di

essa. La conversione di un operatore ordinario in una variante in grado di operare con valori che possono variare nel tempo è nota come *lifting*. Esistono diverse strategie per effettuare il lifting:

- **lifting implicito**: viene effettuata una trasformazioebe automatica di operazioni su singoli valori in operazioni su flussi di valori senza la necessità di scrivere codice per gestire questa trasformazione. Il lifting implicito rende la programmazione reattiva trasparente agli utilizzatori.

- **lifting esplicito**: in questo caso il linguaggio fornisce una serie di primitive che possono essere utilizzate per effettuare il lifting.

- **lifting manuale**: il linguaggio non fornisce operatori per effettuare il lifting. Il programmatore deve ottenere manualmente il valore corrente di un valore variabile nel tempo, che poi può essere utilizzato con gli operatori del linguaggio.

3.6 Entità osservabili e osservatrici

Nel contesto della programmazione reattiva, le fonti di dati o eventi che emettono informazioni vengono chiamate **entità osservabili**, mentre le **entità osservatrici** sono quelle che si collegano a queste fonti per monitorare e reagire ai dati o agli eventi emessi. In questa sezione vengono descritte le due entità, evidenziando le loro caratteristiche.

Entità osservabile Un'entità osservabile rappresenta un flusso continuo di dati o eventi che può essere emesso in modo regolare o completamente asincrono. Questo flusso può essere caratterizzato da una quantità variabile di elementi, incluso zero, uno o un numero infinito di essi. Questo concetto è fondamentale per gestire dati in tempo reale le interazioni utente in applicazioni moderne, consentendo una gestione dinamica dei flussi di informazioni.

Tuttavia, l'entità osservabile va oltre la semplice emissione di dati o eventi. Essa può concludersi in due modi principali: notificando un completamento avvenuto o segnalando un errore anomalo durante la sua esecuzione. Il completamento

avvenuto è tipicamente utilizzato quando il flusso di dati ha termini definiti e finiti, ad esempio dopo il completamento di una sequenza di operazioni. D'altra parte, l'errore può essere segnalato in caso di malfunzionamenti imprevisti, come errori di connessione di rete o eccezioni di programmazione.

Se consideriamo un'entità osservabile con un flusso infinito di elementi, come ad esempio gli eventi di click dell'utente in un'applicazione Android, è importante notare che non sarà mai in grado di emettere un messaggio di completamento, questo perché il completamento è riservato solo al termine naturale del flusso di eventi disponibili, il quale, se infinito, non si verifica mai. Tuttavia, è possibile che l'entità osservabile termini in modo anomalo, segnalando un errore, nel caso si verifichino anomalie durante la sua esecuzione. In tal caso, l'entità osservabile con un flusso infinito terminerà anticipatamente e non emetterà ulteriori elementi.

Entità Osservatrice Un'entità osservatrice può instaurare una connessione (sottoscrizione) con un'entità osservabile al fine di monitorare e gestire i dati o gli eventi trasmessi da quest'ultima secondo specifiche politiche reattive. Gli elementi all'interno di un flusso sono emessi in modo sequenziale, uno dopo l'altro, e vengono elaborati dall'entità osservatrice in modo consecutivo. Questa caratteristica impedisce la comparsa di corse critiche, mantenendo l'ordine delle operazioni e garantendo una gestione reattiva dei dati. Nel caso in cui un grande numero di dati o eventi venga emesso rapidamente dall'entità osservabile e l'entità osservatrice non sia in grado di elaborarli con la stessa velocità, è possibile adottare strategie come l'utilizzo di una *coda degli eventi* o il meccanismo di *buffering* per gestire in modo efficiente il flusso di dati e prevenire perdite o sovraccarichi. Un'entità osservabile può essere sottoscritta da zero, una o più entità osservatrici contemporaneamente, permettendo una gestione flessibile e dinamica del flusso di dati o eventi all'interno di un'applicazione reattiva.

3.7 Programmazione reattiva distribuita

Un linguaggio per la programmazione reattiva distribuita permette la ripartizione dei dati e del grafo delle dipendenze computazionali fra più nodi di una rete in un ambiente distribuito. Prendendo come esempio l'espressione `var3 = var1 +`

`var2`, le tre variabili e il relativo grafo delle dipendenze potrebbero trovarsi su nodi diversi della rete.

La progettazione di un linguaggio reattivo distribuito è di gran lunga più complessa rispetto ad un linguaggio reattivo tradizionale. La principale difficoltà risiede nel fatto che in questi contesti occorre considerare fattori come latenza, congestione della rete, guasti di essa e la mancanza di un orologio globale. Un linguaggio reattivo distribuito, come già anticipato in Sezione 3.4, non riesce a garantire la completa rimozione delle inconsistenze dei dati nel grafo delle dipendenze, portando quindi a glitch. La presenza di un grafo delle dipendenze in un programma reattivo distribuito, oltre a portare alla presenza di glitch, tende ad accoppiare strettamente i componenti dell'applicazione, rendendoli quindi meno resistenti ad errori e riducendone la scalabilità complessiva [MSM19]. Una soluzione che potrebbe essere adottata è utilizzare un approccio centralizzato, nel quale un'entità avente un orologio centrale è responsabile dell'ordinamento degli aggiornamenti nel grafo delle dipendenze. Questo approccio introduce un *singolo punto di rottura* e un eccessivo numero di comunicazioni dovute al fatto che tutte le parti coinvolte devono comunicare con una singola entità ogni qualvolta ricevono e propagano dati, limitando in questo modo la scalabilità dell'architettura.

3.8 Il Manifesto Reattivo

Il 16 settembre 2014 è stata rilasciata la seconda versione del *Manifesto Reattivo* [1], un documento redatto da esperti del settore tra cui Jonas Bonér, Dave Farley, Roland Kuhn e Martin Thompson. Questo manifesto, piuttosto che essere un trattato sulla programmazione reattiva, si presenta come una guida pratica per la progettazione di sistemi reattivi, delineando le caratteristiche chiave che tali sistemi dovrebbero possedere. È importante comprendere che il manifesto non promuove un'unica metodologia o paradigma di programmazione reattiva, ma piuttosto si concentra su principi generali che possono essere applicati a una vasta gamma di architetture e tecnologie. L'obiettivo è garantire che i sistemi reattivi siano capaci di affrontare sfide come la scalabilità, la resilienza e la responsività, comuni

[1] `https://www.reactivemanifesto.org/`

nell'ambito delle applicazioni moderne. La necessità di redigere il manifesto è stata spinta dai rapidi cambiamenti nel panorama tecnologico. Mentre in passato le applicazioni erano caratterizzate da un'architettura monolitica, tempi di risposta relativamente lunghi e gestione di volumi limitati di dati, oggi vi è una realtà in cui le applicazioni devono essere distribuite su una vasta gamma di dispositivi, supportare grandi volumi di dati e fornire risposte quasi istantanee agli utenti. Gli autori del manifesto descrivono un sistema reattivo come **flessibile**, a **basso accoppiamento** e **scalabile**: questo lo rende più semplice da sviluppare e più malleabile nel tempo. Tali sistemi sono più **tolleranti ai guasti** e reagiscono ad essi in modo elegante e non brusco. Sono altamente **responsivi**, offrendo agli utenti un feedback concreto. Il manifesto delinea quattro principali caratteristiche, rappresentate in Figura 3.3, che un sistema deve assolutamente possedere per essere considerato reattivo:

- **responsività**: il sistema, se è in generale possible dare una risposta ai client, la dà in maniera tempestiva. La responsività è la pietra miliare dell'usabilità e dell'utilità del sistema; essa presuppone che i problemi vengano identificati velocemente e gestiti in modo efficace. I sistemi responsivi sono focalizzati a minimizzare il tempo di risposta, individuando per esso un limite massimo prestabilito di modo da garantire una qualità del servizio consistente nel tempo. Il comportamento risultante è quindi predicibile, il che semplifica la gestione delle situazioni di errore, genera fiducia negli utenti finali e predispone ad ulteriori interazioni con il sistema;

- **relisienza**: il sistema resta responsivo anche in caso di guasti. Ciò riguarda non solo i sistemi ad alta disponibilità o mission-critical, infatti, accade che ogni sistema che non è resiliente si dimostrerà anche non responsivo in seguito ad un guasto. La resilienza si acquisisce tramite replica, contenimento, isolamento e delega. I guasti sono relegati all'interno di ogni componente, isolando così ogni componente dagli altri e quindi garantendo che il guasto delle singole porzioni del sistema non comprometta il sistema intero. Il recupero di ogni componente viene delegato ad un altro componente (esterno) e l'alta disponibilità viene assicurata tramite replica laddove necessario. I

client di un componente vengono dunque alleviati dal compito di gestirne i guasti;

- **elasticità**: il sistema rimane responsivo sotto carichi di lavoro variabili nel tempo. I sistemi reattivi possono adattarsi alle variazioni nella frequenza temporale degli input incrementando o decrementando le risorse allocate al processamento degli stessi. Questo porta ad architetture che non hanno né sezioni contese né colli di bottiglia, favorendo così la distribuzione o la replica dei componenti e la ripartizione degli input su di essi. I sistemi reattivi permettono l'implementazione predittiva, oltre che reattiva, di algoritmi scalabili perché fondati sulla misurazione real-time della performance. Tali sistemi, raggiungono l'elasticità in maniera cost-effective su commodity hardware e piattaforme software a basso costo;

- **orientato ai messaggi**: i sistemi reattivi si basano sullo scambio di messaggi asincrono per delineare per ogni componente il giusto confine che possa garantirne il basso accoppiamento con gli altri, l'isolamento e la trasparenza sul dislocamento e permetta di esprimere i guasti del componente sotto forma di messaggi al fine di delegarne la gestione. L'utilizzo di uno scambio esplicito di messaggi permette migliore gestibilità del carico di lavoro, elasticità e controllo dei flussi di messaggi mediante setup e monitoraggio di code di messaggi all'interno del sistema. Uno scambio di messaggi trasparente rispetto al dislocamento rende possibile, ai fini della gestione dei guasti, l'utilizzo degli stessi costrutti e semantiche sia su cluster che su singoli host. Uno stile di comunicazione non bloccante fa sì che l'entità ricevente possa solo consumare le risorse, il che porta ad un minor sovraccarico sul sistema.

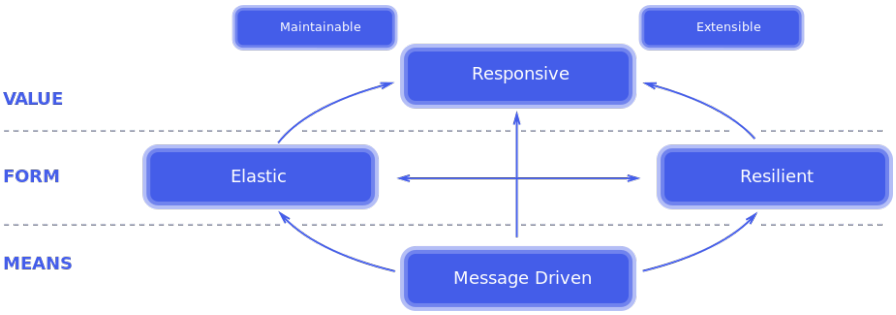

Figura 3.3: Caratteristiche fondamentali di un sistema reattivo.

Capitolo 4

Programmazione reattiva in Kotlin

4.1 Kotlin

Kotlin [1] è un linguaggio di programmazione, sviluppato dall'azienda JetBrains [2] nel 2011, il quale si pone come alternativa a linguaggi come Java e Scala, che girano su *Java Virtual Machine* (JVM). La crescita di Kotlin è stata piuttosto rapida, affermandosi come linguaggio di programmazione di riferimento per applicazioni Android. È possibile configurare Kotlin per essere eseguito su JVM, ma può essere compilato anche in codice Javascript o direttamente in nativo tramite la struttura di compilazione LLVM [LA04] per essere eseguito su dispositivi Android. Kotlin è un linguaggio a tipizzazione statica che unisce i costrutti tipici della programmazione ad oggetti con alcuni di quelli della programmazione funzionale. Una delle caratteristiche principali di Kotlin è la sua **concisione** ed **espressività**: in media è necessario scrivere il 40% di codice in meno rispetto alla controparte Java [3] per raggiungere le stesse funzionalità. Questo è possibile poiché i pattern più comuni sono supportati già a livello di linguaggio. Un'altra caratteristica fondamentale è la **null-safety**. In Kotlin i tipi definiscono se un oggetto può essere nullo o meno. Inoltre Kotlin è tipizzato staticamente, ciò significa che tutti i tipi sono

[1] `https://kotlinlang.org/`
[2] `https://www.jetbrains.com/`
[3] `https://kotlinlang.org/docs/faq.html`

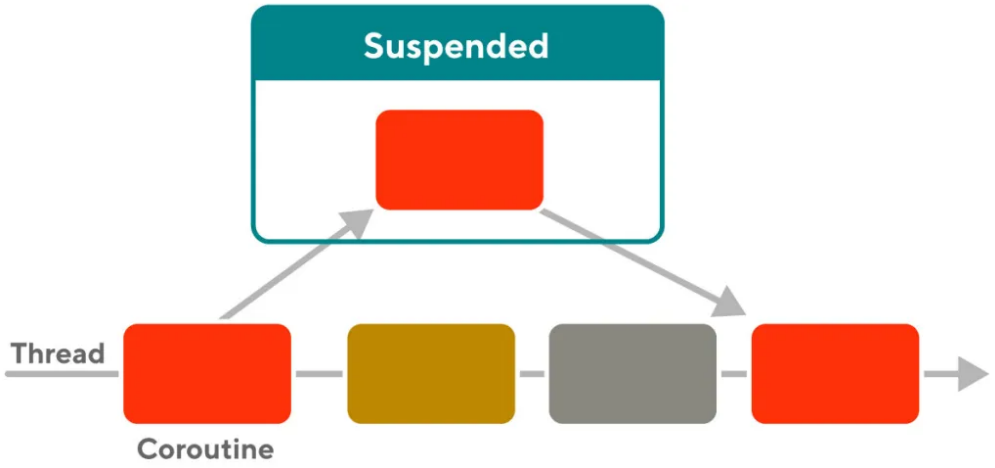

Figura 4.1: Quando una coroutine è sospesa, lo scheduler di Kotlin mette quest'ultima in uno stato sospeso e continua a processare le altre coroutines.

conoscibili a tempo di compilazione, siano essi dichiarati esplicitamente o inferiti dal compilatore stesso. Gli eventuali errori vengono quindi intercettati prima dell'esecuzione del codice. Il codice scritto in Kotlin è interoperabile con il codice scritto in linguaggio Java, è possibile quindi richiamare codice Kotlin da codice Java e viceversa.

4.2 Coroutines

Sebbene siano utilizzate da oltre 50 anni, non esiste una definizione universale su cosa sia una *coroutine*. Kotlin definisce una coroutine come un'istanza di una *suspending function*, ovvero una funzione che può sospendere temporaneamente la propria esecuzione, concettualmente simile ad un *thread*, per il fatto che entrambi prendono un blocco di codice da eseguire e hanno un ciclo di vita simile. Le coroutines tuttavia, non sono limitate ad un singolo thread. La loro esecuzione può essere sospesa in un thread e può essere ripristinata in un altro. La Figura 4.1 mostra come, a seguito della sospensione di una coroutine, il thread possa continuare ad eseguire altre coroutines. La coroutine sospesa può infine essere ripristinata anche su un thread differente.

4.2.1 Classificazione

Ana Lucia De Moura e Roberto Ierusalimschy [dMI09] classificano le coroutines secondo la loro implementazione:

- **trasferimento del controllo simmetrico/asimmetrico**: riguarda il modo in cui le coroutines gestiscono il flusso di esecuzione e la comunicazione con altre coroutines o con il thread principale. Nel trasferimento del controllo *simmetrico*, le coroutine cooperano tra loro per controllare l'esecuzione. Le coroutines con trasferimento del controllo simmetrico sono generalmente implementate utilizzando un meccanismo cooperativo, dove una coroutine può essere sospesa e ripresa solo quando decide di farlo. D'altra parte, nel trasferimento del controllo *asimmetrico*, una delle coroutines coinvolte detiene il controllo sull'esecuzione e può assegnarlo ad altre coroutines in modo asincrono. In questo modello, una coroutine può essere sospesa o ripresa anche senza il suo consenso diretto, poiché il controllo è gestito in modo asincrono dal sistema o da un'entità esterna;

- **implementazione stackful/stackless**: influenza il modo in cui vengono gestite le coroutines e la profondità dello stack delle chiamate. In un'implementazione *stackful*, ogni coroutine ha il proprio stack separato, simile a quello di un *thread*. Ciò consente a ciascuna di esse di eseguire chiamate ricorsive senza causare un *overflow* dello stack, poiché ogni coroutine ha il proprio stack dedicato. Tuttavia, questo approccio richiede una gestione importante delle risorse, poiché ogni coroutine richiede un'allocazione di memoria separata per lo stack. In un'implementazione *stackless*, le coroutines condividono lo stesso stack o utilizzano un modello di gestione dello stack non ricorsivo, come i *coroutine frames*. Questo modello rende la gestione delle coroutine più efficiente in termini di utilizzo delle risorse, ma può limitare la profondità delle chiamate ricorsive e potenzialmente causare un overflow dello stack se la profondità delle chiamate è troppo grande;

- **supporto di prima classe/limitato**: influisce sul grado di flessibilità e controllo che gli sviluppatori hanno quando utilizzano le coroutines. Con il *supporto di prima classe*, le coroutines sono trattate come qualsiasi altro

tipo di dato nel linguaggio di programmazione. Ciò significa che possono essere passate come argomenti a funzioni, restituite come valori di ritorno da funzioni e assegnate a variabili. Il *supporto limitato* per le coroutine porta a restrizioni sulle operazioni che è possibile eseguire con le coroutines.

4.2.2 Limitazioni della programmazione asincrona

La programmazione asincrona è soggetta agli stessi problemi della programmazione in generale; allo stesso tempo, presenta una serie di problemi unici, non rilevanti per il codice sincrono [EBAU21]. In questa sezione verranno discussi brevemente due dei problemi che influenzano maggiormente lo spazio di progettazione per un linguaggio di programmazione con supporto alla programmazione asincrona: "**colorazione delle funzioni**" e **gestione degli errori**.

Una differenza importante tra diverse implementazioni di computazione asincrona riguarda la gestione della divisione tra codice asincrono e sincrono. Ci si riferisce a questo problema come "*colorazione delle funzioni*". Questo è un concetto comune che assume diverse forme e formalizzazioni, ma in sostanza si riduce a un numero di regole semplici:

- ogni programma o funzione è assegnata ad un particolare colore, ad esempio rosso o blu;
- il codice blu rappresenta la computazione sincrona e può essere acceduto da entrambi i colori;
- il codice rosso rappresenta invece la computazione asincrona, e può essere acceduto solo dal codice di colore rosso;
- esistono costrutti speciali che consentono di chiamare il codice rosso da quello blu, in alternativa, il codice delle *entry function* è rosso.

Per risolvere questo problema si potrebbe pensare ad un linguaggio in cui tutto il codice è rosso, e tutte le computazioni sono quindi asincrone. Nella realtà, però, introdurre computazione asincrona in tutte le funzioni porta a diversi problemi:

- la computazione asincrona introduce complessità rispetto alla chiamata di normali funzioni;

- molti linguaggi di programmazione sono sincroni per natura e introducono costrutti asincroni solo per determinati scopi. La mancanza di una chiara distinzione tra funzioni sincrone e asincrone porta a problemi di comprensione del programma;

- La gestione degli errori e la loro propagazione in codice rosso è un task particolarmente complesso. Se tutto il codice fosse rosso, sarebbe richiesto una gestione avanzata degli errori in tutto il codice.

Un problema cruciale nella programmazione asincrona riguarda la gestione degli errori. Quando si hanno più esecuzioni asincrone in corso contemporaneamente e una di esse fallisce, il modo per recuperare e proseguire l'esecuzione non è così lineare come nel codice sincrono. Nel codice standard, esiste una chiara relazione tra chiamante e chiamato: gli errori nel chiamato si propagano verso l'alto al chiamante e vengono gestiti lì o continuano a propagarsi. Tuttavia, nel codice asincrono, questa relazione non è più presente e senza di essa non c'è un punto di responsabilità definito su chi debba gestire quali errori. È anche possibile che sia necessario propagare la gestione degli errori verso il basso, ad esempio per annullare calcoli non necessari. Per affrontare questo problema, si possono utilizzare le funzionalità offerte dal linguaggio di programmazione per creare un meccanismo di gestione degli errori sostitutivo, tuttavia, tutte queste soluzioni non sono ottimali per la gestione degli errori verso il basso. Un approccio più strutturato alla gestione degli errori asincroni incorporerebbe sia la gestione verso l'alto che quella verso il basso.

Esistono due approcci principali utilizzati nella programmazione asincrona attuale. Il primo si basa sugli **alberi di supervisione**. Questo metodo prevede l'organizzazione esplicita delle attività asincrone in alberi padre-figlio, che corrispondono alla struttura desiderata di propagazione degli errori verso l'alto e verso il basso. In caso di errore, è possibile scegliere se isolare, propagare o riavviare il sotto-albero di attività interessato. Il secondo approccio è chiamato **concorrenza strutturata**, propone di applicare i principi della programmazione strutturata alla programmazione asincrona. In questo caso, se un'attività A avvia un'attività B, la durata di B non può superare quella di A. Questo schema stabilisce una relazione di "*launcher-launchee*" invece di quella *chiamante-chiamato* e descrive

come propagare gli errori e le cancellazioni verso il basso. Rispetto agli alberi di supervisione, la concorrenza strutturata è meno flessibile, poiché la strategia di gestione degli errori è predefinita. Tuttavia, è anche meno verbosa e corrisponde meglio al modo in cui solitamente viene scritto il codice asincrono.

4.2.3 Goals

Kotlin è stato concepito come un linguaggio di programmazione pragmatico, pensato per essere utilizzato quotidianamente e per agevolare gli sviluppatori nel completare i loro compiti attraverso le sue funzionalità e strumenti. Questo stesso approccio viene applicato al suo supporto per la programmazione asincrona, con obiettivi principali ben definiti.

Prima di tutto, Kotlin mira a garantire **l'indipendenza dalle specifiche implementazioni di basso livello** delle piattaforme. Poiché è progettato per essere un linguaggio multipiattaforma, costruire un supporto asincrono basato su altre implementazioni (come ad esempio le futures su JVM) potrebbe generare diversi problemi, soprattutto in termini di interoperabilità tra le diverse piattaforme.

Inoltre, Kotlin si impegna a essere altamente **adattabile alle implementazioni esistenti**. Essendo relativamente nuovo, il linguaggio presta particolare attenzione all'interoperabilità con il codice già esistente, con un'enfasi speciale sul lavoro con il codice Java sulla piattaforma JVM in modo trasparente. Considerando che esistono già approcci consolidati per la gestione del codice asincrono su piattaforme specifiche (come ad esempio le promise in JavaScript o le operazioni di input/output non bloccanti su JVM), Kotlin mira a integrare senza soluzione di continuità tali API nel suo ecosistema.

Infine, Kotlin punta a fornire un **supporto per la programmazione asincrona pragmatica**. L'avvento e la diffusione dell'approccio async/await, rispetto ad altri stili di programmazione asincrona, hanno evidenziato l'importanza della leggibilità del codice.

4.2.4 Coroutine builder

I **coroutine builder** sono funzioni che consentono di creare e avviare coroutine in Kotlin. In Kotlin sono disponibili diversi coroutine builder, i più utilizzati sono:

- **launch**: è utilizzato per avviare una nuova coroutine in modo asincrono. Quando si utilizza *launch*, la coroutine viene avviata e viene eseguito il codice all'interno della coroutine in modo indipendente dal flusso principale del programma. Questo significa che la coroutine può essere eseguita in background senza bloccare il thread principale. *Launch* non restituisce alcun valore, quindi è utilizzato principalmente per operazioni di tipo *fire-and-forget*; l'esecuzione ritorna un oggetto `job`, sul quale si può eseguire il metodo `join()`, che blocca il thread chiamante fino a quando la coroutine non è terminata. Le eccezioni che accadono in una coroutine eseguita con *launch* sono propagate alla coroutine padre, ma non alle coroutines sorelle. Se viene lanciata una eccezione e non viene gestita, questa verrà ignorata silenziosamente.

- **async**: è utilizzato per avviare una coroutine che restituisce un valore, cioè un'operazione che produce un risultato. A differenza di *launch*, *async* restituisce un oggetto `Deferred`, che rappresenta il risultato futuro dell'operazione asincrona. Il valore può essere ottenuto utilizzando la funzione `await()` su un oggetto `Deferred`. Anche in questo caso il thread chiamante è bloccato finché non è stato prodotto il risultato. Questo è utile quando si desidera eseguire un'operazione asincrona e recuperare il risultato prodotto da essa. Le eccezioni in questo caso sono immagazzinate all'interno dell'oggetto `Deferred`. Se eseguiamo il metodo `await()`, l'eccezione viene rilanciata e può essere gestita con un normale blocco `try-cath` o altri meccanismi di gestione delle eccezioni.

- **runBlocking**: è un coroutine builder che fa da ponte fra il codice al suo interno e il mondo esterno alle coroutine. Il thread che esegue questo builder viene bloccato per tutta la durata della chiamata, finché tutte le coroutine al suo interno non hanno completato la loro esecuzione.

4.2.5 Coroutine scope e concorrenza strutturata

Un **coroutine scope** fornisce un modo per gestire il ciclo di vita delle coroutines, consentendo di avviare, gestire o cancellare gruppi di coroutines in modo coerente e sicuro. Ciascuna di esse deve essere gestita all'interno di uno scope e quando

questo viene cancellato tutte le coroutines al suo interno vengono cancellate. I coroutine scope possono essere nidificati, in questo caso quando uno scope di livello superiore viene cancellato, vengono cancellate tutte le coroutines al suo interno e tutti gli scope interni vengono cancellati ricorsivamente. La radice di questo albero di dipendenze è chiamata `GlobalScope`. Anche se è possibile eseguire le coroutines all'interno dello scope globale, è consigliato creare nuovi scope secondo la necessità e renderli figli e genitori l'uno dell'altro in modo logico, sicuro e efficiente. In sintesi, il coroutine scope permette di gestire le coroutines in modo strutturato, definendo il ciclo di vita delle stesse. Questo garantisce che le risorse vengano gestite in modo corretto e che le attività vengano terminate in modo ordinato, migliorando l'affidabilità e la manutenibilità del software e risolvendo i problemi della gestione degli errori nella programmazione asincrona descritti nella Sezione 4.2.2

4.2.6 Coroutine context

Il **coroutine context** è un insieme persistente di oggetti definiti dall'utente che possono essere associati alla coroutine stessa. Questo insieme può includere oggetti responsabili delle politiche di threading della coroutine, del logging, degli aspetti di sicurezza e di transazione dell'esecuzione della coroutine, nonché dell'identità e del nome della stessa. Se pensiamo alle coroutines come thread leggeri, il contesto di una coroutine può essere paragonato a una raccolta di variabili locali del thread. Tuttavia, a differenza delle variabili locali del thread, il contesto di una coroutine è immutabile, ma è possibile aggiungerne elementi usando l'operatore `plus`. Questo non rappresenta una grave limitazione, poiché le coroutines sono così leggere che è facile lanciarne una nuova quando c'è la necessità di modificare qualcosa nel contesto. La libreria standard non include implementazioni concrete, ma fornisce un'interfaccia, consultabile al Listato 4.1, in modo che tutti questi aspetti possano essere definiti in modo componibile. Concettualmente, il coroutine context può essere considerato una sorta di ibrido tra un insieme e una mappa: gli elementi hanno chiavi come in una mappa, ma le chiavi sono direttamente associate agli elementi, similmente ad un insieme. Ogni elemento di un coroutine context è esso stesso un contesto. Questo consente la creazione di contesti composti. In Figura 4.2 è possibile notare come lo scope di una coroutine figlio venga definito partire dal

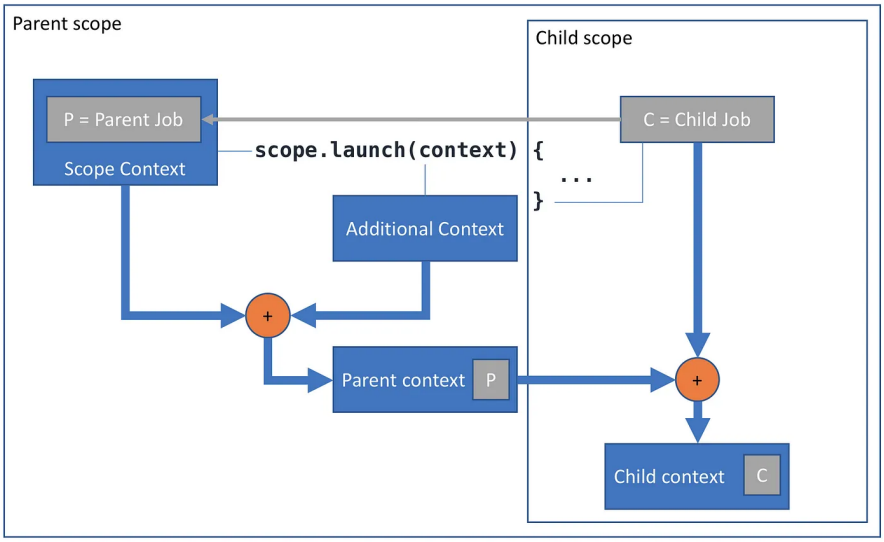

Figura 4.2: La nuova coroutine crea la propria istanza di `job` e definisce il proprio contesto a partire dal contesto padre più il job creato.

contesto dello scope padre e aggiungendo la propria istanza di `job`.

Coroutine Dispatcher

Un coroutine context include un **coroutine dispatcher**, il quale determina quale o quali thread saranno utilizzati per l'esecuzione delle coroutine. È responsabile dell'allocazione delle risorse di sistema necessarie per l'esecuzione delle coroutines e della coordinazione del loro funzionamento su thread o altri thread pool disponibili

Listing 4.1: Interfaccia definita per il coroutine context, presente nel package `kotlin.coroutines`.

```
interface CoroutineContext {
    operator fun <E : Element> get(key: Key<E>): E?
    fun <R> fold(initial: R, operation: (R, Element) -> R): R
    operator fun plus(context: CoroutineContext): CoroutineContext
    fun minusKey(key: Key<*>): CoroutineContext

    interface Element : CoroutineContext {
        val key: Key<*>
    }

    interface Key<E : Element>
}
```

nel sistema. I dispatchers più comuni disponibili in Kotlin sono:

- **Dispatchers.Default**: viene utilizzato per eseguire le coroutine su un pool di thread con un numero di thread pari al numero di processori disponibili nel sistema o un multiplo di esso. È adatto per operazioni di CPU-bound, come elaborazioni intensive o calcoli.

- **Dispatchers.IO**: viene utilizzato per eseguire le coroutines su un pool di thread dedicato alle operazioni I/O-bound, come operazioni di lettura/scrittura su file, operazioni di rete, o chiamate API. Questo dispatcher ha un pool di thread più grande rispetto a Dispatchers.Default per consentire un numero maggiore di operazioni di I/O contemporaneamente.

- **Dispatchers.Main**: è specifico per le applicazioni Android e viene utilizzato per eseguire le coroutine sul thread principale della User Interface (UI). È importante utilizzare questo dispatcher quando si effettuano operazioni che coinvolgono l'interfaccia utente per evitare blocchi o rallentamenti dell'UI.

- **Dispatchers.Unconfined**: Questo dispatcher esegue le coroutine in modo sequenziale sul thread chiamante, senza alcun thread pool dedicato. È utile in situazioni in cui si desidera eseguire il codice sul thread chiamante fino a quando non è necessario passare ad un altro thread.

Tutti i coroutine builder, come `launch` e `async`, accettano opzionalmente un coroutine context come parametro che può essere usato per specificare esplicitamente il dispatcher e altri elementi per la nuova coroutine. Quando un coroutine builder è utilizzato senza parametri, eredita il contesto (e quindi il dispatcher) dal coroutine scope dal quale è stato eseguito.

4.3 Kotlin Flow

Kotlin Flow è una libreria che offre un'API reattiva per gestire i flussi di dati asincroni in applicazioni Kotlin. Basato sul modello della programmazione reattiva, Kotlin Flow permette agli sviluppatori di creare, trasformare e combinare flussi di eventi in modo dichiarativo e componibile. Android descrive un flow come un tipo

in grado di emettere valori in sequenza, a differenza delle suspending function, le quale restituiscono un solo valore. È possibile per esempio utilizzare un flow per ricevere aggiornamenti in tempo reale da un database. Concettualmente un flow è un flusso di dati che può essere consumato in modo asincrono. Un flow è molto simile ad un iteratore che produce una sequenza di valori, ma utilizza suspending function per produrre e consumare valori in modo asincrono. Le entità coinvolte nei flow sono tre:

- **produttore**: produce i dati che sono aggiunti al flusso;
- **intermediario** (opzionale): può modificare ogni valore emesso nel flusso o il flusso stesso senza consumare i valori;
- **consumatore**: consuma i valori dal flusso.

4.3.1 Tipi di flow

Nel contesto dei Kotlin Flow, possiamo dividere i flussi in due categorie: **cold flow** e **hot flow**. I cold flow emettono valori sono quando avviene una sottoscrizione. Ogni qualvolta un osservatore si sottoscrive ad un cold flow viene avviata la produzione dei valori dall'inizio del flusso di dati. Ogni sottoscrizione a questo tipo di flow ha quindi il proprio flusso di dati privato, il quale è indipendente dagli altri. Un hot flow, al contrario, emette valori continuamente, indipendentemente dalla presenza o meno di sottoscrizioni attive. Quando un osservatore si sottoscrive a un hot flow, inizia a ricevere i valori emessi da quel momento in avanti, senza influenzare la produzione dei valori stessi. Non esistono flussi privati per ciascuna sottoscrizione, quindi tutti gli osservatori ricevono gli stessi valori emessi.

Cold Flow

Come è possibile notare in Figura 4.3, tutti i tipi di flow estendono dall'interfaccia *Flow*, il cui codice è consultabile al Listato 4.2. Il *Single Abstract Method conversion* (SAM conversion), è un meccanismo messo a disposizione da Kotlin mediante il quale è possibile implementare interfacce funzionali utilizzando funzioni lambda, offrendo una sintassi più concisa e leggibile. L'interfaccia `FlowCollector` è un'interfaccia funzionale, ha quindi un solo metodo, che è `emit(value: T)`. Quando

Listing 4.2: Codice dell'interfaccia Flow

```
public interface Flow<out T> {

    public suspend fun collect(collector: FlowCollector<T>)
}
```

l'osservatore invoca il metodo `collect` sul flow, sta quindi implementando il metodo `emit` appena descritto. Tutti i flow builder messi a disposizione, di cui si parlerà successivamente, hanno come receiver implicito un `FlowCollector`. Questo permette di emettere nuovi elementi sul flusso chiamando semplicemente il metodo `emit`, il quale contiene la logica per consumare l'elemento. Tutti i flow builder di tipo cold emettono tutti i valori ad ogni nuova sottoscrizione, creando così un flusso privato per ogni singolo osservatore.

Hot Flow

Come anticipato, a differenza dei cold flow, i quali riemettono tutti i valori ad ogni nuova sottoscrizione, gli hot flow emettono elementi in continuazione. Esistono due tipi di hot flow: **sharedFlow** e **stateFlow**. In questo caso a livello implementativo sia l'emissione che la collezione degli elementi funziona in maniera differente rispetto ai cold flow.

SharedFlow Uno SharedFlow rappresenta un flusso di dati condiviso che può essere letto da più collector contemporaneamente. È prevalentemente utilizzato quando si desidera condividere flussi di dati tra più parti dell'applicazione. Internamente, SharedFlow ha un *buffer* in grado di mantenere valori che saranno poi collezionati dagli osservatori. Quando uno sharedFlow viene dichiarato, è possibile settare il parametro `replay`, il quale definisce il numeri di valori precedenti che vengono emessi ai nuovi collector all'atto della sottoscrizione. Ad esempio, se `replay` è settato a 2, il flusso emetterà gli ultimi due valori ai nuovi collector. Un altro parametro che è possibile settare è `extraBufferCapacity`, il quale è utile per permettere agli osservatori più lenti di ricevere i valori senza sospendere gli emettitori. L'ultimo parametro è `onBufferOverflow`, utile per scegliere la strategia con la quale gestire l'emissione di nuovi elementi in caso di *buffer overflow*. Quando

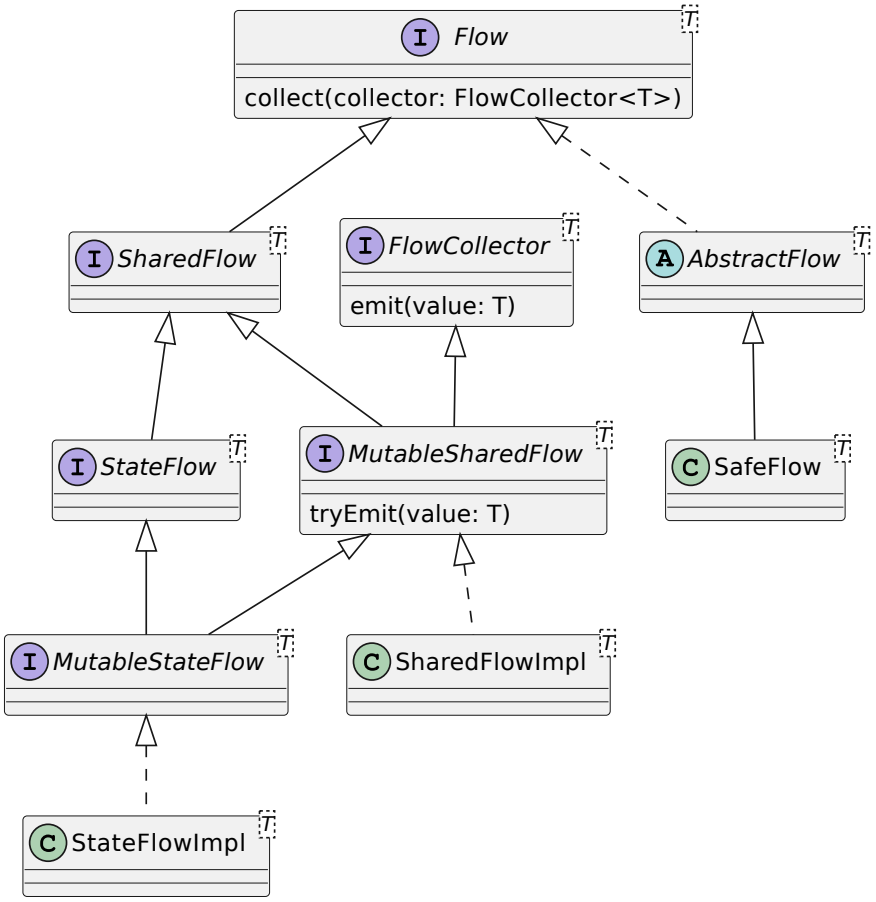

Figura 4.3: Diagramma delle classi dei Kotlin Flow.

Listing 4.3: Codice delL'implementazione del metodo collect negli sharedFlow.

```
override suspend fun collect(collector: FlowCollector<T>): Nothing {
    ...
    try {
        ...
        while (true) {
            var newValue: Any?
            while (true) {
                newValue = tryTakeValue(slot)
                if (newValue !== NO_VALUE) break
                ...
            }
            ...
        }
    } finally {
        ...
    }
}
```

un produttore invoca il metodo `emit`, il valore viene aggiunto al buffer. Il metodo `collect`, come è possibile notare nel Listato 4.3, esegue un `ciclo while` che non termina mai, il che significa che il metodo `collect` è bloccante e perciò necessita di essere eseguito in una coroutine dedicata. Ad ogni iterazione, viene controllato se esiste un nuovo valore da collezionare. Se questo esiste viene richiamato il metodo `emit` proprio come nel caso dei cold flow, descritti nella Sezione 4.3.1.

StateFlow Uno StateFlow rappresenta uno stato in sola lettura con un singolo valore aggiornabile che emette aggiornamenti ai suoi collector. È particolarmente utile quando è necessario rappresentare qualsiasi tipo di stato. A differenza dello SharedFlow, non ha un buffer interno. Lo stato viene gestito tramite un campo `value`, il quale viene sovrascritto ad ogni cambiamento di stato. Questa caratteristica impedisce ai consumatori troppo lenti di leggere tutti gli elementi emessi sul flusso. Infatti, nel caso di due emissioni consecutive sul flow, un osservatore troppo lento potrebbe accorgersi solo dell'ultimo stato. Per dichiarare uno StateFlow è necessario settare uno stato iniziale. Quando viene invocato il metodo `emit` da un produttore, in realtà quello che accade è l'aggiornamento del campo `value`. Per quando riguarda la raccolta degli elementi, possiamo notare nel Listato 4.4, che il metodo `collect` è anche in questo caso un metodo bloccante, e che per controllare la disponibilità di nuovi valori, effettua un confronto fra il vecchio stato e quello attualmente presente nel campo `value`.

Listing 4.4: odice delL'implementazione del metodo collect negli stateFlow.

```
override suspend fun collect(collector: FlowCollector<T>): Nothing {
    ...
    try {
        ...
        while (true) {
            val newState = _state.value
            ...
            if (oldState == null || oldState != newState) {
                collector.emit(NULL.unbox(newState))
                oldState = newState
            }
            ...
        }
    } finally {
        ...
    }
}
```

4.3.2 Creazione di un flusso

Esistono diversi modi per inizializzare un flow, alcune dei metodi principali:

- **cold flow**:
  - `flowOf()`: è possibile creare un flow fornendo una serie di valori come argomenti.
  - `asFlow()`: serve per convertire una collezione, come ad esempio una lista, in un flow.
  - `flow{}`: è possibile creare un flow emettendo direttamente i valori all'interno della lambda.
- **hot flow**:
  - `mutableSharedFlow<T>()`: per dichiarare uno SharedFlow occorre specificare il tipo di dato del flusso.
  - `mutableStateFlow(value)`: per inizializzare uno StateFlow è necessario fornire un valore iniziale.

Esempi di codice per inizializzare un flow sono disponibili al Listato 4.5.

Listing 4.5: Esempi di come i flow builder possono essere utilizzati per inizializzare nuovi flow.

```
val coldFlow1 = flowOf(1, 2, 3, 4, 5)


val coldfFlow2 = (1..5).asFlow()


val coldFlow3 = flow {
    (0..10).forEach {
        emit(it)
    }
}

val sharedFlow = MutableSharedFlow<Int>()

val stateFlow = MutableStateFlow(0)
```

4.3.3 Operatori

Gli operatori sono funzioni specializzate progettate per manipolare, trasformare e controllare i flussi. Gli operatori possono essere divisi in due categorie principali:

- **operatori intermediari**: permettono di manipolare il flusso, consentendo di creare pipeline di operazioni che verranno eseguite solo quando si attiva un operatore di terminazione. Gli operatori intermedi includono:
  - operatori di trasformazione come **map**, **flatmap** e **transform**;
  - operatori di filtraggio come **filter**, **distinct** e **take**;
  - operatori di combinazione come **zip** e **combine**;
  - operatori di gestione degli errori come **catch** e **retry**;
  - operatori di gestione del tempo come **debounce**;
- **operatori di terminazione**: il comportamento di un operatore di terminazione è differente per quanto riguarda hot flows e cold flows. In un cold flow ogni sottoscrizione riavvia l'emissione dei dati dall'inizio della sorgente dati. Un operatore di terminazione quindi avvia l'esecuzione del flusso. In un hot flow, i dati sono già in corso di generazione, indipendentemente dalla presenza di sottoscrizioni. In questo caso un operatore di terminazione consente di iniziare a consumare i dati da qualsiasi punto essi si trovino, senza riavviare l'emissione dall'inizio. Gli operatori di terminazione includono:

- **collect**: consuma i valori emessi dal flusso e li elabora.
- **toList, toSet, toMap**: raccolgono i valori emessi dal flusso e li convertono nella collezione desiderata.
- **reduce, fold**: riducono e aggregano rispettivamente i valori emessi dal flusso in un singolo valore.

Capitolo 5

Progettazione e implementazione del prototipo

5.1 Analisi

Trattandosi dello sviluppo di un prototipo, avente come fine ultimo quello dell'integrazione, è opportuno mantenere un modello del dominio che sia il più fedele possibile a quello del simulatore Alchemist. Tuttavia, occorre definire ulteriori vincoli affinché le tecniche di programmazione reattiva possano essere utilizzata adeguatamente.

La programmazione reattiva introduce aspetti asincroni non presenti attualmente in Alchemist. Uno degli aspetti fondamentali per una esecuzione corretta di una simulazione a step è la consistenza dello stato del sistema, uno step non può cominciare fintanto che l'ambiente allo step precedente non si trova in uno stato consistente. Questo serve a garantire che le decisioni prese in uno step siano influenzate dallo snapshot acquisito allo step precedente e non da una rappresentazione parziale di esso. Occorre inoltre considerare il vincolo di causalità fra eventi: ad ogni step l'evento che viene eseguito deve essere quello con il tempo di occorrenza minore, garantendo così la corretta sequenza di esecuzione e la coerenza temporale della simulazione. L'esecuzione di tutte le azioni di un evento devono avvenire in maniera sequenziale. Questo assicura che ciascuna azione possa prendere decisioni basandosi su uno stato consistente del sistema. La consistenza

dell'ambiente all'inizio di ciascuno step assicura anche che tutte le condizioni che un evento necessita per essere eseguito possano essere valutate basandosi su una rappresentazione fedele di esso In una simulazione ad eventi discreti un evento necessità di uno stato consistente dell'environment per aggiornare il proprio tempo. Solo quando l'ambiente è coerente, il tempo di accadimento di un evento può essere aggiornato in base alle regole definite nel modello di simulazione, assicurando che l'evoluzione di essa avvenga in modo corretto, riflettendo il comportamento del sistema che si intende simulare. La simulazione, come in Alchemist, avanza per step discreti, in ognuno dei quali viene eseguito l'evento imminente e il tempo della simulazione avanza in relazione al tempo di accadimento dell'evento.

Come anticipato, si è deciso di rimanere fedeli al modello del dominio presente in Alchemist, astraendo però dalla metafora chimica descritta in Sezione 2.11. A seguito di questa scelta, nel modello del dominio del prototipo non esiste il concetto di **reazione**, il quale viene sostituito con il concetto di **evento**. Non esiste nemmeno il concetto di **molecola**, il quale è sostituito da un generico **content**. Restano invariati invece gli altri concetti. Come si può notare in figura, l'**environment** contiene al suo interno un insieme di **nodi** e tutti i **vicinati**, i quali vengono creati grazie ad una regola di collegamento (**linking rule**). L'environment è anche responsabile di mantenere la **posizione** di ciascun nodo, consentirne l'aggiunta, la rimozione e lo spostamento. Ogni nodo ha al suo interno **eventi**, i quali possono verificarsi ad un determinato istante, e **content**, ovvero contenitori di valori. Ogni evento a sua volta contiene **condizioni**, le quali devono essere soddisfatte per garantire l'esecuzione dell'evento stesso, e le **azioni** che verranno eseguite a seguito della sua esecuzione. Ogni evento ha anche un **tempo putativo**, ovvero il tempo al quale è schedulata la sua prossima occorrenza. È possibile osservare il modello appena descritto in Figura 5.1.

È prevista anche la presenza di un **engine**, ovvero il motore della simulazione, responsabile di far partire la simulazione e farla avanzare, e di uno **scheduler**, responsabile di mantenere una lista degli eventi futuri, ordinati per tempo di accadimento.

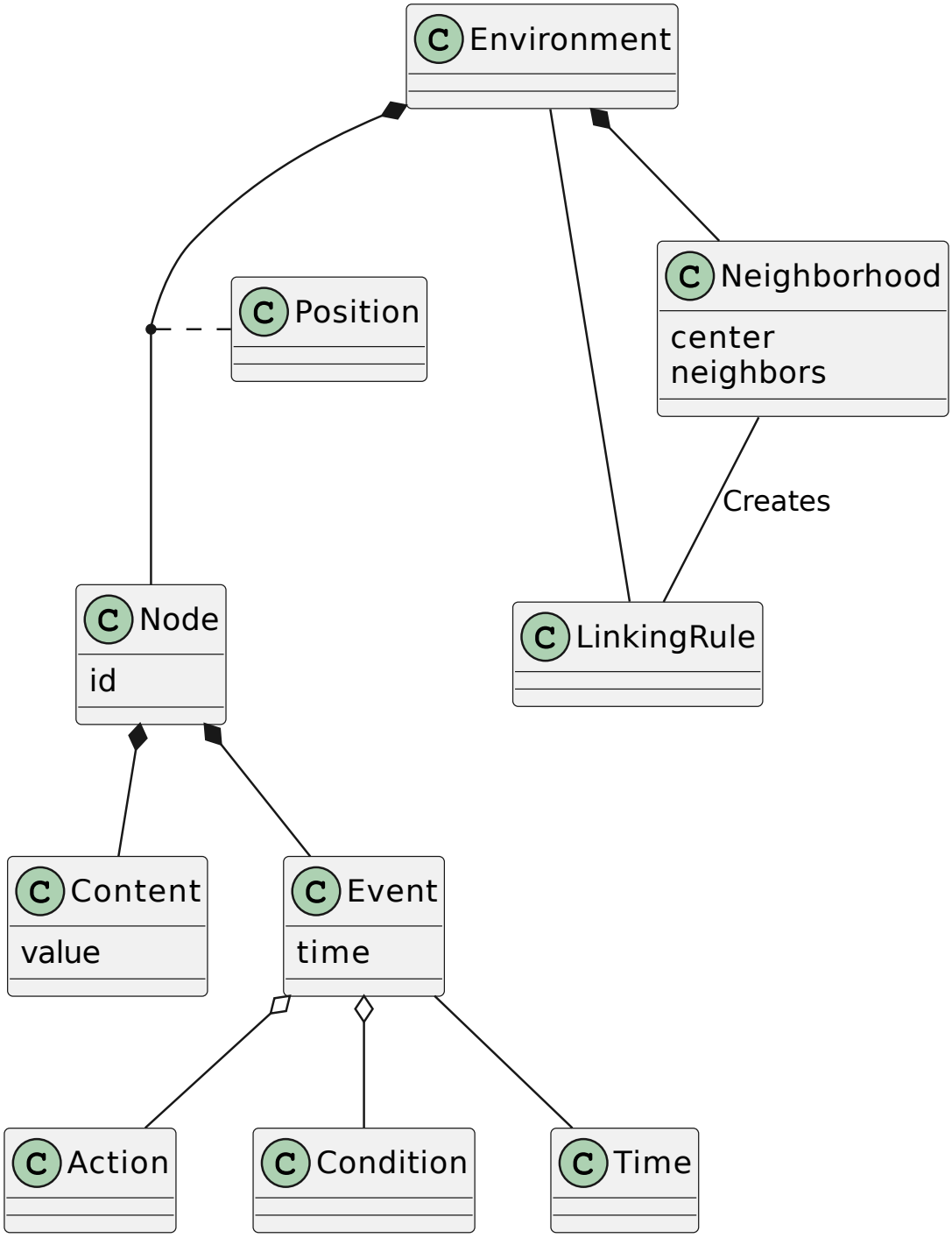

Figura 5.1: Modello del dominio modellato attraverso il diagramma delle classi di analisi.

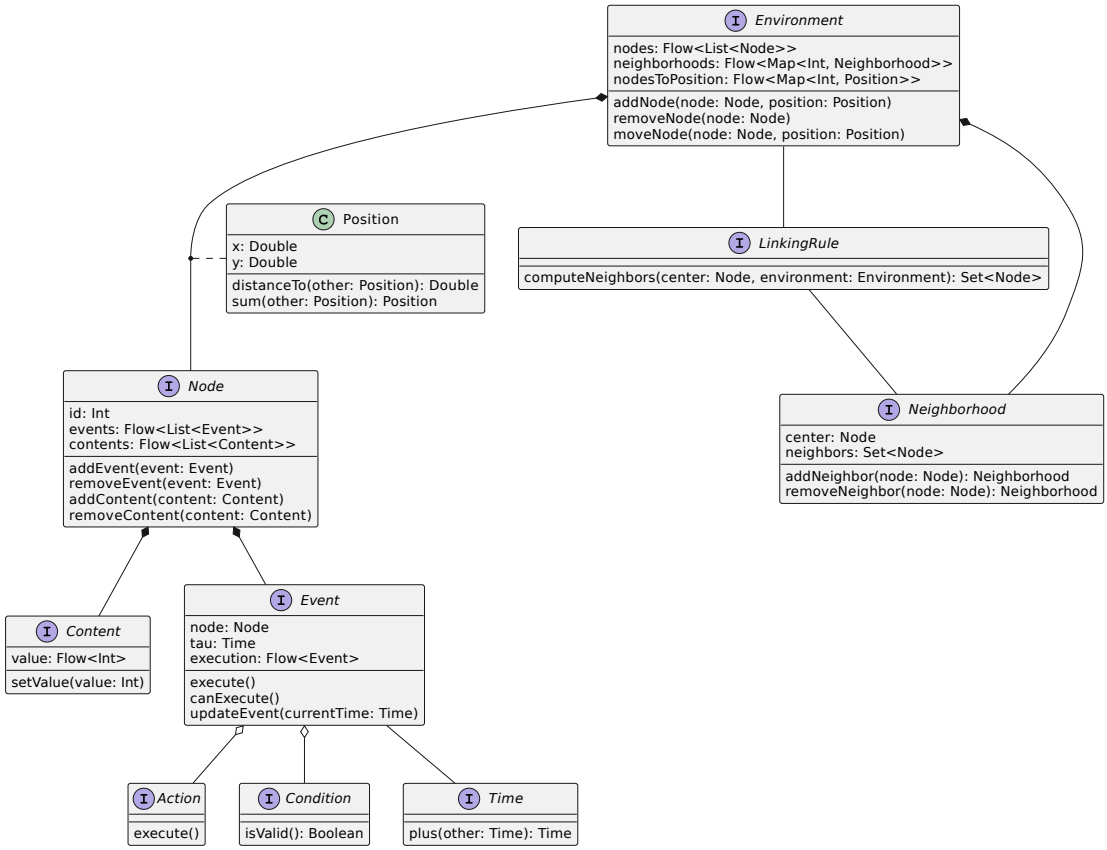

Figura 5.2: Diagramma delle classi del prototipo emerso in fase di progettazione.

5.2 Progettazione

In Figura 5.2 è possibile notare il diagramma delle classi emerso in fase di progettazione. Come è possibile notare, l'environment consente di poter osservare i vicinati, i nodi, e le posizioni di essi. Anche i nodi consentono di osservare gli eventi e i contenuti al loro interno. Il singolo contenuto consente di osservare il proprio valore. Tutto all'interno dell'ambiente deve essere osservabile. Questo garantisce che la linking rule possa osservare elementi arbitrari all'interno dell'ambiente per formare i vicinati. Ciascun evento emette un valore sul proprio flusso quando viene eseguito. Gli eventi dipendenti da esso, ricevendo notifica dell'esecuzione, possono aggiornare il proprio tempo di occorrenza. Ulteriori considerazioni relative alla progettazione verranno discusse in questa sezione.

5.2.1 Engine e Scheduler

L'*engine* è responsabile di far partire la simulazione e del suo avanzamento. In fase di inizializzazione deve aggiornare il tempo putativo degli eventi presenti nell'*environment*, dopodiché li aggiunge allo *scheduler*. Una volta terminata la fase di inizializzazione fa partire l'effettiva simulazione, la quale termina nel caso in cui sia stato raggiunto il limite massimo di step eseguibili o non ci sia più nessun evento programmato per essere eseguito. Ad ogni step della simulazione, l'*engine*:

1. preleva, se presente, il prossimo evento da eseguire dallo scheduler;
2. aggiorna il tempo della simulazione con il tempo dell'evento da eseguire;
3. controlla che le condizioni dell'evento da eseguire siano soddisfatte;
4. esegue effettivamente l'evento;
5. aggiorna il tempo putativo dell'evento appena eseguito;
6. comunica allo scheduler di ordinare la lista di eventi futuri, poiché i tempi putativi di alcuni eventi potrebbero essere cambiati.

Il funzionamento appena descritto e osservabile in Figura 5.3.

Si noti che, a seguito della chiamata per eseguire l'evento, il controllo deve ritornare all'engine solo nel momento in cui tutte le azioni dell'evento sono state effettivamente eseguite, tutte le dipendenze fra eventi sono state aggiornate e gli eventi dipendenti da quello eseguito hanno aggiornato il proprio tempo putativo.

Lo scheduler deve mantenere una lista degli eventi futuri ordinati per tempo di accadimento. Oltre a ciò, deve fornire la possibilità di rimuovere eventi dalla propria lista e di aggiungerne di nuovi.

5.2.2 Eliminazione grafo delle dipendenze

Come anticipato in sezione Sezione 2.11, Alchemist gestisce le dipendenze fra gli eventi utilizzando un grafo delle dipendenze, nel quale i nodi sono le reazioni (eventi) e gli archi connettono una reazione r a tutti i nodi che dipendono da essa, ovvero quelli il cui tempo di attivazione deve essere aggiornato a seguito dell'esecuzione

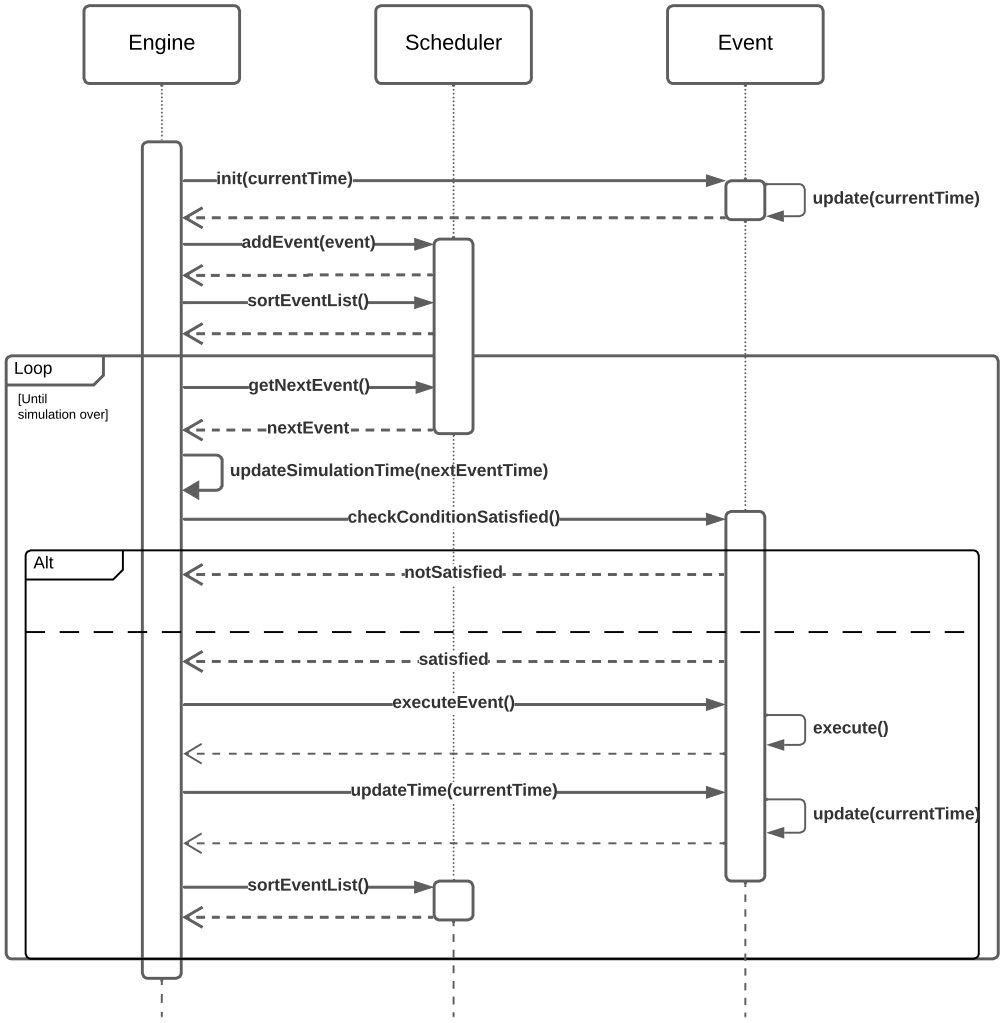

Figura 5.3: Diagramma di sequenza che modella l'inizializzazione e l'avanzamento degli step nella simulazione.

di r. L'utilizzo di un grafo delle dipendenze in un simulatore ad eventi discreti per l'aggiornamento degli eventi dipendenti richiede elevata capacità di calcolo, specialmente quando il sistema presenta una complessità elevata o un grande numero di eventi da gestire. Per affrontare la complessità computazionale derivante dalla gestione di un grafo delle dipendenze è stato scelto di eliminare tale grafo, optando invece per un approccio basato sulla programmazione reattiva. In questo contesto, ogni evento osserva gli eventi da cui dipende e reagisce di conseguenza aggiornando il proprio tempo di occorrenza, riducendo così la necessità di una gestione esplicita delle dipendenze tramite grafo. Un evento può osservare gli altri eventi presenti all'interno dello stesso nodo e gli eventi presenti nei nodi del vicinato. Utilizzando la programmazione reattiva, gli eventi possono essere modellati come flussi di dati che si propagano attraverso il sistema. Ogni evento può essere definito come un'unità indipendente che reagisce ai cambiamenti negli eventi osservati, senza la necessità di mantenere un grafo delle dipendenze centralizzato. Poiché eventi e nodi possono essere rimossi dall'environment e i vicini possono cambiare in base alla regola di collegamento scelta, è fondamentale riconoscere che l'insieme di eventi osservati da un singolo evento può variare durante l'esecuzione della simulazione. Pertanto, oltre agli eventi da cui dipende direttamente, un evento deve essere in grado di reagire ai cambiamenti nell'insieme degli eventi presenti nel suo nodo, nonché ai cambiamenti nei nodi del suo vicinato.

5.2.3 Gestione dei cambiamenti nell'ambiente

Come anticipato in precedenza, la creazione dei vicinati è affidata alla *linking rule*, la quale ne definisce le regole per la creazione. La creazione dei vicinati può essere eseguita osservando la posizione dei nodi nell'*environment*, i valori dei content all'interno dei nodi stessi, o altre parti arbitrarie dell'environment. Per far fronte alla complessità della creazione e dell'aggiornamento dei vicinati, è stato scelto di adottare un approccio basato sulla programmazione reattiva. Ciò significa che la linking rule è implementata in modo da osservare reattivamente l'ambiente circostante e reagire di conseguenza agli eventi che coinvolgono i nodi. In pratica, la linking rule monitora ciò di cui ha bisogno per la creazione dei vicinati e osserva l'insieme dei nodi presenti nell'environment in modo da rilevarne eventuali mo-

difiche come l'aggiunta o la rimozione e aggiornare di conseguenza i vicinati dei nodi interessati. Potrebbe essere necessario osservare altri attributi dei nodi, come i valori contenuti al loro interno, per definire in modo più preciso i vicinati. Per esempio, se i vicinati vengono definiti in base alla loro posizione, occorre osservarne anche il cambiamento di posizione durante la simulazione.

5.2.4 Criticità nell'uso degli Hot Flow in simulazioni a step

In una simulazione a step è essenziale garantire uno stato consistente prima di procedere allo step successivo al fine di garantire risultati accurati ed affidabili. L'utilizzo degli hot flow, con la loro natura asincrona e non deterministica, rende difficile garantire questa consistenza dello stato, poiché non esiste un momento definito in cui tutti gli osservatori hanno elaborato i dati e lo stato è completamente aggiornato. Come anticipato in Sezione 4.3.1, la chiamata alla funzione `collect`, la quale controlla costantemente la presenza di nuovi elementi da consumare, non termina mai ed è quindi necessario eseguire ciascuna di esse in una coroutine dedicata. Questa coroutine, rimarrà attiva fino al termine della simulazione, o fino a quando l'osservatore necessiterà dei dati emessi su tale flusso. Quando un nuovo elemento viene emesso su un flusso hot, non è possibile controllare con precisione quando tutti gli osservatori hanno finito di elaborarlo; come è possibile notare in Figura 5.4, il controllo ritorna all'entità che emette sul flusso non appena tutti gli osservatori hanno ricevuto notifica della presenza dell'elemento. Ci si potrebbe quindi trovare nella situazione in cui inizia un nuovo step senza che gli aggiornamenti degli eventi dipendenti o la creazione dei nuovi vicinati sia stata effettivamente terminata. Ogni osservatore, oltretutto, può a sua volta reagire alla presenza di un elemento su un flusso, emettendo a sua volta elementi su un altro flusso, creando così un albero di chiamate asincrone che può diventare complesso da gestire e sincronizzare.

Nel contesto di questo prototipo, il termine dello step non è il solo punto in cui è necessaria consistenza nello stato del sistema. L'occorrenza di un evento scatena l'esecuzione di una o più azioni, le quali potrebbero necessitare dello stato attuale dell'ambiente per prendere decisioni su come modificarlo. Un altro punto nel quale è necessaria consistenza è quando un evento intende notificare gli eventi da esso

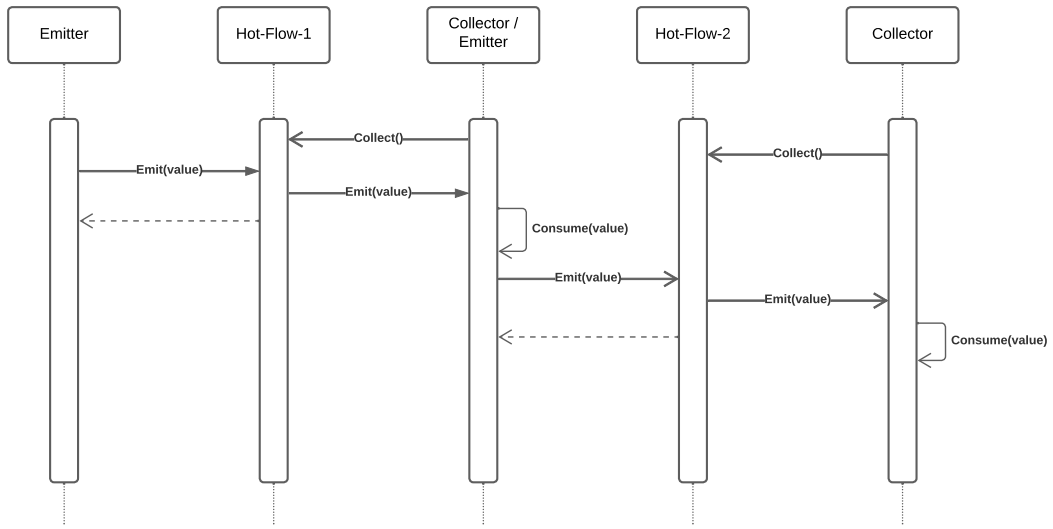

Figura 5.4: Diagramma di sequenza che mostra come il controllo a seguito di un emissione su un flow ritorni all'emettitore senza possibilità di controllare quando l'elemento viene effettivamente consumato.

dipendenti della sua effettiva esecuzione. Qui la consistenza è necessaria poiché le azioni eseguite potrebbero aver modificato la topologia dei vicinati, portando quindi ad una variazione nella lista degli eventi dal quale ogni singolo evento dipende. La garanzia che tutti gli eventi dipendenti abbiano aggiornato il proprio tempo putativo prima di proseguire con lo step successivo è fondamentale per garantire che la lista di eventi futuri sia ordinata nel modo corretto. Quando si gestisce una simulazione a step, è essenziale mantenere un controllo accurato del tempo e degli eventi che devono accadere in specifici momenti temporali. Se un evento dovesse aggiornare il suo tempo dopo la fine dello step in cui dovrebbe effettivamente farlo, potrebbe causare un problema di inconsistenza temporale nell'ambiente di simulazione. Si immagini di avere un evento programmato per accadere in un certo momento temporale all'interno di uno step. Dopo la fine di questo step, l'evento potrebbe aggiornare il suo tempo per essere pianificato per il prossimo step. Tuttavia, se questo aggiornamento non avviene correttamente, potrebbe verificarsi una situazione in cui, al successivo step, l'evento si ritrovi nella lista degli eventi futuri con un tempo minore rispetto al tempo corrente.

5.3 Implementazione

Nel contesto di una simulazione è importante che gli osservatori ricevano tutti lo stesso stato al fine di garantire la coerenza dei risultati. Si è deciso quindi di utilizzare flow di tipo hot, descritti in Sezione 4.3.1. Come descritto precedentemente, i possibili hot flow utilizzabili nella libreria Kotlin Flow sono **MutableSharedFlow** e **MutableStateFlow**. Si è scelto di utilizzare i primi nel caso in cui il flusso dovesse rappresentare lo stato attuale di una determinata risorsa, mentre i secondi sono stati utilizzati nel caso in cui si volesse comunicare un determinato evento che è accaduto all'interno della simulazione. Le scelte in dettaglio verranno analizzate successivamente nel capitolo. Il codice del simulatore è disponibile su Github [1].

5.3.1 Soluzione al problema degli hot flow in simulazioni a step

Come spiegato in Sezione 5.2.4, il problema principale risiede nel fatto che la libreria Kotlin Flow non mette a disposizione meccanismi per controllare l'avvenuto consumo di un elemento pubblicato sul flusso. Si è deciso quindi di implementare un meccanismo per risolvere questa necessità. L'idea di base è fare in modo che a seguito di ogni emissione sul flusso, si attenda una notifica da parte degli osservatori dell'avvenuta consumazione dell'elemento. Così facendo, il controllo torna all'entità che ha emesso sul flusso solo quando tutti gli osservatori hanno effettivamente consumato l'elemento. Utilizzando questo approccio non occorre preoccuparsi di possibili alberi di chiamate asincrone, poiché, nel caso in cui un osservatore emettesse elementi a sua volta su un ulteriore flusso, anch'esso aspetterebbe le notifiche dai consumatori di tale flusso, e il controllo tornerebbe all'emettitore alla radice solo quando tutto l'albero è stato risolto con successo. Per fare ciò, si sono utilizzate due classi ad-hoc: *AwaitableMutableStateFlow* e *AwaitableMutableSharedFlow*, il cui codice è disponibile al Listato 5.1. Le suddette classi implementano rispettivamente le interfacce *MutableStateFlow* e *MutableSharedFlow*, delegando però l'effettiva implementazione a *StateFlowImpl* e *SharedFlowImpl*. Entrambe le

[1] `https://github.com/giacomoaccursi/Reactive-DES`

classi implementate eseguono l'*override* del metodo `emit`, aggiungendo la logica per attendere le notifica descritta sopra, e definiscono il metodo `notifyConsumed`, utilizzato dall'osservatore per comunicare l'effettivo consumo dell'elemento. La logica per attendere il consumo degli elementi emessi è incapsulata all'interno del flow stesso, l'attesa è perciò trasparente lato emettitore. È stata creata anche una classe *AwaitableMutableFlow*, la quale è estesa da entrambe le classi implementate, per evitare ripetizione di codice. In Figura 5.5 è possibile notare come le classi implementate appena descritte si interfacciano con le classi definite nella libreria Kotlin Flow.

5.3.2 Gestione reattiva dipendenze fra eventi

In seguito si intende descrivere come tramite la programmazione reattiva si è implementata la gestione delle dipendenze fra eventi. Come anticipato, ogni evento è dipendente dagli eventi che accadono all'interno dello stesso nodo e dagli eventi che accadono nei nodi del vicinato. La natura di questo insieme di eventi è dinamica, poiché durante la simulazione nodi ed eventi possono essere rimossi e la topologia del vicinato può cambiare. Ciascun evento, tiene traccia degli eventi da cui dipende tramite l'utilizzo di due mappe, entrambe aventi come chiave l'evento osservato e come valore il `job` associato alla coroutine nella quale è stata avviata l'osservazione. La prima mappa, *observedLocalEvents*, tiene traccia degli eventi locali al nodo, mentre la seconda tiene traccia degli eventi appartenenti ai nodi del vicinato. Come si può notare nel Listato 5.2, ciascun evento si mette in ascolto in attesa di cambiamenti nella lista degli eventi presenti nel nodo in cui risiede. Ad ogni cambiamento, inizia ad osservare nuovi eventi aggiunti e smette di osservare gli eventi che sono stati rimossi. Nel Listato 5.3, invece, è consultabile il codice riferito all'osservazione degli eventi presenti nei nodi facenti parte il vicinato.

5.3.3 Gestione reattiva aggiornamento dell'environment

Come specificato in Sezione 5.2.3, per aggiornare i vicinati, la *linking rule* deve osservare i cambiamenti nella lista dei nodi presenti all'interno dell'environment e, a seconda della strategia di creazione dei vicinati scelta, altre parti arbitrarie dell'environment. All'interno del prototipo è possibile trovare l'implementazione

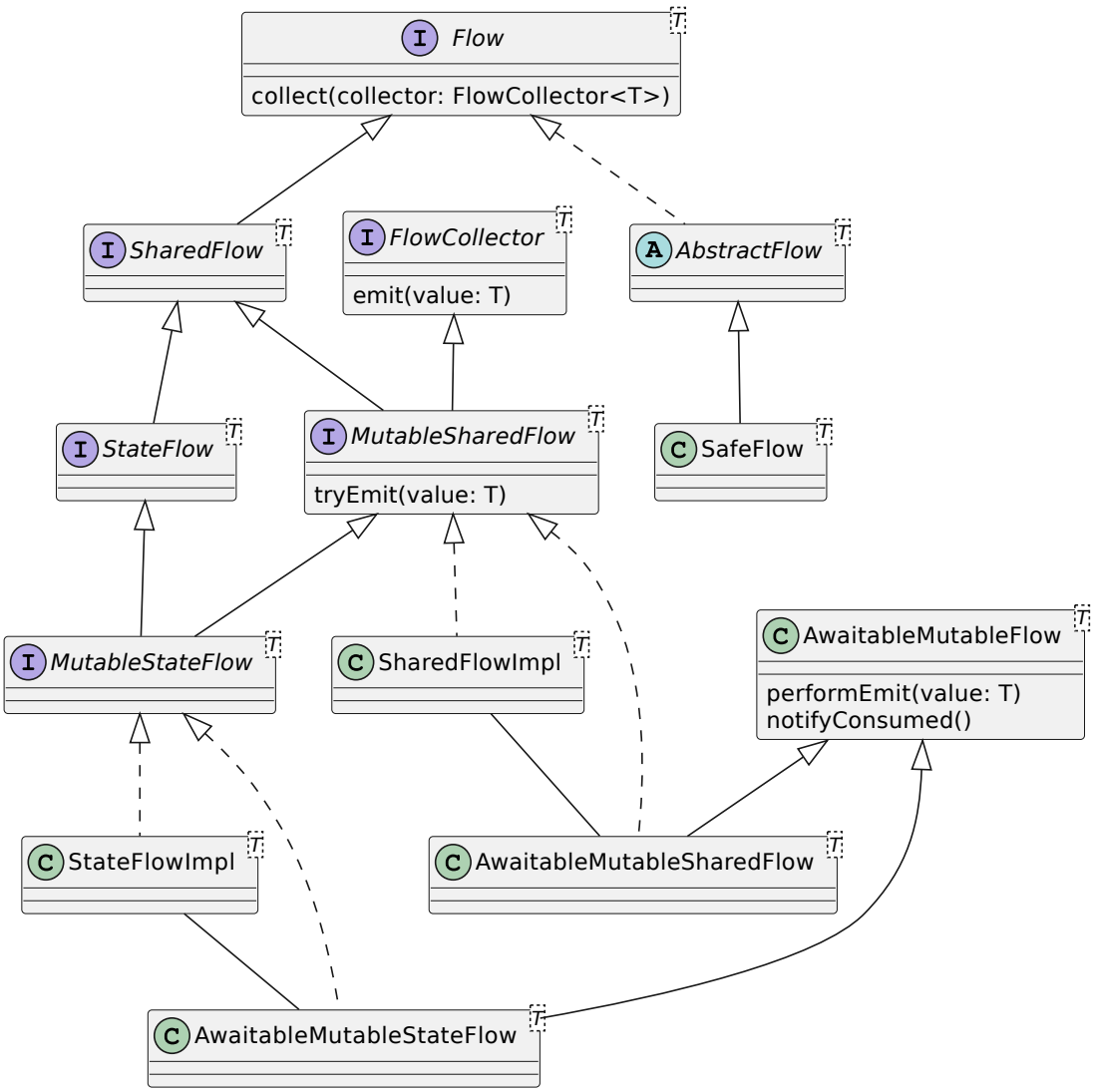

Figura 5.5: Diagramma delle classi dei Kotlin Flow estesa con le classi implementate ad-hoc.

Listing 5.1: Implementazione delle classi *AwaitableMutableFlow*.

```
abstract class AbstractAwaitableMutableFlow<T>(
    private val flow: MutableSharedFlow<T>,
    private val ioDispatcher: CoroutineContext = Dispatchers.IO,
) {
    private var emitLatch: CountDownLatch = CountDownLatch(0)
    /**
     * Perform emit waiting for notification.
     */
    suspend fun emitAndWait(value: T) {
        emitLatch = CountDownLatch(flow.subscriptionCount.value)
        flow.emit(value)
        withContext(ioDispatcher) {
            emitLatch.await()
        }
        emitLatch = CountDownLatch(0)
    }
    /**
     * A function for notifying the consumption.
     */
    fun notifyConsumed() {
        emitLatch.countDown()
    }
}

class AwaitableMutableSharedFlow<T>(
    private val sharedFlow: MutableSharedFlow<T>,
) : AbstractAwaitableMutableFlow<T>(sharedFlow), MutableSharedFlow<T> by
    sharedFlow {

    override suspend fun emit(value: T) {
        this.emitAndWait(value)
    }
}

class AwaitableMutableStateFlow<T>(
    private val stateFlow: MutableStateFlow<T>,
) : AbstractAwaitableMutableFlow<T>(stateFlow), MutableStateFlow<T> by stateFlow {

    override suspend fun emit(value: T) {
        this.emitAndWait(value)
    }
}
```

Listing 5.2: Codice utilizzato per l'osservazione degli eventi locali al nodo.

```
private fun observeLocalEvents() {
    coroutineScope.launch {
        node.events.run {
            this.onSubscription { initLatch.countDown() }.collect {
                // Events that no longer belong to the neighborhood are identified.
                val removed = observedLocalEvents.keys - it.toSet() - setOf(this@EventImpl)
                // Events that are now part of the neighborhood are identified.
                val added = it.toSet() - setOf(this@EventImpl) - observedLocalEvents.keys
                // Observations of events on which it no longer depends are deleted.
                removed.forEach { event ->
                    observedLocalEvents[event]?.cancelAndJoin()
                    observedLocalEvents.remove(event)
                }
                // A new observation is started for each new event on which it is dependent.
                added.forEach { event ->
                    val job = launch {
                        val executionFlow = event.observeExecution()
                        executionFlow.run {
                            this.collect { event ->
                                // Following the execution of an event on which it is dependent, the execution time must be updated.
                                updateEvent(event.tau)
                                // Notification to the flow of actual consumption.
                                this.notifyConsumed()
                            }
                        }
                    }
                    observedLocalEvents[event] = job
                }
                // Notification to the flow of actual consumption.
                this.notifyConsumed()
            }
        }
    }
}
```

Listing 5.3: Codice utilizzato per l'osservazione degli eventi presenti nei nodi del vicinato

```
private fun observeNeighborEvents() {
    coroutineScope.launch {
        environment.neighborhoods.run {
            this.onSubscription {
                initLatch.countDown()
            }.mapLatest {
                // The updated neighborhood is mapped to a set of nodes.
                it[node.id]?.neighbors.orEmpty()
            }.collect { neighbors ->
                // Stop to observe the event that no longer belong to the
                    neighborhood.
                stopToObserveOldNeighbors(neighbors)
                // Events that are now part of the neighborhood are identified and
                     observed.
                val addedNeighbors = neighbors - observedNeighbors.keys
                addedNeighbors.forEach { node ->
                    val job = launch {
                        node.events.run {
                            this.collect { events ->
                                stopToObserveRemovedEvents(events)
                                val added = events.toSet() -
                                    observedNeighborEvents[node]?.keys.orEmpty()
                                added.forEach { event ->
                                    val job = launch {
                                        val executionFlow = event.observeExecution
                                            ()
                                        executionFlow.run {
                                            this.collect { ev ->
                                                // Following the execution of an
                                                    event on which it is dependent
                                                    , it need to update its
                                                    execution time.
                                                updateEvent(ev.tau)
                                                // Notification to the flow of
                                                    actual consumption.
                                                this.notifyConsumed()
                                            }
                                        }
                                    }
                                    observedNeighborEvents[node]?.set(event, job)
                                }
                                this.notifyConsumed()
                            }
                        }
                    }
                    observedNeighbors[node] = job
                    observedNeighborEvents[node] = hashMapOf()
                }
                this.notifyConsumed()
            }
        }
    }
}
```

Listing 5.4: Codice utilizzato per l'osservazione dei nodi presenti all'interno dell'environment e della loro posizione.

```
private fun observeNodes() {
    startToObserveFlow(environment.nodesToPosition)
}

private fun observeNodesPosition() {
    startToObserveFlow(environment.nodes)
}

private fun startToObserveFlow(flow: AwaitableMutableStateFlow<*>) {
    coroutineScope.launch {
        flow.run {
            this.onSubscription {
                initLatch.countDown()
            }.collect {
                // The new neighborhoods are recalculated based on the actual
                    nodes and their positions.
                val newNeighborhoods = environment.getAllNodes().associate { node
                    ->
                    node.id to SimpleNeighborhood(node, computeNeighbors(node,
                        environment))
                }
                environment.updateNeighborhoods(newNeighborhoods)
                // Notification to the flow of actual consumption.
                this.notifyConsumed()
            }
        }
    }
}
```

di una linking rule che crea i vicinati in base alla posizione dei nodi nell'environment. Ciò non toglie la possibilità di aggiungere altre strategie al simulatore e scegliere ad ogni simulazione la strategia desiderata. Nel Listato 5.4 è presente il codice con il quale la classe *PositionLinkingRule* osserva la lista di nodi presenti nell'environment e la loro posizione all'interno di esso, ricreando ogni volta i vicinati corretti.

5.3.4 Fase di inizializzazione degli osservatori

Utilizzando flussi hot, occorre una fase di sincronizzazione iniziale nella la quale vi sia la certezza che al termine della creazione dell'istanza di un osservatore, qualunque esso sia, esso sia pronto immediatamente a collezionare gli elementi. Questa necessità è data dal fatto che, come descritto in precedenza, ogni osservazione viene eseguita all'interno di una coroutine separata, e pertanto il controllo

del flusso torna immediatamente al chiamante, il quale potrebbe emettere elementi prima che l'osservatore sia effettivamente pronto per collezionarli. Per ovviare a tale problema si è fatto uso del metodo `onSubscription`, fornito da *SharedFlow* e *StateFlow*. Il metodo in questione viene invocato quando la connessione al flow da parte dell'osservatore è stata effettivamente stabilita e consente di definire azioni da eseguire in quel determinato momento. In fase di inizializzazione dell'osservatore viene inizializzato un *CountDownLatch*, il cui valore è pari al numero di flow su cui l'osservatore dovrà mettersi in ascolto. Al momento di ogni effettiva sottoscrizione viene decrementato il latch. Così facendo si ha la certezza che una volta terminata l'inizializzazione di un osservatore, questo sia effettivamente pronto per consumare gli elementi emessi sul flow.

5.3.5 Engine e scheduler

Il codice utilizzato per al gestione di un singolo step della simulazione è consultabile al Listato 5.5. In questa fase prototipale è stato scelto di implementare lo *scheduler* di eventi attraverso una semplice lista. Se il prototipo dovesse poi essere integrato nel simulatore Alchemist, questa soluzione sarebbe limitante dal punto di vista delle performance poiché ad ogni step viene effettuato un ordinamento completo della lista di eventi, in tal caso si renderebbe necessario l'utilizzo di meccanismi più efficienti, come quello già presente in Alchemist, descritto in Sezione 2.11.3.

5.4 Visualizzazione aggiornamento dipendenze fra eventi

Al fine di dimostrare l'effettivo funzionamento del prototipo sviluppato e mostrare che è possibile gestire le dipendenze fra eventi attraverso l'uso della programmazione reattiva, eliminando la necessità del grafo delle dipendenze, è stato creato un semplice esportatore che si occupa, ad ogni step, di collezionare lo snapshot dell'ambiente al tempo corrente. Ogni snapshot contiene lo step e il tempo correnti, i nodi dell'environment e le loro relazioni in termini di vicinato, le dipendenze di ciascun evento e la lista di eventi futuri con relativo tempo di accadimento. Attraverso un piccolo programma scritto in codice Python, utilizzando le librerie *Mat-*

Listing 5.5: Codice per l'esecuzione di un singolo step all'interno della simulazione.

```
private suspend fun doStep() {
    val nextEvent = scheduler.getNext()
    if (nextEvent == null) {
        status = Status.TERMINATED
    } else {
        val scheduledTime = nextEvent.tau
        if (scheduledTime.toDouble() < currentTime.toDouble()) {
            error("next event is scheduled in the past")
        }
        currentTime = scheduledTime
        if (nextEvent.canExecute()) {
            nextEvent.execute()
            nextEvent.updateEvent(currentTime)
            scheduler.eventsUpdated()
        }
    }
    currentStep += 1
}
```

plotlib e *NetworkX*, è stata creata una rappresentazione grafica di tutti gli snapshot esportati, consentendo di visualizzare le dipendenze in modo più intuitivo.

In Figura 5.6 si può notare come lo snapshot allo step 0, ovvero il momento in cui i nodi e gli eventi sono stati aggiunti all'environment ma la simulazione non è ancora partita effettivamente. In alto è visualizzato lo step corrente e il tempo di simulazione corrente. Il grafico di sinistra mostra i nodi, connessi con una linea continua con i nodi facenti parte il vicinato. In questo step tutti i nodi sono vicini fra loro, quindi tutti i nodi hanno una linea che li connette con tutti gli altri. Il fatto che tutti i nodi siano vicini è un caso particolare della simulazione in questione e non un vincolo. Il grafico di destra mostra invece le dipendenze fra eventi. Ogni evento ha una freccia uscente verso gli eventi da cui dipende. Dato che nel prototipo sviluppato gli eventi sono dipendenti dagli eventi presenti nel proprio nodo e nei nodi vicini, è possibile notare come tutti gli eventi inizialmente siano dipendenti da tutti quelli presenti nell'environment. Il colore di ciascun evento corrisponde al colore del nodo nel quale risiede. In basso invece è visualizzata la lista degli eventi futuri, ordinata per tempo di accadimento.

In Figura 5.7 è possibile visualizzare l'evoluzione del sistema allo step 1 a seguito dell'esecuzione dell'evento 5, ovvero l'evento con il tempo di accadimento più basso. Il tempo viene fatto avanzare e corrisponde al tempo putativo dell'evento eseguito. Come si può notare, il nodo 1 non è più vicino degli altri nodi, ragion

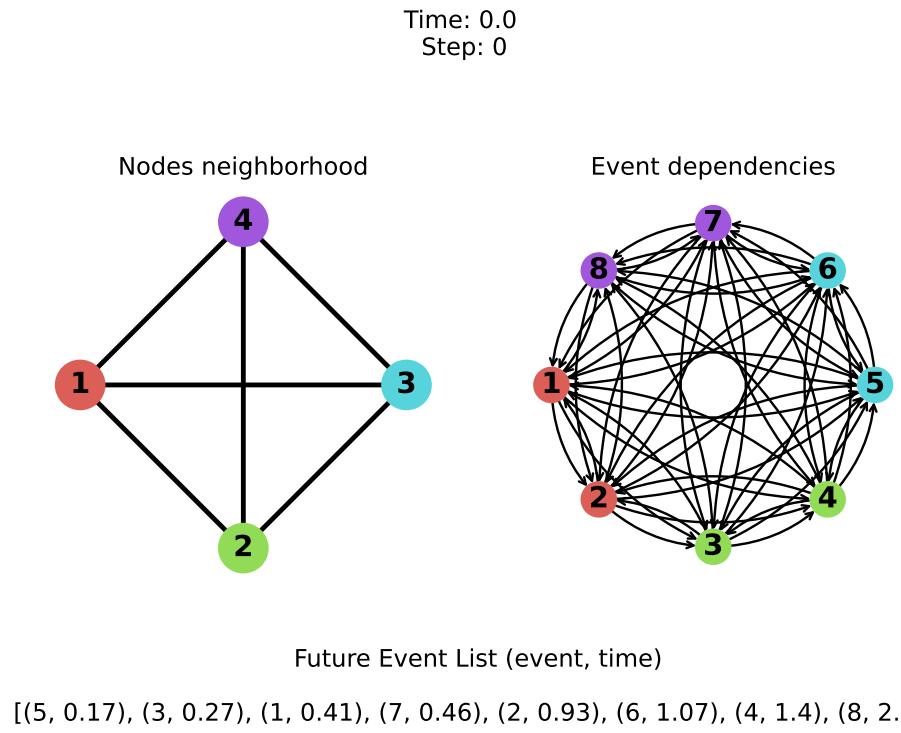

Figura 5.6: Rappresentazione grafica dello snapshot dell'ambiente allo step 0.

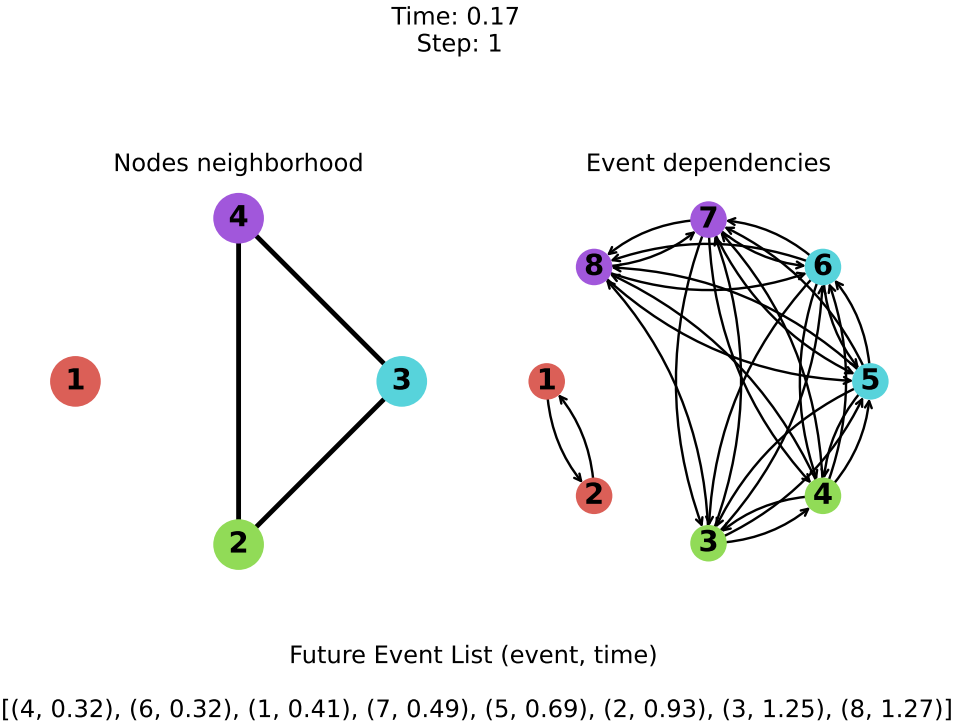

Figura 5.7: Rappresentazione grafica dello snapshot dell'ambiente allo step 1.

per cui gli eventi presenti al suo interno non hanno più relazioni di dipendenza con gli eventi presenti negli altri nodi. A seguito dell'esecuzione dell'evento, gli eventi dipendenti hanno aggiornato il proprio tempo di occorrenza e la lista è stata riordinata opportunamente.

5.5 Considerazioni sui risultati raggiunti

Il prototipo sviluppato [Acc24] è stato concepito con l'intento di essere poi integrato all'interno del simulatore Alchemist. Prima di immergersi nella fase di sviluppo del codice per Alchemist, è stato ritenuto fondamentale dedicare del tempo alla valutazione preliminare della fattibilità di tale progetto. Questo approccio ha permesso di acquisire una comprensione approfondita dei requisiti tecnici, delle possibili criticità e delle opportunita associate all'integrazione. Tuttavia, dato che

si tratta di un prototipo, è importante sottolineare che esso astrae la complessità presente in Alchemist. Questo significa che al momento risultata difficile effettuare un confronto diretto fra la soluzione presentata e il simulatore. Eventuali valutazioni delle performance dovranno essere effettuate una volta completata l'integrazione. L'implementazione della programmazione reattiva, caratterizzata da un flusso di lavoro asincrono, ha portato alla luce alcune criticità quando è stata combinata con la simulazione a step. In una simulazione a step, è fondamentale mantenere la consistenza dell'ambiente, specialmente prima di procedere allo step successivo. La natura asincrona della programmazione reattiva può causare situazioni in cui gli eventi vengono elaborati in tempi variabili, rendendo difficile garantire che lo stato dell'ambiente sia coerente e stabile. Si potrebbe definire l'approccio adottato per risolvere tale problema come un "workaround". Questo perché la programmazione reattiva mira a gestire gli eventi in modo asincrono, senza dover attendere il completamento di un'operazione prima di procedere con altre attività. Aspettare che tutti gli osservatori consumino un elemento introduce una dipendenza sincrona che può rallentare l'esecuzione complessiva del programma.

5.5.1 Lavori futuri

Trattandosi di un prototipo, avente come obiettivo l'analisi della fattibilità della creazione di un simulatore ad eventi discreti le cui dipendenze e aggiornamento dell'environment avvenissero in maniera reattiva, alcuni aspetti legati soprattutto alle performance sono stati trascurati. Questi aspetti, in sede di integrazione con il simulatore Alchemist dovranno sicuramente essere presi in considerazione per la riuscita del progetto. Uno dei punti che si è scelto di trascurare è la creazione dei vicinati da parte della *linking rule*. Attualmente essa, ad ogni cambiamento al quale è interessata, ricrea tutti i vicinati a partire dalle informazioni dei nodi dell'environment. Ovviamente questa scelta non è ottimale dal punto di vista delle performance. Per migliorare ciò si potrebbe prendere in considerazione l'aggiornamento dei vicinati solo dei nodi interessati. Questo miglioramento richiede che la linking rule mantenga internamente uno stato corrente di ciò a cui è interessata, e una volta ricevuto l'aggiornamento, capisca autonomamente quali sono stati i cambiamenti. Un'altra implementazione piuttosto "naive" è quella dello schedu-

ler, responsabile di mantenere una lista ordinata dei prossimi eventi da eseguire. Utilizzare una semplice lista, che verrà riordinata ad ogni step, porta a problemi di performance, soprattutto in casi in cui il numero di eventi futuri è piuttosto elevato. Implementazioni come quella attualmente presente nel simulatore Alchemist sono sicuramente più efficienti.

Capitolo 6

Conclusioni

In questa tesi è stata eseguita una panoramica sulle simulazioni ad eventi discreti, esplorando le loro fondamenta teoriche, metodologie di progettazione e implementazione, nonché le principali applicazioni in diversi settori. Inoltre, è stata analizzata e approfondita la programmazione reattiva, sia nel contesto generale che nel contesto specifico dell'applicazione al linguaggio Kotlin. Il prodotto della tesi condotta è la prototipazione di un simulatore ad eventi discreti reattivo. Il prototipo segue la modellazione del dominio presente in Alchemist, nel quale dovranno poi essere integrate le funzionalità sviluppate, eliminando la necessità del grafo delle dipendenze fra eventi, utilizzando tecniche di programmazione reattiva al fine di rendere implicite queste dipendenze. La programmazione reattiva è stata utilizzata anche per l'aggiornamento dei vicinati all'interno dell'ambiente. Per raggiungere tale scopo è stata necessaria una estensione delle funzionalità messe a disposizione dalla libreria Kotlin Flow, al fine di gestire in modo adeguato la pubblicazione e il consumo degli elementi sul flusso. Il prototipo sviluppato non è da intendersi come un prodotto finito, bensì come una base di partenza per integrare le funzionalità di cui si è interessati nel simulatore Alchemist; ragion per cui, alcuni aspetti relativi all'ottimizzazione delle performance sono stati trascurati. Lo sviluppo di tale prototipo ha permesso di evidenziare le criticità e le opportunità delle strategie adottate.

Bibliografia

[Acc24] Giacomo Accursi. Reactive des, 2024.

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Spring Joint Computing Conference*, volume 30 of *AFIPS Conference Proceedings*, pages 483–485. AFIPS / ACM / Thomson Book Company, Washington D.C., 1967.

[BB91] Albert Benveniste and Gerard Berry. The synchronous approach to reactive and real-time systems. *Proc. IEEE*, 79(9):1270–1282, 1991.

[BCC+13] Engineer Bainomugisha, Andoni Lombide Carreton, Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. A survey on reactive programming. *ACM Comput. Surv.*, 45(4):52:1–52:34, 2013.

[BINN10] Jerry Banks, John S. Carson II, Barry L. Nelson, and David M. Nicol. *Discrete-Event System Simulation, 5th New Internatinal Edition*. Pearson Education, 2010.

[BS03] Bapat and Sturrock. The arena product family: enterprise modeling solutions. In *Proceedings of the 2003 Winter Simulation Conference, 2003.*, volume 1, pages 210–217 Vol.1, 2003.

[Chr13] Bronislav Chramcov. The optimization of production system using simulation optimization tools in witness. *International Journal of Mathematics and Computers in Simulation*, 7:95–105, 01 2013.

[CLK95] Sung-Do Chi, Ja-Ok Lee, and Young-Kwang Kim. Discrete event modeling and simulation for traffic flow analysis. In *1995 IEEE International Conference on Systems, Man and Cybernetics. Intelligent Systems for the 21st Century*, volume 1, pages 783–788 vol.1, 1995.

[Coo08] Gregory H. Cooper. *Integrating Dataflow Evaluation into a Practical Higher-Order Call-by-Value Language*. PhD thesis, Brown University, USA, 2008.

[CVAP22] Roberto Casadei, Mirko Viroli, Gianluca Aguzzi, and Danilo Pianini. Scafi: A scala DSL and toolkit for aggregate programming. *SoftwareX*, 20:101248, 2022.

[DM64] Bernard Dimsdale and Harry M. Markowitz. A description of the SIMSCRIPT language. *IBM Syst. J.*, 3(1):57–67, 1964.

[dMI09] Ana Lúcia de Moura and Roberto Ierusalimschy. Revisiting coroutines. *ACM Trans. Program. Lang. Syst.*, 31(2):6:1–6:31, 2009.

[EBAU21] Roman Elizarov, Mikhail A. Belyaev, Marat Akhin, and Ilmir Usmanov. Kotlin coroutines: design and implementation. In *Onward!*, pages 68–84. ACM, 2021.

[Ell09] Conal M. Elliott. Push-pull functional reactive programming. In Stephanie Weirich, editor, *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell, Haskell 2009, Edinburgh, Scotland, UK, 3 September 2009*, pages 25–36. ACM, 2009.

[FT01] Alois Ferscha and Satish Tripathi. Parallel and distributed simulation of discrete event systems. 03 2001.

[Fuj90] Richard Fujimoto. Parallel discrete event simulation. *Commun. ACM*, 33(10):30–53, 1990.

[Fuj95] Richard Fujimoto. Parallel and distributed simulation. In William R. Lilegdon, David Goldsman, Christos Alexopoulos, and Keebom Kang, editors, *Proceedings of the 27th conference on Winter*

simulation, WSC 1995, Arlington, VA, USA, December 3-6, 1995, pages 118–125. IEEE Computer Society, 1995.

[GB00] Michael A. Gibson and Jehoshua Bruck. Efficient exact stochastic simulation of chemical systems with many species and many channels. *The Journal of Physical Chemistry A*, 104(9):1876–1889, 2000.

[Gil77] Daniel T. Gillespie. Exact stochastic simulation of coupled chemical reactions. *The Journal of Physical Chemistry*, 81(25):2340–2361, 1977.

[Har96] Stephan Hartmann. The world as a process: Simulations in the natural and social sciences. In Rainer Hegselmann, editor, *Modelling and Simulation in the Social Sciences from the Philosophy of Science Point of View*. 1996.

[Hel08] Magdy Helal. A hybrid system dynamics-discrete event simulation approach to simulating the manufacturing enterprise. 2008.

[HM68] Fred C. Holland and Reino A. Merikallio. Simulation of a multiprocessing system using GPSS. *IEEE Trans. Syst. Sci. Cybern.*, 4(4):395–400, 1968.

[Hug99] R. I. G. Hughes. *The Ising model, computer simulation, and universal physics*, pages 97–145. Ideas in Context. Cambridge University Press, 1999.

[Hum90] Paul Humphreys. Computer simulations. *PSA: Proceedings of the Biennial Meeting of the Philosophy of Science Association*, 1990:497–506, 1990.

[Hum04] Paul Humphreys. *Extending Ourselves: Computational Science, Empiricism, and Scientific Method*. Oxford University Press, New York, US, 2004.

[IBH02] Arne Ingemansson, Gunnar Bolmsjö, and Ulrika Harlin. A survey of the use of the discrete-event simulation in manufacturing industry. *Journal of Xiamen University*, pages 193–195, 2002.

[Jef85] David R. Jefferson. Virtual time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, 1985.

[KMO98] K Kendall, C Mangin, and E Ortiz. Discrete event simulation and cost analysis for manufacturing optimisation of an automotive lcm component. *Composites Part A: Applied Science and Manufacturing*, 29(7):711–720, 1998.

[KTZP21] Stefan Kassen, Holger Tammen, Maximilian Zarte, and Agnes Pechmann. Concept and case study for a generic simulation as a digital shadow to be used for production optimisation. *Processes*, 9(8), 2021.

[LA04] Chris Lattner and Vikram S. Adve. The LLVM compiler framework and infrastructure tutorial. In *LCPC*, volume 3602 of *Lecture Notes in Computer Science*, pages 15–16. Springer, 2004.

[Law15] Averill M. Law. *Simulation Modeling & Analysis*. McGraw-Hill, New York, NY, USA, 5 edition, 2015.

[LP79] Jacques Leroudier and Michel Parent. Discrete event simulation modelling of computer systems for performance evaluation. *Mathematics and Computers in Simulation*, 21(1):50–79, 1979.

[MSM19] Florian Myter, Christophe Scholliers, and Wolfgang De Meuter. Distributed reactive programming for reactive distributed systems. *Art Sci. Eng. Program.*, 3(3):5, 2019.

[ÖB09] Onur Özgün and Yaman Barlas. Discrete vs. continuous simulation: When does it matter? 2009.

[OM08] Günther Ossimitz and Maximilian Mrotzek. The basics of system dynamics : Discrete vs continuous modelling of time. 2008.

[PMV13] Danilo Pianini, Sara Montagna, and Mirko Viroli. Chemical-oriented simulation of computational systems with ALCHEMIST. *J. Simulation*, 7(3):202–215, 2013.

[PVB15] Danilo Pianini, Mirko Viroli, and Jacob Beal. Protelis: practical aggregate programming. In Roger L. Wainwright, Juan Manuel Corchado, Alessio Bechini, and Jiman Hong, editors, *Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015*, pages 1846–1853. ACM, 2015.

[Ril13] Linda Ann Riley. Discrete-event simulation optimization: a review of past approaches and propositions for future direction. In *2013 Summer Simulation Multiconference, SummerSim '13, Toronto, Canada - July 07 - 10, 2013*, page 47, 2013.

[SM] George Q Sun and Wang Ming. Manufacturing simulation.

[SMKY13] Azura Che Soh, Mohammad Hamiruce Marhaban, Marzuki Khalid, and Rubiyah Yusof. A discrete-event traffic simulation model for multilane-multiple intersection. In *2013 9th Asian Control Conference (ASCC)*, pages 1–7, 2013.

[TCGATC21] Renato Torres-Carrión, Carlos Gonzaga-Acaro, and Hernán Torres-Carrión. Design and simulation in omnet ++ of a wireless sensor network for rural exploitation zones. In *2021 16th Iberian Conference on Information Systems and Technologies (CISTI)*, pages 1–6, 2021.

[TD95] Francesco Tisato and Flavio DePaoli. On the duality between event-driven and time-driven models. *IFAC Proceedings Volumes*, 28(22):31–36, 1995. 13th IFAC Workshop on Distributed Computer Control Systems 1995(DCCS '95), Toulouse-Blagnac, France, 27-29 September.

[VH08] András Varga and Rudolf Hornig. An overview of the omnet++ simulation environment. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications,*

Networks and Systems & Workshops, Simutools '08, Brussels, BEL, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[Wie98] Frederick Wieland. Parallel simulation for aviation applications. In Deborah J. Medeiros, Edward F. Watson, John S. Carson II, and Mani S. Manivannan, editors, *Proceedings of the 30th conference on Winter simulation, WSC 1998, Washington DC, USA, December 13-16, 1998*, pages 1191–1198. WSC, 1998.

[Win03] Eric Winsberg. Simulated experiments: Methodology for a virtual world. *Philosophy of Science*, 70(1):105–125, 2003.

[WLH05] S. Wang, K.Z. Liu, and F.P. Hu. Simulation of wireless sensor networks localization with omnet. In *2005 2nd Asia Pacific Conference on Mobile Technology, Applications and Systems*, pages 1–6, 2005.

[ZS19] Liliia Ziganurova and Lev N. Shchur. Synchronization aspects of the optimistic parallel discrete event simulation algorithms. In *International Conference on Data Analytics and Management in Data Intensive Domains*, 2019.