ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

Simulazione di organizzazioni a decisioni distribuite

Tesi di laurea in:
Advanced Software Modelling and Design

Relatore
Prof. Danilo Pianini

Candidato
Pietro Lelli

Correlatore
Prof. Guido Fioretti

IV Sessione di Laurea
Anno Accademico 2023-2024

Abstract

Le organizzazioni moderne, caratterizzate da gerarchie piatte, affrontano crescenti complessità nei processi decisionali e produttivi. La simulazione rappresenta un approccio efficace per analizzare e ottimizzare sistemi complessi, consentendo di esplorare scenari "what-if" e supportare decisioni. Tuttavia, gli strumenti esistenti spesso presentano alcuni problemi, quali l'eccessiva specializzazione settoriale e i costi elevati. Questa tesi propone lo sviluppo di un simulatore generico, flessibile e open source. L'architettura del sistema, infatti, è stata progettata per adattarsi a molteplici domini, tra cui l'assistenza sanitaria e l'industria. I casi di studio sviluppati dimostrano l'efficacia del simulatore nell'assegnamento di carichi di lavoro e nel miglioramento delle prestazioni complessive del sistema di produzione. In particolare, sono emersi risultati significativi dall'applicazione di alcune modalità operative, come l'assegnamento basato sulla lista d'attesa più breve. Pur trattandosi di una versione preliminare, il sistema offre una base valida per futuri sviluppi, quali l'introduzione di funzionalità avanzate e di altri elementi del dominio più complessi. Questo progetto evidenzia il potenziale delle simulazioni come strumenti di supporto decisionale alle organizzazioni moderne, consentendo di migliorarne la flessibilità e l'efficienza nei processi operativi.

Grazie a chi mi ha supportato in questo percorso.

Contents

Chapter 1

Contesto e motivazioni

Le imprese, ad oggi, si trovano ad affrontare una situazione complessa, dovuta alle imprevedibili richieste del mercato, alla crescente personalizzazione dei prodotti e alla fluttuazione degli ambienti di produzione [GOSSR10]. In particolare, le organizzazioni con decisioni distribuite devono bilanciare flessibilità, efficienza e adattabilità per operare con successo in settori dinamici come la sanità, la logistica e l'industria manifatturiera.[GOSSR10] [ND06]

Le simulazioni, in particolare quelle basate su modelli ad agenti, Agent-Based Model (ABM), sono state utilizzate come strumento di analisi e miglioramento di tali sistemi, consentendo di esplorare scenari complessi e identificare soluzioni ottimali. Come evidenziato da Nilsson e Darley (2006), l'ABM consente di creare scenari "what-if" realistici che aiutano i decisori a comprendere meglio le dinamiche dei sistemi e a ottimizzare le politiche operative [ND06]. Questo approccio è stato ampiamente utilizzato in contesti industriali, dimostrando la capacità di migliorare la pianificazione produttiva [GOSSR10]. Tuttavia, molte applicazioni di simulazione sviluppate in passato sono state progettate per settori specifici. Ad esempio, KanbanSimTM, un simulatore basato su Kanban, si è rivelato utile nell'industria automobilistica per ottimizzare i flussi produttivi e i livelli di inventario, ma risulta incapace di adattarsi alle variazioni degli ordini e non è riutilizzabile in contesti diversi da quello descritto [GOSSR10]. Nonostante le funzionalità sopra analizzate, molti simulatori esistenti hanno alcune rilevanti limitazioni. Da un lato, si basano su modelli che semplificano eccessivamente la realtà, assumendo

linearità o staticità dei processi [ND06]. Dall'altro lato, lo sviluppo di software proprietari estremamente specialistici, soprattutto a partire dagli anni 2000, ha reso questi strumenti costosi e inaccessibili per molte organizzazioni [ND06]. Questo approccio si è spesso concentrato su domini specifici, mentre le organizzazioni moderne, caratterizzate da strutture flessibili e da dinamiche decisionali decentralizzate, richiedono simulatori capaci di adattarsi a domini differenti.

Molte delle sfide incontrate in settori apparentemente diversi, come l'assistenza sanitaria e la produzione industriale, condividono elementi comuni, tra cui la gestione delle risorse, la necessità di coordinazione tra entità autonome e la capacità di rispondere rapidamente a cambiamenti esterni [GOSSR10]. Questi elementi rappresentano il fondamento per il progetto descritto in questa tesi, che si propone di sviluppare un simulatore flessibile, capace di essere applicabile a domini differenti e di superare alcuni limiti degli strumenti esistenti. Infatti, a differenza dei simulatori tradizionali focalizzati su contesti settoriali specifici, questo approccio si concentra sull'individuazione e modellazione di elementi comuni a diversi domini, in modo da rendere l'architettura e l'applicativo facilmente estendibili e riutilizzabili. Inoltre, l'adozione di tecnologie open source permetterà superare il nodo economico, che rende difficile l'utilizzo degli applicativi già implementati, rendendo così il simulatore più accessibile. Questa combinazione di flessibilità, accessibilità e applicabilità a domini differenti può rappresentare un miglioramento significativo rispetto agli strumenti attualmente disponibili.

Chapter 2

Organizzazioni a decisioni distribuite

2.1 Introduzione

A partire dagli ultimi decenni del XX secolo, un numero crescente di aziende sta sperimentando ambienti sempre più turbolenti e imprevedibili. A tali cambiamenti, le aziende hanno reagito ideando forme organizzative che si discostano in modo sostanziale dalle tradizionali burocrazie. Il numero di livelli gerarchici si è ridotto, gli individui sono stati organizzati in team, i confini sono diventati sempre più sfumati sia all'interno delle aziende che tra le aziende stesse, e l'ideologia aziendale viene impiegata come ulteriore collante di organizzazioni altrimenti logore. La filosofia che sostiene questa tendenza è che le imprese dovrebbero cavalcare l'incertezza invece di cercare di proteggersi da essa, generando adattamenti ad hoc per adattarsi ai cambiamenti dell'ambiente [AC13] [BBCL16] [Bon10] [Bro75].

Questo tipo di burocrazie flessibili [CF22] [Dol10] sono state inizialmente etichettate come adhocrazie per la loro capacità di adattarsi ai cambiamenti dell'ambiente, per la loro capacità di adattare la loro configurazione a problemi specifici [DSV04], e poi come organizzazioni a rete per le loro relazioni in continuo cambiamento con altre organizzazioni [PP98], e infine come organizzazioni agili quando il networking si estende all'interno dell'organizzazione stessa [Min83] [JJ13].

È notevole che, dal momento che le organizzazioni agili si rimodellano con-

tinuamente, le loro caratteristiche devono essere espresse in termini di processi ricorrenti piuttosto che di una tipica struttura invariante [Fio12].

Mentre la teoria economica concepisce le gerarchie come isole di pianificazione all'interno di un mare di interazioni di mercato [FZ14], le reti agili sono organizzazioni che possono in certo grado contenere concorrenza all'interno delle loro mura [PP98]. Questo ha senso perché, proprio come le economie pianificate evitano inutili duplicazioni ma irrigidiscono l'innovazione, le gerarchie tradizionali potrebbero non essere adatte ai modelli di innovazione aperta in continua evoluzione [Gol16]. In particolare, le reti agili possono consentire un certo grado di sperimentazione parallela nella misura in cui ha senso esplorare opzioni divergenti, mentre le duplicazioni inutili possono essere ridotte se questa necessità viene meno [Mar82]. Quindi, nelle organizzazioni di rete agili, la scelta tra esplorazione e sfruttamento [Mar82] (o entrambe) è anche una scelta di regole di governance.

Per quanto riguarda il classico focus sul coordinamento attraverso l'adattamento reciproco (o il feedback), il coordinamento attraverso la supervisione (o il piano) e il coordinamento attraverso la standardizzazione come elementi costitutivi delle organizzazioni [Mum06] [PRK12] [Her14], le reti agili frenano la coordinazione per supervisione, preferendo condizioni di contorno standardizzate in cui i team possono operare liberamente piuttosto che specificare i dettagli delle attività di ogni singolo individuo [KS93].

Le istanze vanno dalle architetture software che vincolano e incanalano le attività dei programmatori senza specificarle [Bro75], alla stesura di un elaborato insieme di norme che regolano le interazioni all'interno del team e tra i team [CF22], così come l'organizzazione di aste interne per progetti che richiedono la collaborazione del team lungo determinate dimensioni specifiche [CVR24]. Si può obiettare che il pilotaggio indiretto tramite "fatti oggettivi" è uno strumento manageriale molto classico, in effetti [Gou54] [Kle96]. Tuttavia, le organizzazioni basate sul lavoro di squadra lo articolano in modi nuovi. La filosofia della produzione snella è portata al limite. Proprio come il sistema Andon di Toyota visualizza lo stato di tutte le squadre lungo la linea di produzione, le organizzazioni agili utilizzano le tecnologie informatiche per rendere disponibili le informazioni su ciascun team a tutti i team del proprio e ai livelli gerarchici superiori [Min83]. Secondo la produzione snella, le informazioni devono essere disponibili pubblicamente

per evidenziare i problemi, invece di risolverli con risorse "allentate" in eccesso [PS88]. Tuttavia, ciò può avere la conseguenza di spingere i dipendenti a sviluppare soluzioni per conto proprio, una pratica nota come miglioramento continuo [Mum06], oppure sovraccaricare i team a tal punto da rendere il lavoro intenso, una pratica nota come gestione dello stress [PS88].

Tentativamente, il miglioramento continuo è più probabile a livelli gerarchici elevati e nelle industrie high-tech o creative, mentre la gestione dello stress può essere prevista quando non si cerca il contributo intellettuale dei dipendenti. In entrambi i casi, l'effetto netto è la riduzione del numero di livelli gerarchici. Le reti agili portano all'estremo, al punto che aziende con migliaia di dipendenti possono avere solo due livelli gerarchici [Min83]. Ai team vengono assegnati obiettivi impegnativi [MM19]. Le decisioni di assunzione e licenziamento, così come le decisioni relative alla responsabilità e alla retribuzione individuale e di gruppo, sono delegate ai team stessi [DSVW19]. Mentre l'autoselezione dei membri del team per facilitare l'accettazione della retribuzione collettiva è stata sperimentata ben prima dell'agile [KS93], la delega della decisione di licenziamento, o anche la semplice possibilità di licenziare i dipendenti, è in netto contrasto con la precedente tradizione umanistica, in cui l'impegno si otteneva in cambio di un lavoro interessante e di un impiego a vita [MM22] [Gou54].

Le reti agili possono fornire reti di sicurezza per i team che falliscono, ma per lo scopo pratico di mantenere la forza lavoro competente piuttosto che per ragioni idealistiche [CVR24]. Tuttavia, si può osservare che il relativo successo degli esperimenti sociotecnici più idealistici degli anni '60 e '70 sono stati almeno favoriti dalla carenza di forza lavoro [Mum06].

Una caratteristica sorprendente degli elementi sopra citati è che i principi organizzativi riguardano i team e non gli individui. Tuttavia, le strutture che si trovano nelle reti agili non sono solo gerarchie con team al posto degli individui, perché è probabile che gli individui siano membri di più team allo stesso tempo, eventualmente assumendo ruoli diversi. Ad esempio, il leader di un team può essere un follower in altri team e viceversa. La Figura 1 illustra questo concetto con un organigramma minimo (a sinistra) che prevede due soli livelli gerarchici, vale a dire un organo di governo A e quattro squadre 1, 2, 3 e 4. Al centro e a destra, due diagrammi di appartenenza in cui le squadre sono rappresentate da

ovali che si intersecano se hanno membri comuni. Le intersezioni implicano membri dell'organizzazione che sono attivi in più team, mentre le inclusioni significano che tutti i membri di un team sono anche membri di un team più ampio.

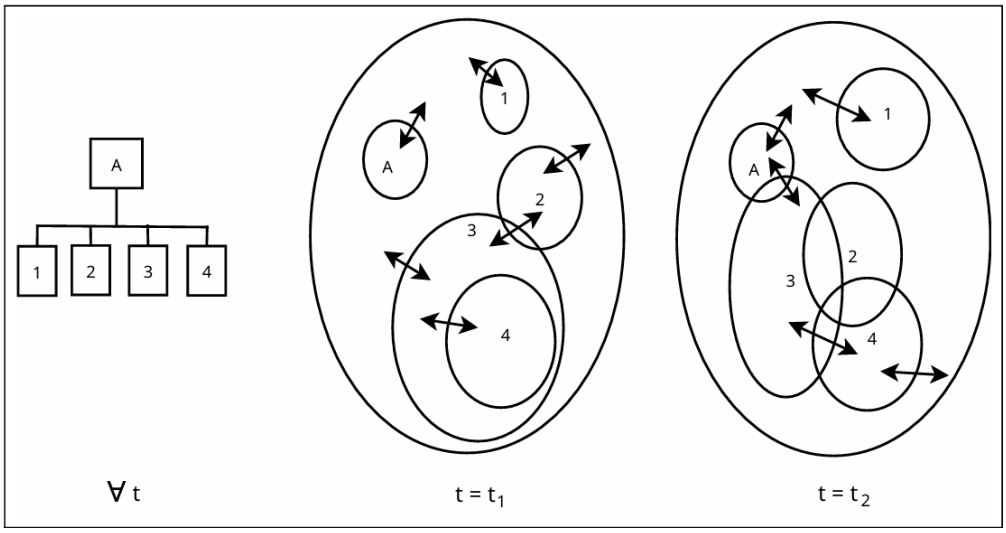


Figure 2.1: Rete agile rappresentata da un organigramma variabile nel tempo a sinistra e da due diagrammi di appartenenza a t=t1 e t=t2 al centro e a destra.

Le tre sezioni della Figura 1 rappresentano una stessa organizzazione. Tuttavia, mentre la differenza gerarchica che separa l'organo di governo dagli altri team rimane nel tempo, l'appartenenza al team cambia nel tempo. Pertanto, i due diagrammi di appartenenza della Figura 1 rappresentano la stessa organizzazione in tempi diversi. Questo quadro potrebbe essere ulteriormente complicato includendo la possibilità che i team vengano generati o sciolti nel tempo, il che influirebbe sia sui diagrammi di appartenenza che sull'organigramma. Nella Figura 1, le frecce rappresentano il coordinamento per aggiustamento reciproco (feed-back), mentre le linee rette dell'organigramma rappresentano il coordinamento per supervisione. Le reti agili utilizzano entrambe le modalità, ma cercano di mantenere al minimo la supervisione dei team, anche se la supervisione avviene all'interno dei team nella misura in cui questi sono incaricati di prendere decisioni di assunzione e licenziamento. La standardizzazione delle attività viene evitata quando possibile, mentre la standardizzazione delle condizioni al contorno (direttamente o indirettamente) influisce sulla transizione tra i diagrammi di appartenenza con il tempo.

2.2 Decisioni distribuite nelle organizzazioni agili

Il concetto di decisioni distribuite si sposa con il funzionamento delle organizzazioni agili. In queste strutture, il processo decisionale non è accentrato nelle mani di pochi, ma si diffonde all'interno dell'organizzazione. Ad esempio, la responsabilità è distribuita: chi si trova più vicino alle informazioni rilevanti è anche chi ha il potere di prendere decisioni. Questo approccio elimina la necessità di continui passaggi gerarchici e consente di agire con maggiore rapidità ed efficacia [1].

Un altro aspetto fondamentale è la collaborazione interfunzionale. I team, spesso composti da persone con competenze e ruoli diversi, lavorano insieme per affrontare problemi e arrivare a soluzioni condivise. Questo non solo migliora la qualità delle decisioni, ma favorisce anche un senso di appartenenza e partecipazione. Infine, le unità operative o i team sul campo possono rispondere rapidamente alle sfide che incontrano, senza dover attendere direttive dall'alto. Questo rende l'organizzazione più flessibile e reattiva [2].

[1] https://www.sherpany.com/it/risorse/riunioni-dei-dirigent/benefici-organizzazione-agile-aziendale/

[2] https://tech4future.info/agile-organization-cose-principi/

Chapter 3

Analisi

3.1 Analisi del dominio

3.1.1 Descrizione del dominio

Nel contesto delle organizzazioni distribuite, è fondamentale definire un modello che rappresenti fedelmente gli elementi e i collegamenti tra le diverse componenti descritte di seguito.

Unità produttive

Le unità produttive costituiscono il nucleo operativo del sistema. Sono composte da squadre di lavoro che possono includere lavoratori e macchinari. Ogni unità produttiva ha l'obbiettivo di svolgere attività specifiche, a seconda delle proprie capacità di esecuzione. La capacità delle unità produttive di interagire è regolata da grafi di interazione, definiti da un pianificatore, un individuo di più alto livello gerarchico. Tali interazioni possono includere la comunicazione tra unità per coordinare attività, condividere risorse o formare alleanze.

Inoltre, una caratteristica molto importante delle unità produttiva è rappresentata dalla dinamicità, infatti possono sciogliersi, riformarsi o aggregarsi autonomamente in base a vincoli predefiniti. Questo meccanismo consente una gestione dinamica degli ordini.

Oltre a questo, ogni unità produttiva può disporre di un magazzino locale che

consente di anticipare il lavoro e quindi gestire in modo più efficiente il carico operativo, sfruttando anche periodi di inattività.

Ordini

Gli ordini rappresentano richieste per la produzione di beni o servizi e sono costituiti da una o più ricette che ne definiscono l'esecuzione. Per prendere decisioni efficaci ruguardanti gli ordini, è necessario osservare i tempi di lavorazione e le file d'attesa delle unità produttive, e quindi adattare dinamicamente il proprio percorso in risposta a nuove informazioni, come la modifica delle ricette o la riorganizzazione delle priorità.

Un aspetto distintivo degli ordini è la loro capacità di evolvere. Ad esempio, possono includere nuovi step o modificare le ricette durante l'esecuzione. Un esempio significativo potrebbe essere quello che avviene in ambito medico, quando una diagnosi incompleta o un peggioramento del paziente porta a ulteriori analisi e cambiamenti terapeutici, modificando così l'ordine durante la sua esecuzione.

Ricette

Le ricette rappresentano le modalità di esecuzione di ogni ordine; infatti, possono contenere diversi step da eseguire e vari tipi di dipendenze tra di essi. Di seguito i principali tipi di dipendenze e modalità di esecuzione che le ricette possono contenere:

- Dipendenze di sequenza: alcune attività devono necessariamente precederne altre.
- Biforcazioni (passi OR): permettono di scegliere tra alternative equivalenti.
- Infornate: richiedono l'esecuzione contemporanea di un determinato numero di elementi.
- Passi paralleli (passi AND): prevedono l'esecuzione simultanea di più sequenze operative che si ricongiungono successivamente.

Pianificatore

A differenza di un pianificatore tradizionale, in questo dominio il pianificatore non interviene direttamente nel micromanagement delle attività. La sua funzione principale è quella di creare il contesto in cui ordini e unità produttive possano operare in modo autonomo. Le sue principali responsabilità quindi includono:

- Selezionare l'insieme delle ricette disponibili.
- Definire i grafi delle interazioni tra unità produttive, regolando le loro possibili collaborazioni.
- Stabilire protocolli per la formazione, lo scioglimento e la riorganizzazione delle unità produttive e dei team.
- Pianificare gli approvvigionamenti delle risorse nel magazzino.

Un'ulteriore funzione del pianificatore potrebbe essere la definizione di vincoli per gestione di meccanismi di interazioni tra unità produttive più avanzati, come ad esempio la formazione di cordate o la partecipazione congiunta ad aste interne. Questi strumenti possono essere utilizzati per coordinare risorse e attività tra le unità, migliorando l'efficienza di produzione e favorendo un modello organizzativo piatto.

3.2 Analisi del simulatore

L'analisi del tipo di simulazione e linguaggio da utilizzare rappresenta una passo molto importante per la riuscita del progetto. Di seguito, verranno analizzate le motivazioni che rendono Alchemist un simulatore adeguato al progetto.

Generalità e adattabilità del simulatore Un elemento distintivo di Alchemist è la sua generalità, è possibile infatti mappare gli elementi del suo meta-modello alle entità di un corrispettivo concetto nel dominio di interesse. Questa caratteristica consente di adattare il simulatore a diversi domini. Nel caso specifico delle organizzazioni a decisioni distribuite, questo significa poter modellare le entità,

come unità produttive e ordini che vedremo più nel dettaglio successivamente, e associarle agli elementi del modello già presenti in Alchemist.

Approccio Event-Driven Un altro aspetto rilevante di Alchemist è la sua natura event-driven. Questo approccio si adatta al dominio delle organizzazioni a decisioni distribuite. Entrando più nel dettaglio, l'approccio event-driven si focalizza sulla relazione causale tra eventi e le azioni svolte dal sistema, ponendo l'accento sul motivo per cui accade qualcosa. Al contrario, il modello time-driven si concentra sul momento in cui le azioni avvengono, evidenziando il "quando" accade qualcosa. Nel primo caso, le azioni vengono eseguite il più rapidamente possibile all'arrivo degli eventi e la gestione del tempo può avvenire tramite segnali temporali trattati come eventi esterni asincroni. Nel secondo caso, invece, le azioni vengono eseguite in base a un programma prestabilito, e gli eventi esterni vengono gestiti attraverso meccanismi di polling [TD95].

Esperienza pregressa con il simulatore da parte del gruppo di lavoro Un'altra motivazione chiave è stata l'esperienza pregressa nel gruppo di lavoro con il simulatore Alchemist. Questo aspetto si è rivelato determinante in più occasioni, permettendo di superare con rapidità le difficoltà tecniche incontrate durante la fase di progettazione e, soprattutto, implementazione del simulatore.

3.3 Il simulatore Alchemist

3.3.1 Introduzione

Alchemist [PMV13] è un simulatore ad eventi discreti (DES) [BW08], ossia un simulatore in cui gli eventi vengono eseguiti uno alla volta avanzando il tempo di simulazione conseguentemente, come spiegato precedentemente. Questo tipo di approccio consente di modellare con facilità eventi collocati nel tempo attraverso distribuzioni temporali. Gli scenari tipici possono riguardare fenomeni molto differenti, dalle reazioni chimiche al movimento dei pedoni, esprimendo il fenomeno da simulare nei termini del modello di Alchemist. Questo modello è composto di entità astratte, che è necessario concretizzare in base alla specifica simulazione che

si vuole ottenere.

Alchemist è derivato da un modello prettamente chimico, di conseguenza, la terminologia utilizzata per indicare le varie entità si rifà a quella solitamente utilizzata in chimica, come spiegato successivamente.

3.3.2 Meta-Model

Model

Ci sono varie entità all'interno di Alchemist, mostrate nella seguente figura e successivamente spiegate [1]:

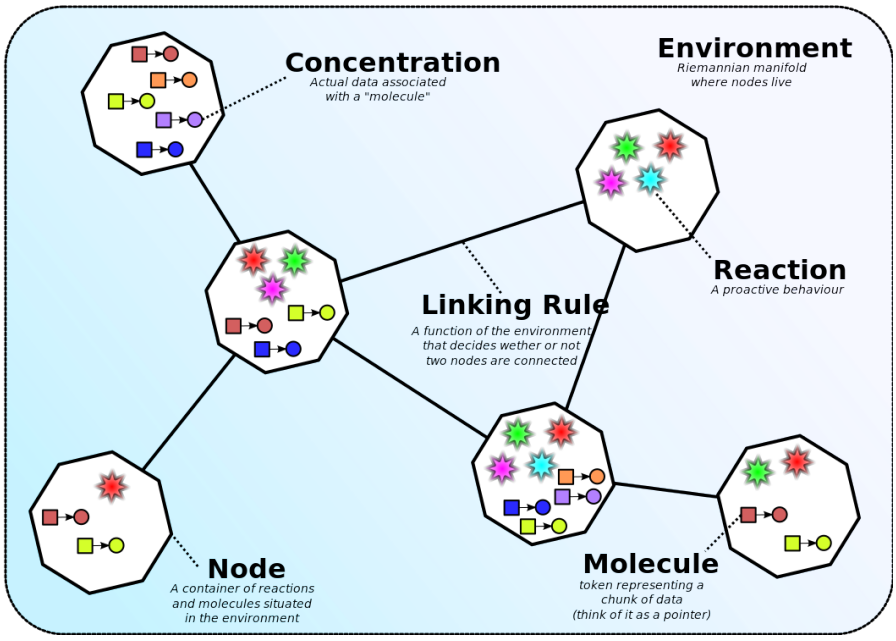


Figure 3.1: Alchemist model

Molecule Una molecule rappresenta il nome di un dato contenuto all'interno di un nodo. Se Alchemist fosse un linguaggio di programmazione imperativo, una molecola corrisponderebbe al concetto di nome di una variabile.

[1] `https://alchemistsimulator.github.io/explanation/metamodel/index.html`

Concentration La concentration è il valore associato a una particolare molecola. Se Alchemist fosse un linguaggio di programmazione imperativo, la concentrazione corrisponderebbe al concetto di valore associato a una variabile.

Node Un node è un'entità all'interno dell'ambiente e può contenere molecole e reazioni.

Environment L'environment rappresenta l'ambiente all'interno del quale avviene la simulazione. Esso funge da contenitore per i nodi ed è in grado di:

- Determinare dove si trovano i nodi nello spazio, ossia la loro posizione.
- Calcolare la distanza tra due nodi.
- Supportare il movimento dei nodi.

Linking rule Una linking rule definisce quali nodi fanno parte del vicinato di un determinato nodo.

Neighborhood Un neighborhood è un'entità composta da un nodo centrale e un insieme di nodi che rappresentano il vicinato.

Reaction Una reaction è un evento che può modificare lo stato dell'ambiente. Ogni nodo ha un insieme di reazioni.

Ogni reazione è definita da:

- Una lista di condizioni.
- Una o più azioni.
- Una distribuzione temporale (TimeDistribution).

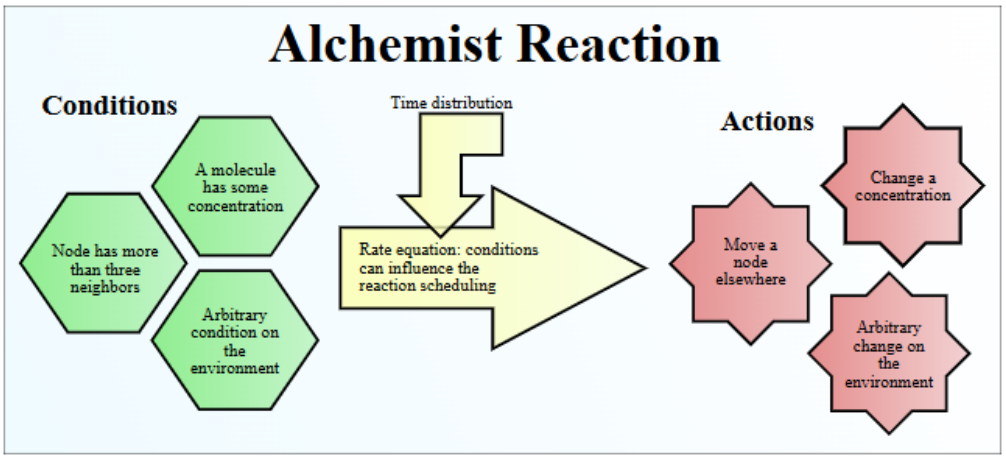


Figure 3.2: Alchemist reaction

Condition Una condition è una funzione che prende come input l'ambiente attuale e restituisce un valore booleano e un numero. Se la condizione non è soddisfatta (ossia, il suo output è falso), la reazione a cui è associata non può essere eseguita. Il numero restituito può influenzare o meno la velocità della reazione, a seconda della reazione e della sua distribuzione temporale.

Action Un'action modella un cambiamento nello stato dell'ambiente.

Incarnation

In Alchemist le entità del dominio possono essere concretizzate in modi differenti. Sulla base di ciò, è stato introdotto il concetto di Incarnation, o incarnazione, che è una specializzazione del modello di Alchemist che aiuta a definire specifici tipi di simulazione, specificando le implementazioni concrete delle entità di Alchemist per una certa categoria di simulazioni, operando di fatto una traduzione model-to-model.

3.3.3 Il motore

Il motore di simulazione adottato in Alchemist si basa sul Next Reaction Method di Gibson e Bruck [GB00], un algoritmo ottimizzato per simulare sistemi complessi. Questo approccio consente di selezionare la prossima reazione da eseguire

in tempo costante e di aggiornare le strutture dati in tempo logaritmico. Nonostante la sua efficienza, il Next Reaction Method originale non supporta operazioni dinamiche, come l'aggiunta, la rimozione o lo spostamento di nodi e reazioni, necessarie per simulare scenari computazionali più complessi. Alchemist ha quindi esteso l'algoritmo, introducendo modifiche a due strutture dati fondamentali: la coda prioritaria indicizzata e il grafo delle dipendenze [PMV13]. Questa estensione consente di gestire dinamicamente reazioni e dipendenze, rendendo il motore adatto a simulazioni che includono eventi complessi.

3.3.4 Utilizzo di Alchemist

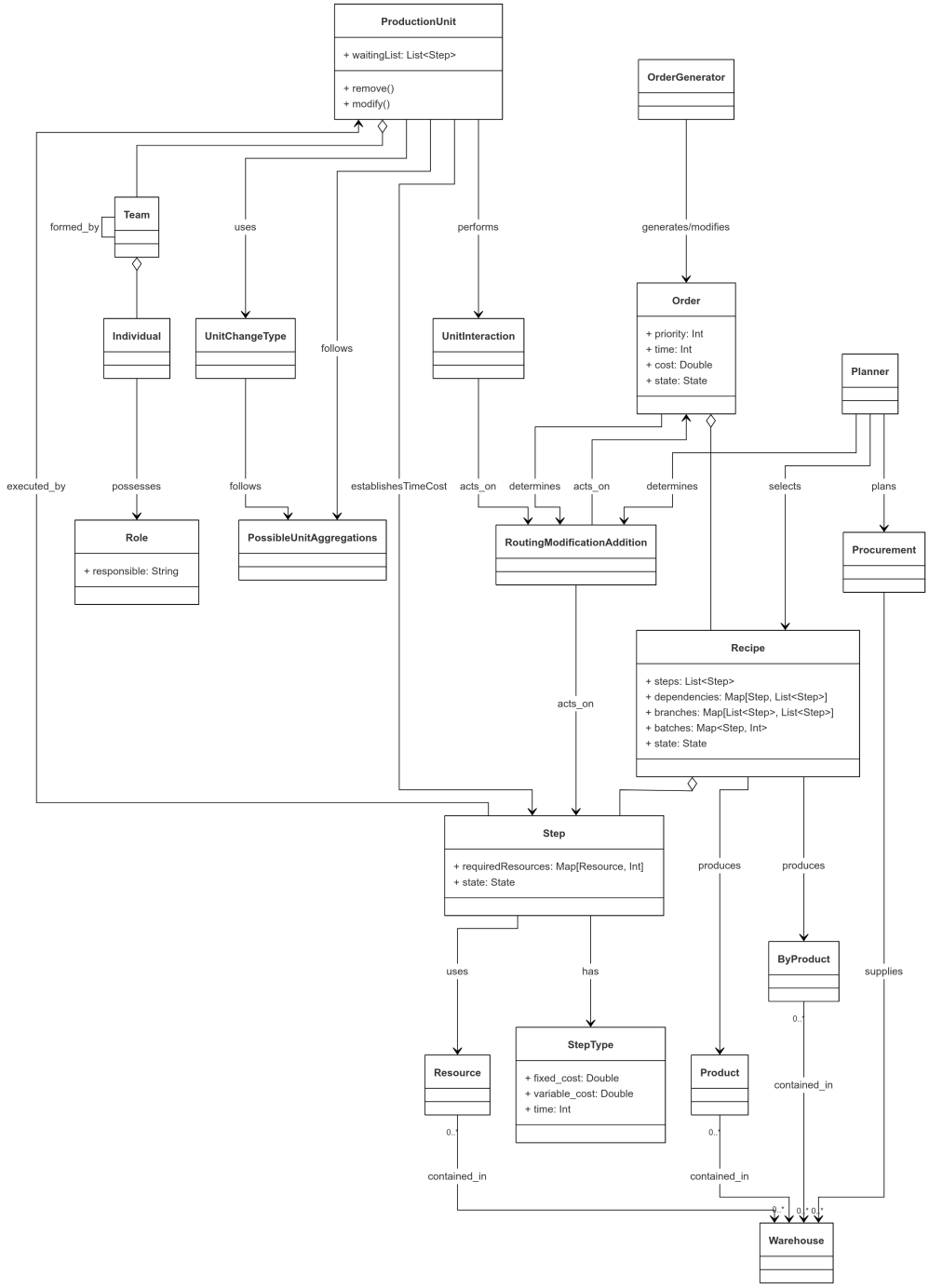
Alchemist può essere utilizzato in due diverse modalità: interattiva e batch. La differenza sta nel fatto che in modalità interattiva viene eseguita una sola simulazione per volta, mentre in modalità batch possono essere eseguite più simulazioni con una sola invocazione di Alchemist. In entrambi i casi, la simulazione viene configurata mediante un documento in formato YAML [BKEI05], all'interno del quale, utilizzando gli opportuni tag, è possibile stabilire quanti nodi disporre nell'ambiente, in che posizione e quali molecole debbano contenere.

Andando più nello specifico, è possibile specificare quale debba essere il tipo di Environment da utilizzare, con quale criterio i nodi debbano essere collegati tra di loro per formare un Neighborhood e quale sia il modo in cui la posizione di un nodo all'interno dell'ambiente viene espressa. Per ogni reazione viene specificata la relativa distribuzione temporale. Infine, è possibile specificare l'incarnazione. All'interno del file YAML di configurazione è inoltre possibile definire delle variabili, aventi ciascuna sia un valore di default che un ulteriore range di possibili valori. In modalità batch un sottoinsieme di queste variabili viene utilizzato per generare una serie di simulazioni differenti, utilizzando ad ogni iterazione un valore all'interno del range di valori ammissibili e il valore di default per tutte le altre, mentre in modalità interattiva vengono utilizzati unicamente i valori di default per tutte le variabili.

3.4 Modellazione del dominio

3.4.1 Class diagram

Di seguito il diagramma delle classi realizzato a seguito dello studio del dominio precedentemente descritto.



3.4.2 Mappatura del dominio sul simulatore

Nel contesto della simulazione, è necessario mappare gli elementi descritti nel dominio sulle entità fornite dal simulatore Alchemist. Questa mappatura consente di tradurre i concetti di alto livello descritti nel dominio in entità e azioni concrete che possono essere simulate all'interno dell'ambiente di Alchemist. Ad esempio, ogni unità produttiva nel dominio, che rappresenta l'elemento principale della produzione, sarà tradotta in un `NodeProperty`. Questi nodi durante l'esecuzione della simulazione interagiscono tra loro e con l'ambiente. Per quanto riguarda il processo di approvvigionamento, che nel dominio rappresenta il rifornimento delle risorse necessarie per il funzionamento delle unità produttive, esso sarà mappato come un'`Action`. Le Action sono azioni che vengono eseguite dai nodi durante la simulazione e che permettono di modificare lo stato dell'ambiente. Allo stesso modo, anche l'esecuzione dei vari step di produzione e la generazione degli ordini saranno tradotte in Action che verranno eseguite nel corso dell'esecuzione. Nel dominio è necessario inoltre un ambiente che contenga le variabili condivise tra le diverse unità produttive, come gli ordini e il magazzino. Questo sarà tradotto nel simulatore come un `Environment`. L'Environment rappresenta l'ambiente di simulazione e conserva tutte le informazioni globali, accessibili da tutti i nodi. Infine, è necessario definire una politica di collegamento tra le unità produttive. Questa permetterà di definire con chi le unità produttive possono interagire. Questa politica di collegamento sarà tradotta nel simulatore utilizzando la LinkingRule `FullyConnected`. Questa funzione permetterà di creare una rete in cui tutte le unità produttive saranno collegate tra loro. Gli altri elementi del dominio, che non trovano un corrispettivo diretto nel simulatore Alchemist, saranno implementati da zero.

Chapter 4

Design

4.1 Funzionalità implementate

4.1.1 Modello

Order

Nel sistema di gestione degli ordini è necessario tenere traccia di una serie di ricette, monitorando quando ciascuna di esse è completata per poter concludere l'intero ordine. Un altro aspetto critico è dato dal fatto che l'ordine deve essere gestito in modo tale che, una volta completato, venga spostato dalla lista degli ordini in corso a quella degli ordini completati. La classe `Order` affronta questi problemi. Ogni ordine è associato a una lista di ricette, rappresentate da oggetti `Recipe`. Per completare un ordine, è necessario che tutte le ricette in esso contenute siano finalizzate. Lo stato di avanzamento dell'ordine viene rappresentato attraverso un attributo `state`. Quando tutte le ricette sono completate, lo stato dell'ordine cambia, segnalandone il completamento. A questo punto, l'ordine viene rimosso dalla lista degli ordini in corso e viene aggiunto a quella degli ordini completati.

```
1  package simulation.model
2
3  class Order(
4      val idCode: String, var recipes: List<Recipe>,
5      val environment: DistributedDecisionEnvironment,
6      var state: State = State.TOBEASSIGNED
7  ) {
8      fun completeOrder() {
9          state = State.COMPLETE
10         environment.completedOrders.add(this)
11         environment.orders.remove(this)
12     }
13 }
```

Recipe

Nel sistema di gestione delle ricette è necessario tenere traccia di una serie di step che devono essere completati per ottenere il risultato desiderato. Ogni ricetta è legata a un ordine specifico e occorre monitorare quando tutti gli step sono completati per poter etichettare la ricetta come conclusa. Inoltre, la ricetta deve produrre un risultato che sarà poi aggiunto al magazzino. La classe `Recipe` risolve questi problemi. Ogni ricetta contiene una lista di oggetti `Step`, ciascuno rappresentante una singola attività che deve essere completata. Inoltre, la ricetta tiene traccia della relazione con l'ordine che la contiene tramite l'attributo `orderParent`. Lo stato di avanzamento della ricetta è gestito attraverso l'attributo `state`, che funziona in modo simile a quello utilizzato per gli ordini, segnalandone il completamento quando tutti gli step sono stati finalizzati. Una volta che tutti gli step sono completati, la ricetta viene considerata conclusa, e il risultato, definito dall'attributo `result`, viene aggiunto al magazzino, completando il ciclo di vita della ricetta.

```
package simulation.model

class Recipe (
    val idCode: String,
    val orderParent: Order,
    var steps: List<Step>,
    var state: State = State.TOBEASSIGNED,
    var result: Result
) {
    fun completeRecipe() {
        state = State.COMPLETE
        orderParent.environment.warehouse.addResult(result)
    }
}
```

In una versione futura, la classe `Recipe` potrebbe essere arricchita con ulteriori attributi, come indicato nel diagramma delle classi:

- `dependencies`: Una mappa che rappresenta le dipendenze tra gli step, indicando quali attività devono essere completate prima che altre possano iniziare.

  `dependencies : Map<Step, List<Step> >`

- `branches`: Una mappa che rappresenta le possibili biforcazioni nell'esecuzione degli step, consentendo scelte alternative durante il processo di esecuzione.

  `branches : Map<List<Step>, List<Step> >`

- `batches`: Una mappa che indica quante istanze di ciascuno step devono essere eseguite contemporaneamente.

  `batches : Map<Step, Int>`

Step

La gestione di una ricetta richiede la suddivisione del lavoro in singole unità, dette step, ognuna delle quali rappresenta una specifica attività da completare. Per eseguire uno step è necessario definirne il tipo, monitorarne lo stato e verificare che le risorse necessarie siano disponibili. Inoltre, è importante assicurarsi che lo stato dello step, e di conseguenza quello della ricetta e dell'ordine di cui fa parte, sia aggiornato in modo corretto durante l'esecuzione. La classe `Step` risolve questi problemi rappresentando il singolo elemento costitutivo di una ricetta. Ogni

step è caratterizzato da un attributo `type`, che specifica il tipo di attività da svolgere, e da una lista di risorse necessarie all'esecuzione, rappresentata dall'attributo `requiredResources`. Per poter tracciare l'avanzamento del processo la classe include anche un attributo `state`, che monitora lo stato dello step e si integra con gli attributi state delle classi `Recipe` e `Order`. Durante l'esecuzione, la classe fornisce metodi per gestire il completamento dello step, verificare la disponibilità delle risorse e aggiornare i vari stati delle componenti. Di seguito una breve spiegazione del loro funzionamento e i relativi codici sorgenti:

- **execute()**: verifica se tutte le risorse necessarie sono disponibili e in quel caso completa lo step corrente utilizzando il metodo `completeStep()`. Inoltre se tutti gli step della ricetta sono completati, completa la ricetta; analogamente, se tutte le ricette dell'ordine sono completate, completa l'ordine.

```
fun execute() {
  val orderParent = recipeParent.orderParent
  completeStep()
  if(recipeParent.steps.all { it.state == State.COMPLETE }) {
    recipeParent.completeRecipe()
  }
  if (orderParent.recipes.all { it.state == State.COMPLETE }) {
    orderParent.completeOrder()
  }
}
```

- **checkResources()**: controlla la disponibilità delle risorse necessarie per eseguire lo step. Restituisce un oggetto `Pair` che indica se tutte le risorse sono sufficienti e una mappa contenente le risorse che effettivamente utilizzerà. Questo metodo è utile per verificare in anticipo se uno step può essere eseguito.

```
fun checkResources(): Pair<Boolean, MutableMap<Resource, Int>> {
  var areEnoughResources = true
  val neededResources = mutableMapOf<Resource, Int>()
  requiredResources.forEach { (resource, quantity) ->
    val matchingResource = environment.warehouse.resources.keys.find { it.idCode
        == resource.idCode }
    if (matchingResource != null && environment.warehouse.resources[
        matchingResource]!! >= quantity) {
      neededResources += Pair(matchingResource, quantity)
    } else {
      areEnoughResources = false
    }
  }
  return Pair(areEnoughResources, neededResources)
}
```

- **completeStep()**: metodo privato utilizzato per completare lo step corrente. Rimuove le risorse necessarie dal magazzino, aggiorna lo stato dello step a `COMPLETE` e aggiorna il costo totale e il tempo totale nell'environment condiviso. Questo metodo viene richiamato internamente da `execute()`.

```
private fun completeStep() {
  val (areEnoughResources, neededResources) = checkResources()
  if (areEnoughResources) {
    neededResources.forEach { (resource, quantity) ->
      environment.warehouse.getResource(resource, quantity)
    }
    state = State.COMPLETE
    environment.totCost += type.cost
    environment.totTime += type.time
  }
}
```

Warehouse

La gestione delle risorse e dei risultati prodotti rappresenta un elemento molto importante per la simulazione di questo dominio. Ogni step richiede risorse specifiche per essere eseguito, mentre il completamento delle ricette genera risultati che devono essere archiviati. È necessario quindi un elemento che gestisca le risorse disponibili e i risultati ottenuti, permettendo di aggiungere o rimuovere elementi. La classe `Warehouse` rappresenta il magazzino del sistema e possiede le funzionalità sopra descritte. Al suo interno sono presenti due liste principali, una per le risorse e una per i risultati, e i relativi metodi di gestione.

ProdUnit

La simulazione del sistema richiede dei nodi capaci di eseguire gli step in base alle proprie capacità, garantendo così il progresso dell'intero processo produttivo. Ogni unità deve identificare i tipi di step che è in grado di gestire e possere una lista d'attesa per le attività da completare. La classe `ProdUnit` modella un'unità produttiva e funge da nodo della simulazione. Ogni unità è caratterizzata da una lista di capacità, rappresentata dall'attributo `capabilities`, che specifica i tipi di step che può eseguire. Basandosi su queste capacità, l'unità gestisce una `waitingList`, ovvero un elenco di step in attesa di essere eseguiti.

```
package simulation.model

class ProdUnit (
    override val node: Node<Any>,
    val environment: DistributedDecisionEnvironment,
    val idCode: String,
    private val capabilities: List<String>
) : NodeProperty<Any> {

    var waitingList: List<Step> = listOf()
    private val queuingPolicy: QueuingPolicy<Step> = ExecuteFirstPolicy()
}
```

Durante la simulazione, l'unità produttiva fornisce i metodi necessari per gestire ed eseguire gli step presenti nella propria lista d'attesa, contribuendo così all'avanzamento del processo produttivo. Di seguito una breve spiegazione del loro funzionamento e i relativi codici sorgenti:

- **isCapableOfExecute(step: Step)**: verifica se l'unità produttiva è in grado di eseguire un determinato step. La capacità è determinata dalla presenza dei tipi di capacità nella lista `capabilities`.

```
fun isCapableOfExecute(step: Step): Boolean {
  return when {
    "ALL" in capabilities -> true
    "A" in capabilities && step.type == StepType.TYPE_A -> true
    "B" in capabilities && step.type == StepType.TYPE_B -> true
    "C" in capabilities && step.type == StepType.TYPE_C -> true
    else -> false
  }
}
```

- **addStepToWaitingList(step: Step)**: aggiunge uno step alla lista d'attesa dell'unità di produzione, a condizione che il suo stato sia `TOBEASSIGNED`. Una volta aggiunto, lo stato dello step viene aggiornato a `ASSIGNED`.

```
fun addStepToWaitingList(step: Step) {
  if(step.state == State.TOBEASSIGNED) {
    waitingList = waitingList + step
    step.state = State.ASSIGNED
  }
}
```

- **executeFromWaitingList()**: seleziona e esegue il prossimo step dalla lista d'attesa, secondo la policy definita (`ExecuteFirstPolicy`). Lo step scelto deve avere stato `ASSIGNED` e devono essere presenti nel magazzino tutte le risorse necessarie. Una volta eseguito, lo step viene rimosso dalla lista d'attesa.

```
fun executeFromWaitingList() {
  val step = queuingPolicy.getNext(waitingList) { step -> step.state ==
      State.ASSIGNED && step.checkResources().first }
  if (step != null) {
    step.execute()
    waitingList = waitingList.filter { it.idCode != step.idCode }
  }
}
```

Result

Nella gestione del processo produttivo, è essenziale tenere traccia dei risultati prodotti da ogni ricetta. La classe `Result` è progettata per rappresentare il risultato ottenuto dal completamento di una ricetta. Quando tutti gli step associati a una ricetta vengono finalizzati, il risultato corrispondente viene automaticamente aggiunto al magazzino.

StepType

Ogni step presenta caratteristiche specifiche, come il tempo richiesto per l'esecuzione e il costo associato. Per gestire questi aspetti è necessario un modello che permetta di definire e organizzare i diversi tipi di step. La classe `StepType` risolve questo problema utilizzando un'enumerazione. Ogni `StepType` rappresenta una tipologia

di step ed è associato a un valore predefinito di tempo e costo. Gli `StepType` attualmente implementati includono un insieme di tipi predefiniti, ma è possibile estendere o modificare questa lista intervenendo direttamente nel codice. Gli StepType previsti in questo momento sono i seguenti:

- `TYPE_A("A", 1.0, 10.0)`
- `TYPE_B("B", 2.0, 20.0)`
- `TYPE_C("C", 3.0, 30.0)`

Resource

Nella simulazione del sistema ogni step richiede risorse specifiche per poter essere eseguito. Diventa quindi fondamentale gestire queste risorse nel magazzino, sia quando vengono utilizzate sia quando vengono aggiunte. La classe `Resource` soddisfa questa esigenza, rappresentando una risorsa necessaria per l'esecuzione di uno step all'interno del processo. Durante la simulazione queste risorse vengono aggiunte al magazzino attraverso l'azione `AddResourceToWarehouseAction`, spiegata più nel dettaglio successivamente.

4.1.2 Environment

Durante una simulazione è necessario gestire diversi elementi condivisi, come ordini, risorse e risultati. È quindi fondamentale disporre di un ambiente che permetta di aggiornare e gestire questi elementi in modo dinamico e coerente. La classe `DistributedDecisionEnvironment` rappresenta proprio questo ambiente di simulazione e risolve questi problemi. Infatti è progettata per contenere e gestire gli oggetti condivisi tra tutti i nodi di simulazione. Essa contiene vari elementi, come la lista `orders`, che memorizza gli ordini in corso. Questa viene aggiornata rimuovendo gli ordini completati e aggiungendo quelli generati. Per gestire le risorse e i risultati condivisi l'ambiente contiene un'istanza di magazzino, rappresentato dall'attributo `warehouse`. Inoltre l'ambiente permette di tracciare il costo totale e i tempi totali di esecuzione degli step mediante le variabili `totCost`

e `totTime`, aggiornandoli dinamicamente ogni volta che uno step è stato completato. Infine, l'environment contiene il metodo per la generazione degli ordini. Questa funzione utilizza poi un generatore di ricette e un generatore di step. Permette quindi di generare ordini completi a partire da due parametri richiesti da file di configurazione, ovvero il numero di ricette in ogni ordine e il numero di step in ogni ricetta. Una volta passati questi parametri, si occcupa di creare un ordine con al suo interno una lista di n ricette che a loro volta contengono m step, questi step vengono generati assegnandogli casualmente delle risorse necessarie. Anche la lista di possibili risorse necessarie è passata come parametro da file di configurazione.

Di seguito le varie funzioni per la generazione rispettivamente di ordini, ricette e step.

```
fun generateOrder(resourceIds: List<String>, numRecipesInOrder: Int,
    numStepInRecipe: Int) {
  val recipeList = mutableListOf<Recipe>()
  val order = Order(
    "Order" + (0..100).random(),
    recipes = listOf(),
    environment = this,
    state = State.TOBEASSIGNED
  )
  for (i in 1..numRecipesInOrder) {
    recipeList.addLast(generateRecipe(order, resourceIds, numStepInRecipe))
  }
  order.recipes = recipeList
  orders.add(order)
}
```

```
private fun generateRecipe(orderParent: Order, resourceIds: List<String>,
    numStepInRecipe: Int): Recipe {
  val stepList = mutableListOf<Step>()
  val randomRecipeValue = (0..10).random()
  val recipe = Recipe("Recipe$randomRecipeValue", orderParent, stepList, result =
      Result("Result$randomRecipeValue"))
  for (i in 1 .. numStepInRecipe) {
    stepList.addLast(generateStep(recipe, resourceIds))
  }
  return recipe
}
```

```
private fun generateStep(recipeParent: Recipe, resourceIds: List<String>): Step {
  val requiredResources: MutableMap<Resource, Int> = mutableMapOf()
  if (resourceIds.isNotEmpty()) {
    requiredResources[Resource(resourceIds.random())] = (1..5).random()
  }
  return Step(
    "Step" + (0..100).random(),
    recipeParent,
    randomStepType(),
    recipeParent.orderParent.environment,
    requiredResources
  )
}

private fun randomStepType(): StepType {
  val random = (0..2).random()
  return when (random) {
    0 -> StepType.TYPE_A
    1 -> StepType.TYPE_B
    2 -> StepType.TYPE_C
    else -> StepType.TYPE_A
  }
}
```

4.1.3 Actions

Le Action rappresentano le azioni che il simulatore esegue con una certa frequenza (time-distribution), scelta tramite file di configurazione. Sono state implementate le seguenti azioni:

- **AddResourceToWarehouseAction**: aggiunge al magazzino una quantità casuale tra 1 e 5 di una risorsa, scelta casualmente tra le possibili risorse (scelte da file di configurazione).
- **OrderGeneratorAction**: utilizza il generatore di ordini per generare un nuovo ordine e assegnarlo all'environment condiviso tra tutti i nodi.
- **AssignStepToProdUnitAction**: si occupa di trovare ed assegnare il prossimo step ad una unità produttiva, sono state implementate tre modalità di assegnamento ed è possibile scegliere quale utilizzare da file di configurazione:

1. `RandomProdUnit`: l'unità produttiva che esegue l'azione sceglie casualmente un'unità produttiva, quindi assegna alla sua lista d'attesa il primo step che è in grado di eseguire, non ancora assegnato e in base alle sue capabilities.

```
private fun assignRandomStep() {
    val prodUnitToAssign = environment.nodes.mapNotNull { it.asProperty
        <Any, ProdUnit>() }
        .random()
    environment.orders
        .asSequence()
        .filter { it.state != State.COMPLETE }
        .flatMap { it.recipes }
        .filter { it.state != State.COMPLETE }
        .flatMap { it.steps }
        .firstOrNull { s -> s.state == State.TOBEASSIGNED &&
            prodUnitToAssign.isCapableOfExecute(s) }
        ?.let { prodUnitToAssign.addStepToWaitingList(it) }
}
```

2. `Self`: l'unità produttiva che esegue l'azione analizza la lista degli ordini e assegna alla propria lista d'attesa il primo step che è in grado di eseguire, non ancora assegnato e in base alle sue capabilities.

```
private fun assignNextStep() {
    environment.orders
        .asSequence()
        .filter { it.state != State.COMPLETE }
        .flatMap { it.recipes }
        .filter { it.state != State.COMPLETE }
        .flatMap { it.steps }
        .firstOrNull { s -> prodUnit.isCapableOfExecute(s) && s.state
            == State.TOBEASSIGNED }
        ?.let { prodUnit.addStepToWaitingList(it) }
}
```

3. `ShortWaitingList`: come prima cosa viene identificata l'unità produttiva con lista d'attesa più corta e, successivamente, analizzando la lista degli ordini condivisa, viene assegnato il primo step possibile a tale unità produttiva, considerando le sue capacità.

```
private fun assignStepToProdUnitWithShortWaitingList() {
    val prodUnitToAssign = environment.nodes.mapNotNull { it.asProperty<Any,
        ProdUnit>() }
        .minByOrNull { it.waitingList.size }
    environment.orders
        .asSequence()
        .filter { it.state != State.COMPLETE }
        .flatMap { it.recipes }
        .filter { it.state != State.COMPLETE }
        .flatMap { it.steps }
        .firstOrNull { s -> s.state == State.TOBEASSIGNED && prodUnitToAssign?.
            isCapableOfExecute(s) == true }
        ?.let { prodUnitToAssign?.addStepToWaitingList(it) }
}
```

- **RunOneStepAction**: si occupa di far eseguire uno step della lista d'attesa all'unità produttiva. Per identificare quale sia lo step da eseguire si controlla se le risorse necessarie ad eseguire quello step sono sufficienti, altrimenti si considera lo step successivo, finchè non si trova uno step eseguibile. In questo modo l'unità produttiva non rimane bloccata sul primo step, consentendo così di ottimizzare e velocizzare l'esecuzione degli elementi della lista d'attesa. Una volta identificato lo step, vengono rimosse dal magazzino le risorse necessarie e viene modificato lo stato in `COMPLETE`. Se si tratta dell'ultimo step da completare di una ricetta, anche lo stato della ricetta viene etichettato come completato e il risultato corrispondente viene aggiunto al magazzino. Inoltre, se la ricetta completata risulta l'ultima ricetta da completare, anche lo stato dell'ordine che la contiene viene etichettato come completato e l'ordine viene rimosso dagli ordini condivisi e aggiunto ad una lista di ordini completati.

4.2 Scelte implementative

4.2.1 QueuingPolicy

Per garantire un sistema facilmente estendibile e adattabile, è stata progettata la politica di accodamento utilizzando l'interfaccia `QueuingPolicy<T>`. Questa scelta progettuale offre diversi vantaggi in termini di flessibilità e manutenibilità. L'interfaccia `QueuingPolicy<T>` definisce una politica di accodamento generica, in

questo modo è possibile aggiungere facilmente nuove politiche creando altre classi che implementano l'interfaccia. Ad esempio sarebbe possibile creare una nuova politiva che utilizzi una coda di tipo LIFO.

L'uso della funzione `condition: (T) -> Boolean`, come parametro opzionale nel metodo `getNext`, permette di definire condizioni di selezione personalizzati per ogni politica, rendendo più semplice la personalizzazione della policy.

```
1 package simulation.policy
2
3 interface QueuingPolicy<T> {
4     fun getNext(list: List<T>, condition: (T) -> Boolean = { true }): T?
5 }
```

```
1 package simulation.policy
2
3 class ExecuteFirstPolicy<T> : QueuingPolicy<T> {
4     override fun getNext(list: List<T>, condition: (T) -> Boolean): T? {
5         return list.firstOrNull(condition)
6     }
7 }
```

4.2.2 Generatore di ordini nell'Environment

Nel sistema di simulazione sviluppato, il generatore di ordini è stato inserito all'interno dell'environment anziché essere implementato come un object separato. Questa scelta progettuale è stata dettata dalla necessità di supportare l'esecuzione di molteplici simulazioni in parallelo. Un'implementazione del generatore come object avrebbe implicato la sua natura di singleton, causando la condivisione di esso tra tutte le istanze di simulazione. Questo avrebbe portato a problemi tra le simulazioni, compromettendo l'accuratezza dei risultati. Inserendo invece il generatore direttamente nell'environment, ogni simulazione mantiene la propria istanza indipendente, garantendone la corretta esecuzione.

4.3 Simulazione

4.3.1 File di configurazione

Di seguito una breve spiegazione degli elementi principali del file di configurazione.

Questa configurazione di base consente di posizionare un'unità produttiva che non esegue nulla. Per aggiungerne altre è necessario inserire altri elementi alla variabile `prodUnits`.

```
incarnation: protelis
environment:
  - type: simulation.DistributedDecisionEnvironment
network-model:
  type: FullyConnected
_prodUnits: &prodUnits
  - type: Point
    parameters: [ 1, 0 ]
    properties:
      - type: simulation.model.ProdUnit
        parameters: [ "ProdUnit1", [ "ALL" ] ]
    contents:
      - molecule: prodUnit
deployments:
  *prodUnits
```

Inoltre, è possibile assegnare ai nodi posizionati un'azione da svolgere, in questo caso `RunOneStepAction`, che permette di eseguire uno step dalla propria lista d'attesa. Per assegnare altre azioni è necessario aggiungere altri elementi di tipo `Action` alla variabile `prodUnitsProgram`.

```
incarnation: protelis
environment:
  - type: simulation.DistributedDecisionEnvironment
network-model:
  type: FullyConnected
_prodUnitsProgram: &prodUnitsProgram
  - time-distribution: 5
    type: Event
    actions:
      - type: simulation.action.RunOneStepAction
        parameters: 1
_prodUnits: &prodUnits
  - type: Point
    parameters: [ 1, 0 ]
    properties:
      - type: simulation.model.ProdUnit
        parameters: [ "ProdUnit1", [ "ALL" ] ]
    contents:
      - molecule: prodUnit
    programs:
      - *prodUnitsProgram
deployments:
  *prodUnits
```

Infine, è possibile aggiungere alla configurazione del simulatore un exporter che si occuperà di generare file con estensione .csv contenente i dati di simulazione specificati. Di seguito un esempio di utilizzo:

```
export:
  type: CSVExporter
  parameters:
    fileNameRoot: "export"
    exportPath: "data"
    interval: 10.0
  data:
    - time
    - molecule: "ResultsSize"
      aggregators: [ mean, max, min ]
```

4.3.2 Parametri personalizzabili

- Resources: lista delle possibili risorse usata nell'azione `AddResourceToWarehouseAction` per l'approvvigionamento e nell'azione `OrderGeneratorAction` per la generazione degli step con le risorse necessarie ad eseguirli.
  - `[ ]` non usa risorse
  - `["R0", "R1", "R2", "R3"]` il simulatore usa queste risorse.

```
_resources: &resources
["R0", "R1", "R2", "R3"]
```

- Numero di ricette in ogni ordine (secondo parametro) e numero di step in ogni ricetta (terzo parametro).

```
- time-distribution: 1
    type: Event
    actions:
      - type: simulation.action.OrderGeneratorAction
        parameters: [*resources, 2, 3]
```

- Numero di unità produttive.
- Capacità dell'unità produttiva (quali step può eseguire):
  - `[ ''A'', ''B'', ''C'', ''ALL'']` per modificare le possibilità è necessario modificare l'enumerazione nelle classi StepType e Environment.

```
properties:
  - type: simulation.model.ProdUnit
    parameters: ["ProdUnitA", ["A"]]
```

- Modalità di assegnamento degli step alle liste d'attesa delle unità produttive:
  - `RandomProdUnit`
  - `Self`
  - `ShortWaitingList`

```
- time-distribution: 5
    type: Event
    actions:
      - type: simulation.action.AssignStepToProdUnitAction
        parameters: ["ShortWaitingList"]
```

- Frequenza di ogni azione modificando la relativa time-distribution:
  - Aggiunta delle risorse al magazzino.
  - Assegnamento degli step alle unità produttive.

- Generazione dell'ordine.
- Esecuzione, da parte dell'unità produttiva, di uno step dalla propria lista d'attesa.

Chapter 5

Casi di studio

In questo capitolo verranno analizzati differenti scenari di utilizzo del simulatore implementato. Le simulazioni effettuate hanno come obbiettivo lo studio di come differenti modalità di assegnamento e parametri di esecuzione influenzano l'efficienza e il comportamento delle unità produttive e quindi dell'organizzazione a decisioni distribuite.

La simulazione esplora principalmente le tre modalità con cui gli step vengono assegnati alle unità produttive: RandomProdUnit, Self, e ShortWaitingList. Ognuna di queste modalità viene analizzata al variare dei parametri personalizzabili, in modo da evidenziare quali sono i vantaggi e le limitazioni di ognuna di esse. Per fare questo è stata utilizzata la modalità batch di esecuzione, che permette di generare una serie di simulazioni differenti, utilizzando ad ogni iterazione un valore all'interno del range di valori definito. I risultati ottenuti vengono poi esportati tramite CSVExporter, aggregati ed analizzati.

Di seguito saranno mostrati i casi d'uso specifici analizzati e le relative osservazioni.

5.1 Numero di risultati prodotti al variare della modalità di assegnamento

5.1.1 Caso 1: stesse frequenze di assegnamento tra unità produttive

Questa configurazione prevede l'utilizzo di tre unità produttive in grado di eseguire tutti i tipi di step, ad ognuna di queste è assegnata la stessa frequenza di assegnamento. Eseguendo differenti simulazioni con modalità di assegnamento differenti è possibile individuare quale sia la modalità ottimale che consente di ottenere risultati migliori.

```
variables:
  assign_mod: &assign_mod
    type: ArbitraryVariable
    parameters: ["RandomProdUnit", ["RandomProdUnit", "Self", "ShortWaitingList"]]
  seed: &seed
    min: 1
    max: 100
    step: 1
    default: 1
seeds:
  scenario: *seed
  simulation: *seed

_prodUnitsProgram: &prodUnitsProgram
  - time-distribution: 5
    type: Event
    actions:
      - type: simulation.action.RunOneStepAction
  - time-distribution: 5
    type: Event
    actions:
      - type: simulation.action.AssignStepToProdUnitAction
        parameters: *assign_mod

_prodUnits: &prodUnits
  - type: Point
    parameters: [ 1, 0 ]
    properties:
      - type: simulation.model.ProdUnit
        parameters: [ "ProdUnit1", [ "ALL" ] ]
    contents:
      - molecule: ResultsSize
        concentration: 0
    programs:
      - *prodUnitsProgram
  - type: Point
    parameters: [ 1, 0 ]
    properties:
      - type: simulation.model.ProdUnit
        parameters: [ "ProdUnit2", [ "ALL" ] ]
    contents:
      - molecule: ResultsSize
        concentration: 0
    programs:
      - *prodUnitsProgram
  - type: Point
    parameters: [ 1, 0 ]
    properties:
      - type: simulation.model.ProdUnit
        parameters: [ "ProdUnit3", [ "ALL" ] ]
    contents:
      - molecule: ResultsSize
        concentration: 0
    programs:
      - *prodUnitsProgram
```

Modalità RandomProdUnit

La modalità `RandomProdUnit` assegna gli step alle unità produttive in modo casuale. Questo approccio porta a risultati generalmente inferiori rispetto alle altre modalità, poichè è possibile che alcune unità produttive rimangano con la propria lista d'attesa vuota per alcuni momenti durante l'esecuzione. Questo squilibrio si traduce in una sotto-utilizzazione delle risorse disponibili e in una minore ottimizzazione delle esecuzioni. Inoltre, questo problema si ripresenta nel momento in cui il generatore di ordini termina la sua esecuzione e le unità produttive devono smaltire la propria lista d'attesa. Anche in questo caso alcune unità produttive rischiano di rimanere con la lista d'attesa vuota, mentre altre potrebbero avere code molto lunghe, aumentando così il tempo necessario a terminare gli ordini.

Numero di risultati prodotti in media in 100 iterazioni: 1310.

Modalità Self

La modalità `Self` consente alle unità produttive di scegliere autonomamente gli step da aggiungere alla propria lista d'attesa. Questa strategia garantisce risultati migliori rispetto alla modalità RandomProdUnit, in quanto le unità produttive difficilmente rimangono senza lavoro da svolgere. L'assegnamento autonomo permette alle unità produttive di ottimizzare il proprio carico di lavoro, riducendo al minimo i tempi morti. Tuttavia, utilizzando questa modalità, i differenti tempi di esecuzione degli step portano alcune unità produttive ad accumulare elementi nella propria lista d'attesa. Anche in questo caso si ripresenta il problema già presentato con la modalità precedente, nel momento in cui il generatore di ordini termina la sua esecuzione e le unità produttive devono smaltire la propria lista d'attesa.

Numero di risultati prodotti in media in 100 iterazioni: 1496.

Modalità ShortWaitingList

La modalità `ShortWaitingList` punta a bilanciare il carico di lavoro tra le unità produttive, assicurandosi che le liste d'attesa abbiano dimensioni sempre uniformi. I risultati ottenuti sono comparabili a quelli della modalità Self, con il vantaggio aggiuntivo di una maggiore equità nella distribuzione del lavoro. Questo approccio

si rivela particolarmente utile nella fase finale, quando gli ordini smettono di essere generati e le unità produttive devono smaltire le liste d'attesa, in questo caso una distribuzione equilibrata richiede meno tempo per completare tutti gli ordini.

Numero di risultati prodotti in media in 100 iterazioni: 1496.

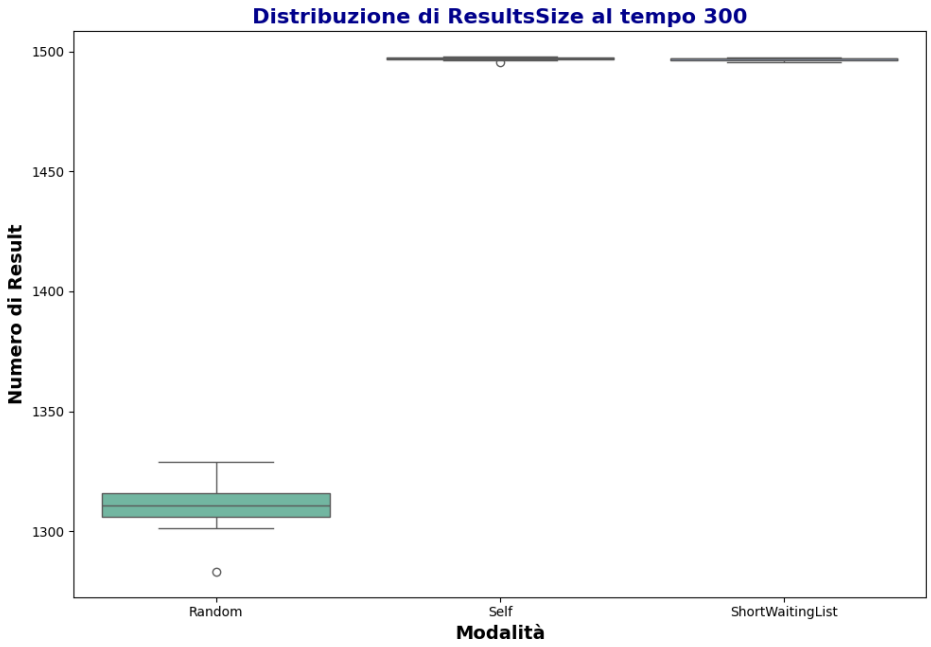


Figure 5.1: Numero di risultati prodotti per ogni modalità

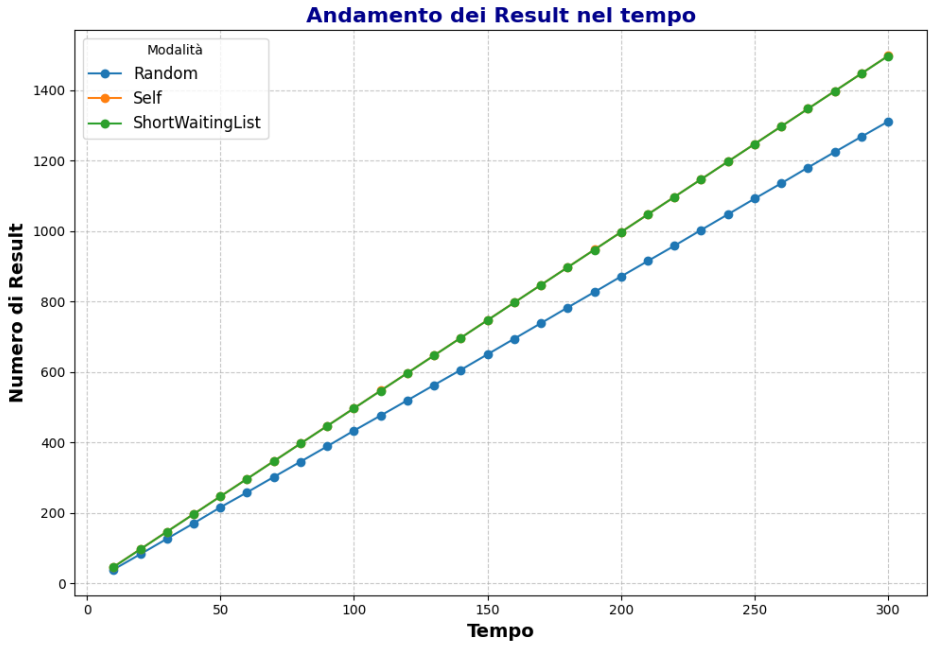


Figure 5.2: Andamento dei risultati prodotti per ogni modalità

5.1.2 Caso 2: frequenze di assegnamento diverse tra unità produttive

Questa configurazione prevede l'utilizzo di tre unità produttive in grado di eseguire tutti i tipi di step. In questo caso però alla ProdUnit1 è stata assegnata una frequenza di assegnamento più alta rispetto a quella precedente, mentre alle altre unità produttive è stata assegnata una frequenza di assegnamento più bassa. Questo caso mette in evidenza il comportamento delle diverse modalità di assegnamento quando le frequenze di assegnamento variano significativamente tra unità produttive dello stesso tipo.

```
variables:
  assign_mod: &assign_mod
    type: ArbitraryVariable
    parameters: ["RandomProdUnit", ["RandomProdUnit", "Self", "ShortWaitingList"]]

_prodUnitsProgram: &prodUnitsProgram
  - time-distribution: 5
    type: Event
    actions:
      - type: simulation.action.RunOneStepAction
        parameters: 1
  - time-distribution: 8
    type: Event
    actions:
      - type: simulation.action.AssignStepToProdUnitAction
        parameters: *assign_mod
_prodUnitsProgram2: &prodUnitsProgram2
  - time-distribution: 5
    type: Event
    actions:
      - type: simulation.action.RunOneStepAction
        parameters: 1
  - time-distribution: 2
    type: Event
    actions:
      - type: simulation.action.AssignStepToProdUnitAction
        parameters: *assign_mod
_prodUnits: &prodUnits
  - type: Point
    parameters: [ 1, 0 ]
    properties:
      - type: simulation.model.ProdUnit
        parameters: [ "ProdUnit1", [ "ALL" ] ]
    contents:
      - molecule: ResultsSize
    programs:
      - *prodUnitsProgram
  - type: Point
    parameters: [ 1, 0 ]
    properties:
      - type: simulation.model.ProdUnit
        parameters: [ "ProdUnit2", [ "ALL" ] ]
    contents:
      - molecule: ResultsSize
    programs:
      - *prodUnitsProgram2
  - type: Point
    parameters: [ 1, 0 ]
    properties:
      - type: simulation.model.ProdUnit
        parameters: [ "ProdUnit3", [ "ALL" ] ]
    contents:
      - molecule: ResultsSize
    programs:
      - *prodUnitsProgram2
```

Modalità RandomProdUnit

Nella modalità `RandomProdUnit`, l'assegnamento casuale non mette in evidenza ulteriori problemi oltre a quelli già evidenziati con il caso di studio precedente.

Numero di risultati prodotti in media in 100 iterazioni: 1157.

Modalità Self

Nella modalità `Self`, invece, le unità produttive con frequenze di assegnamento più basse tendono a rimanere con la lista d'attesa vuota, mentre quella con frequenza più alta accumula una lista d'attesa molto lunga. Questo fenomeno compromette l'efficienza complessiva del sistema, creando disparità nella distribuzione del carico di lavoro e portando così il sistema a risultati molto peggiori rispetto alla modalità RandomProdUnit.

Numero di Product completati in media in 100 iterazioni: 497.

Modalità ShortWaitingList

Al contrario, la modalità `ShortWaitingList` si dimostra ottimale in questa configurazione. L'unità produttiva con frequenza di assegnamento più alta assegna parte del suo carico alle altre, evitando che queste rimangano inattive o che si accumuli troppo lavoro nella propria lista d'attesa.

Numero di risultati prodotti in media in 100 iterazioni: 1197.

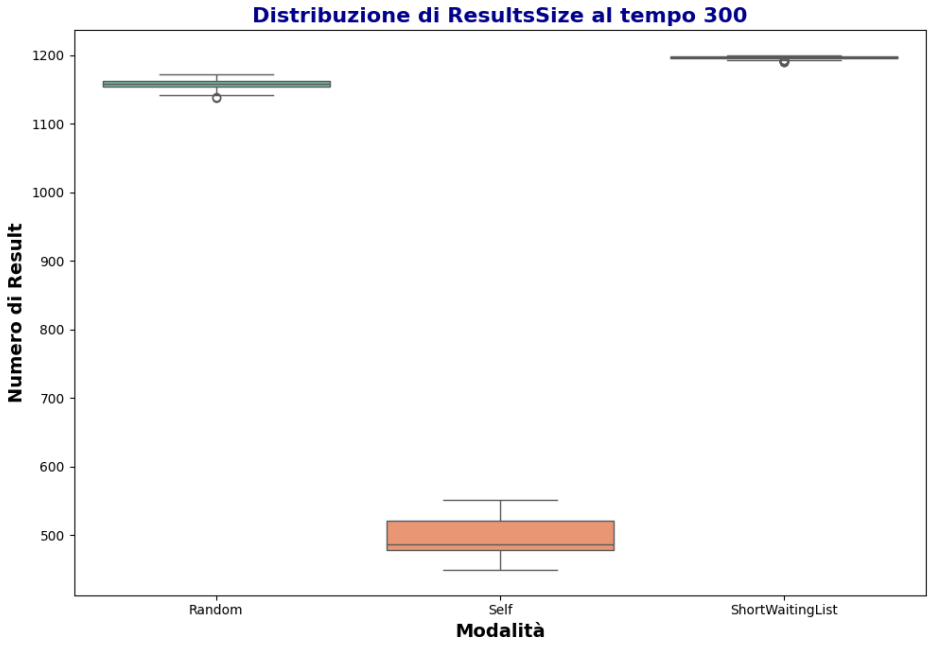


Figure 5.3: Numero di risultati prodotti per ogni modalità

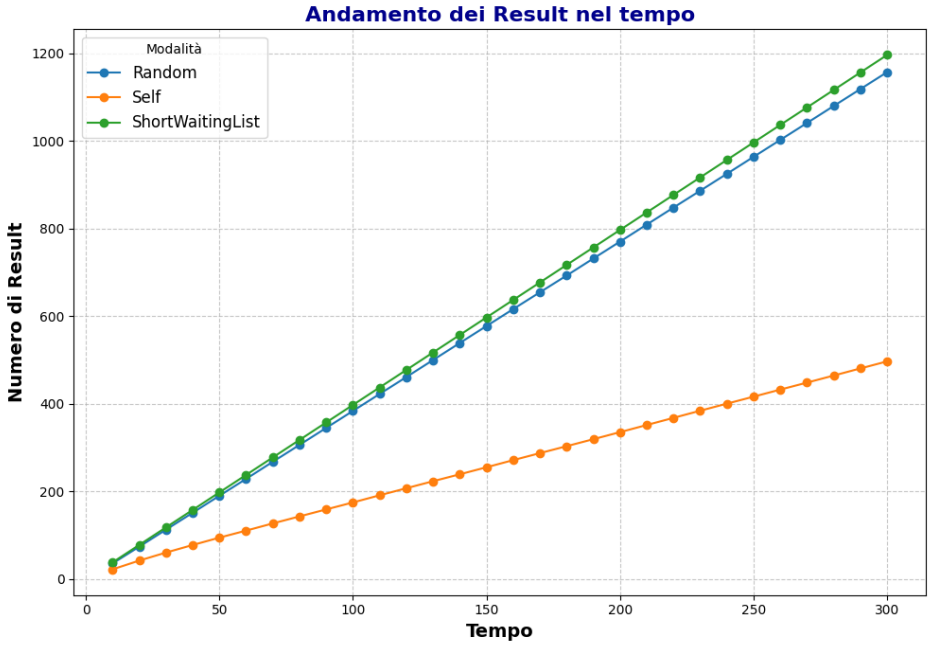


Figure 5.4: Andamento dei risultati prodotti per ogni modalità

5.2 Specializzazioni delle unità produttive

In questo caso d'uso, vengono analizzate tre unità produttive con capacità differenti:

- La **ProdUnit1** può eseguire solo step di tipo A.
- La **ProdUnit2** può eseguire solo step di tipo B.
- La **ProdUnit3** è in grado di eseguire step di tipo A, B e C.

Questa configurazione mette in evidenza la seguente dinamica di ottimizzazione: nonostante l'unità produttiva ProdUnit3 sia progettata per eseguire tutti i tipi di step, durante la simulazione tende a specializzarsi nell'esecuzione di step di tipo C. Questa specializzazione è il risultato di un comportamento del sistema, che ha come obbiettivo il completamento degli ordini. La simulazione assegna gli step alle unità produttive tenendo conto delle loro capacità specifiche. Inizialmente, quindi, ProdUnit3 esegue tutti i tipi di step, ma, con il progredire della simulazione, emerge una specializzazione dell'unità produttiva. Questo accade perchè tutti gli step di tipo C non possono essere eseguiti da altre unità produttive, quindi per completare gli ordini l'unità produttiva 3 inizia ad eseguire solo questo tipo di step. Questo comportamento permette di migliorare l'efficienza del sistema nel completamento degli ordini.

```
_prodUnitsProgram: &prodUnitsProgram
  - time-distribution: 5
    type: Event
    actions:
      - type: simulation.action.RunOneStepAction
        parameters: 1
  - time-distribution: 1
    type: Event
    actions:
      - type: simulation.action.OrderGeneratorAction
        parameters: [*resources, 2, 3]
  - time-distribution: 5
    type: Event
    actions:
      - type: simulation.action.AssignStepToProdUnitAction
        parameters: ["ShortWaitingList"]

_prodUnits: &prodUnits
  - type: Point
    parameters: [ 1, 0 ]
    properties:
      - type: simulation.model.ProdUnit
        parameters: [ "ProdUnit1", [ "A" ] ]
    contents:
      - molecule: CompletedStepA
      - molecule: CompletedStepB
      - molecule: CompletedStepC
    programs:
      - *prodUnitsProgram
  - type: Point
    parameters: [ 1, 0 ]
    properties:
      - type: simulation.model.ProdUnit
        parameters: [ "ProdUnit2", [ "B" ] ]
    contents:
      - molecule: CompletedStepA
      - molecule: CompletedStepB
      - molecule: CompletedStepC
    programs:
      - *prodUnitsProgram
  - type: Point
    parameters: [ 1, 0 ]
    properties:
      - type: simulation.model.ProdUnit
        parameters: [ "ProdUnit3", [ "ALL" ] ]
    contents:
      - molecule: CompletedStepA
      - molecule: CompletedStepB
      - molecule: CompletedStepC
    programs:
      - *prodUnitsProgram
```

5.2.1 Risultati della simulazione

Di seguito sono riportati i risultati della simulazione in 100 iterazioni:

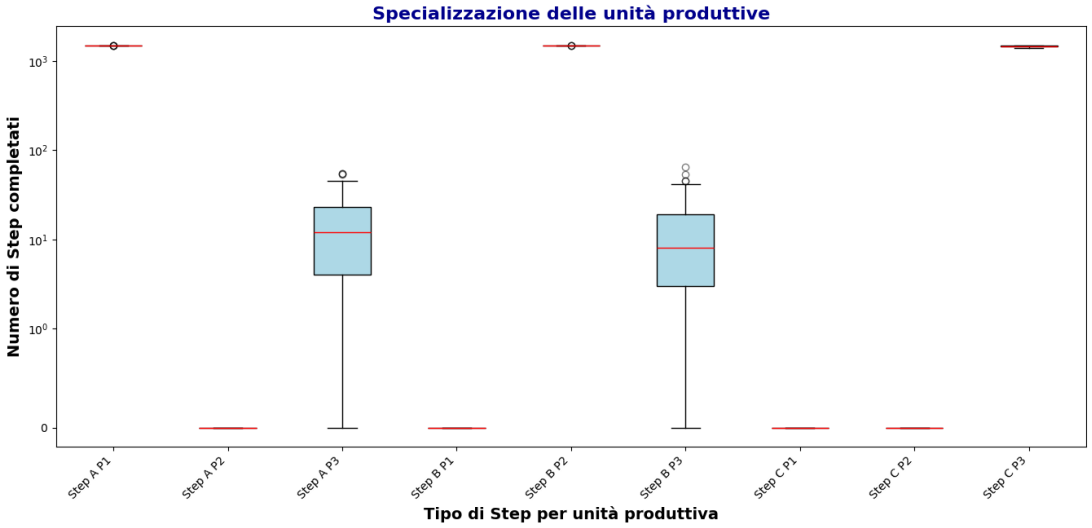


Figure 5.5: Numero di StepType eseguiti da ogni unità produttiva

Questi risultati mettono in evidenza una chiara specializzazione dell'unità produttiva ProdUnit3 in step di tipo C. La configurazione del sistema con altri parametri di simulazione, come l'utilizzo di tre unità produttive che sono in grado di eseguire tutti i tipi di step o la modifica delle frequenze delle azioni, non mette in evidenza ulteriori casi di specializzazione.

Chapter 6

Sviluppi futuri e conclusioni

6.1 Sviluppi futuri

Di seguito vengono presentati alcuni sviluppi futuri che potrebbero essere implementati, ampliando la versione base. Le sezioni relative alle dipendenze, biforcazioni e infornate andrebbero ad arricchire la parte del modello dedicata agli ordini, alle ricette e agli step, permettendo una simulazione più precisa e dettagliata di questi aspetti. Inoltre, la sezione successiva potrà integrare la parte relativa alle unità produttive, introducendo la possibilità di aggiungere nuovi elementi al modello, come team e individui. Questo consentirebbe di ampliare le interazioni tra le unità produttive, estendendo quelle già presenti, come `RandomProdUnit`, `Self` e `ShortWaitingList`.

6.1.1 Dependencies (Dipendenze)

Le dipendenze all'interno di una ricetta rappresentano i vincoli logici che determinano l'ordine in cui gli step devono essere eseguiti. Questa sezione introduce le modifiche necessarie per aggiungere tali relazioni al sistema già implementato.

Aggiunta del seguente attributo alla classe Recipe

```
var dependencies:  Map<Step, List<Step»
```

Modifica del metodo di ProdUnit

`fun executeFromWaitingList()`

1. Prende step dalla `waitingList`.
2. Controlla se gli step presenti nella mappa `dependencies` della ricetta sono stati completati *(step.state == StepType.COMPLETE)*.
3. Se la condizione precedente è vera, esegue lo step.

6.1.2 Branches (Biforcazioni)

Le biforcazioni permettono di definire alternative per l'esecuzione di uno o più step, selezionando la migliore possibilità di esecuzione in base a criteri specifici. Questa sezione introduce le modifiche necessarie per aggiungere il concetto di biforcazioni al sistema già implementato.

Aggiunta del seguente attributo alla classe Recipe

`var branches: Map<Step, List<Step»`

Modifica del metodo di ProdUnit

`fun executeFromWaitingList()`

1. Prende step dalla `waitingList`.
2. Aggiunta metodo che, a partire da una lista di step ne restituisce uno considerando diversi fattori, come:
   (a) Risorse del magazzino
   (b) Costo.
   (c) Tempo.
3. Confronta lo step preso dalla `waitingList` con quelli associati in `branches`.
4. Esegue lo step ottimale.

6.1.3 Batches (Infornate)

Le infornate consentono di eseguire lo stesso step più volte in modo raggruppato. Questa sezione descrive le modifiche necessarie all'integrazione delle infornate nel sistema già implementato.

Aggiunta del seguente attributo alla classe Recipe

```
var batches:  Map<Step, Int>
```

Modifica dei metodi di Step

`fun checkResources()`

Modifica del metodo aggiungendo un parametro `Int` che rappresenta il numero associato allo step nell'infornata (1 se lo step non è presente in batches).

`fun execute()`

Modifica del metodo aggiungendo un parametro `Int` che rappresenta il numero associato allo step nell'infornata (1 se lo step non è presente in batches).

Modifica del metodo di ProdUnit

`fun executeFromWaitingList()`

1. Prende step dalla `waitingList`.
2. Controlla se lo step in questione è presente nella mappa `batches`.
3. Se la condizione precedente è vera:
   (a) Chiama `checkResources` passando l'intero associato allo step in `batches` o `1`.
   (b) Se la condizione precedente è vera, chiama `step.execute` passando l'intero associato allo step in `batches` o `1`.

6.1.4 Approvvigionamento di risorse su richiesta

Un ulteriore miglioramento potrebbe riguardare la gestione dell'approvvigionamento delle risorse. Attualmente, questa fase avviene attraverso l'utilizzo dell'azione `AddResourceToWarehouseAction`. Tuttavia, si potrebbe ottimizzare il processo consentendo che l'approvvigionamento avvenga su richiesta diretta dell'unità produttiva, qualora questa non disponga delle risorse necessarie per eseguire uno step presente nella propria lista d'attesa. Per implementare questa funzionalità, è necessaria la modifica del metodo `executeFromWaitingList` della classe ProdUnit. In dettaglio, quando l'unità produttiva tenta di eseguire uno step e rileva l'assenza di risorse sufficienti, dovrebbe inviare una richiesta per rifornire il magazzino con le quantità mancanti delle risorse necessarie.

6.1.5 Nuovi tipi di interazione tra unità produttive

Un'altra estensione interessante riguarda l'introduzione di nuovi tipi di interazione tra le unità produttive, includendo la possibilità di formare aggregazioni temporanee. Nella versione base del modello, alle unità produttive vengono assegnate delle capabilities attraverso il file di configurazione. Si potrebbe estendere il modello introducendo i concetti di team e individui. I team si comporrebbero autonomamente a partire da individui disponibili, determinando in questo modo le capabilities necessarie. Questa estensione permetterebbe alle unità produttive di formare team in base ai tipi di step presenti nella propria lista d'attesa, assicurandosi di disporre delle capacità richieste per completarli. Per implementare questa funzionalità, sono necessarie le seguenti modifiche al simulatore di base:

- Introduzione della classe `Individual`, che rappresenta un singolo individuo.
- Introduzione della classe `Team`, che rappresenta un gruppo di individui.
- Implementazione di un metodo `reorganizeProdUnitWithCapabilities`:

  Questo metodo, a partire da un'unità produttiva e una lista di capacità, aggiunge un team composto dagli individui disponibili necessari per soddisfare le capacità mancanti.

- Modifica del metodo `executeFromWaitingList` della classe `ProdUnit`:

  Quando l'unità produttiva non dispone delle capabilities necessarie per eseguire uno step, deve richiedere la formazione di un team con le capacità mancanti utilizzando il metodo sopra definito.

Questa estensione renderebbe possibile l'aggiunta di ulteriori tipi di interazioni complesse tra unità produttive, come ad esempio la gestione di aste interne.

6.2 Conclusioni

Il progetto descritto in questa tesi si pone l'obbiettivo di simulare le organizzazioni moderne, caratterizzate da gerarchie piatte. Attraverso l'implementazione del simulatore è stato possibile analizzare e ottimizzare i processi operativi, riconducibili a domini differenti. Il simulatore sviluppato, infatti, offre una soluzione generica e facilmente estendibile, progettata per adattarsi a vari domini.

Nonostante la versione presentata non contenga tutte le funzionalità del dominio analizzato, l'architettura del sistema consente di aggiungere nuovi elementi del dominio e funzionalità più complesse, permettendo di simulare scenari più articolati e ottenere risultati più rilevanti.

I risultati ottenuti nei casi di studio evidenziano la capacità del simulatore di adattarsi a diverse configurazioni di assegnamento, frequenze operative e specializzazioni produttive. Le simulazioni hanno mostrato come alcune modalità operative permettano di bilanciare il carico di lavoro, ridurre i tempi morti e migliorare quindi l'efficienza complessiva del sistema di produzione. In particolare, la modalità di assegnamento `ShortWaitingList` si è dimostrata molto efficace negli scenari analizzati. Tuttavia, è importante sottolineare che le simulazioni realizzate rappresentano esempi semplici di utilizzo, futuri sviluppi potrebbero includere l'introduzione di approvvigionamenti di risorse su richiesta, la gestione di team dinamici e l'introduzione di tipologie di ricette più complesse, che includano dipendenze di vario tipo.

In conclusione, questa tesi dimostra l'efficacia e il potenziale delle simulazioni nell'affrontare le complessità delle organizzazioni moderne, fornendo validi strumenti di supporto decisionale.

Bibliography

[AC13] Pierpaolo Andriani and Jack Cohen. From exaptation to radical niche construction in biological and technological complex systems. *Complexity*, 18(5):7–14, 2013.

[BBCL16] Ethan Bernstein, John Bunch, Niko Canner, and Michael L. Lee. Beyond the holacracy hype. *Harvard Business Review*, 94(Jul-Aug):38–49, 2016.

[BKEI05] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. Yaml ain't markup language (yaml™) version 1.1. Tech. rep., yaml.org, 2005.

[Bon10] Giovanni Bonifati. 'more is different', exaptation and uncertainty: Three foundational concepts for a complexity theory of innovation. *Economics of Innovation and New Technology*, 19(8):743–760, 2010.

[Bro75] Frederick P. Brooks. *The Mythical Man-Month: Essays on software engineering*. Addison-Wesley, Reading (MA), 1975.

[BW08] Eduard Babulak and Ming Wang. Discrete event simulation: State of the art. *International Journal of Online Engineering*, 4(2), 2008.

[CF22] Giacomo Cabri and Guido Fioretti. Flexibility out of standardization. *International Journal of Organization Theory & Behavior*, 25(1–2):22–38, 2022.

[CVR24] Corinna Coupette, Jilles Vreeken, and Bastian Rieck. All the world's a (hyper)graph: A data drama. *Digital Scholarship in the Humanities*, 39:74–96, 2024.

[Dol10] Timothy E. Dolan. Revisiting adhocracy: From rhetorical revisionism to smart mobs. *Journal of Futures Studies*, 15(2):33–50, 2010.

[DSV04] Nicholas Dew, Saras D. Sarasvathy, and Sankaran Venkataraman. The economic implications of exaptation. *Journal of Evolutionary Economics*, 14(1):69–84, 2004.

[DSVW19] Lucia Darino, Marcus Sieberer, Arthur Vos, and Owain Williams. Performance management in agile organizations. *McKinsey & Company*, pages 1–8, April 2019.

[Fio12] Guido Fioretti. Two measures of organizational flexibility. *Journal of Evolutionary Economics*, 22(5):957–979, 2012.

[FZ14] Teppo Felin and Todd R. Zenger. Closed or open innovation? problem solving and the governance choice. *Research Policy*, 43(5):914–925, 2014.

[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.

[Gol16] Eliyahu M. Goldratt. *Standing on the shoulders of giants: Production concepts versus production applications*. Routledge, 2016.

[GOSSR10] Paulina Golinska, Joanna Oleskow-Szlapka, Agnieszka Stachowiak, and Pawel Rudiak. Agent-based model of kanban flows in the environment with high demand variances. In Yves Demazeau, Frank Dignum, Juan M. Corchado, Javier Bajo, Rafael Corchuelo, Emilio Corchado, Florentino Fernández-Riverola, Vicente J. Julián, Pawel Pawlewski, and Andrew Campbell, editors, *Trends in Practical Applications of Agents and Multiagent Systems*, chapter XXXII, pages 267–275. Springer Verlag, Berlin, 2010.

[Gou54] Alvin W. Gouldner. *Patterns of Industrial Bureaucracy*. Free Press, Glencoe, 1954.

[Her14] Tor Hernes. *A Process Theory of Organization*. Oxford University Press, Oxford, 2014.

[JJ13] Thomas Jønsson and Hans Jeppe Jeppesen. Under the influence of the team? an investigation of the relationships between team autonomy, individual autonomy and social influence within teams. *The International Journal of Human Resource Management*, 24(1):78–93, 2013.

[Kle96] Art Kleiner. *The Age of Heretics*. Nicholas Brealey Publishing, London, 1996.

[KS93] Jon R. Katzenbach and Douglas K. Smith. *The Wisdom of Teams: Creating the high-performance organization*. Harvard Business Review Press, Boston, 1993.

[Mad07] Janusz Madejski. Survey of the agent-based approach to intelligent manufacturing. *Journal of Achievements in Materials and Manufacturing Engineering*, 21(1):67–70, 2007.

[Mar82] James G. March. Exploration and exploitation in organizational learning. *Organization Science*, 2(1):71–87, 1982.

[Min83] Henry Mintzberg. *Structure in Fives: Designing effective organizations*. Prentice Hall, Englewood Cliffs, 1983.

[MM19] Joost Minaar and Pim De Morree. *Corporate Rebels: Make work more fun*. Corporate Rebels Nederland, Amsterdam, 2019.

[MM22] Joost Minaar and Pim De Morree. *Start-Up Factory: Haier's RenDanHeYi model and the end of management as we know it*. Corporate Rebels, Amsterdam, 2022.

[MS58] James G. March and Herbert A. Simon. *Organizations*. John Wiley & Sons, New York, 1958.

[Mum06] Enid Mumford. The story of socio-technical design: Reflections on its successes, failures and potential. *Information Systems Journal*, 16(4):317–342, 2006.

[ND06] Fredrik Nilsson and Vince Darley. On complex adaptive systems and agent-based modelling for improving decision-making in manufacturing and logistics settings. *International Journal of Operations & Production Management*, 26(12):1351–1373, 2006.

[PMV13] D. Pianini, S. Montagna, and M. Viroli. Chemical-oriented simulation of computational systems with alchemist. *Journal of Simulation*, 7(3):202–215, jan 2013.

[PP98] Joel M. Podolny and Karen L. Page. Network forms of organization. *Annual Review of Sociology*, 24:57–76, 1998.

[PRK12] Phanish Puranam, Marlo Ravendraan, and Thorbjørn Knudsen. Organization design: The epistemic interdependence perspective. *Academy of Management Review*, 37(3):419–440, 2012.

[PS88] Mike Parker and Jane Slaughter. Management by stress. *Technology Review*, pages 37–44, October 1988.

[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.